Capturing Proof Process

Andrius Velykis

A thesis submitted for the degree of
Doctor of Philosophy

School of Computing Science
Newcastle University
Newcastle upon Tyne, UK

June 2015

Abstract

Proof automation is a common bottleneck for industrial adoption of formal methods. Heuristic search techniques fail to discharge every proof obligation (PO), and significant effort is spent on proving the remaining ones interactively. Luckily, they usually fall into several proof families, where a single idea is required to discharge all similar POs. However, interactive formal proof requires expertise and is expensive: repeating the ideas over multiple proofs adds up to significant costs.

The AI4FM research project aims to alleviate the repetitive effort by "learning" from an expert doing interactive proof. The expert's proof attempts can give rise to reusable *strategies*, which capture the ideas necessary to discharge similar POs. Automatic replay of these strategies would complete the remaining proof tasks within the same family, enabling the expert to focus on novel proof ideas.

This thesis presents an architecture to capture the expert's proof ideas as a *high-level proof process*. Expert insight is not reflected in low-level proof scripts, therefore a generic ProofProcess framework is developed to capture high-level proof information, such as *proof intent* and important *proof features* of the proof steps taken. The framework accommodates *branching* to represent the actual proof structure as well as layers of abstraction to accommodate different *granularities*. The full *history* of how the proof was discovered is recorded, including multiple *attempts* to capture alternative, failed or unfinished versions.

A prototype implementation of the ProofProcess framework is available, including integrations with Isabelle and Z/EVES theorem provers. Two case studies illustrate how the ProofProcess systems are used to capture high-level proof processes in examples from industrial-style formal developments. Reuse of the captured information to discharge similar proofs within the examples is also explored.

The captured high-level information facilitates extraction of reusable proof strategies. Furthermore, the data could be used for proof maintenance, training, proof metrics, and other use cases.

Acknowledgements

Robertai. Dar kartą ačiū.

I would like to thank everyone who has provided me with valuable advice and support throughout my doctoral studies. First and foremost, I wish to thank my supervisor, Cliff Jones, without whom none of this would have happened. I am grateful for the opportunities to meet all the people, see all the places and participate in academic life. Cliff has taught me much and (sometimes unknowingly) allowed me to go after many things related to this research. Most of all, I am grateful to Cliff for being a great friend to me and my family and seeing me through the whole long process.

Leo Freitas has guided me to the world of theorem proving and brought me to Newcastle and to this PhD. I am indebted to him for where I am now. I wish to thank Leo for being a friend, a colleague and a coding partner in hacking tools for Z. I am grateful to him for testing—and actually using!—my tools while they were very raw and kept running out of memory.

I would also like to extend thanks to Gudmund Grov for the technical discussions, to Iain Whiteside, Alan Bundy and all the members of the AI4FM research project for the useful input to the ProofProcess framework and the collaborations in the wider effort of learning proof strategies.

Immeasurable gratitude goes to my wife Roberta, who has created life and held everything together while I was looking for words to go into the thesis.

Thanks to everyone who has endured me taking this adventure. Maybe surprisingly... I would do it again!

This work is supported by UK EPSRC grants EP/H024050/1 (AI4FM) and EP/J008133/1 (TrAmS-2).

Short contents

Contents

List of figures

List of ProofProcess steps

PART I

Reusing proof

CHAPTER 1

Introduction

1.1 Extracting interactive proof strategies

The research presented in this thesis has its origins and directions (and funding) within the AI4FM research project, which aims to "learn" from experts doing interactive proof to improve automation in formal verification [AI4, BGJ09, BGJ10, JGB10, FJ10]. The scope is specifically narrowed to proofs encountered in formal industrial-scale developments (rather than proofs in formalised mathematics), which exhibit repetition and similarities.[1]

Formal specification and verification are used to increase assurance in software and hardware systems, particularly in high-integrity, safety- and business-critical applications. Verification is performed by discharging relevant (often automatically generated) *proof obligations* (POs), which establish properties about the specifications and the developed system. For widespread adoption of formal methods, proof automation is key. The current generation of theorem provers feature powerful AI techniques and advanced heuristics: as a result, they can discharge or assist with a large proportion of these POs. Unfortunately, a small proportion of a very large number is still a very large number. The residual POs require manual, interactive proof and command a large development effort.[2]

This significant interactive proof effort is the problem that the AI4FM project aims to address. Rather than trying to discover general automated reasoning

[1] An extended motivation of this research is presented in Chapter 2, particularly in Section 2.1.

[2] For example, the residual 8% of interactive POs in the *Paris Metro Line 14* development comprise 2250 POs and have taken over seven man-months to complete [Abr07] (see also Section 2.1.4).

heuristics or trying to tweak the formal model with the hope of making it more suitable for the provers, the aim is to accept the challenging POs and assist with the interactive proof construction. A particular opportunity arises from the observation that POs in industrial-style formal developments can be grouped into "families". In each family of POs, usually a single proof idea is necessary and all other POs in the family are discharged "in a similar way". Reusing this single idea within the family would improve the automation of interactive proof—this is the goal of the AI4FM research project:

Hypothesis H_{AI_4FM} *We believe that it is possible to extract strategies from successful proofs that will facilitate automatic proofs of related POs.*

An expert would construct proofs interactively and the AI4FM system would capture and "learn" his strategy, which could be replayed automatically or would be easily adaptable by non-expert users. This "learning" in general should not be equated with *machine-learning*. In fact, the best scenario would see the expert do an interactive proof of a *single* PO within a family. Then the AI4FM system would learn his strategy and discharge other similar POs automatically. With a single source proof to learn from, generalisation is a more suitable approach than machine-learning. To facilitate this, it is necessary to ascertain the expert's idea on discovering the proof: the questions of what constitutes "the idea" and how to capture it are the focus of this PhD research.

1.2 Thesis contribution

Simply put, the main goal of this work is to provide a rich *data source* about interactive proof, so that strategies can be subsequently extracted. Low-level proof scripts (particularly constructed in the procedural proof style preferred in industrial-scale developments) do not carry enough information to facilitate this: they provide instructions to the theorem prover rather than describe the proof. There is a need for a holistic approach at a higher level of abstraction, capable of capturing an expert's proof insight as well as the particulars of driving the proof. Developing such an approach is the aim of this PhD research:

Hypothesis H_1 *Enough information can be collected from interactive proofs to facilitate understanding of expert's high-level reasoning as reusable proof strategies.*

The main questions arising from the hypothesis include: what constitutes the important and sufficient information about the expert's proof and the corresponding high-level ideas; how can they be represented; and how does it relate to the actual theorem prover commands. This thesis presents the following contributions to address the questions:[3]

1. A generic ProofProcess model to describe an expert's *proof process* is introduced. It uses *proof intent* tags and sets of *proof features* to describe high-level proof steps. The model supports complex proof *structure* and *granularity*, enabling description at different levels of abstraction. Abstract proof steps encapsulate the low-level steps within reasoning systems (e.g. actual theorem prover commands). Multiple proof *attempts* capture the process of proof discovery.

2. Novel facets of the proof process are considered as important proof features. Not limited to just the current goal, the proof features also capture "meta-information": e.g. the use of lemmas and their features, proof origin, provenance, specification domain, proof guidance and more.

3. An extension of the core ProofProcess model to capture *proof history* is presented. While not on the critical path to strategy extraction, this additional data can facilitate a variety of other use cases.

Unfortunately, the task of automatically obtaining the expert's actual proof insight resembles mind-reading. Some of the high-level description of the proof process needs to be manually provided by the expert during the interactive proof. However, there are opportunities for automatic analysis—the second hypothesis describes this research direction:

Hypothesis H_2 *Certain information about the proof process can be inferred automatically, via analysis of the proof context and previous proofs.*

This thesis presents approaches and ideas to infer some information about the proof process via analysis of low-level proof commands:

[3]The list of thesis contributions is continued within this section, highlighted using shaded areas in the text.

4. A non-invasive approach to infer proof branching *structure* via analysis of goal term changes is presented.
5. An approach to identify diverging proof *attempts*, recognise proof re-runs and extend existing attempts with new proof steps is introduced.
6. Approaches to automatically infer or derive various types of *proof features* are proposed.
7. Contributions to the development of an abstract model of *proof strategies* and a high-level description of the AI4FM "system" as part of the AI4FM Newcastle team. Work jointly published in [FJV14, JFV13, FJVW13, FJVW14].

The proposed comprehensive description of an expert's interactive proof process uses a large number of data points. Furthermore, the target industrial-scale proofs can be of notoriously large size and involve complex proof objects. Tool-based solutions are needed to support the development and evaluation of the proof process capture approach as well as to provide the captured data for strategy extraction (i.e. the main goal of this work).

Implementation of a proof process capture system forms a major part of the contributions. Furthermore, an objective of building a *generic* framework is achieved by providing integrations with two different theorem proving systems.

8. A prototype implementation of the ProofProcess framework to capture interactive proof process has been developed. The system features a generic core for recording, storing and manipulating the data with an accompanying user interface based on the Eclipse platform.
9. A reusable File History framework has been made available, which provides an efficient solution to record proof script history in a prover-aware manner. It can be used independently of the ProofProcess system.
10. A ProofProcess framework integration with the Isabelle theorem prover [NPW02] has been built. It tracks the asynchronous proof construction and records details about the interactive proof.

11. A ProofProcess framework integration with the Z/EVES theorem prover [Saa97] has been built. It tracks the interactive proof process and captures the prover-specific details.

12. A general-purpose Isabelle/Eclipse prover IDE for the Isabelle theorem prover has been released. It provides infrastructure for the Isabelle ProofProcess integration but is also used standalone.

13. The tools from the Community Z Tools project [MU05] have been improved, including a new integration with the Z/EVES thereom prover. This is joint work with Leo Freitas at Newcastle. The developments provide infrastructure for the Z/EVES ProofProcess integration but the tools are also used standalone.

The proposed proof process capture approach and the developed systems need to be evaluated. The ProofProcess model aims to describe *any* proof. Thus rather than looking at small and fine-tuned examples, the evaluation should consider real-world proofs, which can be inefficient and "ugly". A particular interest lies in the interplay between general-purpose, well-known proof strategies and domain- or problem-specific proof steps. Furthermore, the different approaches to highlighting the key features of the proof context that trigger the expert to take a particular proof strategy need to be explored. This thesis contributes case studies covering these questions. The case studies provide high-level descriptions of different proofs and then propose how the captured proof process information is reused to construct proofs of similar lemmas.

14. A case study in capturing a high-level proof process description using Isabelle/HOL is presented. The case study uses the proposed abstractions to describe a procedural style proof of a lemma from a formal development of a heap memory management specification.

15. The case study also shows how similar proofs can be constructed using analogy. The captured high-level proof process information is reused to guide the proof, suggest analogous concepts and infer new proof steps required to bridge the proof differences.

16. A second case study in capturing high-level proof process using Z/EVES is presented. It covers a proof by an inexperienced user from a formal development of a separation kernel specification. The case study demonstrates that a high-level description helps understanding and streamlining the proof. Furthermore, the case study shows that even an awkward proof can spawn reusable proof strategies.

17. Additional use-cases for the captured proof process data are proposed in the areas of proof metrics, proof maintenance and tutoring, etc.

Publications

Some of the work presented in this thesis as well as contributions to the wider AI4FM research have been previously published or presented in the following:

- Leo Freitas, Cliff B. Jones, and Andrius Velykis. Can a system learn from interactive proofs? In Andrei Voronkov and Margarita Korovina, editors, *HOWARD-60. A Festschrift on the Occasion of Howard Barringer's 60th Birthday*, pages 124–139. EasyChair, 2014. [FJV14][4]

- Andrius Velykis. Capturing and inferring the proof process (Part 2: Architecture). In Alan Bundy, Dieter Hutter, Cliff B. Jones, and J Strother Moore, editors, *AI meets Formal Software Development*, number 12271 in Dagstuhl Seminar Proceedings, pages 27–27. Schloss Dagstuhl, 2012. [Vel12a]

- Andrius Velykis. Inferring the proof process. In Christine Choppy, David Delayahe, and Kaïs Klaï, editors, *FM2012 Doctoral Symposium*, Paris, France, August 2012. [Vel12b]

- Cliff B. Jones, Leo Freitas, and Andrius Velykis. Ours is to reason why. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *LNCS*, pages 227–243, Shanghai, China, September 2013. Springer. [JFV13]

[4] [FJV14] has been submitted in early 2012, but the publication took a long time to appear.

- Leo Freitas, Cliff B. Jones, Andrius Velykis, and Iain Whiteside. How to say why (in AI4FM). Technical Report CS-TR-1398, School of Computing Science, Newcastle University, October 2013. [FJVW13]

- Leo Freitas, Cliff B. Jones, Andrius Velykis, and Iain Whiteside. A model for capturing and replaying proof strategies. In Dimitra Giannakopoulou and Daniel Kroening, editors, *VSTTE 2014*, volume 8471 of *LNCS*, pages 183–199. Springer, 2014. [FJVW14]

The implementations and tools developed during this research are open-source and available at the following:

- ProofProcess framework and prover integrations source code is available at `http://github.com/andriusvelykis/proofprocess`.

- Isabelle/Eclipse has been released and is available from its website at `http://andriusvelykis.github.io/isabelle-eclipse`.

- Community Z Tools improvements including the Z/EVES integration are available from the CZT website at `http://czt.sourceforge.net`.

1.3 My journey[5]

The journey to create and let go of this volume of text in front of you and the ideas inscribed within (as well as the tens of thousands of lines of code safely versioned in Git repositories) has not been direct. In this section, I would like to mention several key moments that have determined the direction of my PhD research, the results of which are presented in this thesis.

The scope of the AI4FM project of which I am a member is very large and challenging. From the outset, the work has been partitioned between the two principal research teams: the Newcastle team provides the proof analysis, proof description and the specification of the overall system; whereas the Edinburgh team builds upon the previous experience with proof planning and AI to define a strategy language and how strategies are extracted ("learned") from the proofs.

I have a background in software engineering in industry as well as experience with formal specification and proofs using the Z notation and Z/EVES (e.g. [VF10]).

[5] This section is written in the first person, as the journey through PhD research is quite personal, whereas the research results now belong to the world.

Because of this, I was eager to utilise my skills in the proof analysis task, which has culminated with this thesis on *capturing proof process*. I am extremely grateful to my colleagues, who have been very supportive of this direction and allowed me to indulge in tool development. However, with the opportunity also came the responsibilities: I have inserted myself into the critical path within the project of "crunching" the interactive proofs to facilitate strategy extraction. One could try drawing parallels between the scope of the task and the page count of this thesis.

An important decision to my work has been the choice of theorem provers. The Rodin toolset has been at the centre of the motivation for this research project. However, the lack of experience with the toolset within the project has mitigated against the use of it. Instead, we looked into proofs done using the Z/EVES prover: it is small and we have access to a number of industrial-style proofs done by the members of AI4FM (myself included). The strategy language developments by the AI4FM Edinburgh team, however, have been building upon the implementation of IsaPlanner and used the more popular but more mathematics-oriented Isabelle prover. Eventually Isabelle became the prover of choice within the project, with new formal verification experiments in AI4FM being done using Isabelle/HOL. However, this choice increased the challenges in my work due to Isabelle's power, expressivity and the blistering pace of development. I have designed and developed the ProofProcess framework to be generic and support both provers. Building an extensible platform and anticipating future extensions to other provers, such as the Rodin toolset, required more effort but resulted in a better-designed, modular and reusable framework.

The choice of the theorem provers[6] resulted in a significant effort being spent on infrastructure development. Z/EVES did not have a convenient API to access the data, whereas Isabelle has been migrating to a new Isabelle/Scala API during the period of this research. Furthermore, the proposed architecture of proof process capture goes beyond any proof script style, thus there was no obvious functionality to reuse and new tools needed to be developed.

Development of the general-purpose theorem proving infrastructure as well as building the new proof process capture functionality (and the theoretical approach) on top of it means that my research, rather than being a pin-point advancement of the overall knowledge, is about *building a platform*. This is a very exciting task, but it appears that finishing and nicely wrapping-up such work within the

[6] And maybe the eagerness to build new things.

timespan of a single PhD research is quite impossible. Nevertheless, this research has identified a number of additional use cases for the captured proof process data as well as opportunities for further extending and polishing the current system. For the next step of tackling the "learning from the expert" problem, the foundations of a proof process capture platform are in place.

The developments of proof strategies and the overall AI4FM system in both AI4FM teams have influenced the direction of how my ProofProcess system was developed. In fact, it now sits conveniently in the middle between the abstract model where strategies are described using high-level metadata and the graph-based implementation of strategies that can be used to replay them, developed in AI4FM Newcastle and Edinburgh, respectively. The large scope of the research and coming from the different sides of the problem prolongs the "meeting in the middle" in AI4FM. However, the ProofProcess architecture, proof process abstractions as well as the low-level links to the theorem prover conveniently position my research results to bridge this gap.

1.4 Thesis outline

This thesis is organized into four main parts: motivation, architecture, implementation and evaluation. The listing of thesis contributions above follows the same structure. The following outline provides a brief overview of each part.

The next chapter completes Part I and extends the introduction by presenting a detailed motivation for this PhD research. It argues that industrial-style formal verification proofs are amenable for strategy reuse. Furthermore, the chapter provides an overview of the related work in representing and reusing proofs and proof strategies.

Part II proposes an architecture and a model to capture the interactive proof process and the expert's high-level insight associated with it. It discusses how a proof process capture system could work; and what comprises a high-level description of a proof process (and how it is represented). Furthermore, a proof history extension is proposed and ways to infer proof process abstractions automatically are discussed. Finally, the discussion on proof strategies links this work with other developments within AI4FM.

The proposed architecture is implemented as the prototype ProofProcess framework, which is described in Part III. The description covers the implementation of

the generic core framework as well as the integrations with the Isabelle and Z/EVES theorem provers.

Part IV provides sizeable descriptions of two case studies used to evaluate the proposed proof process capture framework. Each case study features a high-level description of an industrial-style proof and a discussion on how this information can be reused for similar proofs. The case studies also provide more details about their respective prover integrations. Furthermore, the last chapter summarises the results presented in this thesis and presents an outlook for future research based on the foundations laid in this thesis. It also explores further opportunities to use the captured proof process data.

Finally, Appendix A presents the formal proof process models from the architecture description in a single place for a quick overview.

CHAPTER 2

State of the art and related work

The AI4FM project and this PhD research aim to improve the automation of interactive proof in areas where general heuristics are insufficient. The idea is to capture and benefit from experts' high-level reasoning, stepping back from low-level proof tactics. This chapter provides an extended motivation for this research with the argument that industrial-type proofs encountered during formal analysis and verification of software are amenable to capturing, extracting and reusing proof strategies. Furthermore, this chapter outlines the background and related work: what are the challenges of interactive formal verification; how proofs and proof strategies are represented and abstracted; and how previous proofs are reused to improve automation.

2.1 Formal verification of software

The increasing ubiquity of computer systems in everyday life increases the pressure to produce higher quality software. Formal methods enable the industry to increase confidence in the developed software, via formal specification, analysis and verification. However, in order to increase the uptake of formal methods, particularly beyond safety-critical or business-critical applications, research and development are needed to improve the automation, usability and availability of

techniques, tools and experience of using formal methods. This section presents an overview of formal development and verification of software systems as well as challenges arising in applying them in an industrial setting, particularly from the complexity and size of the formal models, the number of proof obligations, etc. An important observation for the AI4FM research is that proofs can be grouped into "families" that use the same proof idea: a phenomenon enabled by the automated generation of proof obligations, similarities in data structures, etc. The existence of such similarities and proof families raises the opportunity to reuse these proof ideas and has been the trigger for the AI4FM project as well as this PhD research.

2.1.1 Formal methods in industry

Computer systems are taking over many aspects of everyday life, from things people interact with directly (computers, phones, cars) to foundations supporting modern life (e.g. infrastructure controllers, financial systems, communication protocols, etc.). Hardware and software systems grow in scale, functionality and are becoming very complex. The increasing reliance on computer systems and their ubiquity mean that even subtle errors can manifest as extremely expensive[1] system failures or threaten human safety.

Avoiding errors, particularly in safety-critical and high-integrity applications, is a major goal of software engineering: developers need to be able to construct systems that are reliable despite their complexity. A notion of *correct* software emerges: systems that are constructed with full assurance and knowledge of what they do, how they do it and why they work.

One way of achieving such goals is by using formal methods, which are mathematically based languages, techniques and tools used throughout the life cycle of software and hardware computer systems. The mathematical nature of formal methods enables users to produce precise and unambiguous documentation for domain modelling, requirements engineering, specification, design, development, testing and maintenance. Formal verification can be used to reason about properties of the model and to achieve the necessary confidence about the system and the

[1] For example, an error in the floating point unit of the Intel Pentium processor [SB94] cost the company $475 million. Rueß et al. [RSS96] show that formal specification and verification can be done to prove correctness of this non-trivial division algorithm and its hardware implementation, whereas testing alone would not likely have caught the error. Following this, formal verification has been performed for subsequent processors at Intel [OZGS99, Har06], AMD [MLK98], etc.

development process. The use of formal methods does not guarantee *correctness*,[2] but goes a long way towards increasing understanding of a system, revealing inconsistencies and incompleteness, and clarifying the ambiguities [CW96, WLBF09].

The use of different formal methods in industry is well documented by Clarke and Wing [CW96] as well as in a recent survey by Woodcock et al. [WLBF09]. Furthermore, case studies are available from one of the largest "deployments" of formal methods in industry as part of the DEPLOY research project [DEPa, RT13]. The surveys show that the use of formal methods helps detect faults, improves the quality of the software as well as confidence and understanding of it. Furthermore, in many cases the project cost has been reduced by the use of formal methods, particularly the time spent on system integration and testing.

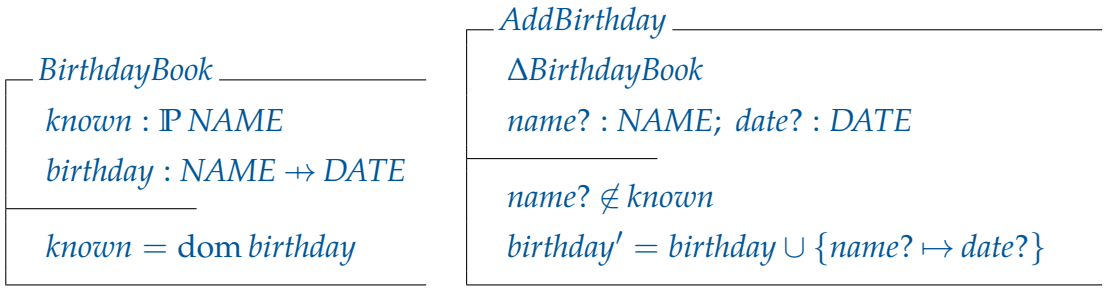
The documented successes, however, come mostly from using formal methods as a *specification* tool: i.e. to model and mathematically specify the system, execute and inspect the formal model, etc. However, *verification* of systems, either by model-checking the specification or using formal proof to reason about its properties, is undertaken less frequently.[3] Surveys highlight that users in industry prefer model-checking their specifications or employing other "push-button" solutions. However, model-checking is inherently limited when verifying complex models due to state explosion. Formal proof, on the other hand, requires expertise and good tool support to be used successfully. Even then, verifying formal specifications using interactive proof is a time-consuming effort.

Formal verification of software, nevertheless, is the goal of the research community that leads towards achieving the ideal of *correct* software. Further improvements in formal methods techniques, application, automation, and—particularly—tools are needed for successful uptake in industry. Tony Hoare has proposed the Grand Challenge in Verified Software, uniting researchers as part of a global long-term Verified Software Initiative to advance the state of art in formal verification [Hoa03, HM05, JOW06, BHW06, SW08].

The research described in this thesis and within the wider AI4FM project aligns with these goals: the aim is to improve the automation of formal verification by reusing interactive proof ideas. The following sections provide more background information about formal specification and verification via interactive proof.

[2]The concept of *correctness* in formal verification is used to state that the system matches the specification. Validating that the specification represents what the user intended is another task.

[3]For example, approximately 20% of the surveyed projects used formal proof, 35% used model-checking, as reported in [WLBF09].



(a) State of a birthday book.

(b) Operation to record a new birthday.

Figure 2.1: Specification of a birthday book "system" in Z notation. Refer to [Spi92] for details on the specification and the Z notation.

2.1.2 Formal specification and verification

A committee of researchers led by Leavens [LAB+06] laid out a roadmap towards achieving the verified software grand challenge via enhanced programming languages and formal methods. At the core of this research direction is formal specification and *correctness by construction* [HC02]: systems are constructed by producing specifications at each level of abstraction, with verification at each level of refinement, from requirements or domain modelling to implementation in code or hardware. The survey in [LAB+06] provides an overview of the state of the art, approaches and future directions to formal specification and verification. Instead of repeating them, only the aspects contributing to similarities in verification proofs are discussed here, as they provide the opportunities for proof reuse.

Formal specification languages such as VDM [Jon90], Z [Spi92, WD96], B [Abr96], or Event-B [Abr10] provide notations to specify a mathematical model of the system. Such a model consists of a definition of a state space, which describes the involved concepts and their properties, as well as operations that manipulate the state and describe the behaviour. Structuring mechanisms to construct models resemble structures in programming languages (e.g. classes and procedures).

A classic example that models an abstract system to record birthdays using the Z notation [Spi92] is listed in Figure 2.1. The *state* of the system is specified as a collection of variables, describing the important concepts. System properties can be specified as *invariants*: e.g. the *BirthdayBook* state requires that all known names have birthdays recorded for them in Figure 2.1(a). The system behaviour can be specified using *operations*: relations between the before- and after-state of a

system. For example, in Figure 2.1(b), the process of adding a new birthday entry is specified by stating that the birthday map in the after-state ($birthday'$) consists of the before-state birthdays with the new entry added. Furthermore, to ensure correct execution, an operation *precondition* requires that the new name is not yet known in the system.

Industrial-scale formal specifications [CB08, DEPb] feature significantly more variables, complex invariants and properties. The formal methods often provide mechanisms to describe system components in a modular manner: e.g. the smaller components are self-contained in regards to their variables and properties—the high-level components are then constructed out of the smaller ones with properties spanning multiple components.

Proof obligations

The process of constructing the formal specification unambiguously using mathematical concepts helps to gain insight and better understanding about the specified system. However, to verify the correctness of the specification, the properties of interest should be specified as *proof obligations* (POs) with an associated formal proof. Such proof obligations could be used to answer questions about system behaviour (e.g. checking that heap memory allocation followed by the deallocation of the same region gives back the original heap memory structure) or verify the key system properties (e.g. that there are no deadlocks), etc. The main types of POs are discussed below.

The proof obligations would be specified mathematically as predicates over some scope of the overall specification.[4] The formal proof would use the definitions from the specification and the semantics of the underlying formal notation to produce an argument that the proof obligation can be discharged within some logic framework. For better assurance, the proofs are mechanised using *theorem provers*: the proof can be discovered automatically or constructed interactively.

In addition to proving important properties about the system, other types of proof obligations are used to ensure that the specification is constructed correctly, that the formal development process is adequate, etc. These proof obligations can be generated automatically from the specifications: the following paragraphs discuss the main types.

[4]The question of whether the specified proof obligation means exactly what the user intends still needs to be raised—an absolute *correctness* cannot be guaranteed.

Partial functions and well-definedness

Partial functions are used to specify and reason about computer programs: e.g. mathematical division, lookups in finite data structures such as arrays or maps, partially-terminating recursive functions, etc. Some formal approaches [BCJ84, BBM98] tackle partiality by using a three-valued logic, where predicates can be *true*, *false* or *undefined*. Other proof frameworks tackle the issue by ensuring that all partial functions are *well-defined* at the point of their use [Abr10, Saa97].

In the latter case, well-definedness proof obligations (also called "domain checks") are generated for applications of partial functions within the specification. Discharging such proof obligations requires showing that the available invariants or operation preconditions restrict the arguments of partial function applications to the domain where the function is defined. If the specification lacks such preconditions, the proofs cannot be completed.

Operation satisfiability

Operation satisfiability proof obligations require showing that operation preconditions are strong enough to make the postconditions feasible. These proofs are also known as *feasibility* proofs. Satisfiability proofs ensure that the operation behaviour is known and valid for all preconditions that allow the operation. The general shape of the satisfiability proof obligations in VDM (adapted from [Jon90, Appendix E]) is the following:

$$\forall \sigma \in \Sigma \cdot \mathit{pre\text{-}OP}(\sigma) \;\Rightarrow\; \exists \sigma' \in \Sigma \cdot \mathit{post\text{-}OP}(\sigma, \sigma')$$

Such proof obligations can be generated automatically for each operation within the formal specification. The proofs generally follow the same pattern of finding an instantiation that satisfies the precondition. In many cases, the instantiation can be directly derived from the operation specification.

Refinement

Key to the *correct-by-construction* formal system development are refinements between specifications [Hoa72, Jon90, Mor94, BvW98]. The approach enables construction of system specifications at appropriate levels of abstraction. Refinement relationships between specifications ensure that the concrete representation of the

system correctly refines the more abstract ones. This provides a link between the abstract system description and the low-level implementation.

Layering the system specification is beneficial to tackling the system complexity and introducing additional details gradually during the system development process. Furthermore, properties can be proved at the abstract level and preserved at the low-level implementation via the refinement process. The approach is well supported by the established formal methods, albeit attitudes about the number of refinement steps differ when it comes to proof.

To ensure that the refinement is correct, a relationship between the specifications needs to be established and it be shown that the corresponding data structures and operations preserve the properties of the abstract specification. In some approaches, the refinement steps are done by a series of small transformations, guaranteeing refinement correctness [Mor94], or the refinement steps are small enough that the proofs can be found automatically [ABH+10]. In other cases (the *posit-and-prove* approach [Jon90, Abr96]), the refinement steps correspond to important levels of abstraction, rather than being chosen to simplify the proofs. To show that the refinement between two specifications is correct, a number of proof obligations are used (adapted below from [Jon90, Appendix E]).

Given the states Abs and Rep from the abstract and representation (refined) specifications, respectively, retrieve function $retr : Rep \to Abs$ establishes the relation from the concrete to the abstract. The *adequacy* proof obligation is used to show that every abstract case is represented:

$$\forall a \in Abs \cdot \exists r \in Rep \cdot retr(r) = a$$

Furthermore, the corresponding refined operations must be valid wherever the abstract ones are, and their results must fall within the scope allowed by the abstract specification: i.e. refined operations must have (possibly) wider preconditions and narrower postconditions. The following proof obligations are used to show this:

$$\begin{aligned}
&\forall r \in Rep \cdot pre\text{-}AOp(retr(r)) \;\Rightarrow\; pre\text{-}ROp(r) \\
&\forall r, r' \in Rep \cdot \\
&\quad pre\text{-}AOp(retr(r)) \wedge post\text{-}ROp(r, r') \;\Rightarrow\; post\text{-}AOp(retr(r), retr(r'))
\end{aligned}$$

The refinement proof obligations are established and proved for all data structures and operations between the specifications.

2.1.3 Theorem proving

Mechanised theorem proving is used to construct a machine-checked argument about posited POs. Industrial-scale formal developments are too large to be able to construct hand-written proofs or rely on manual inspection for verification. Furthermore, mechanising the proofs may uncover issues with the specification. For example, the original development of the Mondex specification used hand-written proofs [SCW00]. Redoing proofs with a theorem prover (e.g. see [JW08]) uncovered problems with the specification, such as missing invariants, etc.

The many aspects and techniques of mechanised theorem proving are covered in depth by Robinson and Voronkov [RV01] as well as more recently by Harrison [Har09]. Both handbooks focus on *automated* reasoning, but also provide extensive overviews of *interactive* theorem proving. Automated theorem proving is preferred for industrial applications, however significant interactive proof effort still remains, as discussed in the next section.

Interactive proof is the focus of the research described in this thesis and within the AI4FM research project. Geuvers [Geu09] provides an introduction to interactive theorem proving, the history of the subject and an overview of various provers. Some of the main concepts relevant to this thesis are outlined below.

Correctness of a proof in mathematics can be determined by reducing it to a series of very small steps, each of which can be verified simply and irrefutably. Achieving such proof granularity, however, is infeasible in human proofs, thus proofs in mathematical texts are usually of a higher level. When it comes to computer-mechanised proofs, there are several approaches to ensuring the correctness. A theorem proving system could produce an "independently checkable proof object" (e.g. *proof term* [BN00]), which could be verified by another (smaller and better-understood) prover. Alternatively, the LCF approach [GMM+78] uses a small kernel of inference rules, which implementation can be verified for correctness. All proofs are constructed using this kernel: advanced *tactics* can be built on top of it that use only the low-level inference rules to ensure correctness.

The proofs in formal verification comprise *proof scripts*: a collection of proof commands. The proof scripts do not necessarily correspond to the proof in logic. They are instructions to the theorem prover on how to rediscover the proof. Some

proof tactics involve automated proof search: adding new lemmas can derail a proof that had previously been successfully found.

Interactive proof representation and abstraction is central to the research described in this thesis. Section 2.2 continues the overview of interactive theorem proving, proof strategies, proof explanation, etc.

2.1.4 Verification challenges

Proof obligations (and associated proofs) in formal verification of computer systems are not difficult *mathematically*. Nevertheless, the sheer number of proof obligations that need interactive proof (and expertise required to discharge them) in industrial-size formal developments is a significant obstacle for industry to widely embrace the use of formal methods.

Abrial [Abr07] reports on two developments in the railway domain: formal development and verification of safety-critical parts of software for a Paris Metro line and a driverless shuttle train at a Paris airport. The formal developments were done using the B formal method with Ada code generated from the specification. The verification amounted to 27,800 and 43,610 proofs in respective cases. Even though most of the proof obligations were discharged automatically by the underlying automated theorem provers, the remaining 8% of proofs were done interactively, amounting to over seven man-months of effort for the verification of the Metro line (the figures are 3% and over four man-months for the airport shuttle—a lower figure due to improved tool automation).

Similar figures are reported in the results of the DEPLOY project [RT13]. Pilot verification of a "start-stop" system in a car consisted of over 4,000 POs, with about 10% of them requiring interactive proof. Similar exercise in a business information systems domain saw 20–30% of POs requiring interactive proof, a case study in microprocessor verification had this number at 36% (over 1,700 interactive proofs).

Even good levels of automation leave a large absolute number of interactive proofs to discharge. The engineer experiences reported in [RT13] indicate an overwhelming preference for "push-button" solutions: either by model-checking or automated theorem proving. Interactive proof requires expertise in theorem proving, and switching to interactive proof is perceived as friction to the overall formal development and system design process. Any improvement to formal verification automation levels is welcomed by industry.

Improvements in automated theorem provers or model-checking can help raise

the level of automation in formal verification. More powerful computers and available memory help with automated proof search or accommodate larger state space for model checking. Furthermore, using more powerful provers can help: Schmalz [Sch12] uses Isabelle/HOL with fine-tuned proof tactics as an automated theorem prover for the Rodin toolset, discharging 76–95% of the proofs.

However, certain aspects of the industrial-style proofs limit the success of the current-generation automated theorem provers. For example, higher-order functions encountered in specifications or finding non-trivial instantiations (e.g. in satisfiability proofs) are difficult to tackle automatically. Furthermore, industrial-style developments feature a large number of new concepts (data structures, variables): new lemmas need to be added about their properties. This would increase automation, but discovering such lemmas is another difficult task, as reported by industrial partners in DEPLOY [RT13]. Furthermore, in large-scale specifications, adding new lemmas for every new datatype results in a very large number of lemmas, which can choke automated proof search.[5] Other partners in DEPLOY report that adjusting the model (e.g. using set operations instead of quantifiers) improves automation [RT13]. Adjusting the model to be more amenable to proof,[6] however, poses the risk of making a mistake in the specification as redefinition may no longer align to what it was originally intended to specify.

Bourke et al. [BDKK12] discuss the experiences and challenges of large-scale formal verification that involves extensive interactive proofs: the pervasive system-level verification of Verisoft [AHL+08], and the operating system microkernel verification in the L4.verified project [KEH+09], among others. Both projects use the Isabelle/HOL theorem prover; the L4.verified repository consists of 390,000 lines of proof with 22,000 lemmas (POs), the Verisoft project has published about 500,000 lines of proof with 8,800 lemmas. This represents an effort of a very large scale: verifying the microkernel in L4.verified took about 20 person-years in total: nine person-years invested in formal language frameworks, proof tools, automation and prover libraries; verification proof was 11 person-years (cf. the implementation of the microkernel was 2.2 person-years) [KEH+09].

The experiences from these verification efforts echo the previous reports about

[5] The Rodin toolset [ABH+10] implements filtering heuristics to remove "unrelated" lemmas from automatic proof search.

[6] Additional abstraction layers, preconditions or invariants could constrain the proof search and increase automation levels. In case of actual issues with the model, Ireland and Llano et al. [IGLB13, Lla12] propose *reasoned modelling critics* that suggest how the model can be changed in some cases of failed proofs.

the importance of automation to improve productivity. Domain-specific automation is needed for industrial-scale developments: the researchers have developed custom proof tactics to achieve this. Proof style and proof comprehension are important to avoid reinvention of proof strategies and ease the introduction of new proof engineers to the project. Interestingly, the OS developers eventually elected to learn interactive theorem proving and contributed to the overall proofs (cf. proof-averse engineers in DEPLOY). The average time to become productive with interactive proof is reported to be two to three months, having advice and training from existing formal methods experts [AJK+12].

In addition to the interactive proof effort needed for the formal verification, significant work is needed to redo the proofs when the specification changes. This occurs when verification identifies errors in the specification, which has to be reverified after fixing, or when the requirements or the design change, altering the specification. Furthermore, introduction of new features can affect existing components and their proofs need to be redone. Klein et al. [KEH+09] report that in the worst case, a change affecting less than 5% of the code base required redoing proofs equivalent to 17% (one person-year) of the entire original proof effort. The issue has also been identified by DEPLOY partners [RT13] and some steps towards the automation of reverification have been done [Meh08] (see also Section 2.3).

2.1.5 Proof families

Interactive proof in industrial-scale formal verification raises significant challenges. However, there are opportunities to reuse proof ideas where proofs can be clustered into "families". In each family of proofs, usually a single high-level proof idea is needed: all other POs in the family are discharged "in a similar way".[7]

The properties of industrial-type formal verification described earlier give rise to a lot of similarities in proofs. For example, the automatically generated proof obligations produce goals of the same shape: e.g. when proving operation satisfiability, proofs always feature search for instantiations of operation post-conditions. Refinement proof obligations between two layers of abstraction use the same retrieve function, thus proofs involve the same definition, etc.

The way formal specifications are constructed also gives rise to similarities, stemming from the same definitions being featured in proofs. Large developments

[7] Anecdotal evidence from a DEPLOY project partner showed that out of 100 residual interactive proofs in an industrial application, five were distinct with difficult proofs, while the remaining 95 "followed the pattern" of one of these five but still required a manual proof.

feature complex, nested data structures that are built up of smaller components. The same state has a number of operations describing its behaviour: all these operations will share variables and invariants coming from the definition of the state and the datatypes (e.g. two operations differ only by a couple of predicates describing different behaviours, but have a large number of common invariants from the state definition). Such construction of formal specification results in proofs that feature a lot of the same definitions and predicates. To improve the automation of interactive proof, lemmas can be defined on the datatypes that encapsulate the important properties. However, in general, the similarities in proof arising from featuring the same concepts are significant.

Proof families could also be identified in reverification of a specification. When the specification changes, its proofs are affected. However, the change may affect only part of the goal and the unaffected parts would be discharged in the same (or similar) pattern to the earlier proof. Furthermore, the specification changes may require adjusting definitions or lemmas used in the proof, but the general idea may remain the same: the new proof would follow the same idea as the original.

The large number of similarities results in theorem proving experts replicating the smaller number of actually new proof ideas for similar proofs. This is different from mathematical proofs, where the problem is usually small but difficult. The opportunity to reuse proof ideas within proof families provides the main motivation for the AI4FM research project and, consequently, for the research described in this thesis. Capturing proof ideas for reuse is the main problem: this thesis focuses on proof abstraction to capture the proof insight. The next section provides an overview of how proofs are represented and abstracted in related research.

2.2 Proof representation and abstraction

To reuse interactive proofs, the embodied proof ideas need to be captured, extracted and encoded as strategies for replay. This section discusses how proofs are constructed, represented and abstracted in interactive theorem proving. Furthermore, existing approaches to encode reusable proof ideas are presented.

2.2.1 Proof style

Harrison [Har96] provides an overview and discussion on interactive proof style, particularly on differences between *procedural* and *declarative* styles. Procedural

proofs instruct the theorem prover more or less explicitly on how to advance the proofs and normally consist of a sequence of proof tactics. When constructing a declarative proof, the user states the facts to be proved and the theorem prover fills in the gaps automatically (if possible).

When comparing the styles, Harrison [Har96] concludes that the declarative style of constructing interactive proof has many advantages, particularly in regards to readability and maintainability. Procedural proofs are difficult to comprehend without re-running them, whereas declarative proofs state what needs to be proved explicitly. Improvements or changes to proof tactics or the proof context can derail a previous procedural proof and inspecting "what should have happened" is difficult as proofs need to be re-run. Declarative proof, on the other hand, signposts the proof and the user always knows what facts need to be proved and may adjust the automation to reach them.

Nevertheless, when it comes to formal verification (the "big, ugly, concrete proofs" [Har96], rather than "clean" mathematical proofs), Harrison comes to the conclusion that the procedural style is more suitable. Formal verification users tend to ignore the proof after the prover accepts it, hence reducing the importance of readability. Furthermore, such proofs feature large terms and quoting them explicitly in the declarative style may be out of the question. Furthermore, the procedural style supports customised and parameterised proof tactics that are fine-tuned to deal with the particular verification problems or within the given domain. In addition to these arguments, procedural proof style can be easier for proof discovery, even blind exploration. In the industrial context, this approach may be easier for proof engineers to try common proof strategies and see where they take the proof, rather than devise proof plans for the declarative approach.

Another distinction in proof style is the proof direction: *backward* vs. *forward* chaining. The backward style goes from the conclusion to the premises, replacing the goal with sub-goals that need to be proved. The forward style starts with the premises and builds up the argument until the conclusion goal is reached. The proof direction often aligns with the particular proof style: e.g. procedural proofs usually employ backward reasoning, where each tactic application transforms the open goal and produces sub-goals for the next tactic to tackle; forward reasoning is easier to encode in a declarative style, where subsequent facts are stated explicitly, then handpicked and building up to the final goal. Some systems support a *mixed* style of reasoning: e.g. *mural* [JJLM91] allows construction of the proof from both directions, neatly visualised as actual forwards and backwards directions in a

natural deduction style; Isabelle/Isar supports switching between the styles, where a declared forward reasoning fact can be proved in a procedural backward style.

The research presented in this thesis focuses on industrial-style formal verification proofs, which, as argued earlier, tend to be procedural and employ backward reasoning. However, some of the concepts proposed in this thesis, such as proof process abstraction using insight and important proof features or proof history capture (Chapters 4–5), would improve proof readability and maintainability (identified as disadvantages of the procedural proof style by Harrison [Har96]).

2.2.2 Representing proofs

Proof scripts are rarely adequate to represent the proof process or capture expert insight on constructing proofs. They are aimed at providing instructions to the theorem prover on how to discover the proof. More structured approaches, such as the Isabelle/Isar proof language [Wen02], target the declarative proof style and provide a human-readable proof document, trying to replicate the style of mathematical proofs. Other approaches discussed here expand on the low-level proof script with information about the wider proof process, provide enhanced representations or different focus on the proof.

Denney et al. [DPT06] introduce *hiproofs* (hierarchical proofs) to express the hierarchical structure of tactics used by a theorem prover to construct a proof tree. The representation introduces a containment relationship within parts of the proof tree, providing decomposition of tactics responsible for constructing the proof. A diagrammatic notation to represent hiproofs is also proposed, depicting tactic calls and the underlying proof tree. The authors restrict hiproofs to a tree structure and provide appropriate semantics for what is a valid hiproof, how they are folded and unfolded, etc. The focus is on proofs in formal logic, rather than in the various theorem prover implementations, hence the restrictions on the representation.

Hiproofs are further developed by Aspinall et al. [ADL10]. A hiproof syntax is introduced and the graphical representation is formalised. This enables defining *tactics* for hiproofs—*hitacs* (discussed further below). Furthermore, Whiteside [Whi13] defines additional hiproof constructs (e.g. *swap* to swap goals) as well as providing a normal form for hiproofs.

Hiproofs provide a clean representation for proofs, identifying sequences of proof steps, proof branches as well as a hierarchy over the proof tree to represent higher-level tactics. In fact, the proof process tree structure proposed in this thesis

(Section 4.3) is inspired by hiproofs. The proposed ProofProcess tree structure relaxes some restrictions to support real-life theorem provers (i.e. the *implementations*, from which hiproofs shy away): e.g. multiple input goals are permitted; the structure enables encoding proof branch merges and thus, technically, no longer represents just trees; non-deterministic proof steps are permitted (steps without a fixed arity of output goals), etc. Furthermore, the ProofProcess structure focuses on higher- and lower-level *proof steps* rather than actual prover tactics as envisaged by Denney et al. [DPT06].

Some work has been done to capture hiproofs automatically from tactic-based theorem provers. Obua et al. [OAA13] present two tools: Tactician and HipCam to record hiproofs in the HOL Light theorem prover. Both can extract the tactic (and their sub-tactic) calls as well as the corresponding construction of the actual proof tree. The implementations approaches differ, but both are invasive: the Tactician tool requires modifying proof tactics used in the proof, whereas the HipCam tool modifies the kernel of the HOL Light prover to access details about proof structure.

Autexier et al. [ABD+06] step away from traditional proof scripts and propose a richer "proof data structure" (PDS) for the Ωmega [BCF+97, SBB+02] system. For each lemma, the system utilises a directed acyclic graph structure to represent its proof. The proof is encoded in a reductive representation (each justification step reduces a goal to sub-goals). Furthermore, the lemmas used in a proof are linked within a wider *proof forest*: i.e. a goal that has already been proved previously would be linked to that previous proof.

Furthermore, PDS provides facilities for capturing facets of a *proof process*: in particular, the support for proof *granularity* and alternative *attempts*. The Ωmega system focuses on *proof planning*, thus different granularity can be used to provide a high-level proof sketch and then refine its steps with lower-level proof plans. For each goal, the proof structure supports recording alternative justifications, e.g. recorded at a different granularity, or completely different attempts altogether. Refer to [ABD+06] for the full description of PDS.

Using different proof attempts for proof development as well as constructing an argument using hierarchically structured justifications in a natural deduction proof has been supported by the *mural* system [JJLM91]. Users could switch between unfinished attempts when exploring the proof. However, when the proof is finalised, the other attempts are cleaned up.

2.2.3 Proof strategies

When it comes to improving automation of interactive theorem proving, the main focus is on capturing and encoding the general patterns in proof so that they can be reused efficiently. Proof *tactics* can be considered low-level implementations of reusable proof ideas, whereas *proof planning* or *proof strategies*[8] often denote higher-level encoding of proof ideas.

Proof planning

Proof planning [Bun88, Bun91] utilises high-level specifications (proof *methods*) of general-purpose proof tactics (or any proof steps) to construct proof plans. A proof planner uses the methods to discover a proof plan: a high-level encoding of a strategy to construct the proof. A proof plan can be used to guide the theorem prover, significantly reducing the need for search. Associated with some proof methods are collections of *proof critics* [IB96]. They provide common patches to help with recovery from a failed application of the method. *Rippling* [BBHI05] is a powerful proof method in proof planning, very successful in inductive proofs. Furthermore, tool support is available for the various proof planning techniques, namely the IsaPlanner [DF03] tool for the Isabelle theorem prover [NPW02] or the Ωmega system [BCF+97]. IsaPlanner implements proof planning and rippling techniques with automated proof search.

From proof tactics to strategies

Tactic-based theorem provers provide general-purpose *tactics* that encode common reasoning patterns or proof search algorithms. Users can extend the system by constructing new tactics; for LCF-based systems, new tactics are constructed in a sound way using the existing tactics or the minimal inference kernel. However, writing such tactics is not trivial and usually requires switching to a prover's implementation language (e.g. tactics for Isabelle are written using the ML programming language). This can deter users and problem-specific reusable tactics are written only when the need is obvious and the benefit is substantial.[9] Research on *tactic languages* focuses on improving the state of the art of developing new proof tactics.

[8] There is no canonical definition of *proof strategies*, different research uses the name to encode ideas of varying levels of abstraction.

[9] For example, for the formal verification of the seL4 microkernel containing 22,000 lemmas and proofs, only 10 to 20 new tactics have been developed [BDKK12].

Aspinall et al. [ADL10] propose a tactic language for hiproofs: *Hitac*. The language supports proof search by providing constructs to assert the control flow (match the current goal), alternate tactics or call them recursively, etc. Evaluation of *Hitac* tactics constructs hiproofs, preserving the link between the tactics and the low-level proof tree. Whiteside [Whi13] extends the tactic language with lemma applications and uses it within his generic declarative proof language *Hiscript*.

Proof-oriented tactic languages are becoming more prominent in popular theorem provers. Eisbach [MWM14] is a proof method language for Isabelle/Isar, whereas Ltac [Del02] and Mtac [ZDK+13] are tactic languages for the Coq theorem prover. These languages integrate with the logic and proof language of respective theorem provers as well as providing concepts to specify pattern matching on goals, combining other tactics or proof methods, backtracking and other proof search functionality, etc. Autexier and Dietrich [AD10, Die11] propose a tactic language for declarative proof, used in the Ωmega proof planning system. Whereas previously mentioned tactic languages are used for procedural-style tactics, Autexier and Dietrich propose a declarative way of writing tactics. When executed, such tactics generate a declarative proof script that is then checked by the prover.

In an approach similar to proof planning, Heneveld [Hen06] suggests specifying when tactics can be applied using a proof feature language—the combination of tactic with features would represent a strategy. The proof feature information is then actively used to suggest applicable proof strategies or run them automatically (cf. tactic languages above, where users can create new tactics but use them manually and explicitly). During a proof, multiple strategies could be triggered by the same set of proof features, so different weights are given to determine which one is the most applicable. Given a set of features and schemas, the weights could be learned automatically.

❦ ❦ ❦

With increasing availability of user-friendly ways to specify proof strategies within theorem provers, users have more opportunities to reuse proof ideas. However, identifying and specifying reusable strategies is still a difficult, manual task.[10] Furthermore, current support for pattern matching and control flow is limited to the current goal only—other information, such as availability of suitable lemmas or proof process meta-information, is not accessible for strategy definitions.

[10] For example, Melis and Siekmann [MS99] utilise students and call for a consortium of interested researchers or organisations to encode reusable mathematics as proof planning methods.

2.3 Proof reuse

Existing attempts to reuse previous proofs can be seen as taking either the "formal methods" or the "artificial intelligence" route. The former aims to generalise or otherwise adapt a proof for similar problems. The latter is useful with a large corpus of available proofs: they are data-mined for common patterns that can be extracted as general-purpose reusable strategies. This section provides a brief overview of the research in these directions.

2.3.1 Proof generalisation and analogy

Reusing proofs for sufficiently similar ones requires generalisation of the source proof, which would make target proof obligations (POs) instantiations of the generalisation. Felty and Howe [FH94] describe a generic approach to generalisation that uses *metavariables* to generalise variables in source POs. These metavariables can be instantiated with single variables or complex expressions in the target proof, allowing replay of the source proof. Johnsen and Lüth [JL04] take generalisation further by suggesting that PO assumptions are made explicit and then abstracting over function symbols and type constants in addition to metavariables. With this approach, source proof reuse becomes generally applicable: e.g. proofs about one data type can be reused with other types. Higher-order *anti-unification* [KSGK07] can be used to generalise terms from different domains and create analogies based on the least general generalisation. These generalisation approaches are illustrated with small examples of proof reuse, usually about proofs in mathematical theory (set, group theory, data types, etc). More research is needed to apply them for POs arising in industrial-scale formal methods applications.

Reif and Stenzel [RS93] present a method (implemented within the KIV system [BRS+00]) to reuse program verification proofs in certain cases when changes in the initial program require the proofs to be redone. The approach reuses old proof fragments (proof subtrees) by moving them to new positions, corresponding to the new positions of program text fragments. The approach targets syntax-driven program verification and requires user interaction for non-reusable parts. Similarly, Mehta [Meh07, Meh08] extends the Rodin toolset to support the reuse of old proofs when a model change triggers PO regeneration. The approach reuses original proof skeletons, which contain reasoner calls (e.g. rule applications) as well as dependencies of hypotheses for each call. Supported PO changes include

adding/removing unused (explicit) hypotheses and variable renaming.

Using *analogy* for proofs can help reuse a proof of a source theorem to construct the proof of a similar target theorem. For example, Vadera [Vad95] implements a *proof follower* for the *mural* theorem proving system. To reuse a proof, first a symbol-to-symbol mapping is established between the source and target theorems. This mapping is then used to perform a backward proof of the target theorem by analogy with the source proof. The approach can suggest which lemmas to use based on which lemmas were used in the original proof: they are either found in the library, or the user is asked to fully instantiate them. The matching between lemmas and terms used in the proofs is quite strict to avoid diverging from the source proof, however some restrictions are loosened in particular cases (e.g. lemma hypotheses are ignored if an exact analogous lemma cannot be found). However, strictness of matching and close following of the proof makes the use of the proof follower quite limited.

Melis and Whittle [MW99] have found more success in using analogy for proofs when working at a higher level of abstraction of *proof plans*. The approach aims to produce a proof plan of the target theorem that is similar to a given source theorem. Proof plans carry additional information about proof, such as justifications of why a particular proof method was selected by the proof planner. When trying to replay the same proof method, these justifications are checked to determine whether the method applies in the target proof. If they fail, a gap is left in the proof plan for the user to fill in interactively. Furthermore, the additional high-level proof information in proof plans is used to constrain the analogy mapping (rather than forcing a strict matching by default) or to suggest new lemmas needed by the target proof plan. Finally, special heuristics are provided to reformulate the source proof plan to make it applicable during replay. The proposed approach is implemented for the CLaM [BvHHS90] proof planning system and the results show improved automation in inductive theorem proving [MW99].

2.3.2 Data-mining proofs

Reusing previous proof information by data-mining for common patterns has seen the most success in automated theorem proving (ATP). A significant problem to automated proof search is premise selection: large sets of available lemmas explode the search space. The MaLARea system [USPV08] has been successful in ATP large-theory competitions by employing a machine learning component to

determine the most suitable lemmas. The approach tries to match the symbols (constants, functions, etc.) and the structure of each conjecture with lemmas that have been used in the successful automated proof of that conjecture.

The ideas are taken further with the MaSh tool [KBKU13]: the machine-learning component for Isabelle's Sledgehammer tool. Sledgehammer can send the current proof context to multiple automated theorem provers and then provide a list of lemmas comprising found proofs that can be reconstructed within Isabelle. The set of features characterising each lemma is extended in MaSh to include type information, the theory to which the lemma belongs, the kind of rule (e.g. simplification), whether the fact is local, whether any quantifiers exist, etc. The set of proofs on which the tool performs data-mining is filtered to exclude basic logic facts, automatically derived definition proofs, and proofs involving more than 20 lemmas. Interestingly, human-constructed proofs are not the best candidates for learning: to improve success, ATP-constructed versions of proofs are preferred (and can be explicitly re-run by the user).

Data-mining existing proofs has also been tried for interactive theorem proving with the aim of extracting proof strategies. This approach follows an assumption that certain sequences of low-level proof steps (e.g. procedural proof tactics or lemma applications) form reusable strategies: i.e. if a similar problem is encountered, the expert would take the same proof steps. Therefore machine-learning techniques are employed to find the common patterns of proof step sequences.

Jamnik et al. [JKPB03] use this approach to learn new *proof methods* in mechanised mathematics (set, group theory) from proofs constructed using the Ωmega proof planner. The proofs and discovered patterns consist of sequences of lemma applications. The learning algorithm uses the least-general generalisation of a set of *well-chosen* example proofs.

Duncan [Dun07] uses data-mining to find proof patterns within the libraries of the Isabelle theorem prover. The expressivity of available tactics, however, raises obstacles to using proof scripts as the data source. For example, advanced proof search tactics such as `auto` are used with different effect in various situations, thus treating them as atomic would make data-mining miss a lot of patterns. Furthermore, tactic parameters (e.g. quantifier instantiations) would over-specify the source proofs and should be ignored. To circumvent the expressivity of proof scripts, Duncan uses low-level *proof term* representations of each proof, constructed using lemmas combined with low-level inference rules.

Both Jamnik et al. [JKPB03] and Duncan [Dun07] generalise the found proof

patterns into tactics/proof methods that can be used in interactive proof. However, there are some limitations to the success of the approach. For example, the identified new proof tactics in [Dun07] are comprised of quite general low-level lemmas, such as implication introduction *impI* or *modus ponens mp*. The created tactics, therefore, are very general-purpose and involve basic theorem proving concepts that users learn quite quickly anyway. The approach is not successful for domain- or problem-specific strategies. Strategies identified in [JKPB03] are more interesting, but require a good selection of source proofs from which to learn: i.e. all proofs have to be from the same *family*.

Komendantskaya and Heras [KHG13, HK14] combine conjecture features and the tactic-based learning into a more comprehensive approach in the ML4PG tool (with results on learning from Coq/SSReflect [GM10] proofs). The selection of proof features characterising each proof step is limited: the principle ones are the three top symbols, names and a number of tactics, and number of subgoals. Pattern recognition tools require that the number of selected features is limited and fixed, thus selecting appropriate ones is important. ML4PG uses clustering machine-learning algorithms to discover patterns in proofs. It can be used to identify patterns in proof libraries or to suggest which proofs are similar to the currently started one. However, the approach requires the user to take the initial steps in the proof so that similar ones are suggested. Furthermore, some of the clusters contain lemmas and proofs where similarity is not always obvious or useful. ML4PG does not attempt to generalise the found clusters into tactics but defers to the user to recognise the similarity and perform the proof by analogy. Finally, the success of learning is higher when a small number of tactics is used (e.g. as in the SSReflect library). For expressive theorem provers, the number of different tactics makes finding patterns difficult.

❦ ❦ ❦

The success of *learning* proof strategies in the context of interactive proof is quite limited. The extracted strategies are too general, it is difficult to capture when they need to be applied, and the expressivity of theorem provers is an obstacle to identifying patterns. Furthermore, data-mining requires a corpus of available proofs from which to learn. Using this approach for a new problem is difficult: problem-specific strategies cannot be learned from a single or a handful of proofs.

The success of analogical reasoning in proof planning [MW99] shows that using higher-level information about proof is important. Capturing such high-level

insight from an expert doing interactive proof would facilitate proof reuse. Furthermore, this information could also be useful for machine-learning approaches as new or improved characterising features. For example, rather than using the top three symbols, it would be more accurate to use the user-tagged ones, or the ones actually affected by the proof. Capturing this high-level insight about an expert's proof process is the aim of the research presented in this thesis.

2.4 Strategy reuse in AI4FM

The development of proof strategies and their reuse within the AI4FM research project (to which the research presented in this thesis also belongs) emphasises the importance of proof meta-information and abstraction. The proposed proof strategies provide a high-level description of how to advance matching proofs. This description utilises various proof meta-information to encode strategy applicability and replay. The particular strands of AI4FM research most relevant to this thesis are the development of an abstract formal model of the AI4FM system, which includes the proposed strategy representation (e.g. in [FJVW14]) as well as the *proof-strategy graphs* [GKL13], which provide a a concrete representation of proof strategies and their replay functionality.

This work is closely related to the research presented in this thesis: the aim of the proposed proof process capture system is to facilitate extraction of proof strategies within the wider AI4FM system. The thesis presentation follows this direction of information flow: Chapters 3–6 present the architecture of the proof process capture system, followed by the discussion in Chapter 7 of the "strategies side" of AI4FM and how the captured proof process information can be reused in the form of proof strategies.

PART II

Architecture for proof capture

CHAPTER 3

Capturing proof insight

This research aims to capture how an expert does interactive proof so that strategies can be "learned" and reused for similar proofs. The *similarity* between proofs encountered in formal verification can rarely be attributed solely to their syntactic resemblance, but rather to the existence of a "general idea" to discharge proofs within the same family. To extract this idea as a reusable proof strategy, it is important to look at the proofs more holistically and at a higher level of abstraction than just the low-level commands of the final proof script.

This part presents an architecture and a high-level approach to capturing the whole interactive *proof process*. This chapter argues that abstraction is important to record the high-level proof insight while preserving the links to the actual proof commands. This enables transferring proof *knowledge* between similar proofs rather than trying to copy the proof scripts. An interaction model to capture such data is also proposed. It couples background "snooping" of prover information with having the expert, who is doing the proof, mark the important features and insight. Chapters 4–5 explore the model of the captured proof process data, together with extensions to record—and re-run—the full proof history. Some of this high-level data can be inferred automatically, either through special heuristics or by learning from previously captured proofs: Chapter 6 explores some of the approaches. Finally, Chapter 7 discusses how the captured information can be used to extract reusable proof strategies.

3.1 Requirements and overview

The need for a comprehensive proof process *capture* arises from the limited effectiveness of the proof scripts as the source for strategy extraction. The view within the AI4FM project is that the "reusability" of proofs lies somewhere higher than the proof script commands, specifically within the high-level proof insight and ideas that lead the expert user to finishing the proof. During interactive proof, the expert materialises these ideas by selecting low-level prover commands that are verifiable by the theorem proving system. The high-level insight, unfortunately, rarely survives this translation and is at best represented within the proof script in some form of free-style comment. While some of the insight may be reconstructible via inspection, it is more sensible to capture and record it while the proof is being developed and the proof direction is fresh in the expert's mind.

To extract and replicate the expert's approach to tackling a specific proof as reusable proof strategy, it is important to capture the key proof aspects that have shaped the decision process. In interactive proof, the direction taken in the proof is affected by a number of proof features, including the current goal[1] and hypotheses,[2] the available proof context and other conditions. Identifying, capturing and structuring all these important factors is the main goal of this PhD research as described in the H_1 hypothesis (repeated here):

> *Enough information can be collected from interactive proof to facilitate understanding of experts' high-level reasoning as reusable proof strategies.*

The name *proof process* is used in this thesis to label the comprehensive account of all this important information spanning the whole proof development: an expert's *proof process* is the story of how the proof(s) were discovered. A full account of a particular proof process needs to capture higher-level features of the proof at various levels of abstraction, portray the full proof development, as well as record how the proof is mechanised within a particular theorem proving system. The most important facets of a proof process are listed below:

- Proof *granularity*: the same proof can be described at several different layers of abstraction: from high-level proof plan to actual prover commands (proof

[1] The word "goal" in this thesis usually means the goal and hypotheses (i.e. what the expert sees in the theorem prover), particularly when used by itself. When talking about specific parts of the "goal", it is differentiated from the *hypotheses/assumptions* that are used to prove the *goal*.

[2] The words "hypothesis" and "assumption" are used interchangeably in this thesis.

tactic steps). The different layers would be linked compositionally, i.e. several lower-level proof steps constitute a single higher-level step. Capturing and interpreting a proof at some arbitrary (even variable) granularity would benefit proof description and reuse, e.g. a proof strategy for similar proofs could have the same high-level proof steps but alternative steps at lower levels of abstraction in each proof.

- Proof *structure*: proof scripts rarely make the proof branches of independent sub-goals (e.g. the base and step cases of inductive proofs) distinctive. The proof steps are normally[3] written in a linear fashion and produce a proof script as a sequence of proof commands. Identifying and capturing a correct proof structure provides a clearer indication of a user's intent as well as yielding a better separation of possibly reusable parts of the proof.

- Multiple proof *attempts*: a development of a single proof often involves multiple attempts at the goal. These can include failed attempts (versions of the proof that are either stuck or just unfinished) as well as different successful versions. All attempts are valuable: a "cleaned up" proof may be more elegant but less general, more efficient but too reliant on automation, while the original success may represent a clumsy but more reusable attempt. Furthermore, failed attempts may still contain generally reusable proof strategies.

- Proof step *intent*: high-level description of the expert's proof direction. The proof steps are translated to prover commands in the proof script, however these low-level commands often fail to describe what the proof step actually does. Proof tactics are mainly generic and can be used in a variety of situations. By capturing the *high-level* description of a proof step (especially when proof commands are grouped into a single high-level step), the *meaning* of the step is defined more clearly. The *intent* can be a simple tag that gives a name to the underlying reasoning of the proof step. By complementing the *intent* with further proof insight abstraction via proof *features* (below), an abstract description of a proof step can be captured.

- Proof *features*: everything that drives the expert's choices. The "why" information about each proof step is particularly important to capture for strategy reuse: "why is a particular proof step taken", "what is required for such a

[3]For example, the *apply*-style proofs in Isabelle, all proofs in Z/EVES, etc.

proof step", "what is the expected feature of the step result", etc. The collection of such proof step triggers, preconditions and postconditions would provide an abstract specification of a proof step that could be mechanised and reused for similar proofs. The proof features describing such important information can be the shape of the goal, the available lemma, certain datatypes or record structures, and many more facets of the proof environment. Such information is usually readily available in the expert's mind *during* the proof process, however deducing it from the finished proof is difficult, even for the same person doing the proof. Capturing the proof features, either automatically or via user interaction, is of high importance to construct useful proof abstractions.

- *Validity* of proof abstractions and prover-specific encodings: all proof process abstractions must be justified by verified steps within a theorem proving system. The link with the underlying prover must be established within other facets of the proof abstractions as well, e.g. the validity of recorded proof branch structures, the definitions of proof features would include terms and proof objects from the proof state and the proof context, the links with actual proof script text could also be recorded, and so on.

The following brief examples (the shaded text and the associated figures) aim to illustrate some of these facets of a proof process, however refer to Chapter 4 for the full discussion. Also, for more detailed examples, refer to case studies in Chapters 11–12. The example proof fragments below are taken from the case study in Chapter 11, particularly lemma *dispose1_disjoint_above*, listed in Figure 11.4.

Figure 3.1 outlines the different *granularity* of a high-level "zooming" proof step. When it is most abstract, it is a single **Zoom** step, but can be decomposed into two high-level proof steps: **Expand definition** and **Cleanup**. Each of these can be further unpacked, revealing lowest-level proof commands in Isabelle/HOL: `apply (unfold F1_inv_def)` and `apply (elim conjE)`. Various abstractions of proof steps are also listed in Figure 3.2.

The capture of proof process *structure* transforms a linear proof script fragment in Figure 3.2(a) in to a tree structure in Figure 3.2(b), separating proof commands that tackle individual sub-goals. Furthermore, each proof step is given an *intent* (**bold font** in each box), describing the actual high-level idea of the proof step. These can talk about the general proof approach (e.g **Trivial assumption**) or be

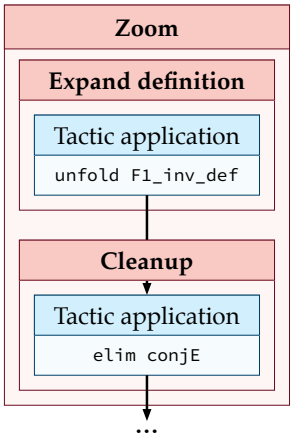


Figure 3.1: "Zooming" steps from the proof of lemma *dispose1_disjoint_above*.

specific to the actual problem domain (e.g. **Show disjointness of above region**). All high-level proof steps are justified by the actual verified theorem prover commands (blue boxes), ensuring the *validity* of the captured proof process.

Within each proof step, the *proof features* describe what triggered it, why the particular proof step was taken. For example, the **Zoom** step records *Preferred level of discourse* (`nat1_map`) feature (among others), indicating that complex function definitions should be expanded to the "maps" level. The **Split disjointness** proof step (step H7) records the specific goal shape (*Goal shape* (`disjoint (?s1 ∪ ?s2) ?s3`)) and the availability of a matching lemma (*Used lemma* (`disjoint_union`)) as key features. For details, see ProofProcess steps H1 and H7, respectively, in Chapter 11.

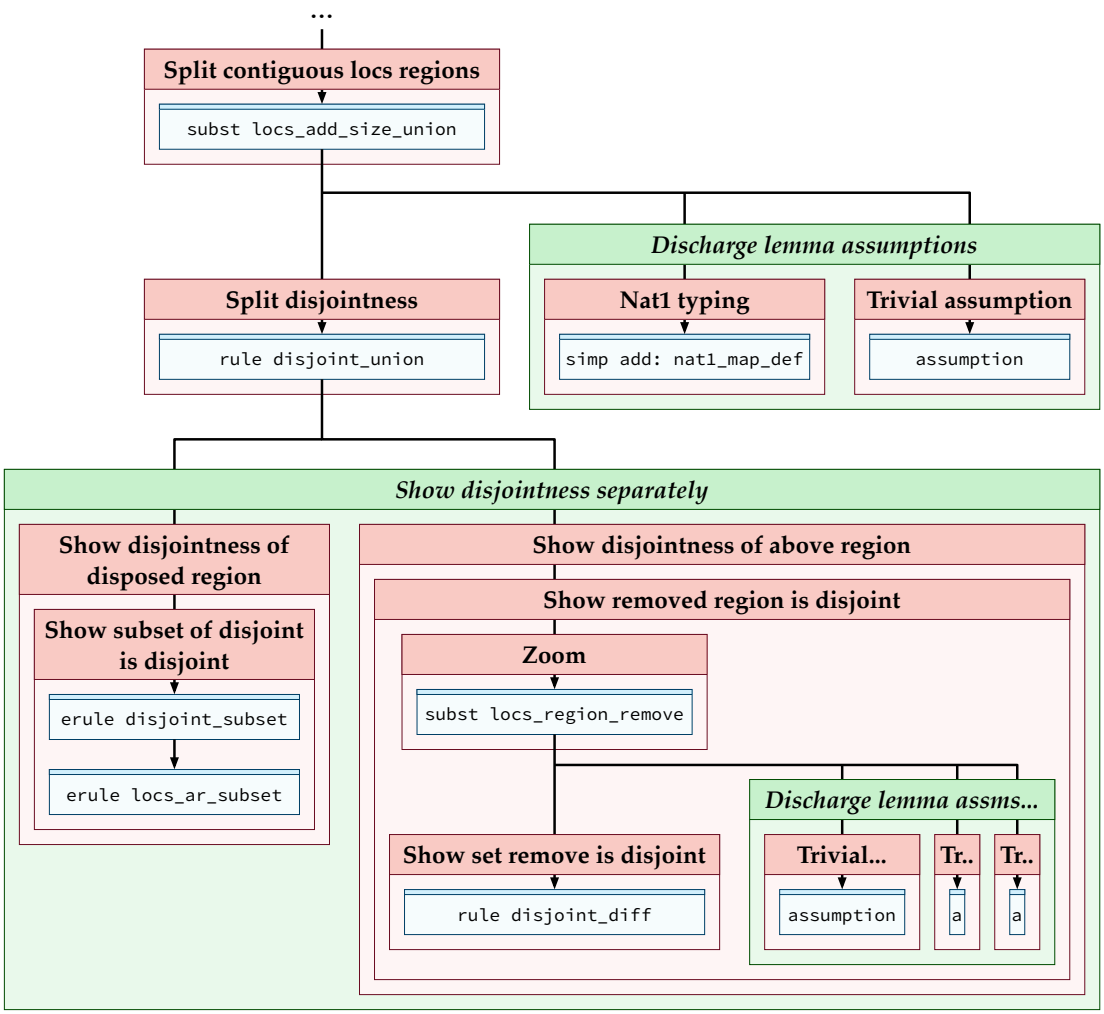
During the proof discovery process of each lemma, the proof process of each different attempt is captured in the same style. Figure 3.3 lists some possible attempt types on a sample formal proof process.

Some of the proposed abstractions, such as the different *granularity*, the explicit proof *structure* as well as proof *intent* descriptions of high-level proof steps can be accommodated within established proof structures such as *hiproofs* [DPT06].[4] However, the full account of a *proof process*, including the capture of full proof development as multiple attempts as well as abstraction via proof features requires more elaborate approaches to proof organisation and structure. Chapter 4 presents

[4]The proof intent would be encoded within composite nodes in *hiproofs*.

```
...
apply (subst locs_add_size_union)
apply assumption
apply (simp add: nat1_map_def)
apply (rule disjoint_union)
apply (erule disjoint_subset)
apply (erule locs_ar_subset)
apply (subst locs_region_remove)
apply assumption
apply assumption
apply assumption
apply (rule disjoint_diff)
done
```

(a) Extract from the proof script.



(b) Captured ProofProcess data.

Figure 3.2: Fragment of proof process capture for lemma *dispose1_disjoint_above*.

- **Lemma** `feasibility-po-20`
  - Attempt: blind failed
  - Attempt: unfinished
  - Attempt: failed subgoal A
  - Attempt: failed subgoal A (different failure)
  - Attempt: successful
  - Attempt: successful, cleaned-up
  - Attempt: successful, alternative, using new lemma
  - ...

Figure 3.3: Sample lemma proof attempt types.

such an approach and an abstract model to capture and accommodate the required proof process abstractions.

The focus on abstraction (high-level proof steps and descriptions, important proof features, etc.) to describe an expert's proof process comes from the AI4FM approach to extracting proof strategies and assisting the user with similar proofs. The aim is that as soon as an *expert* user completes a single proof in a family of similar proofs, an attempt can be made to solve the other proofs within the same family automatically—"in a similar manner"—i.e. with the extracted strategy from the first proof. In an ideal scenario, this would require the expert user to do only the "new" different proofs without wasting effort on similar ones. The possible availability of just a single proof as the source for strategy extraction undermines the use of the majority of AI methods (e.g. machine learning). Therefore, the captured proof process abstractions serve as a way to generalise the single proof into a reusable proof strategy. Note that when the collection of captured proof process data grows, AI methods could be used to further generalise the strategies or extract new ones. However, the focus is on *generalising* strategies from a small number of proofs rather than data-mining from a large corpus of data. The approach aims to find strategies reusable within a family of proofs rather than hunting for too-generic and obvious strategies that are applicable everywhere (cf. data-mining strategies from proof scripts in Section 2.3.2).

The proposed abstractions provide a generic approach to capturing and representing a high-level proof process. The combination of proof intent and proof features together with abstractions of proof steps and structure can be used to

describe proofs within different theorem proving and reasoning systems. The use of these abstractions provides a basis for a generic framework to record, represent and analyse proof processes. Note that the differences in logics, "ways of doing proof" and basic definitions make the interworking of proof processes from different provers unlikely:[5] i.e. the data captured or strategies extracted from one theorem proving system are likely to be ineffective (even unusable) in another. Nevertheless, the proposed high-level proof concepts are generic and thus enable design of generic proof process analysis techniques as well as reuse of implementation components. The framework applications (implementations) need to provide integration with selected theorem provers to capture and represent their proof process using the proposed high-level concepts. A specific implementation would also record the appropriate low-level details (e.g. term representations, proof commands, etc.) and provide the functionality to interpret and manipulate the prover-specific information. The implementations of generic framework components can be reused for shared platforms: e.g. the prototype ProofProcess systems for Isabelle and Z/EVES share a large number of components (see Section 8.3.1).

The proposed abstractions of an interactive proof process would provide a high-level view of how a proof is achieved. Chapter 4 presents details of a model to represent the captured proof process with the required high-level features. It forms the basis of a proof process capture *system*.[6] Recording all the proof process data, however, is not a straightforward activity. Therefore, before delving into the details of the proof process representation, it is important to discuss how such a system would work to capture both the high-level proof insight and the necessary details. The next section proposes a three-way interaction model between the theorem proving system, the user and the proof capture system to achieve this.

3.2 Interaction: the prover, the expert and the apps

The aim of the ProofProcess[7] system is to "follow" the user doing interactive proof and be able to record all necessary details from the theorem proving system as well as to capture the input about the high-level insight from the user. This requires

[5] These differences would prevent the use of a single proof process database for different theorem provers without additional translation effort.

[6] This part focuses on the abstract architecture of a proof process capture system. The prototype implementation of the proposed system is described in Part III.

[7] The ProofProcess name is used to refer to both the proposed architecture and the prototype implementation of the proof process capture system/framework.

the system to capture "live" data from both the prover and the user doing proof as well as perform further analysis to "make sense" of how the proof is advanced and completed. This section proposes an interaction model that describes how such a system would work, what would be captured from the prover and where user input is expected. Furthermore, interaction with other systems that would utilise the captured data is also explored.

The aim within the AI4FM project is to "learn" from the person doing interactive proof and reuse this knowledge to discharge similar proofs automatically. The success of learning a good proof strategy thus depends on a successful proof being constructed as well as on the person doing the proof: awkward proofs would yield awkward strategies. Figure 3.4 presents an example scenario where a theorem proving *expert* is called in to produce a proof, which is used to extract a proof strategy for the *engineers* to reuse later. The scenario describes an industrial setting where there is a more prevalent distinction between a theorem proving *expert* and an *engineer*, who does the formal specification, but is less capable or less fond of proving difficult theorems.[8] From such examples, it is convenient to talk about an *expert* doing the proof, but it can certainly be the case that an engineer working on proofs behaves as the expert and provides excellent—and reusable—proof insight.

In the context of this thesis, the notion of "expert doing proof" is used to emphasise the ideal scenario where the captured proof process information describes good, generalisable proof attempts with enough high-level insight marked. The information captured during the interactive proof will serve as the source for strategy extraction and other uses. Thus the quality of the captured information, and, in turn, the quality of the extracted strategies, can be assumed to be only as good as the proof process of the user doing the actual proof. If the proof process is convoluted and not general enough, or if it lacks certain properties of generalisation or high-level insight, it affects the possibility of detecting reusable strategies. Nevertheless, these cases do happen and the framework accounts for the whole proof process, including the non-optimal proof attempts with missing abstractions, even failed attempts. These cases can also be useful: e.g. the missing abstractions might be *inferred* during post-capture analysis (see also Chapter 6), while the failed attempts may contribute partial strategies for other proofs. In general, a proof process capture system would provide a *progressive enhancement* of

[8] Anecdotal evidence from industrial partnerships suggests that engineers and modellers strongly prefer automatic push-button solutions (e.g. model-checking, automatic theorem provers, etc.) to interactive theorem proving.

A formal specification of a software component is developed in an industrial setting by an engineer familiar with the problem domain. To verify the formal specification, a theorem proving system is used. Proof obligations (POs) are generated about the properties of the specification to be discharged using the prover. A significant proportion of the POs are discharged automatically by the prover using existing heuristics, but a large number of *difficult* POs remain (see Section 2.1). The engineer lacks skills in theorem proving and has limited experience with formal proofs to construct these proofs interactively.

An expert is called to obtain the missing interactive proofs. For an open PO, the expert is able to determine the correct proof direction, choose the appropriate proof commands and define the necessary additional lemmas (e.g. generic lemmas as well as ones specific to the formal model). All this process is recorded by an AI4FM proof process capture system. The expert specifies the high-level proof steps and marks the important proof features for each one. The proof process capture system is able to construct a high-level representation of the proof with important proof insight marked, all the while preserving information about the actual proof commands used in the prover. Next, this data is generalised further to create a *strategy* of how the proof had been achieved.

With the strategy of the first proof being available, the expert can tackle other "new" proofs without wasting effort on proofs similar to the first one. The similar proofs can be completed by the engineer using the AI4FM tools to *replay* the extracted strategies. An appropriate strategy, if available, is suggested to the engineer at any point in the proof. The non-matching steps are either adapted automatically, or the engineer is asked to provide certain adjustments manually (e.g. to define a similar lemma). Furthermore, when the formal specification changes, the extracted strategies may be reused to adjust the existing proofs without needing to call the expert in.

Figure 3.4: A scenario of strategy extraction from an expert's proof.

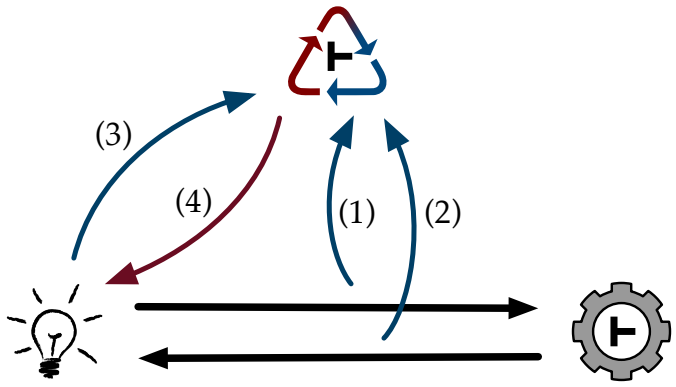


Figure 3.5: AI4FM proof capture process.

functionality: certain basic information can be captured and inferred automatically (e.g. from the prover data, via goal analysis or from previous proof process data), while user input and assistance would be most beneficial and almost required (at least initially) for complex high-level insight (proof intent and features).

Figure 3.5 provides an overview of how AI4FM—particularly the ProofProcess system proposed in this thesis—tracks expert interactions with a theorem proving system. The symbols, clockwise, depict the expert (lightbulb—for inspiration), the proof process capture system (AI4FM logo: *recycling deductions*) and the prover (cogwheel around the turnstile symbol). The basic interaction between the expert and the prover is marked by the horizontal arrows. The ProofProcess system "listens" to this interaction and also captures additional expert guidance. The specific interactions comprise the following (numbered in Figure 3.5):

1. Track the expert's interactions with the theorem proving system.
2. Collect the proof results from the expert's actions.
3. Ask the expert to provide high-level proof process information.
4. Suggest important high-level features and strategies to the expert.

The next sections explore these interactions in more detail: tracking the prover communication as well as capturing the expert's high-level insight.

3.2.1 Wire-tapping theorem prover

Integration with the theorem prover enables the proof process capture system to access the low-level details about the proof being done. By intercepting and recording the communication between the expert and the prover, both sides of the

proof can be captured: the proof commands chosen by the expert as well as the results of these proof commands from the prover. These directions are represented, respectively, by arrows 1 and 2 in Figure 3.5.

Tracking proof activities

In the majority of proof assistants, the expert's interactions consist of editing the proof script (and the formal specification) and then "submitting" it to the prover for verification. The proof part includes selecting conjectures to prove and then choosing proof commands (tactics) and their parameters to discharge these goals. In addition to direct proof activities, the expert may add new lemmas or definitions needed for the proof or even change the existing ones either to fix mistakes or to make it more amenable to proof. As these activities are captured by the proposed system, the overall record would resemble the actual proof script, but the historical dimension (see Chapter 5) would also provide details on the progress and direction: i.e. how the proof script was submitted to the prover, which definitions were added in relation to which proof, when backtracking happened, etc.

If proof assistants provide "advanced" interactivity, capturing such user interactions would yield further information. For example, the user interface to the Z/EVES theorem prover allows selecting part of the goal so that matching lemmas are suggested. The user can select one of the lemmas to be applied (inserted in the proof script and submitted to the prover). Capturing this interaction would help understand the proof process better: e.g. that the user is interested in a particular part of the goal, that the lemma was suggested by the prover, etc. Other advanced interactivity could include, for example, proof "wizards", invocation of automated proof tools, searching for lemmas on the side, consulting documentation, etc. All this is part of the overall proof process and can help infer the high-level ideas.

When capturing the proof being constructed, it is important to focus on the proof "as the expert sees it", rather than on the mathematical representation of the proof. The low-level proof representations, such as *proof terms* in Isabelle, are useful to verify proof correctness, but they do not reveal how the proof was discovered.[9] Recording the proof commands and their results, however, focuses on how the expert sees the proof, what is considered for taking the particular proof steps. A

[9]Proof terms [BN00] capture the low-level λ-structure of proofs and can be constructed by the Pure inference kernel in Isabelle. The representation, however, uses low-level inference rules. The proof trees are usually massive; and advanced proof tactics can add a large number of inferences during proof search, losing relationship with the high-level proof ideas.

proof process representation based on proof commands and user activities has close ties with the expert's reasoning process and lends itself better to generalisation using high-level proof steps and other proposed abstractions (see Chapter 4). Similar approach to annotate proof commands when describing proofs is also taken by other tools (see Section 13.4.2).

Extracting proof details

To capture the effects of expert interactions, the system records how the proof state changes in the prover, e.g. by tracking the before- and after-goals of each successful proof step or marking the failed ones. The prover is also queried for other proof details, in particular what lemmas have been used by the chosen proof tactics, what constitutes the proof context, etc. In most theorem proving systems, the interaction between the expert and the prover can be recorded automatically. Modern proof assistants (e.g. Isabelle, Z/EVES Eclipse) provide application programming interfaces (APIs) that allow implementing proof capture systems as add-ons to the theorem provers.[10] The integration can be described as "wire-tapping" the prover communications, where the proof process capture system is notified of every prover event and is able to query the necessary data to record.

However, further analysis of the captured proof process data would require a deeper link with the prover. For example, to check if a term is of some given shape, term *unification* needs to be performed, which may require internal theorem prover functionality and logic manipulation capabilities as well as the associated proof context to interpret the terms. The theorem prover integration needs to account for this. At the extreme, the theorem prover system itself would need to be altered to expose details necessary for exhaustive capture of important information.[11]

"Wire-tapping" the interaction between the expert and the theorem prover enables the capture of low-level details about the proof, including the proof commands and their results. This process can be done automatically with little impact on the theorem prover performance, e.g. if run as a low-priority activity. The expert may continue doing interactive proof as usual, while additional information is captured alongside with little overhead. The recorded proofs can then be

[10] Chapters 9 and 10 describe integration of the ProofProcess system with Isabelle and Z/EVES theorem provers, respectively. The Proviola tool also intercepts communication to capture proof data (see Section 13.4.2).

[11] Examples of similar tools: Tactician requires modifying proof tactics used in the proof; HipCam modifies the kernel of the HOL Light theorem prover to access details about proof structure [OAA13] (both are used to capture *hiproofs* in HOL Light).

subjected to further analysis, where appropriate abstractions would be inferred (further discussed in Chapter 6). Some of this can be done automatically, in other cases manual intervention is needed to complete the data or confirm the analysis results. The next section proposes how an expert would interact with the proof capture system to provide this information.

3.2.2 Consulting the expert

Arrows 3 and 4 in Figure 3.5 represent the user interaction of the proposed ProofProcess system. Users are expected to assist with the generalisation of the captured proof process. While some aspects of the proof process can be inferred from the proof results automatically, getting the high-level proof insight "out of the expert's head" is not straightforward—if at all possible. Example 3.4 describes a common scenario where an expert is called to do the difficult proofs. The expert's efforts initially are directed towards "new" proofs, which do not have already-captured "siblings". In these cases, assistance from the AI4FM or similar tools is very limited to non-existent: the expert has to work mostly from scratch (the cases where *something* is available are discussed later in the section).

Getting user input

The main interaction scenario with the expert would involve the ProofProcess system prompting the expert on "why are you doing this?" (arrow 3 in Figure 3.5). The expert would have opportunities to provide high-level insight information about *why* the proof is progressing in the chosen direction while doing the interactive proof. This involves generalising and naming the proof steps (e.g. an expert's proof step may consist of several prover commands) as well as identifying specific proof features that trigger and support the chosen proof direction. Example proof features include a specific term that exists in the goal, the fact that a special lemma is key to progress (the existence of the lemma as well as its features is important), and so on. Refer to Section 4.2 for examples of different proof features proposed in this thesis as well as Chapters 11–12 for more detailed case studies.

The optimal user interface to capture user input is under consideration, but options can range from attention-grabbing prompts (e.g. the expert would be asked to name the proof step or mark the important proof features, possibly with some suggestions available) to some less intrusive solutions. An example of the latter is displaying the captured low-level proof process within the proof assistant UI. The

data would be pre-populated with *suggested* high-level insight, which the expert could adjust or supplement with additional proof features during the proof. A less intrusive solution would avoid breaking the concentration of the expert during the proof, but might miss out on some immediate details. In any case, the expert would be able to group the proof commands into higher-level steps, select-and-mark the important proof features in the captured goal terms, and so on. Alternatively (or interchangeably), a top-down approach to proof process development could also be taken: i.e. the expert would first postulate high-level proof steps and then develop appropriate proof commands to "instantiate" the high-level steps in the prover. In both approaches, the end result would be a proof attempt consisting of high-level insight as well as low-level proof commands. Both abstraction and instantiation of the proof process are needed to extract the reusable strategies.

Suggesting to the user

The reverse arrow in the interaction between the expert and the ProofProcess system mainly represents the automatic *suggestions* about high-level aspects of the captured proof process—the activity of "inferring the proof process" (arrow 4 in Figure 3.5). The red arrow colour indicates the analysis (and possibly "learning") activities taking place to produce these suggestions.[12] The suggestions can be produced by analysing the proof state or even by drawing from how the expert marked previous similar proofs.

One of the aims of this research, as proposed in the H_2 hypothesis, is to investigate how certain aspects of proof process capture can be automated. The automated *generalisation* of the proof process is of particular interest: identification of important proof features, suggestion of proof intent and other abstractions. For example, the important terms in the goal (variables, expressions, predicates) can be "guessed" automatically by comparing the goal differences in a proof step, or by looking at the parameters of a proof command. Similar analysis of the proof context can be done to find important lemmas and other proof features. Such analysis can be done even when the expert starts from scratch with a proof, thus helping from the very beginning. The inferred features would speed up the process of collecting the high-level proof information considerably, as the expert would spend less time marking the features and therefore would be more inclined to do it. However, such algorithms can be overzealous in producing suggestions: i.e. *all*

[12] Cf. other proof capture interactions discussed earlier are blue in Figure 3.5, as they represent a more straightforward *recording* of formal proof process.

possible important proof features would be suggested. The expert would still need to select the ones that are actually important and discard the rest. Therefore, the system would produce *suggestions* rather than provide a fully automatic solution. In addition to trimming the selection of suggestions, the expert would also need to mark the other proof features that the ProofProcess system lacks support to infer.

Other suggestions about the current proof process can be attempted by looking at the previously captured proof processes. The system may find patterns among the previously captured data, which can be matched against the current proofs. For example, if the proof command `apply (elim conjE)` was tagged as **Cleanup** proof intent in a previous proof (e.g. in Figure 3.1), when the same command is used again, the same proof intent could also be suggested. Similarly, the same proof features can be marked automatically for the same proof commands, and so on. Section 6.6 continues the discussion on matching with existing proof data.

Eventually, an extracted proof strategy (or multiple) may match the current goal. The user would select a preferred one for replay. If successful, the proof would be completed automatically, with its results captured again, this time populated with high-level information directly from the strategy. Strategy *replay* interactions are discussed further in Section 7.1.

Capturing the appropriate abstractions about the expert's interactive proof process is key to the AI4FM approach to learning proof strategies. In the worst case, a fresh proof idea would require the expert to mark all high-level proof steps and important proof features manually. However, there are avenues where certain high-level information can be inferred automatically. Chapter 6 explores the various approaches to inferring (parts of) the proof process in more detail. The perfect ProofProcess system would leverage the automation capabilities and infer as much of the proof process as possible, making the proof process capture and the subsequent strategy extraction an easy and useful effort to the expert.

3.2.3 *Apps* for the captured proof process

The captured proof process data represents a comprehensive description of an interactive proof process. This information is useful beyond strategy extraction: different standalone applications could benefit from the data.

The core proof process capture functionality can be extended to provide further dimensions of interactive proof. Support for recording the order and timestamps of proof steps would allow *measuring* of the formal proof activities (Section 13.4.3)

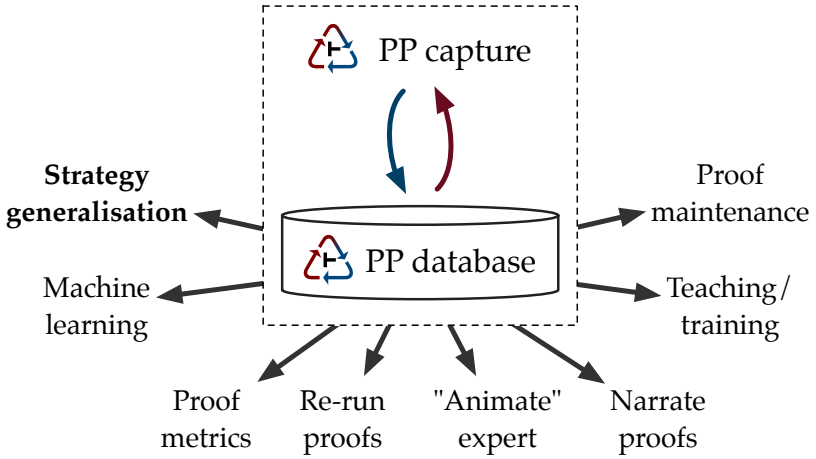


Figure 3.6: ProofProcess system and the proof process (PP) data *apps*.

as well as facilitating a "fly-through" over the whole proof development as an alternative to inspection of individual proofs. Linking the captured data with actual proof script files can facilitate a closer UI integration as well as allowing automatic "animation" of the expert (Section 5.2). These are just several areas of application where the use of captured proof processes is envisaged. Furthermore, in most cases, recording the additional information could be performed in a fully automatic fashion, without affecting the expert doing the proof or the core proof process capture activities.

In the face of these opportunities to record a comprehensive account of the interactive proof process, the role of the proposed ProofProcess system can be extended to support different uses of data. The captured proof process data would comprise a *chronicle* of proof development, providing data to answer the *whats*, *hows*, *whens* and other questions about formal proof. Some of the different uses of the proof process system and the captured data are discussed further in Section 13.4. This section only outlines the interaction between the ProofProcess system and these proposed applications.

Figure 3.6 depicts some of the mentioned applications for the captured proof process data. The wide range of independent applications suggest that rather than building the proof process capture as a single-purpose component (source for strategy extraction) within some massive AI4FM system, the ProofProcess system can be positioned as a *standalone* application. The strategy extraction then becomes one of the "apps"[13] to use the captured data; and the overall interaction model

[13]Similar to how mobile "apps" (applications) in smartphones are often designed to fulfil narrow task sets, rather than being large, Swiss Army knife style applications.

resembles a service-based architecture.

As illustrated in Figure 3.6, the ProofProcess system itself can be considered as two applications: the system to capture and analyse the proof processes as well as a repository (database) to store and access them. A database-like solution would be needed to store the different proof attempts and other historical proof data as well as to accommodate the sheer quantity of proof processes accumulated during a longer use. Furthermore, such separation between capture and storage is warranted when implementations differ: e.g. the capture may require a close prover integration and would likely be implemented within the same platform as the prover, while the database solution may be more generic. Finally, accessing the captured data in a generic way requires a set of APIs to be defined. Some applications, such as strategy extraction, would traverse the whole of captured data to generalise the proof processes into strategies or run machine learning techniques for the same purpose. Others, such as proof maintenance or proof metrics, would be more interested in querying for specific proofs or proof features (see Section 13.3.4 for further discussion on querying proof processes).

The proposed ProofProcess system is more than just a "step" within the AI4FM approach. Proof process capture is central to a number of different possible applications, of which identifying reusable strategies makes up just a small subset. A generic source of data about formal proof can help the users actually doing the proof as well as the people who want to know more about *how* it is done.

CHAPTER 4

Recording proof processes

To succeed in learning and reusing expert's proof ideas, a recording of the interactive proof process must capture the proof insight as abstractions over proof commands, as well as the intended proof structure together with the full history of proof development, rather than just the final low-level proof script. This chapter proposes a model of a proof process that provides said abstractions.

The ProofProcess *model* has been developed to represent generic high-level interactive proof processes. It acts both as the domain model and an abstract data structure used in the framework. This chapter describes the main features of different proof processes and provides a VDM [Jon90] specification of the abstract model. The proposed abstractions include high-level descriptions of proof insight (proof *intent* and *features*), arbitrary granularity of proof steps, proof structuring capabilities as well as identification of different proof attempts that comprise the full proof development. These describe the high-level—"human"—view of the proof process. The validity of the captured high-level proof is ensured by linking it with what actually happened in the theorem prover (proof commands and results verified by the prover). The model ensures that the abstract proof process is justified by the actual proof steps.

The ProofProcess model provides generic high-level abstractions that are theorem prover-independent, but allows for prover-specific extensions to capture the necessary details. There can be many ways to describe proof attempts—the

provided abstractions aim to accommodate any proof process and any proof structure. Designing a *generic* proof process capture framework that is applicable for different theorem provers is one of the goals of this PhD research.

The model described in this chapter focuses on representing and *recording* the proof process. It is extended to support recording of proof *history* in Chapter 5. Ideas on *inferring* parts of the proof process are explored in Chapter 6, while Chapter 7 discusses how proof process data can be used to extract proof *strategies*.

4.1 Proof intent

An expert's insight during the proof is the most valuable part of the proof process capture. It is rarely available within the finished (and likely "polished") proof scripts. Thus existing catalogues of formal development data[1] are of limited usefulness in terms to high-level proof insight.

A finished proof script normally lacks information about "how" or "why" a particular proof command was used. Understanding proofs requires up-to-date knowledge on proof commands, lemmas, and definitions used. Even then, one often needs to re-run the proof script and inspect every goal change in the attempt to recover the general idea of the original proof process. The loss of information about high-level insight encumbers the human comprehension of existing proofs. The information cannot be recovered by machines either: e.g. strategy extraction from proof scripts by data-mining is limited (Section 2.3.2).

Proof script *comments* can sometimes serve to record basic proof ideas for human consumption. A proof author can record the high-level proof plan as an associated text, explaining the strategy and reasoning. However, without links to specific proof steps and additional parsing, this information is not mechanisable and thus is difficult to use for proof automation. Furthermore, the proof insight includes proof structure, abstraction, associated proof elements (lemmas or subterms) and other features. Recording all this as plain-text comments would lose links between proofs and their meta-information.

This thesis proposes to capture the high-level proof insight as combinations of proof *intent* with proof *features*,[2] recorded for each proof step. Proof *intent* assigns

[1] *Archive of Formal Proofs* for Isabelle [KNP]; the *Verified Software Repository* [BHW06]; numerous Event-B examples from the DEPLOY project [DEPb]; formalised base libraries of theorem provers (e.g. Isabelle/HOL); etc.

[2] Proof features are described in Section 4.2.

a name to each proof step, describing its high-level purpose. It can be considered as a *tagging* mechanism for proofs, providing human-readable descriptions for proof commands. The names of proof intents are expected to follow the style one would use to describe the same proof verbally, e.g. they would be answers to questions about the "why" and "how" of the proof process, such as:

- *Why* have these proof steps been chosen?
- *How* is the proof intended to continue?
- *What* are the high-level parts of the said proof?

The name "intent" has been chosen because the collection of such descriptions would tell a story of how the expert *intends* to complete a proof—the high-level proof plan. Note that the *foresight* about how the proof is expected to progress is not prescribed: the intent can equally well be assigned after completing (and fully understanding) the proof. Supplementing the proof with intent information this way would still produce a story of the *intended* proof process for other similar proofs, hence the name.

A collection of proof intents would build up a *vocabulary* of how proof is done. It could be specific to a single proof development, restricted to a certain theorem prover or specification domain, or even collect all possible ways of "doing proof" across different systems and domains. Concepts from mathematical proof and general proof strategies would directly translate to a number of generic intents, e.g. **Induction step case**,[3] **Rippling** [BBHI05], etc. In general, however, the expected proof intents will be specific to either the actual formal development, the specification domain, conventions of the chosen formal method, type of conjecture, particulars of using the theorem prover, or the expert's personal proof style. Such proof intents are much better suited to the approach proposed in this thesis and AI4FM. For example, proof intents specific to the conjecture type or to the current specification describe strategies used to discharge proofs in these specific families. They are not likely to be applicable to generic problems, however. Learning strategies specific for these proof families would improve automation in situations when programming generic heuristics is not worthwhile.

This thesis is not advocating that capture, learning and replay of strategies is somehow superior to developing and programming generic heuristics or advanced

[3]Proof intent names in this thesis are typeset in bold, e.g. **Intent name**.

tactics. Where generic strategies can be identified, developing direct automation techniques would be a better approach than learning the proof strategies from the expert. However, learning strategies can be beneficial for the various problems and proofs not covered by the generic approaches. These actually include the majority of formal developments: even if the essence of the proof is a simple induction or application of the *rippling* strategy, getting down to the essence of the proof or filling in the gaps of the generic strategy would still be domain-specific and require corresponding specific intents to describe it.

Proof *intent* is a quite generic concept, which enables arbitrary "tagging" of proof steps with names representing high-level proof insight. Some examples of intent types are listed in Figure 4.1 (also refer to case studies in Chapters 11–12 for details on actual proof intents used in their proofs). No restrictions are placed on the proof abstraction represented by intents: they can range from very broad proof activities to quite detailed strategy tags. For example, the same proof can be marked with **Induction** intent or a combination of **Induction base case** and **Induction step case** proof intents. These proof intents represent different levels of abstraction and even suggest nesting: e.g. step and base cases would be *contained* within the **Induction** intent. Furthermore, the same proof step can be tagged with multiple intents, each describing a particular facet of the proof step. For example, intents **Show disjointness of disposed region** and **Show subset of disjoint is disjoint** in Section 11.2.4 describe the same high-level proof step: its location within the proof as well as its set-theoretical meaning.

In addition to the "contains" relationship between proof intents of varying abstraction, other relationships can be proposed. Proof intents could be organised into a taxonomy using a "specialises" relationships between them, such as:

- **Peano $\mathbb{N}$ induction** specialises **Induction**;
- **Complete $\mathbb{N}$ induction** specialises **Induction**;
- **Structural induction** specialises **Induction**.

The specialisation allows the establishment of structure within the vocabulary of proof intents. Furthermore, it suggests possible *alternative* proof intents when extracting proof strategies.

In the ProofProcess model, proof intents are represented as an open set of tags. The set may be pre-populated with names for generic proof steps, but mainly will be populated during proof process capture with custom user intents.

General proof:

Induction, **Set up induction**, **Base case**, **Step case**, **Expand definition**, **Insert lemma**, ...

Specialised proof (also some examples in [FW14]):

One-point witnessing, **Invariant breakdown**, **Hidden case analysis**, ...

Proof scope management:

Zoom (expand definitions selectively until the preferred level of discourse), **Extract sub-state**, ...

Datatype (e.g. set-theoretical):

Show set remove is disjoint, **Show subset of disjoint is disjoint**, **Nat1 typing**, ...

Domain/specification:

Show disjointness of disposed region, intents specific to rail/automotive/aero/other domains, ...

Conjecture types:

Existential precondition (for feasibility proof obligations), **Type witnessing** (for axiomatic checks), ...

Formal method/prover:

Expand schema, **Expand operation**, **Trivial assumption**, **Substitute assumption equality**, ...

Making theorem prover "happy":

Reorder assumptions, **Bridge types**, **Split on lemma assumptions**, **Do backward proof** (e.g. simulate a backward proof step when the prover does not support it), ...

Other:

Prove automatically/blindly, **Cleanup**, user-specific approaches to doing proof, ...

Figure 4.1: Examples of proof intent types.

$$Intent = \textbf{token}$$

$$ProofStore :: intents : IntentId \xrightarrow{m} Intent$$
$$\ldots$$

In the abstract model, the actual representation of intent is not important, hence the **token** type. Proof intents are reusable: the same high-level strategy in different proofs would be marked with the same intent.

This section proposed using proof *intent* as tags for expert's high-level ideas about the proof process. During strategy extraction, they will become names of strategies and sub-strategies (see Chapter 7). Intents provide abstraction over proof commands (steps) and give a high-level overview of the proof. Further information about the high-level insight of the proof process, in particular what *describes* the underlying proof strategy, are recorded as proof *features*, which complement the proof intent information.

4.2 Abstraction using proof features

Proof *features* provide instruments to mark important details about each proof step. There is a lot of information associated with each proof step: e.g. the starting goal and proof context, the proof command(s) used, the resulting goal and the new proof context, etc. Proof features provide abstraction over all this information by highlighting only the key details and possibly generalising them further.

Complementing proof *intents*, which name the expert's high-level proof steps (strategies), proof *features* record particulars about what triggered the expert to choose each proof step, what is used or needed by the proof step, and what are the key results. Some examples of proof features are listed in Figure 4.2, but many other types of proof features are expected (see also case studies in Chapters 11–12 for examples). Therefore the mechanism is open-ended and is designed to record all relevant information about the proof process in a generic manner.

In the ProofProcess model, proof *features* are represented as named predicates with parameters that can include any type of proof object: e.g. goal terms, lemmas, etc. The model distinguishes between reusable proof feature definitions (names)

Syntactic/existential:

Has symbol ($\bigcap$), *Top symbol* ($\Rightarrow$), *Goal term* (*extpid′*),
Not (*Assumption term* (*extpid′*)), ...

Generalised shape:

Assumption shape (*?p1* $\wedge$ *?p2*), *Has shape* (*locs_of ?d* (*?n* + *?m*)),
Goal shape (*?f* (*?e*) : $\mathbb{N}_1$), ...

Parameterised:

Type (*x*, $\mathbb{Z}$), *Subterm shape* (*?s*, *?a* $\Rightarrow$ *?b*), ...

Lemma use:

Used lemma (*disjoint_union*), *Lemma shape* (ran *?a* = *?b* $\Rightarrow$ *?a* = *?c*), ...

Link before/after states:

Measure reduction (), predicates involving variable before/after states, ...

Datatype meta-information:

Data invariant (*F1_inv*), *Operation schema* (*DeleteAllProcesses*), ...

Structural (complex datatypes, functions):

Contains (*PTab′*, *extpid′*), *Contains* (*finite*(), *insert*()),
Type (*student*, *?rec_type* :: *lastName* : *Char**), ...

Proof context/environment:

Origin (Feasibility PO), *Provenance* (**Expand definition**),
Domain (Automotive), *Key datatype* (*PTab*), ...

Proof guidance:

Preferred level of discourse (*nat1_map*), *Focus symbol* ($\lhd$), ...

Proof command configuration:

Disabled lemma (*subst_App*), *Simplifier depth* (2), ...

Proof structure:

In case split (), *Case number* (#1), *Number of goals* (0), ...

Arbitrary/user notes, comments:

After relaxing break (), *Consult textbook* (), *No idea where to start* (), ...

Figure 4.2: Examples of proof feature types.

FeatureDef and their instances *Feature*, which instantiate feature definitions with parameters for each proof step: e.g. goal terms, lemmas,[4] etc.

$$
\begin{aligned}
&\mathit{Feature} :: \mathit{name} \quad : \mathit{FeatureId} \\
&\qquad\quad \mathit{params} : \mathit{FeatureParam}^{+} \\
&\qquad\quad \ldots \\
&\mathit{FeatureParam} = \mathit{Term} \mid \ldots \\
&\mathit{ProofStore} :: \mathit{features} : \mathit{FeatureId} \xrightarrow{m} \mathit{FeatureDef} \\
&\qquad\quad \ldots
\end{aligned}
$$

In the text, proof features are written in a free style, as feature names with parameters in parentheses, e.g. *Goal shape* ($?x \in ?S$).

The choice of named predicates to represent proof features gives good flexibility in supporting various types of proof features. The flexibility in representation is important, because there are many different things (and ways of describing them) that can influence the expert's proof process. For example, a specific proof direction can be triggered because the goal has a certain function symbol (e.g. *Top symbol* ($\bigcap$) proof feature) or because the expert had a coffee and went for a quick walk (*Relaxing break* () feature).[5] The latter humorous proof feature is used to emphasise that it is impossible to anticipate everything that affects the proof process. Loosely phrased, it is important to be able to record *everything relevant to the proof process* for *any possible proof intent*.

The requirement for proof features to be expressive (so that unknown types of proof features can be recorded) is countered by the need to mechanise them for use in automated scenarios. In order to be able to infer proof features automatically (Chapter 6) or to use them in strategies (Chapter 7), the different proof feature types need to be known and implemented in the system (e.g. to be able to check if a proof feature matches a goal). Named predicates allow for the encoding of necessary parameters for mechanised proof features: e.g. the name of a proof feature identifies its implementation in the system, which expects a certain order,

[4] Lemmas are also considered to be *Term*s within the system. See Section 4.6 for details.

[5] The latter proof feature was suggested by J Strother Moore during Dagstuhl Seminar 12271: *AI meets Formal Software Development* in 2012. When asked about what triggers him to take a specific proof strategy, he explained that sometimes when stuck in a proof, he goes for a walk and figures out the way to proceed with the proof while walking.

number and types of parameters.[6] Named predicates can also be used to record unknown ("custom") proof features with arbitrary names and parameter lists.

4.2.1 Types of proof features

Similar to *Intent*, proof feature definitions *FeatureDef* constitute a *vocabulary* of proof features for a captured proof process. Some of the different proof feature types and their scope are explored below.

$$\mathit{FeatureDef} = \mathit{KnownFtr} \mid \mathit{CustomFtr} \mid \ldots$$

$$\mathit{KnownFtr} = \mathit{TermFtr} \mid \mathit{ShapeFtr} \mid \mathit{UsedLemmaFtr} \mid \mathit{ContextFtr} \mid \ldots$$

Proof features can cover anything available in the proof context, both the actual proof objects within the theorem prover and meta-information about the proof that might only be available in the expert's mind. This is needed to match the context that the expert considers when doing proof: the problem conjecture itself, all the tools and techniques available, the facts and lemmas already established, knowledge about the specification, conjecture, its datatypes and functions, knowledge about the prover, its configuration, techniques and workarounds, and more. The primary source for important features is the current *open* conjecture or goal (the *before-state* of the proof step). The available (and suitable) lemmas as well as other proof elements will also be a source of important proof features. Moreover, proof features in the *after-state* of the proof step (i.e. after the proof command is applied to the open proof state) would record what outcome has been *expected* from the proof step taken: this information would afterwards specify what results the extracted strategy should produce. The proof feature examples in Figure 4.2 give a glimpse of how this proof context could be described. Several proof feature categories are discussed further in the following paragraphs.

Proof features will be generalised during strategy extraction to specify when a strategy is applicable to the goal. Chapter 7 discusses strategies in more detail, but some ideas on how specific feature types could be generalised in strategies are also discussed in the following paragraphs.

[6] It is necessary to ensure that a proof feature is marked with a correct number, order and types of parameters to match the implementation. This could be enforced by advanced UI solutions, suggested via proof feature descriptions or worked around by flexible parsing of parameters.

Existential (term, symbol) features

One of the main triggers of choosing a proof step is that a *certain term or function* exists within the goal. These terms are often targets of the selected proof commands, e.g. they are rewritten to simpler expressions. The examples could include features such as *Has symbol* ($\bigcap$) (or a specialised "top" version, e.g. *Top symbol* ($\Rightarrow$)). Feature *Goal term* (*extpid'*) marks the existence of a certain term (*extpid'*) in the goal, and so on. Often the existence feature needs to be specialised to indicate that a certain property or type of a term is important: e.g. *Inductable* (x), *Type* (x, $\mathbb{N}_1$), etc.

$$TermFtr = ExiSymFtr \mid TopSymFtr \mid ExiTermFtr \mid TypeFtr \mid \ldots$$

Proof features in general are quite open-ended and their meaning is denoted by the human-interpreted feature *name*. This approach gives flexibility in defining features but may impact automation of their reuse (see discussion on *custom* features later in the section).

In strategies, these proof features are likely to be used as-is, i.e. if the same symbol or term is found in a similar proof (and other proof features align), the same strategy would be suggested. However, one avenue of generalising is replacing the indicated symbol with a similar one: e.g. set union $\cup$ with sequence concatenation $\frown$. For particular problems involving set union or sequence concatenation, high-level proof strategies can be similar.

Shape features

Some of these features could be defined in a more formal manner by using a *term shape* to describe the term or function. Term shape is defined by using placeholders[7] in place of all irrelevant term parts. For example, some of the features mentioned above can be redefined using term shapes: *Has shape* ($\bigcap ?S$), *Goal shape* ($?a \Rightarrow ?g$), *Subterm shape* (x, $?n{:}\,\mathbb{N}_1$). To be able to use this information in strategy replay, some *pattern matching* functionality must exist to check whether a term matches the recorded shape. Luckily, such functionality often is available within theorem provers.[8]

[7] Placeholders are written with a leading question mark: e.g. *?var*.

[8] For example, Isabelle supports defining and instantiating *schema terms* natively.

$$ShapeFtr = MainTermShapeFtr \mid SubTermShapeFtr \mid \ldots$$

In addition to being easier to mechanise, shape features are already generalised for use in proof strategies. In using the shape features, the expert indicates that the encoded properties of terms are important, not the existence of the terms themselves. Thus the extracted strategies would match goals with the same properties, e.g. having the same sub-term shape, the same type, etc. While it may be possible to generalise even further and replace the function symbols in shapes with similar (alternative) ones, it would be better just to go with the expert.

Used lemma features

During proof, suitable lemmas or axioms are used to transform open goals. Therefore these lemmas—their *properties* in particular—need to be recorded as important proof features. The *used lemma* features capture both the fact that *a lemma is needed* to advance the proof, as well as *why* the lemma is needed: e.g. how the lemma applies to the goal, what goal transformations it provides, and other properties.

The fastest way is to mark the fact that a particular lemma was used in a proof step, e.g. *Used lemma* (*locs_add_size_union*). This records that the lemma was important to the proof. The use of a lemma may have been a deliberate choice of the expert: e.g. it could have been manually selected and used in the proof command such as `apply (subst locs_add_size_union)`.[9] In other cases, the lemma may have been used automatically by an advanced proof tactic such as `simp` or `auto` in Isabelle. After inspection, the expert may mark some lemmas among the used ones as particularly important to the success of the automated step.

Furthermore, it may be worthwhile marking the lemma properties which made it suitable for the application. This is easier if the lemma was selected on purpose, as the expert already knows why it was needed. By indicating the important properties of the lemma, e.g. *Lemma shape* (ran $?a = ?b \Rightarrow ?a = ?c$),[10] the expert generalises its use. However, if the lemma was used automatically, figuring out which of its properties are important can make the overhead of marking the proof

[9] See Section 11.2.2 in the heap case study for details on the actual use of this lemma.

[10] This lemma shape indicates that the link between range of a map (ran $?a$) and the whole map ($?a$) is important. See Section 12.2.2 in the kernel case study for details on the actual use of the lemma and its properties.

features too big. Nevertheless, marking the fact that a particular lemma was used may be enough, or the important proof features may be learned or generalised automatically afterwards.

The used lemma proof features can also be used with named local assumptions and hypotheses, which can be treated as lemmas within the proof process. This results in a more general version of the captured proof process: for similar proofs, it would not matter whether the lemma is standalone or one of the hypotheses.

$$UsedLemmaFtr = UsedLemmaNameFtr \mid UsedLemmaShapeFtr \mid \dots$$

In the extracted proof strategies, used lemma features would indicate that a lemma is needed for a successful strategy application. In many cases, the exact same lemma is reused and the user would replay the strategy without issues. Otherwise, a *similar* lemma may be needed. The used lemma features provide hints about the shape of the expected lemma, which the user could easily add (or it could be generated automatically).

Structural features

Composite terms, such as complex function definitions, named *record* data structures, and others, can contribute to interesting proof features. In the goal, such terms are often featured *unexpanded*, as function or record names: e.g. $finite(S)$, where $finite()$ is a constant definition; $PTab \Rightarrow nextupid \in \mathbb{Z}$, where $PTab$ is a Z schema [WD96]. The actually important terms may be nested within these definitions or be otherwise related to their structure, thus proof features are needed to describe the structure of proof objects.

For example, say an instance of the record

$$\begin{aligned} Person :: firstName &: Char^* \\ lastName &: Char^* \end{aligned}$$

is used in a goal unexpanded as $student : Person$. If the important term in a proof step is a field of this record (e.g. $student.lastName$), this must be recorded in a proof feature. Such an approach, however, requires marking terms nested within the definition of records. Situations like these often occur in industrial-style proofs

(Section 2.1), where deeply-nested data structures are used to represent software or domain models. Proofs about them frequently involve sub-terms within records (e.g. verification of security properties when a data field changes) and the proof process must capture this information. Analogous situations arise for composite *functions*: i.e. when the way a function is defined is important to the proof.

The most basic of such structural features record that one definition is contained within another, e.g. *Contains* (*PTab′*, *extpid′*) indicates that the *extpid′* variable can be reached by expanding the *PTab′* schema; feature *Contains* (*finite*(), *insert*()) notes that *finite*() is defined in terms of the *insert*() function; etc.

Some structural proof features can be defined using special types or shapes, which allow encoding structural information. For example, to indicate that term *student* having *lastName* field is the important feature, one could specify:

$$Type\ (student, ?rec\_type\ ::\ lastName : Char^*).$$

Alternatively, an approach similar to *used lemma* features can be employed. The record or function *definition* can be considered to be a lemma that replaces the record name with its contents.[11] Proof features about composite terms then would link the *definition lemma* and its important properties with the goal term.

After-state features

The *after-state* proof features can be used to record the important results of the proof step. This is not a separate category of features: after-state proof features are mostly the same types, just indicated on the goals and proof context that are the results of applying the proof step.

The proof features mentioned before are most concerned with what triggers a proof step (e.g. shape of the goal term) or what are the prerequisites for the execution of a proof step (e.g. existence of a suitable lemma). With after-state proof features, the proof process covers all facets of a proof step. The need for after-state proof features arises when advanced proof tactics are used. Powerful proof search tactics can apply lemmas multiple times or perform unsafe goal transformations. The after-state proof features allow marking the important terms within the *result* goal, thus indicating what outcomes are expected from the proof strategy.[12]

[11] This is actually the case in Isabelle, where definition (*∗_def*) lemmas are generated automatically for each definition.

[12] Specifying after-state proof features is not necessary for simple, deterministic proof tactics. For example, simple lemma application always transforms the goal in the same way.

Using after-state proof features in proof strategies requires running the strategies to produce the results on which the proof features are then evaluated. Because of this, matching strategies requires more effort: e.g. a strategy is applied in the background, the proof features are evaluated but may not match, thus the strategy itself would not match.

Proof context features

In industrial-style proofs (Section 2.1), most of the proof effort is spent on discharging proof obligations (POs) about the posited formal specification. The POs, such as *operation feasibility proofs*, *data reification proofs*, etc., are often automatically generated according to the chosen formal method. The generated POs are of a similar shape and their proofs follow similar high-level strategies for each type: e.g. feasibility proofs involve finding witnesses for operation after-states.

The *proof origin* features indicate where the conjecture originates, such as the type of the proof obligation: e.g. *Origin* (Feasibility PO). Other important proof context features can indicate the *domain* of proof (e.g. strategies in proofs about *railway* systems may find little reusability within the *aerospace* domain), mathematical taxonomy (e.g. the proof involves *set-theoretical* constructs), etc.

The *provenance* features can indicate how the sub-goal was reached in the proof. For example, the *Provenance* (**Expand definition**) feature records that **Expand definition** proof step (strategy) was used to produce the current sub-goal. This information highlights the order of strategies: e.g. the expert indicates that the new proof step is chosen because it needs to follow a previously applied proof step.

$$\mathit{ContextFtr} = \mathit{OriginFtr} \mid \mathit{ProvenanceFtr} \mid \mathit{DomainFtr} \mid \ldots$$

Most of the proof context features would be used directly in strategies. These proof features would be matched against the proof context, not the goal. The information, however, can be readily available: the conjecture *origin* can be marked automatically when the POs are generated, the *domain* can be set once for the whole formal development, while the *provenance* would be updated automatically as strategies are replayed or the expert advances the proof manually. Matching these proof features would then amount to just checking for existence of corresponding information on the conjecture.

Custom proof features

The goal of proof features is to capture all relevant details about the proof process. The paragraphs above outline several "standard" proof feature categories, which are often encountered during proof and are prime candidates for implementing within proof process capture and replay systems. Covering other cases, *custom* proof features aim at capturing "everything else" that does not fall into some predefined feature categories, or simply is not yet implemented within the system.

At the most general, every proof feature could be treated as a "custom" one and the following discussion would apply to them. A "vanilla" proof capture system would register all features without any mechanisation support, treating each one as an uninterpreted named predicate. This would be useful to build up the vocabulary of features and implement the most popular ones afterwards, or to do research on how proofs are described.

Having a system with a well-mechanised set of "standard" proof features still leaves space for the use of *custom* ones. The standard proof features may be inadequate in representing certain concepts in a way that the expert intends. For example, a *Preferred level of discourse* (*nat1_map*) feature is used to indicate that proof should be done at the "maps" level (see Section 11.2.1 for the actual example). Term, shape or structural features are problematic to represent this requirement, because the captured concept is quite far reaching. It means that the higher-level definitions, e.g. ones that contain the "maps" terms, need to be expanded to reveal these terms. However, the other definitions may already be at the "maps" or even lower level, so they should not be touched (or even wrapped into "maps"-level concepts). The proof feature captures the idea successfully, but it is hard to mechanise it accurately or use a combination of other proof features to replace it.

The expert doing the proof may also use non-conventional descriptions of proof details. For example, a proof feature may be domain-specific, adhering to some personal style, or simply "fishing" for adequate words to describe the concepts (e.g. marking list concatenation as "addable"). Taking it even further, the custom features may be used akin to *comments*, as arbitrary notes to capture the proof idea (e.g. *After relaxing break* () feature).

The custom proof features provide extensibility to the framework but limit the machine parsing and automatic reuse capabilities for such features. Nevertheless, the arbitrary definitions still have significant interactive use. For example, if a strategy is extracted from a proof and is reused in an interactive setting, the user

may consult the list of proof features, which include the custom ones. If the custom proof feature is defined in a human-readable form, the user may easily recognise the semantics and infer how to advance similar proofs. Furthermore, the semantics to custom proof features could be added afterwards, e.g. by programming their semantics into the proof capture system or by using machine learning with term details to learn the properties.

$$\begin{aligned} \mathit{CustomFtr} :: \mathit{name} &: \mathit{Text} \\ \mathit{template} &: \mathit{ParamTemplate} \end{aligned}$$

$$\mathit{ParamTemplate} = \textbf{token}$$

The custom features are modelled simply as uninterpreted text names. To ensure consistency of how many and what parameters need to be recorded for specific custom proof features, a parameter *template* can be defined. For the previously discussed types of proof features, such information could be modelled in a similar way, or could be prescribed and checked by the implementing system.

4.2.2 Using proof features

Proof features provide an abstraction over the detailed proof context when capturing the proof process. The mechanism allows identifying and marking parts of the proof context that are important to the proof steps, instead of recording or referencing all available prover data. The proof features and proof intent provide a static view of the proof process that can be inspected without interpreting and analysing the goal and proof context. The captured proof process information can be queried without re-running the proof to access the proof context. Furthermore, proof features eliminate the noise of available proof information by highlighting only the important parts. All this comprises an abstract "specification" of the captured proof process, which could be generalised to a strategy, adapted and reused for similar proof processes. Compare this with inspection of a finished proof script: understanding its underlying proof process will likely require interactively re-running the prover and making sense of the goal transformations and the available proof context. In the case of proving very large conjectures and goals, spotting "what has changed" in a proof step can be laborious, where proof features would quickly pinpoint the important terms.

Obtaining proof features

Proof intent and proof features augment the proof process captured from the prover by recording the high-level proof insight. Asking the expert doing proof to mark proof intent and features is the easiest approach to acquiring this data. However, certain aspects of this information can be captured and inferred automatically. For example, the important terms in the goal can be identified by examining the differences between input and output goals in a proof step. In addition to proof context analysis techniques, the previously captured proof process data could also be consulted. By identifying similarities between the captured and current proof processes, new proof features could be inferred by analogy from the proof features of the captured process. Chapter 6 discusses proof process inference techniques in detail. The distinction between the proof features marked manually by the expert and inferred automatically by the proposed system is represented in the model.

Feature :: ...
type : User | Inferred

The origin of how the feature has been marked is important for the treatment of and trust in features during their reuse. A user may be mistaken in the analysis and assign an incorrect feature (e.g. mark something as being *commutative*, when it is not). However, user-set features embody the actual expert insight and may indicate the most important features that drove the proof step. On the other hand, while automatically inferred features may be more robust, they may pollute the proof process with unnecessary data. Furthermore, treatment of *learned* features (e.g. when the user marks the feature initially and it is then inferred for a similar proof) is also interesting. If the learning (and matching) process can be trusted, these features could carry the significance of the user's initial selection.

"Language" of proof features

This thesis does not propose a formal "language" for proof features, instead opting for a somewhat *freestyle* notation for recording them. A partial reason is that proof process capture gives rise to data about how proofs can be described. Avoiding a prescribed language allows users to capture proof process detail in their own

way. Furthermore, a formal language would add to the overhead of describing the proof, which may deter users from supplying the data.

Nevertheless, the proof feature recording has some simple language features. Negation is expected, as some concepts are easier to describe using it. For example, marking that a certain function is eliminated by the taken proof step can use a *Not* (*Has symbol* ($\bigcap$)) proof feature on the after-state of the proof step.

A set of proof features for a proof step should be treated as conjoined: i.e. the expert marks it important that the goal *Has symbol* ($\bigcap$) **and** *Used lemma* (*Inter_eq*) is available. Other predicate operators, however, are not needed for proof capture: e.g. a disjunction of proof features would mark alternatives, but is not needed for a choice that has already been done.

Operators such as disjunction would be part of proof feature languages in *strategies* (Chapter 7), where different sets of proof features can describe different matching scenarios for when a strategy is applicable. A richer language for proof features in strategies would be used to accommodate results of strategy extraction. For example, the disjoint sets of proof features would be extracted from two separate instances (proof processes) of applying the same strategy. However, during proof process capture, a simple negation and conjunction are enough to describe the already-happened scenarios.

The set of proof features on a proof step should be considered as a whole. This means that the same terms or placeholders (*?var*) in different proof features but on the same proof step represent the same thing. Therefore a number of properties can be marked as proof features on the same term—this link will be preserved during generalisation and strategy extraction. For example, if the expert marks *Type* (x, $\mathbb{N}$), *Assumption shape* ($x \in ?S$) and *Assumption shape* ($?S \neq \{\}$), when the strategy is extracted, it requires there to be a natural number variable, with available assumptions about the same variable being in some set, which is not empty. However, the same term in before- and after-states is considered to be different: a *link* proof feature between the before- and after-states needs to be established in this case.

Proof features as hints

The amount and precision of proof features specified when capturing the proof process affects the quality of extracted strategies. However, marking the proof features manually is a significant overhead to the proof process. The initial goal of

the ProofProcess system is to provide a framework for capturing the proof process information at the level that the expert deems appropriate—the user marks the important parts of the proof.

Marking *everything* contributing to the strategy selection is laborious. However, it gives the most precise description of what the expert considers necessary for successful strategy application. Nevertheless, a smaller number of particularly important proof features may suffice in many cases. This is particularly true for manual strategy reuse: humans are good at spotting patterns and extrapolating, thus by glancing at just the key features, one could easily infer the related proof features necessary for successful strategy reuse.

Furthermore, it can be helpful to consider what is enough to distinguish a strategy among the others. For example, having just two or three proof features marked can already be precise enough to narrow down to a single applicable strategy during strategy reuse. Furthermore, instead of being exhaustive in laboriously marking all corner cases of a particular strategy application, it may be easier just to give hints about when it is generally applicable. When it comes to replay, the lack of corner cases may make the strategy match where it should not, but a quick glance from the user or a replay "in the background" would show that the strategy does not work here and it would not be selected for replay.

Treating proof features as *hints* about the proof process reduces the overhead spent on marking them during proof process capture, but may still produce successful strategies. However, experiments are needed to verify the best way of describing a proof step with minimal overhead.

4.2.3 Collecting proof step abstractions

The *ProofInfo* structure encapsulates the high-level insight about each proof step using the proposed proof *intent* and proof *feature* abstractions. This information can be captured at any level of abstraction (see Section 4.3).

$$
\begin{aligned}
\mathit{ProofInfo} :: \ & \mathit{why} : [\mathit{IntentId}] \\
& \mathit{inFeatures} : \mathit{Feature}\text{-}\mathbf{set} \\
& \mathit{outFeatures} : \mathit{Feature}\text{-}\mathbf{set} \\
& \mathit{narrative} : \mathit{Text} \\
& \ldots
\end{aligned}
$$

Recording proof intent is optional: this allows for cases when the strategy undertaken by the expert is not worth recording or cannot be named or described. It can be still be argued, however, that *no strategy* is also a strategy. For example, a **Prove blindly** proof intent would mark parts of proof as less-than-optimal candidates for learning and reuse.

Proof features for each step are split according to their precedence: *inFeatures* capture preconditions about the proof step, e.g. what are the important parts of the goal or available lemmas, etc.; *outFeatures* cater for the occasional need to record what is expected of the proof step, i.e. the *after-state* proof features.[13]

Furthermore, it is anticipated that not all insight or additional information about why the proof step is taken can be captured by the proof intent/features mechanism. Further comments can be recorded within the proof process as additional *narrative*. While not useful for automatic reuse, the *narrative* can carry convenient description of the proof process: e.g. it could explain how a particular intent is to be interpreted within the proof step. It also acts as a fallback for the proof process capture approach, enabling the recording of additional information that is yet unsupported by the system or the approach in general.

A significant proportion of captured high-level information about the proof process comes directly from the expert. The thesis argues that the quality of extracted strategies depends on the quality of the captured data and indirectly on the level of expertise that the user possesses. To differentiate among experts (or even among the choices of the same expert), proof information *score* can be used. It aims to introduce some proof step measure, enabling the rankings of different attempts or proof steps. The *score* could be used to record the expert's confidence in selecting the intent and proof features, differentiate between the levels of expertise among the users, mark "negative evidence" on strategies leading to dead-ends in proofs (though the strategies could still be successful on their own), etc.

$$
\begin{aligned}
&\mathit{ProofInfo} :: \ldots \\
&\qquad\qquad \mathit{score} : \mathit{Score} \\
&\\
&\mathit{Score} = \textbf{token}
\end{aligned}
$$

[13] The *in/outFeatures* naming of proof feature sets corresponds to the *in/outGoals* for prover steps (Section 4.3.3).

Proof *intent* provides abstraction over low-level proof commands by providing high-level insight description to a proof step. Proof *features* capture important proof process details in an abstract way, aiming to pinpoint relevant particulars of the proof goal and related proof objects, while hiding irrelevant noise within proof results. During strategy extraction, proof features can be used to describe when a strategy should be suggested and applied for replay by matching on the current goal and proof context (see Chapter 7). The next section proposes how these concepts can be used to describe a particular proof process as collections of proof steps at the desired level of abstraction.

4.3 Proof structure

A proof of some conjecture can be seen as a collection of proof steps transforming the open goals of the conjecture until proven. The proposed proof *intent* and *features* can be used to provide abstract descriptions of each proof step. The way of structuring proof into such proof steps, however, can vary: the expert's insight on achieving the proof can be expressed at different levels of abstraction. This thesis proposes a *tree-like structure* to construct an accurate and flexible description of the whole proof. The *ProofTree* structure enables proof step decomposition (via *ProofSeq*), yielding different levels of abstraction, as well as recording of branches in the proof (via *ProofParallel*) when different sub-goals can be tackled independently.

$$ProofTree = ProofEntry \mid ProofSeq \mid ProofParallel \mid ProofId$$

The *ProofEntry* records represent actual proof steps as *leaves* of the tree structure. Finally, *ProofId* elements are used for "plumbing" in special cases of complex proof structures (see Section 4.3.7).

This structure is used to *record* a single proof and thus provides a simple way of expressing how a proof is advanced. If several ways of tackling the same proof are available (attempted), they are recorded as different *Attempts* (Section 4.4), each comprising a single proof recording. Furthermore, the structure is only concerned with recording "how a single proof happens" and extended information such as possible alternative proof steps, capturing proof step repetition and others are left for strategies (Chapter 7). The strategies, however, will extract their information from recorded proof trees—the ones proposed in this section. For example, an

expert may record that to discharge a goal with three conjunctions, its proof requires three steps of "$\wedge$-elimination". Since that is how the proof is done, three subsequent (even though the same) steps of "$\wedge$-elimination" would be recorded. When a strategy is extracted from this proof record, however, it may recognise a *repeating* single step instead. Such a strategy would then be applicable for conjectures with a varying number of conjunctions, while the proof record stays true to what actually happens for the particular conjecture.

4.3.1 Proof step sequences

A proof can be viewed as consisting of several top-level proof steps. The exact way of identifying what are the main steps depends entirely on the expert. Furthermore, each of these "top" steps can be viewed as a collection of more detailed sub-steps. Such recursive decomposition allows the specification of multiple layers of abstraction. The decomposition is modelled as a *proof sequence* structure:

$$\begin{aligned} \mathit{ProofSeq} :: \mathit{info} &: \mathit{ProofInfo} \\ \mathit{steps} &: \mathit{ProofTree}^{+} \end{aligned}$$

The expert's insight at each level of decomposition is recorded using *info*. Other proof tree structures also accommodate such *ProofInfo* data, thus enabling capture of arbitrary abstractions over the proof. Furthermore, at least one child proof step must be indicated in the decomposition. This links the introduced high-level proof abstractions with the actual proof justifications (via the *leaf* *ProofEntry* element, see Section 4.3.3).

From the top-down perspective, *ProofSeq* provides "grouping" of lower-level proof steps. The constructed groups can be arbitrary: e.g. every low-level proof step (a *ProofEntry* element) can be wrapped into its own *ProofSeq* group. Such *ProofSeq* that contains a single proof step can be thought of as *decoration*, as it serves to add metadata rather than to group its steps. This approach enables recording multiple sets of proof intent with features for the same captured proof step. The flexibility enables specifying different facets of the high-level insight. For example, intents **Show disjointness of disposed region** and **Show subset of disjoint is disjoint** in Section 11.2.4 give a domain-specific description as well as a

set-theoretical meaning to the same proof step. For each different set of high-level information, the proof step can be wrapped into an additional *ProofSeq* element.

Other approaches include grouping a number of low-level proof steps into a single high-level *ProofSeq* step. This is particularly useful when several proof commands are needed to perform a single higher-level proof step (e.g. to work around theorem prover limitations). Finally, the grouping can continue until there is just a single *ProofSeq* element representing the whole proof. A single "top" element represents a high-level proof step of the overall strategy of doing the particular proof.

4.3.2 Parallel proof branches

When a proof step produces several sub-goals, these are often tackled independently. For example, proof via induction produces two distinct cases: the base case and the step case. Each case can be proved independently and the order of doing so is not important. Proof branches in the model are captured using a *parallel proof* structure:

$$
\begin{array}{lll}
\mathit{ProofParallel} :: & \mathit{info} & : \mathit{ProofInfo} \\
& \mathit{branches} & : \mathit{ProofTree}\text{-}\mathbf{set} \\
\mathbf{where} & & \\
\mathit{inv\text{-}ProofParallel}(\mathit{mk\text{-}ProofParallel}(\mathit{info}, \mathit{branches})) & \triangleq & \mathit{branches} \neq \{\,\}
\end{array}
$$

The parallel split allows the capture of the associated insight information via *info* and requires at least one branch to be indicated. Each branch is a fully-featured proof tree and can have further step decompositions and parallel proof splits. The order of the branches is not important: each branch can be proved independently.

4.3.3 Proof step justification

The *ProofEntry* represents the lowest abstraction of a proof step. It is no longer decomposable and captures details about goal transformation and the justification of the proof step.

$$
\begin{aligned}
\mathit{ProofEntry} &:: \mathit{info} &&: \mathit{ProofInfo} \\
&\quad\ \mathit{step} &&: \mathit{ProofStep} \\
\mathit{ProofStep} &:: \mathit{inGoals} &&: \mathit{Term}^{+} \\
&\quad\ \mathit{outGoals} &&: \mathit{Term}^{*} \\
&\quad\ \mathit{justification} &&: \mathit{Justification}
\end{aligned}
$$

Like other proof tree elements, *ProofEntry* accommodates *ProofInfo* abstractions. At the lowest level, this is a good place for automatically-inferred or captured proof process information for each low-level proof step (e.g. proof commands). For example, the prototype ProofProcess system implementation (Part III) currently tags each *ProofEntry* with a **Tactic application** intent and populates the *narrative* field with the textual representation of the used proof command.

The *ProofStep* structure records what the low-level proof step actually does (i.e. the goal transformations) as well as the justification for the proof step (discussed later). The *inGoals* capture open goals that are transformed by the proof step. Then *outGoals* record the results of the proof step: the remaining sub-goals. Input goals are considered to be discharged if there are no goals remaining: $\mathbf{len}\ \mathit{outGoals} = 0$. Goal representation using sequences accommodates for different theorem provers. Some provers, such as Isabelle, identify separate sub-goals, while others, such as Z/EVES, always operate on a single goal.[14] Using sequences can encode both representation styles, thus both representations are allowed within the framework and a proof is complete when no goals are remaining.[15] The goals are recorded in a prover-specific *Term* representation (see Section 4.6), though eventually there may be little need to know the actual goals: the important parts of the goal would be lifted to a higher level of abstraction using proof *features* (Section 4.2).

The aim to encapsulate each proof step as standalone requires recording both the target and result goals (named *in*/*out* goals in this thesis) for each proof step. Each proof step could contribute a new small strategy, thus allowing independent self-contained analysis of a proof step is beneficial for strategy extraction. Further-

[14] A special command (`cases`) is used in Z/EVES to split a single goal into multiple sub-goals, but the prover then focuses on just one of them. Another command (`next`) then can be used to switch to the proof of the next sub-goal. Otherwise all goal transformations produce a single output goal.

[15] Z/EVES outputs a "**true**" goal after the proof is done, but the Z/EVES ProofProcess integration does not capture this as a new remaining sub-goal.

more, recording in-goals is necessary for parallel proof tree branches, since it is important to know which goals were actually tackled by each parallel branch.

Each proof step must provide a *justification* to establish trust that the goal transformation is correct. Justifications are used to link the high-level abstractions with the theorem prover (or other reasoning systems) in order to *validate* the proof process. For higher-level proof steps, justifications of inner proof steps would be collected to ensure a correct chain of *in/outGoals* transformations. In general, justifications can be of different types or from different reasoning systems, even within the same proof process.

For example, a proof step can be TRUSTED with respect to some source outside the system: e.g. a mathematics book, a proof expert, etc. Furthermore, a prover-independent proof can be recorded by providing some inference steps (e.g. *NaturalDeduction*) as justification. However, the system is designed to be used in conjunction with proof assistants thus the majority of justifications ought to come from the underlying theorem provers. In these cases, *Justification* captures the actual proof commands (tactics) used by the expert to advance the proof within the theorem prover, including command configuration and context—the full *proof trace*. It should deterministically (and hopefully minimally) record everything necessary to "re-run" the proof step. For example, a Z/EVES *trace* would record the goals, applied tactic, used lemmas, case number and other information (see Section 10.2.2 for details). Proof commands in other theorem provers would capture a similar set of details. When outside tools such as SMT solvers are called, the trace may contain the goal (or the prepared SMT input) as well as SMT configuration that was used, and so on. Links with theorem provers are described in Section 4.6 and prototype implementations are discussed in Part III.

$$\mathit{Justification} = \text{TRUSTED} \mid \text{GAP} \mid \mathit{ProofTrace}$$

$$\mathit{ProofTrace} = \mathit{NaturalDeduction} \mid \mathit{IsabelleTrace} \mid \mathit{ZEvesTrace} \mid \ldots$$

The GAP justification allows indicating proof steps that are not discharged. By indicating "gaps" in the proof, one can partially construct the high-level view of the proof and justify it in a non-linear fashion. Each gap would record an unproven goal transformation. Proof steps with GAP justifications can also be used to record

the yet-unattempted proof branches, in order to ensure correct model invariants (discussed later).

The proposed proof tree structure can be used both in top-down and bottom-up perspectives. In a top-down approach, a high-level proof plan is defined first that can then be decomposed into more detailed steps—until the actual proof steps are used as justification. In a bottom-up approach, existing proof script commands are abstracted and grouped into higher-level proof steps until a high-level description is reached. The latter may actually be the preferred *modus operandi* when doing proof exploration: i.e. when the proof plan is not clear at the start of the proof. By performing the proof somewhat "blindly" and examining the results, the expert may spot the actual high-level proof insight and record these abstractions on the existing proof steps. In both cases the resulting proof tree structure would capture an abstract view of the proof that is justified by the actual proof steps.

4.3.4 Recording proof structure

An example featuring a very simple inductive proof recorded using the proposed proof tree structure elements is given in Figure 4.3.[16] Depending on the level of abstraction that this proof tree is examined at, the proof can be described as:

- Single step proof **Inductive proof**;
- Two-step proof: **Prepare induction** step followed by an **Induction** step;
- Seven-step proof that only records the low-level proof commands used (all *ProofEntry* elements);
- Any partial decomposition in between the single and seven-step proof above.

This shows that the expert has capabilities to describe the proof at all intended levels of abstraction.

The proposed proof tree structure is quite similar to hierarchical proof trees: *hiproofs* [DPT06] or the *Proof Data Structure (PDS)* used in the Ωmega proof assistant [ABD+06]. Hiproofs provide a graphical notation and semantics for a proof tree structure that supports proof tactic decomposition and branching. Isomorphism between hiproofs and the ProofProcess tree structure can be shown by matching tactics with intents, boxes with *ProofSeq* and branches with *ProofParallel* elements. The proposed ProofProcess structure, however, departs from proof tactics

[16] The case studies (Chapters 11–12) feature examples of more complex proof tree structures.

ProofSeq: Inductive proof

- *ProofSeq*: Prepare induction
  - *ProofEntry*: Simplify
  - *ProofEntry*: Extract inductive variable
- *ProofSeq*: Induction
  - *ProofEntry*: Apply induction rule
  - *ProofParallel*: Induction cases
    - *ProofSeq*: Base case
      - *ProofEntry*: Simplify
      - *ProofEntry*: Use hypothesis
    - *ProofSeq*: Step case
      - *ProofEntry*: Rewrite using lemma
      - *ProofEntry*: Use hypothesis

Figure 4.3: Sketch example of a *ProofTree* with recorded intents.

and allows more flexible abstraction with proof intent and features as well as supporting different justification options. The visualisation manner of hiproofs using boxes for abstraction would also serve to render the proof process structure equally well. The PDS structure also shares many similarities but is more complex and includes implementation features. The alternative proof steps in the ProofProcess model are captured using multiple *Attempts* (Section 4.4), whereas PDS encodes them directly into the proof graph.

4.3.5 Flattening the proof tree

Both the *ProofSeq* and *ProofParallel* are tree-node structures in the sense that they contain a number of children sub-trees. However, the semantics of each differ:

- *ProofSeq* provides decomposition into a sequence of proof steps. The "input" of a *ProofSeq* is also the "input" of its first sub-step, while the "output" of the last sub-step becomes the "output" of the whole *ProofSeq*.

- *ProofParallel* represents independent branches and the "input" of the parallel split is partitioned among the branches (each consumes some input sub-goals). The results of each branch are then again collected to represent the "output" of the overall parallel split. For the whole *ProofParallel* to be discharged, each of its branches must be discharged. If there are goals remaining in any of the branches, they become the open goals after the *ProofParallel*.

The "flattening" of the composite proof steps is used to examine goals, features or other details about the abstract step, since the actual goals are recorded in the *leaf ProofEntry* elements. The following VDM specification outlines the flattening of "input" and "output" goals as described above.

$$
\begin{aligned}
&inGoals : ProofTree \to Term^{+} \\
&inGoals(ptree) \triangleq \text{given by cases below}^{17} \\
&inGoals(mk\text{-}ProofEntry(info, mk\text{-}ProofStep(in, out, justif))) \triangleq in \\
&inGoals(mk\text{-}ProofSeq(info, steps)) \triangleq inGoals(\mathbf{hd}\, steps) \\
&inGoals(mk\text{-}ProofParallel(info, branches)) \triangleq \\
&\quad \mathbf{let}\ brGoals = [inGoals(b) \mid b \in branches]\ \mathbf{in}\ \mathbf{dconc}\, brGoals
\end{aligned}
$$

$$
\begin{aligned}
&outGoals : ProofTree \to Term^{*} \\
&outGoals(ptree) \triangleq \text{given by cases below} \\
&outGoals(mk\text{-}ProofEntry(info, mk\text{-}ProofStep(in, out, justif))) \triangleq out \\
&outGoals(mk\text{-}ProofSeq(info, steps)) \triangleq \\
&\quad \mathbf{let}\ last = steps(\mathbf{len}\, steps)\ \mathbf{in}\ outGoals(last) \\
&outGoals(mk\text{-}ProofParallel(info, branches)) \triangleq \\
&\quad \mathbf{let}\ brGoals = [outGoals(b) \mid b \in branches]\ \mathbf{in}\ \mathbf{dconc}\, brGoals
\end{aligned}
$$

[17] *ProofId* cases are given in Section 4.3.7.

Flattening proof features poses a more difficult problem. Proof features of an abstract proof step would be collected in a similar manner to *in/outGoals*, but at every level of abstraction. For example, a *ProofSeq* would add *inFeatures* from its first element to itself, and so on recursively. However, some of the proof features from "middle" steps in a *ProofSeq* list would also make sense at the abstract level: the *used lemma* features, or proof features on sub-terms that have not been changed by the previous proof steps. More research needs to be done to precisely identify the proof features that are eligible for inclusion in the higher-level proof step.

4.3.6 Overall status of a proof

A single proof step is considered *discharged* (justified) if there are no outstanding goals: $\textbf{len}\, outGoals = 0$. The same check can be used for higher-level proof step abstractions by flattening the proof step as described above. So for any proof step, it would be defined as following:

$$isDischarged : ProofTree \to \mathbb{B}$$
$$isDischarged(ptree) \triangleq \textbf{len}\, outGoals(ptree) = 0$$

This means that the overall status of a proof (whether it is discharged, or what goals are remaining in a partial proof) depends on the status of the last proof step. If the last proof step is a parallel split (*ProofParallel*), then the collection of unfinished goals at the ends of branches comprise the status of the overall proof.

The *ProofSeq* invariant ensures that all intermediate goals between proof steps are accounted for: each proof step consumes all output goals of the previous step.

$$ProofSeq :: \ldots$$
$$\textbf{where}$$
$$inv\text{-}ProofSeq(mk\text{-}ProofSeq(info, steps)) \triangleq$$
$$\forall i \in \textbf{inds}\, steps \cdot i > 0 \Rightarrow inGoals(steps(i)) =_m outGoals(steps(i\text{-}1))$$[18]

[18] Here $=_m$ is a multiset equality, as the order must be ignored.

Accounting for all proof goals in the representation introduces interesting scenarios when *ProofParallel* elements are involved. As proof branches can be unfinished (or even not started), special measures need to be taken into account for all goals, and to handle unfinished branches *after* a *ProofParallel* element. The next section explores the different scenarios and approaches to handling them.

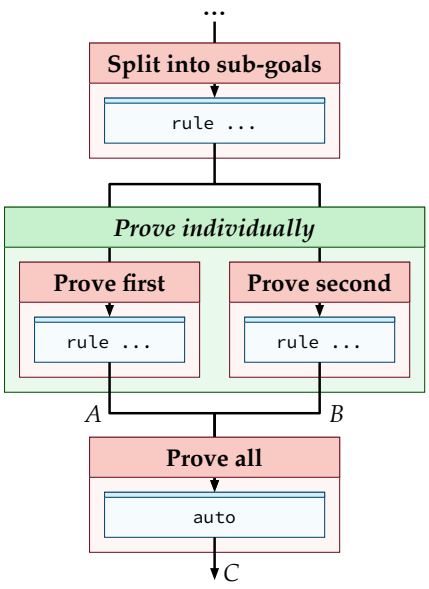
4.3.7 Unfinished proof branches

The proposed *ProofTree* structure mixes both *ProofSeq* and *ProofParallel* tree nodes within the same tree. These elements enable describing a branching structure with grouping capabilities within the same tree. Even more, they can be used for basic *merging* of proof branches; and (with extra "plumbing" extensions) can support quite complex proof structures in general. This section discusses several scenarios arising in handling unfinished proof branches in *ProofParallel* elements, and how they can be addressed using an additional *ProofId* element to "export" proof branches outside their *ProofParallel* node.

Merging branches

Using a tree representation for proofs captures the hierarchical nature of how the goals are transformed: each proof tactic takes a single goal as an input and either discharges it, or produces one or more sub-goals. Each of these is then transformed further by subsequent proof tactics. Many approaches (e.g. *hiproofs* [DPT06]) use a pure tree representation: i.e. each tactic takes only a single input goal; each sub-goal is transformed individually.

When it comes to proof assistants and advanced proof tactics, a pure tree structure cannot always be preserved as the proof is advanced. For example, consider the `auto` proof command in the Isabelle theorem prover (illustrated in Figure 4.4). This proof command can act on all open sub-goals at once. If there are two sub-goals *A* and *B* currently open, applying the `auto` proof command can transform them both and produce a single goal *C* as a result. This sub-goal is a result of transforming either *A* or *B* (the other one would have been discharged completely). Unless the proof process capture is able to trace how the proof state changes inside the prover, it is very hard to determine to which goal, *A* or *B*, to attach the goal *C*. Furthermore, the expert actually perceives that all goals are affected, even though they would have been transformed individually by the theorem prover. As the proposed ProofProcess system aims to capture the proof process "as the expert sees



(a) Graphical representation.

ProofSeq: -

- ...
- *ProofSeq*: Split into sub-goals
- *ProofParallel*: Prove individually
  - *ProofSeq*: Prove first
  - *ProofSeq*: Prove second
- *ProofSeq*: Prove all (*ProofEntry*: `auto`)
- ...

(b) ProofProcess tree structure fragment.

Figure 4.4: Merged proof branches using `auto` proof command in Isabelle.

it", the proof step corresponding to the `auto` application would have two *inGoals* and a single *outGoal*. If *A* and *B* sub-goals come from different proof branches, the single `auto` step has to be represented as a *merge* point for *A* and *B*.[19]

The `auto` merging example can be represented using the ProofProcess tree by adding a proof step after the *ProofParallel* element (illustrated by Figure 4.4). The branches with goals *A* and *B* would be left unfinished within the *ProofParallel*, and the `auto` proof step that follows "consumes" them both. The actual merging is performed by matching the *outGoals* of the parallel step (comprised of *outGoals* of its branches) and the *inGoals* of the following `auto` proof step.

Similar situations arise when capturing proofs from other systems. While in Isabelle a merge of proof branches happens because the expert *perceives* a proof command transforming multiple branches, in Z/EVES a merge can be done explicitly within the proof. For example, a proof goal can be split into branches using a `cases` proof command in Z/EVES [MS97]. Then each proof branch can be advanced individually, switching to the next one using a `next` proof command. The branch can be switched even if it is not finished. Using the `next` proof command at

[19] Actually, if *A* and *B* sub-goals have not yet been split within a *ProofParallel* but are a result of a previous single proof step, it would be represented as a normal *ProofSeq* list instead of a merge point. The ProofProcess architecture allows proof steps to transform multiple goals at once, so the new step would simply consume *everything* the previous proof step produced. To have a merge point, each proof branch *A* and *B* has to have been advanced individually.

the last branch will close the case split and will merge all unfinished branches back into a single goal. To support this, the proof process capture must accommodate *merge* points in the proof structure.

As illustrated in Figure 4.4, a basic merge point can be represented using just the standard *ProofSeq* and *ProofParallel* elements. However, for more complex situations (e.g. when one proof branch is not attempted at all), an extra structural element is needed. The following paragraphs introduce the *ProofId* element and discuss its use for different scenarios.

"Plumbing" using *ProofId* elements

The *ProofId* element is used for structural purposes only, as the last element of an unfinished *ProofParallel* branch. *ProofId* does not represent a proof step and thus does not transform proof goals: i.e. it is an equivalent of a mathematical *Id* function. The element is used to carry proof goals to the next step that actually transforms them, in a sense "exporting" a proof branch outside the *ProofParallel*. Thus it has only one field, *goals*.

$$\mathit{ProofId} :: \mathit{goals} : \mathit{Term}^{+}$$

When considered during proof "flattening" (Section 4.3.5), the list of *goals* represents both the in- and out-goals of a *ProofId* element:

$$\mathit{inGoals}(\mathit{mk\text{-}ProofId}(\mathit{goals})) \triangleq \mathit{goals}$$

$$\mathit{outGoals}(\mathit{mk\text{-}ProofId}(\mathit{goals})) \triangleq \mathit{goals}$$

As explained earlier, the linking of proof steps after a *ProofParallel* split (i.e. as part of a merge point) is done by matching the remaining goals of *ProofParallel* branches and the input goals of the following steps. In general, this can lead to quite complex scenarios where multiple goals are consumed by multiple proof steps (discussed later).

In order to simplify the handling of where each goal goes, the implementation of a prototype ProofProcess system also uses *ProofId* elements to record a *reference* to the next *ProofEntry* element that handles its goals (i.e. "continues" the branch).

It enables a more efficient and a more deterministic resolution of relationships between proof steps (see Section 8.6). As the simplified ProofProcess tree structure representation does not show proof goals, the *ProofEntry* reference of *ProofId* elements is displayed to illustrate how goals match (e.g. as in Figure 4.5).

Complex merging scenarios

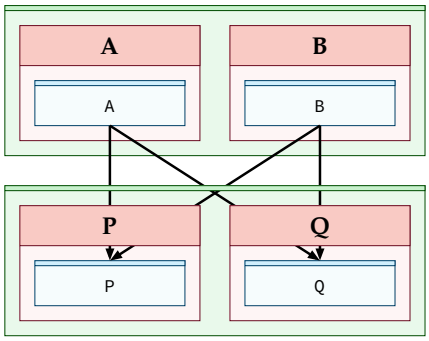
Even simple cases of branch merging are rare in interactive proofs, whereas complex cases are more of a thought exercise rather than actually "seen in the wild". However, the ProofProcess framework aims to capture any possible proof structure and must accommodate the complex cases. Figure 4.5 presents a scenario where each goal from unfinished proof branches is consumed in a criss-cross manner.

Consider a proof with two branches, each of which has two *out* sub-goals: $A(\rightarrow A_1, \rightarrow A_2)$ and $B(\rightarrow B_1, \rightarrow B_2)$. Now consider some arbitrary proof steps, which *in* goals come from different proof branches: $P(\leftarrow A_1, \leftarrow B_1)$ and $Q(\leftarrow A_2, \leftarrow B_2)$. Figure 4.5(a) illustrates the proof step links of matching goals. The links can be represented in a ProofProcess tree structure using sequential *ProofParallel* proof steps, with *ProofId* elements linking the correct proof steps (listed in Figure 4.5(b)).

Unclaimed proof branches

A *ProofParallel* element represents proof branches that transform individual sub-goals. However, in an unfinished proof, some of the proof branches may be yet-"unclaimed": i.e. other sub-goals have been transformed, but some are left untouched since the initial introduction.

For example, consider that the previous proof step had three sub-goals A, B and C, and the *ProofParallel* has branches accounting for only A and B as each one's *inGoals* (illustrated in Figure 4.6). Branch C is "unclaimed" as there are no proof steps transforming it yet. The issue is that the *ProofSeq* invariant introduced in Section 4.3.6 fails because the *inGoals*() of a *ProofParallel* (A and B) do not match the *outGoals*() of the previous step (A, B and C). To construct a correct ProofProcess tree, a new branch needs to be added to account for goal C. One way to instantiate such a branch is via a *ProofEntry* element with a GAP justification, indicating that there is a "gap" in the proof. Alternatively, the branch could have a *ProofId* element with goal C, representing that nothing has been done with the goal, but satisfying the invariant by providing "accountability" of all proof goals.



(a) Graphical representation.

ProofSeq: -

- *ProofParallel*: -
  - *ProofSeq*: **A**
    - *ProofEntry*: A
    - *ProofParallel*: -
      - *ProofId*: ⇝ *ProofEntry*: P
      - *ProofId*: ⇝ *ProofEntry*: Q
  - *ProofSeq*: **B**
    - *ProofEntry*: B
    - *ProofParallel*: -
      - *ProofId*: ⇝ *ProofEntry*: P
      - *ProofId*: ⇝ *ProofEntry*: Q
- *ProofParallel*: -
  - *ProofSeq*: **P** (*ProofEntry*: P)
  - *ProofSeq*: **Q** (*ProofEntry*: Q)

(b) ProofProcess tree structure fragment.

Figure 4.5: Complex inter-linking of proof steps.

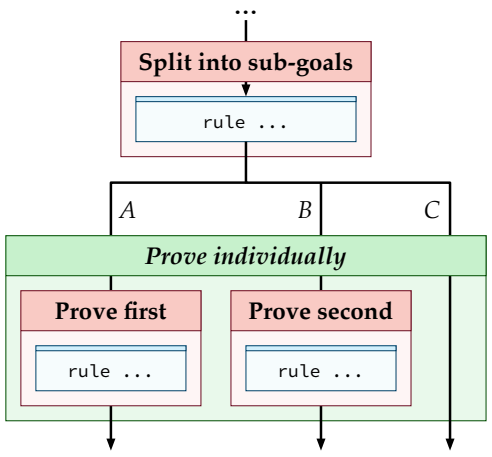


Figure 4.6: *ProofParallel* with an "unclaimed" proof branch.

Grouping partial branches

When discussing proof branches, it is easy to consider a significant split in a proof with large parts of the proof being tackled independently. For example, an induction proof can be partitioned into significant *base case* and *step case* parts. However, new sub-goals frequently are just side-conditions to the "main" proof step. Consider an application of a lemma. If the lemma has a number of assumptions, applying it introduces new sub-goals to prove that these assumptions hold. In many cases, these assumptions are trivial. However, it still takes an extra proof step to discharge them before continuing with the main proof. For illustration, see instances of **Discharge lemma assumptions** in Figure 3.2(b) or the more detailed discussion in Section 11.2.3.

Such side-condition proof branches are captured in the proof process using a *ProofParallel* element, because the assumptions are handled independently of the main goal. However, all branches are *nested* within a *ProofParallel* element equally, hiding the "main" proof branch among the side-condition branches. Such a structure prevents the highlighting of certain top-level steps, since the high-level steps on the "main" proof branch would be nested within the *ProofParallel*.

To circumvent this, the "main" proof branch can be "exported" outside the *ProofParallel* by keeping the branch unfinished but recording the subsequent proof steps *after* the *ProofParallel* element. This would bring the subsequent steps back to the top level of the proof tree, where they could be grouped accordingly to indicate the high-level direction in the "main" proof branch. The side-conditions will be hidden within the *ProofParallel* element, preserving the appearance of a mostly linear proof process. Section 11.2.3 of the heap case study provides a detailed example of such rearranging of the captured proof process tree structure.

Support for graph structures

As shown earlier, the *ProofTree* structure can be used to represent a *merge* of proof branches, no longer preserving a pure *tree* structure. With *ProofId* elements that can record *references* to where the proof branch "continues", the structure can actually support directed acyclic graphs (see Section 8.6).

The tree structure still remains as the main focus of the proof process capture: it is closer to the actual proof, is more suitable to represent high-level grouping of proof steps and is more useful in the majority of proof process styles. However, supporting graph structures facilitates the capture of more complex proof process

structures, even though they are harder to present to the user sensibly. For example, the prototype support for *declarative* Isabelle/Isar proof utilises this capability to link proof steps introducing new assumptions with proof steps that actually use them (Section 13.3.5). More details on converting between a ProofProcess tree structure and a graph structure are available in Section 8.6.

❦ ❦ ❦

The proposed *ProofTree* structure allows the recording of a single view of a particular successful or partial proof. Proof branches can be identified and proof steps can be captured at multiple levels of abstraction. The next section extends the structure to capture the whole proof process by distinguishing between multiple different attempts of the same proof. Note that the discussion on defining a *general* view of a family of proofs (e.g. with alternative or repetitive steps) belongs to *strategies* and their extraction (Chapter 7), not to the recording of proof processes.

4.4 Multiple attempts

Discovering a proof of a conjecture is rarely a straightforward first-attempt effort of writing down the perfect final proof script. An expert may attempt several ideas and approaches to tackling a specific proof, including backtracking after failed attempts to some previous step or even starting afresh. The stories of "fighting with the theorem prover" are common where the proof (together with the associated lemmas and definitions) gets "massaged" to fit the expected format of the theorem prover or the used formal method. Finally, after discovering a proof, it is refined, cleaned-up and otherwise polished to achieve the final proof script. All this information comprises the story of how the expert found a particular proof and contains valuable proof insight. To learn from this proof process and reuse it for similar proofs, the proposed system aims to capture the whole proof development.

An important argument for capturing all versions of a proof is the loss of the original proof insight during *proof clean-up*. Even when following a high-level proof plan, the actual proof often includes exploratory steps of how specific proof commands or lemmas suit the particular proof. The proof may go in smaller steps, feature less automation, resulting in a sub-optimal proof script. The eventual discovery of the proof provides the expert with a better view of the problem. With this knowledge, the proof script is often "cleaned" giving way for better automation and performance, general lemmas and more advanced proof script

commands. For example, a detailed proof with multiple proof commands may get replaced by an equivalent single super-command. In Isabelle proofs, a structured Isabelle/Isar proof may get replaced by an unstructured series of tactic applications or a suitable lemma, and vice versa.

A proof clean-up can indeed produce a more general proof (e.g. via extraction and use of a general lemma) and result in a more general strategy. However, often valuable proof process information is lost. The expert's insight may be captured more accurately within the original "slow" proof (e.g. the problem could be partitioned into significant proof steps with important features marked), rather than by the polished final super-command. The strategy extracted from the original proof may be more amenable for reuse: e.g. when most of the proof can be reused directly and only some sub-steps need to be adapted. A strategy extracted from a super-command, however, might produce an unfavourable proof state when reused: finding out where the strategy diverged would be difficult without taking the smaller steps.

The proposed system aims to accommodate all versions of the proof for a conjecture: the original discovery, the polished final version, alternative, different solutions, and others (some examples are listed in Figure 3.3). Different attempts could also represent different abstractions of the proof, particularly where the different abstractions cannot be related in a compositional manner: i.e. the cases not supported via the *ProofTree* structure (see Section 4.3).

Unfinished and discarded proof attempts can also carry valuable proof insight. The expert may see the proof going in a direction that would not be possible (or would be very difficult) to finish, and may choose to backtrack to some previous step, then take the proof in another direction. Such failed attempts may still contain generally applicable proof steps. Thus failed, abandoned or not-yet-finished attempts should also be captured as part of the proof process.

An *Attempt* is modelled as a rooted *ProofTree* containing a recording of a single actual proof. As explained above, the proof tree may be unfinished: i.e. there may be outstanding goals in the last proof step(s) of the proof tree.

$$\begin{aligned} \mathit{Attempt} :: \mathit{proof} &: \mathit{ProofTree} \\ \mathit{derivedFrom} &: \mathit{AttemptId} \end{aligned}$$

The relationship between attempts (i.e. when a new attempt diverges from a previous backtracked attempt) can be recorded in the *derivedFrom* field. This allows specifying some precedence between attempts. In general, however, proofs contained within each can be viewed as an independent attempt of the same proof. The relationship between attempts can then be established independently by comparing the proof trees for common proof steps.

An argument can be raised for capturing proof insight at the attempt level. These would include high-level features or intent of the whole attempt, e.g. marking the whole thing as a "blind" or "guided" proof attempt. This information can easily be represented on the root *ProofTree* node within an *Attempt*. For example, a root *ProofSeq* element would define high-level insight of the whole attempt as if it was a single-step proof. Then it would be decomposed into lower level proof steps of how the proof is actually done.

All attempts of the same proof are collected in a *Proof* structure, which represents the proof process record of a single proof task (conjecture being proved).

$$
\begin{array}{lll}
\mathit{Proof} :: & \mathit{goals} & : \mathit{Term}^{+} \\
& \mathit{label} & : [\mathit{Name}] \\
& \mathit{attempts} & : \mathit{AttemptId} \xrightarrow{m} \mathit{Attempt}
\end{array}
$$

where

$$
\mathit{inv\text{-}Proof}(\mathit{mk\text{-}Proof}(\mathit{goals}, \mathit{label}, \mathit{attempts})) \triangleq
$$

$$
\forall a \in \mathbf{rng}\, \mathit{attempts} \cdot \mathit{inGoals}(a.\mathit{proof}) =_m \mathit{goals}
$$

The conjecture, for which proof attempts are recorded in the *Proof* structure, is captured as a collection of *goals*. Each attempt provides a proof tree for these goals. Each proof record can be given a *label* value: e.g. to record the name of a corresponding conjecture (lemma, theorem, proof obligation, etc).

When collecting proof attempts, the *goals* rather than the *label* dictate which *Proof* structure is used to host an attempt. This is to ensure that all attempts are actually of the *same* proof. The *Term* type of proof goals is expected to reflect changes of goal terms, proof context or associated libraries (see Section 4.6 for details). Thus, for example, if some sub-term of the goal changes, it will make a different goal that is being proved. The proof attempt would then be captured in a separate *Proof* structure. Furthermore, while *label* could be employed to partition

the attempts according to some criteria, from the query and reuse perspective all these attempts would still be of the same proof and may serve better for proof search if collected together.

Note that the ProofProcess model and the proposed system are not concerned with the *provability* of conjectures (i.e. whether a conjecture is actually a theorem). Nor does it aim to ensure *correctness* and completeness of the captured proof attempts. The proposed system aims to capture the proof process, not to become a theorem prover. Thus incorrect definitions, unfinished or failed proof attempts are expected as part of the overall captured proof process since they do appear during interactive proof. When a strategy is extracted, the proposed AI4FM system would use the strategy to drive some underlying theorem prover automatically, thus all steps would be verified at the theorem prover level and incorrect proof strategies would not proceed. Furthermore, as described earlier, even failed attempts or strategies used for incorrect conjectures can yield generally applicable proof insight, which could be reused for similar proofs of correct conjectures. The *Score* information has been proposed to include considerations about the expert's trust in the chosen strategy. However, in general, the initial system is designed with the assumption of a *perfect* expert. If the captured proof process is of low quality, the extraction and reuse of its strategies would be negatively affected.

The rarely-straightforward process of proof discovery is captured in the proposed system as a series of proof attempts. Similar provisions for storing alternative versions of parts of proof are also available in the *PDS* data structure used by the Ωmega theorem prover [ABD+06]. The ProofProcess model, however, supports not just the alternative versions of a proof, but also captures unfinished or failed attempts as well as aims to record the whole proof development including backtracking and diverging of the proof with a better strategy. Refer to Section 6.3 for further discussion on when a new attempt is derived and how backtracking is recognised in the proposed system.

4.5 Collecting proof processes

The recorded proof attempts are the key information about interactive proof process for the purposes of proof strategy extraction. The proof insight and important features are recorded within each proof attempt, thus extracting strategy information would require querying (Section 13.3.4) over the set of recorded proof

processes, clustering them into families (if available)[20] and generalising over proof features and intents (see Chapter 7 for further details on strategy extraction).

The ProofProcess model does not enforce much structure on how the captured proof processes are to be arranged—the different implementations would choose how the proof processes are aggregated or what further relationships are important between the captured information. Thus at the top level all recorded proof processes are collected into a database-like *ProofStore* object:

$$\begin{aligned} \mathit{ProofStore} :: \mathit{proofs} &: \mathit{Proof}\text{-}\mathbf{set} \\ \mathit{intents} &: \mathit{IntentId} \xrightarrow{m} \mathit{Intent} \\ \mathit{features} &: \mathit{FeatureId} \xrightarrow{m} \mathit{FeatureDef} \end{aligned}$$

The *ProofStore* represents an arbitrary collection of proofs with their attempts together with associated *vocabularies* of reusable proof intent and feature definitions.

The concept of *ProofStore* is abstract by design to allow for different partitioning of proof process collections by different implementations. For example, if *ProofStore* is taken to represent a database of recorded proof process data, the scope of the database can vary for different use cases. One could choose to use a separate proof store for each verification project,[21] which would allow having separate strategies for each project domain. Alternatively, there could be a single *ProofStore* per user and the strategies would represent the style of proof that person employs. A globally shared proof process store would allow for the recording and querying of proof processes from several users: they could employ strategies originated by their colleagues.

Refinements of *ProofStore* could also choose to introduce some partitioning and relationships between collections of captured proof processes. For example, the proof processes could be partitioned into bodies of knowledge, each representing a specification that gives rise to the conjectures being proved (e.g. each body representing a single *theory* in Isabelle). Additional relationships could be established to record how one theory *specialises* (extends) others, when related theories are *morphisms* of each another or possess a "fuzzy" similarity, etc. Such arrangements would give more structure to the captured proof process information, but would

[20] The proposed system is expected to extract some strategies from just a single proof, but more source proofs would help with generalisation.

[21] The prototype implementation of the ProofProcess system currently establishes a single *ProofStore* database per Eclipse project.

also introduce overhead to ensure consistency of the organisation, e.g. when the specifications are refined and conjectures are moved around. Furthermore, a partitioned proof process store would impact search capabilities: which proof process stores are to be queried, how the relationships between them are followed, etc. All this additional information needs to be recorded as metadata. In the current—*flat*—*ProofStore* model, information about partitioning could be recorded as proof features on relevant proof attempts: the *proof context* features would be suitable for this purpose (see *ContextFtr* in Section 4.2.1). The proof process search would then trim the search space by *matching* on appropriate origin proof features.

4.6 Linking with a theorem prover

The abstract model as well as the core of the proposed ProofProcess system are designed to be generic and applicable to different theorem provers. Thus the main concepts are designed to be prover-independent and provide a generic framework to capture, represent and define abstractions of an interactive proof process. The proof tree structure aims to represent proofs done in different proof systems and reasoning frameworks. The combination of proof intent and proof features focuses on capturing high-level proof insight by abstracting the prover-specific details.

While the framework aims to be generic, the actual proof processes would still be based on interaction with theorem provers.[22] Prover-specific data links the proof process abstractions with what actually happens in the prover. This information is very important to extract executable strategies or enable post-collection analysis of the proof process. The system needs to provide a generic solution to include prover-specific information. In the ProofProcess model, prover-specific details are accommodated via two points of abstraction: *Term* and *ProofTrace*. The *Term* concept is used to provide different representation capabilities for logic terms, lemmas and other prover objects, while the *ProofTrace* represents prover-specific details of a proof step: tactics, proof commands, etc.

4.6.1 Recording terms

Terms (e.g. predicates or expressions in proof goals, their sub-terms, etc.) in different theorem provers and proof systems are not represented in a uniform way

[22] The model actually allows any reasoning framework to be the basis and source of the captured proof processes: e.g. manual *natural deduction* proofs could be encoded just as easily by employing some established formal notation for terms and recording inference rules for proof steps.

and may have different meanings and assumptions associated with them. The differences span from definitions of the same concepts to treatment of function partiality (e.g. functions and maps are total in Isabelle but by default partial in VDM) and even to logic fundamentals (e.g. tri-value Boolean operators in VDM). Development of a uniform representation for terms that could cover different proof systems is therefore not feasible in general. Instead, the ProofProcess model and the proposed system opts for keeping the terms specific to their proof systems. This would enable different proof systems to reference the terms within the captured proof processes with richness appropriate for correct term representation.

The prover-specific terms approach does introduce an obstacle in that different proof systems would be unable to interpret terms recorded by other systems. This is a design decision: the captured proof process information and the extracted strategies from one system would be limited to reuse *within the same system*.[23] Allowing strategies to be reusable between different proof systems is not very feasible anyway given the possible fundamental differences in logic and term representation. Some success is expected, however, at the very abstract level of general proof insight. Proof intents and features could provide enough abstraction over prover-specific information to extract *skeletons* of high-level proof plans that would be reusable across different provers. Alternatively, there could be translators to convert terms captured from one prover to be understood by another.

In the ProofProcess model, actual term representations would be defined in prover-specific extensions of the model.[24] At the abstract level, the model does not require knowing the contents of term representations, which are modelled as an uninterpreted given type *Term*:

$$Term = \textbf{token}$$

The abstract notion of *Term* at this level is used to represent various different prover-specific concepts, such as the following:

- Goal terms and sub-terms: predicates, expressions, other complex terms or combinations of sub-terms, etc.

[23] The reuse could be limited even to the same *version* of a theorem proving system, especially when serious changes occur in the development of a theorem prover or its base proof libraries.

[24] See Sections 9.2 and 10.2.1 for prototype extensions supporting terms in Isabelle and Z/EVES theorem provers, respectively.

- Definitions of lemmas, operators, functions and other parts of the formal specification.
- Term *shapes* to represent terms with placeholders.
- Other prover items available in proof context, tactic configurations, etc.

This *Term* data is referenced as parameters of proof features, marking the important parts of the proof process (Section 4.2).

4.6.2 Coping with specification change

When choosing representations for prover-specific *Term*s, it is important to consider their consistency when the current proof context changes. The captured proof process data is "old" by its nature: each previous step is a snapshot of what has happened in the theorem prover. With every new proof attempt the old proof records diverge from the current specification. The definitions or concepts used by previous proofs may no longer be available in the current version of the specification. The divergence is even quicker with the associated proof context, which changes with every step.

Because of this change, each *Term* must record enough information about the proof context and related proof objects to allow for standalone inspection and interpretation. This standalone analysis would happen, for example, when the current proof attempt is compared for similarities with proof process data from older captured attempts. The matching algorithms would need to interpret and compare terms recorded at different points of formal development. Several points about possible term representation options are discussed in the following paragraphs.

There are different options for choosing a specific representation for the *Term* object, in particular the term itself. One way would be to use indices as in the *mural* system [JJLM91]. This would allow for simple and unique references of terms, however the whole snapshot of theory would need to be carried along to know what each index is mapped to.

For many formal notations and provers, a simplistic approach of recording a term would be using its plain-text representation: e.g. terms in Z specifications of the Z/EVES theorem prover could be represented as Z Unicode or LaTeX representations. However, a plain text representation of a term would mean different things at particular points of formal development. To illustrate the issue, consider Z schemas. A plain-text term, such as a goal predicate, would reference some

Z schema using just its name. The issue is that during formal development, the schema itself can easily change (e.g. adding new fields or invariants, changing properties, etc.) while keeping the same name. This means that all terms referencing the schema would also change their internal representations. The same issue would apply for functions, operators, lemmas or other definitional concepts, which definitions could change.[25] To ensure the correct references of terms within the proof process, the *Term* type has an assumption that when internal representation of a term changes, the change must be recorded and taken into account when comparing these *Term*s. The same applies for lemmas: two proof features referencing the same lemma are different if the lemma has changed between them. If a strict approach is taken, two instances of the same specification, if based on different versions of the base theory library, would have to be assumed to be different, unless some way existed to ensure that the meaning of the term had not changed in regards to using it in the proof process. Sets of terms from different proof languages/systems are assumed to be disjoint, since showing semantic equality would prove very difficult.

The possibility of change in the internal representations of recorded terms means that when terms are analysed, they are "matched" rather than checked for equality. The word "match" is used in this thesis to emphasise that all comparisons of terms and other proof process data are rarely absolute and involve "fuzzy" checks. The use of proof features can help with coping with changes in definitions: e.g. *structural features* (Section 4.2.1) can be used to mark the parts of schemas, lemmas and definitions that are important to the proof process. If changes occur in other parts of the definition, the terms would still be comparable. However, the parameters of such proof features would again be *Term*s and may have change-prone internal representations of their own.

The need to record *all* proof context and associated definitions can be relaxed at the term level by limiting the reuse of the recorded proof process and extracted strategies. For example, if the ProofProcess framework is to be used with a single version of a theorem prover and base libraries, it is not necessary to record internal definitions in these libraries as they will not change. The theorem prover would have the same versions of definitions when matching terms. Current versions of the prototype systems presented in this thesis actually employ this simplification by focusing on specific versions of Isabelle and Z/EVES theorem provers.

[25] Although base libraries of theorem provers are unlikely to change at least within the same version of a theorem prover.

4.6.3 Proof step trace

The *ProofTrace* data is used to capture prover-specific information about a proof step. It serves as *justification* of an abstract proof step (see Section 4.3.3) and is used to represent actual prover steps in the model. Recording prover-specific details about a proof step is important to enable *executable* strategies to be extracted. The abstractions presented in this chapter allow for the recording of high-level proof insight, which would comprise the extracted high-level strategies. However, during replay (strategy reuse), it is important to know how the high-level proof step is to be achieved within the prover. The AI4FM system is expected to drive the theorem prover automatically when replaying a strategy. The "prover-meaning" of abstract proof steps is recorded as a *trace* of low-level information about proof commands, tactics and configurations used.

Consider a proof step that sets up induction in a proof. It may be captured as **Apply induction rule** proof intent with important features that an inductive variable i_1 exists in the goal and a lemma L_1 is available with an appropriate induction principle. This could be extracted as a step within a strategy. When the strategy is reused for a similar proof, the system may recognise matching features: e.g. that a similar inductive variable i_2 exists and an appropriate induction lemma L_2 is available. However, just the abstract strategy steps do not contain enough information how to "apply" the strategy to the goal. One way would be to encode the logic within the systems, e.g. how to realise an induction step. However, this is not very feasible as all possible tactics would need to be encoded and the experts would have to stick to a prescribed set of intents. A more flexible way is to *capture* what was done for the abstract proof step originally: e.g. in an Isabelle/HOL proof such induction set-up could be done using the command `apply (induct i1 rule:L1)`. During strategy extraction this could be generalised and when the strategy is reused, the system would figure out that the command needs to be adapted to `apply (induct i2 rule:L2)`, which would advance the proof automatically.

The actual representation of what constitutes enough information to describe how a proof step is done within a theorem proving system—the *ProofTrace*—is not uniform across different provers. Concrete representations would be provided by prover-specific extensions of the model, e.g.:

$$ProofTrace = NaturalDeduction \mid IsabelleTrace \mid ZEvesTrace \mid \ldots$$

It is assumed that each proof trace captures enough information to be "replayable": e.g. the proof command, used tactic and its parameters, also configuration options about the prover and the proof context (e.g. simplifier depth in Isabelle).

ProofTrace may also be used to record details used by other extensions of the proposed system (in addition to strategy extraction). For example, the proof trace could record the position of its proof command within the proof script:

$$
\begin{aligned}
&\mathit{IsabelleTrace} :: \ldots \\
&\qquad\qquad \mathit{source} : [\mathit{TextLoc}] \\
&\mathit{TextLoc} :: \mathit{filePath} : \mathit{File} \\
&\qquad\qquad \mathit{offset} \quad : \mathbb{N} \\
&\qquad\qquad \mathit{length} \quad : \mathbb{N}
\end{aligned}
$$

Recording such information would establish a parallel between the captured proof process and the proof document that is actually being developed. Also, it is used by the proof history capture to reference old versions of specifications and proof documents. The link between the captured proof process and the proof scripts would be beneficial from the user interface perspective, e.g. indicating how the captured information corresponds to what has been input to the prover; as well as a fail-safe for recording the proof process: the associated proof scripts would be preserved in case further inspection is needed for the proof process. The next chapter elaborates this further by proposing a full extension of proof process capture to accommodate the recording of the history of proof development.

CHAPTER 5

Proof history

Proof *history* constitutes an important part of the full account of how proofs are developed. Capturing the order, timing and other details about proof activities alongside the high-level proof process provides new opportunities for proof process analysis and strategy extraction, as well as expanding the possible uses of the captured data into different applications.

Section 5.1 presents an approach to recording proof history as high-level proof activities as well as linking the captured proof steps with the proof scripts where they originate. This establishes a historical dimension to the static proof process view. Section 5.2 proposes how the captured proof timeline can be used to re-run the full proof development: i.e. to "animate" the expert. This enables testing of new analysis techniques or retroactive refining of the captured data. Other interesting opportunities using the proof history include proof *explanation* and proof *metrics*, which are discussed in Sections 13.4.2 and 13.4.3, respectively.

5.1 Recording proof history

Proof history is designed as an add-on to the core ProofProcess model (Chapter 4). The core model provides a static view of the captured proof process, with the aim to facilitate extraction of proof strategies. Neither the core model nor the strategy extraction require proof history: proof attempts aim to be self-sufficient candidates with enough high-level information for strategy extraction. Proof history is therefore a modular extension to the core proof process capture and provides additional opportunities for the automation of strategy extraction as well as for

other applications, such as proof metrics. The modular design is repeated in the prototype implementation (Section 8.3).

5.1.1 Abstract proof log

Proof history can be recorded at two levels: as a *log* of abstract proof activities as well as a history of proof script changes (Section 5.1.2). A high-level *ProofLog* is simply a sequential account of various important formal development *Activities*.

$$\mathit{ProofLog} :: \mathit{activities} : (\mathit{Activity} \times \mathit{Timestamp})^*$$

$$\mathit{Timestamp} = \mathbb{N}$$

Each activity is timestamped to provide a time-based perspective to inspect the captured proof process rather than just a precedence-based one. This could provide some insight into proof durations and similar metrics (see further discussion in Section 13.4.3). The notion of *Activity* within the log can be taken to represent any important action in the development of the formal specification and associated proof scripts. For example, new proof steps are recorded using *ProofActivity* entries, providing a link between the log and the static proof process trees.

$$\mathit{Activity} = \mathit{ProofActivity} \mid \mathit{DefActivity} \mid \ldots$$

$$\mathit{ProofActivity} :: \mathit{proofStep} : \mathit{ProofEntry}$$

Other activities to track within the *ProofLog* can include introduction of new definitions (datatypes, functions, lemmas) in the formal specification (*DefActivity*), changes to the specification, etc.

For example, a proof log tracking all proof and specification activities would reveal that lemma L_1 was introduced and proved while in the middle of some bigger proof P_0. A timeline of such a scenario could be as follows:

1. Proof P_0 is in progress (*ProofActivity* entries logged for each proof command);
2. Definition of lemma L_1 is added to the specification (new *DefActivity* logged);
3. Proof P_1 of lemma L_1 done (*ProofActivity* entries for P_1);

4. Proof P_0 continues, now using lemma L_1.

While the log reveals how the lemma came to be, it is not technically required for the strategy extraction. The proof process would record the successful attempt with a *used lemma* proof feature that lemma L_1 (and its important features) was required to advance the proof. The fact that it was added in the middle of the proof is not important: the similar proofs may need a similar lemma to be added or the lemma could be already available. However, while the time account is not mandatory for strategies, it will facilitate other uses. For example, a "movie" of the formal development process could visualise the lemma addition (Section 13.4.2).

ProofLog is a top-level concept similar to *ProofStore*. Using several proof logs would be subject to similar considerations as the use of *ProofStore*: i.e. there can be different logs per-project, per-user, etc. (cf. Section 4.5). Note that in the case of multiple users (e.g. *global* proof store and log), the sequential activity log may be insufficient to record simultaneous activities, calling for more advanced logging solutions. In general, the *ProofLog* would be paired with the corresponding *ProofStore*, as is implemented in the prototype system (see Section 8.4.3).

Proof activity log can be captured automatically, by recording proof steps (new definition commands, etc.) as they are sent to the prover. The proof history would capture all proof development, including backtracking, failed attempts, fixing definitions, etc. Some manual proof management facilities could be of use, e.g. to drop unnecessary proof activities or to "clean-up" the history. In general, however, logging would be done in the background without user interference.

5.1.2 Proof script history

The abstract proof log records only certain high-level information (i.e. *activities*) about how the development of a formal specification and proof progresses. An additional low-level link between the proof scripts[1] and the captured proof process can be established by recording the *proof script history* and positions of proof steps within. One of the benefits of recording detailed proof script changes is to allow the re-running of the captured proof automatically within the theorem prover. This would allow running additional analysis and enriching the captured data *after* the expert has finished working on the proof (discussed in Section 5.2).

[1]The notion of "proof script" used in this thesis denotes files that can contain the statements of the formal specification alongside the proof commands: i.e. they are not *just* for proofs. For example, Isabelle *theory* files mix definitions and proofs within the same file.

Details about the proof script can be recorded within the *ProofTrace* data (Section 4.6.3), which already captures prover-specific information about the proof command used. For example, it can record the position of the proof step within a proof script, if available. The actual representation would be prover-specific: e.g. the *source* field within *IsabelleTrace* can indicate the position of the proof command text within the Isabelle proof script file:

$$
\begin{aligned}
&\mathit{ProofTrace} = \mathit{NaturalDeduction} \mid \mathit{IsabelleTrace} \mid \mathit{ZEvesTrace} \mid \dots \\
&\mathit{IsabelleTrace} :: \dots \\
&\qquad\qquad \mathit{source} : [\mathit{TextLoc}] \\
&\mathit{TextLoc} :: \mathit{file} : \mathit{File} \\
&\qquad\qquad \mathit{offset} : \mathbb{N} \\
&\qquad\qquad \mathit{length} : \mathbb{N} \\
&\mathit{ZEvesTrace} :: \dots \\
&\qquad\qquad \mathit{source} : [\mathit{TextLoc}]
\end{aligned}
$$

The location of a proof command is recorded as the offset and length of the text comprising the command within the text file (as *TextLoc* record). This approach is applicable for all text file-based formal specifications and proof scripts: e.g. Z Unicode or LaTeX specifications used in Z/EVES, etc. Other theorem proving systems (e.g. Rodin toolset [ABH+10]) use different representations for their proof documents, thus integration with such TP systems would use different representation for recording proof step locations. The general requirements about referencing proof source (e.g. the need for version history, etc.) would be similar, however.

Recording the proof script source must be done considering proof history: the proof scripts change during proof development. The captured proof process data preserves old attempts, which would reference old versions of the proof script. Therefore the *TextLoc.file* cannot reference the "working copy" of the proof document. Instead, the proof command must reference the exact state of the proof script at the moment of execution. The expert may backtrack, alter, even discard the old proof commands. Capturing this process and preserving the history of proof script files is important to achieve the correct link between the proof process and the proof scripts.

The *File* type abstracts the *location* of the proof source, because the file contents can be referenced and accessed in different ways: e.g. the file versions may be stored in the file system, in a database, within a version control system, etc. Furthermore, the abstract model assumes that *File* references ensure the consistency between the contents of the file and the proof steps they are referenced from: i.e. that a correct file version is used; that loading the file version would reinstate the same version of the proof state, etc. Section 8.7 presents a generic File History framework to automatically record proof history and provide suitable *File* reference implementations. The framework provides an approach to capture versions of text-based proof script files in a prover-aware, performance-optimised manner. Alternative implementations of file history provision are also discussed.

In general, proof script history can be recorded automatically, provided a "logging" framework to capture different versions of the proof script files is available (e.g. the File History framework in Section 8.7). The proof command text positions are normally available from the proof assistant: such information is used to highlight the status of the command during proof (e.g. whether any errors occur during command execution). The recording of command positions would also benefit the ProofProcess system user interface: e.g. the captured proof process could be highlighted (overlaid or otherwise marked) in the proof script. The established visual relationship between what is written in the proof script and what is captured would benefit the user. Another important use—re-running the captured proof scripts—is discussed in the next section.

5.2 Re-animating captured proof

Re-animating the captured proofs is an interesting use of all the captured proof process information. The high-level proof process information (Chapter 4) provides a static view of distinct proof attempts, which can be leveraged for extracting reusable proof strategies. Capturing the relationships with the actual proof scripts as well as recording the time perspective will enable users to inspect and automatically recreate the full proof development process leading to these results.

The ability to automate the actions that the expert has performed previously would also be useful during the development of the proof process capture system itself. It would enable the testing of different hypotheses and techniques about automatically inferring the proof process. Some of the techniques are explored in

Chapter 6, however implementing and putting them to use can require significant additional development effort, upgrades in the theorem prover API, etc. With the full proof history being captured and the ability to re-run it later, users can start performing proof capture with minimal proof inference capabilities (e.g. while the system is still in development). When new functionality to infer the high-level proof process is subsequently added, the already-captured proof process could be re-run to upgrade the data using the additional system capabilities. Also, as a future-proofing measure, this would allow capture of additional data from the prover: e.g. if it becomes clear that some unanticipated aspects of the proof are also important and need to be captured. Finally, the approaches of inferring proof intent and proof features could be *evaluated* by re-running the manually-marked proof process and comparing how the automatically inferred data relates to what had been marked manually by the expert.

5.2.1 Re-running proof attempts

Proof process capture aims to record a sufficient static view that can be inspected without running the underlying theorem prover and recreating the original proof state. This includes significant low-level information such as target and result goals, associated proof command configuration, etc. Furthermore, the proof intent and features capture high-level proof insight that is more suited for human consumption. This information is enough for a standalone high-level inspection of the captured proof process.

However, the theorem prover can be required if further prover details are needed. For example, while the rendering of captured goals can be captured initially, examining or extracting sub-terms from within the goals may require prover functionality. Thus if the user wishes to inspect the goal in detail, "zoom into" certain definitions, partition the goal or perform other similar actions, the prover and an associated proof context are needed.[2] To enable such a detailed inspection, the correct proof context needs to loaded, which can be achieved by re-running the proof up to the point of interest. This will feed the required proof data (e.g. associated definitions, other formal specification elements) to the prover.

To re-run a proof attempt up to some recorded proof step (i.e. "load" a particular proof state), the proof script indicated as the *source* of the step's proof trace (e.g.

[2]The proof features are intended to capture some of the *important* proof context (e.g. particular goal details, definitions of important functions, etc.), but the full proof context recording is neither feasible nor actually needed for the standard use cases.

IsabelleTrace.source) is used. The *TextLoc* reference contains both the proof script file and the end location ($= \mathit{offset} + \mathit{length}$) of the text to submit to the prover. Submitting the referenced version of the proof script to the prover would produce the expected proof result.

When re-running a single attempt, several additional things need to be considered. The theorem proving system and base theory libraries used to re-run the proof should match the system and libraries of the original proof process recording. Furthermore, if the linked proof script is part of a larger development (e.g. it imports other formal specification files, such as parent *theories* in the Isabelle prover), correct versions of these related files need to be resolved. If the prover, base libraries or related files differ from the original recording, the proof results are likely to differ as well.

5.2.2 Re-running full proof development

Re-running the full proof development can be used to "animate" the expert. Rather than just getting the final version through the prover, this would recreate the actions of the expert from the beginning to the final version. Full re-run is suited for automatic analysis techniques as well as to view a full "movie" of how the proof has been developed. The latter can be used for teaching and training in interactive theorem proving and is discussed further in Section 13.4.2.

Automatic proof analysis techniques can range from recognising the proof structure or identifying important terms within the goal to using previously captured data to infer high-level insight: i.e. identify what the expert is doing and why. Chapter 6 proposes several approaches and techniques to infer the proof process information automatically. As such techniques are developed, they will become more powerful and more accurate in inferring the correct proof process. Proof history capture and re-run capabilities accommodate such incremental development: they allow running the proof *as it was* without asking the expert to redo the proof manually. For example, the captured proof structure could be refined retroactively by developing better proof process "parsers" and re-running the original proof. All proof insight manually marked by the expert (proof intents, proof features, etc.) needs to be carried over during such refinements of the captured data.

Furthermore, full re-run accommodates the development of new techniques that work on data currently not captured by the ProofProcess system. For example, say it becomes apparent that intermediate backtracking (e.g. when the expert

leaves one proof unfinished to discharge a helper lemma) is important to capturing the proof process. Re-running the full proof development with new analysis functionality would enable capture of this.

Full re-run of the captured proof process uses the sequential log of user interaction activities recorded in the *ProofLog* (Section 5.1.1). Each *ProofActivity* references a particular proof step: this provides an ordering of proof steps. Each proof step in turn has information about the corresponding proof script version and command location. With this data, re-run functionality knows how much of which proof script to submit to the prover. Backtracking is indicated by new versions of the proof script: the new content replaces the old one in the prover. The sequential log of activities would also ensure that all dependencies are consistent: i.e. the proof steps of the "parent" proof script would have been submitted earlier than those of the "child" and the order would have been reflected in the *ProofLog*.

❦ ❦ ❦

The proposed extensions to capturing proof *history* alongside the static high-level insight provide a number of opportunities to inspect and re-evaluate the expert's interactive proof. Particularly, it can be used to test approaches about inferring the proof process automatically. The next chapter proposes some ideas about inferring the proof process structure, identifying important parts of the goal and otherwise trying to figure out automatically what the expert is doing.

CHAPTER 6

Inferring proof processes

Obtaining a high-level description of how an expert does an interactive proof can be seen as a case of mind reading. Chapters 3–4 identify what constitutes a description of an expert's *proof process* and propose a model to capture and record the necessary information. While low-level proof information such as proof commands or goal changes can be queried from a theorem prover, capturing the high-level insight in the expert's mind is difficult. The system can ask the expert to indicate the high-level proof structure as well as provide proof intent and proof feature descriptions manually. Unfortunately, this introduces a significant overhead to capturing the proof process. *Inferring* some of the high-level information automatically would lessen the burden on the expert as well as improve the available high-level proof process data to facilitate strategy extraction.

Investigating how the proof process capture can be automated is one of the aims of this PhD research as described in the H_2 hypothesis (repeated here):

> *Certain information about the proof process can be inferred automatically, via analysis of proof context and previous proofs.*

This chapter proposes several techniques to infer the high-level proof process abstractions: from identifying the proof structure and new attempts to inferring proof intents and important proof features for certain known types or by comparing with previously captured data. The prototype ProofProcess system (Part III)

implements the proposed proof structure capture as well as identification of backtracking and new proof attempts. Building and evaluating the functionality for inferring proof intents, proof features and matching with previously captured proof process data, however, is left for future work.

6.1 Analysis of captured data

As per the proposed interaction model (Section 3.2), the ProofProcess capture system obtains data from the theorem prover and from the user. This includes the user's proof commands as well as the current goal and proof context, the corresponding results and other data. Constructing a high-level *proof process* from this low-level stream of data requires automatic analysis and manual intervention.

The goal of the AI4FM research is to increase the automation of interactive proof by learning strategies from an expert. The extra effort spent on manually specifying important proof process information can be justified, as it could be outweighed by the success in automating similar proofs by strategy reuse, or at least by suggesting hints on how to proceed with the proof based on previously captured information. However, more automation in *inferring* the proof process by analysing the incoming data would go a long way towards better usability of the system. By "filling in" the repetitive parts, it could incentivise the expert to ensure that an adequate high-level description of the proof process is captured.

There is no measure to define when a "perfect" proof process is captured. In fact, it can be argued that providing *hints* rather than a full abstract "specification" of the proof process is more efficient for strategy reuse within a family of proofs (Section 4.2.2). In general, the captured proof process data can be continuously enhanced. The "poorest" proof process consists of a stream of proof commands and proof results from the theorem prover: this information is readily available. Any and all higher-level structure or meta information can be added afterwards, enhancing the quality of the captured proof process as well as facilitating extraction of better, more precise proof strategies. When discussing automatic analysis and inferring capabilities, the process can be partitioned into three parts:

- **Incoming data:** the system "wire-taps" and records the submitted proof commands and the prover output results. Immediate analysis can be done to recognise the proof attempt: whether the user is backtracking, doing a new proof or continuing a proof attempt. Then the proof structure can be

inferred by investigating the results of the proof commands. The expert is presented with a structured representation of the captured proof data.

- **Together with the expert:** proof intent, features and other high-level information can be inferred for the newly-added proof steps. This data is *suggested* to the expert. Inferring proof features can produce significant noise, with certain types not important to the current proof, or low trust in how well a previous proof matches the current one.

  The expert approves the actually important inferred information among the suggestions. Furthermore, the expert may want to adjust the earlier-inferred proof structure to match the high-level insight. Finally, inferring some data may need input from the expert: e.g. the expert marks some proof features manually, then some of the associated ones are inferred automatically.

- **Post-capture:** the captured proof process data is recorded and stored for future access: either to extract proof strategies or for other uses. Additional analysis can be done on this body of data. For example, using machine-learning to infer certain data would require running it on a large dataset and may take a long time. Furthermore, new algorithms can be developed post-capture and re-run to enhance the captured proof process data.[1]

  Unfortunately, the expert is not present when inferring the information post-capture and cannot approve the new data. Therefore the quality of the data generated using post-capture analysis depends on the quality of the inferring functionality and trust in it.

The ability to consult the expert to verify the inferred proof process information allows for the use of broader algorithms and more "guesswork" when producing the suggestions. Other parts can be inferred quite deterministically and thus can be run automatically. The following sections describe some of the approaches.

6.2 Inferring proof structure

Inferring the proof structure from the incoming low-level proof data is mostly concerned with recognising the proof branches. Proof sub-goals can be discharged independently, using separate branches of proof commands. Identifying the proof

[1] Proof *history* (Chapter 5) captures enough information to re-run the full proof development, enabling such delayed development of new analysis functionality.

branches facilitates the clear description of the user's intent. Separation of independent parts of the proof also helps facilitate their individual reuse.

Proof scripts often consist of linear sequences of proof commands. Even when a proof structure is revealed internally, the "overview" representation is still a simple sequence of commands.[2] Theorem provers track which goals are changed by proof commands internally, but this information might not be presented to the user. For example, when constructing a proof in Isabelle, the user sees all open sub-goals at once. When a proof command is submitted, some of the goals may change, others remain the same. By default, the proof commands operate on the first sub-goal. However, some proof commands (e.g. `auto` or `simp_all`) can transform all goals at once; or the user can explicitly restrict commands to particular goals. Inferring the proof structure entails tracking which goals are transformed and untangling the proof commands into a tree structure.

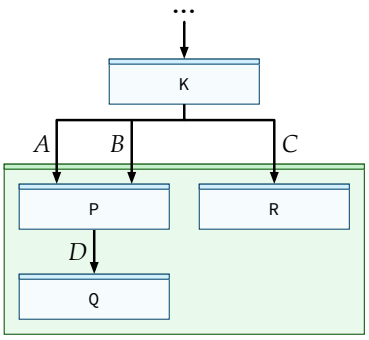
The proposed ProofProcess system aims to work as an add-on to theorem proving systems rather than an invasive replacement. The API provided by the provers is often limited in regards to how much internal structure can be extracted. Therefore the proof structure needs to be inferred from the prover output (i.e. what is displayed to the user). A simple approach is to track the goal changes and match the *input* goals of a proof step with the *output* goals of the previous one.

Consider a proof with three open sub-goals (e.g. using Isabelle): A, B, C. A user submits a proof command P to the prover. The result contains only two sub-goals: D and C. Goal C remains the same after the proof step, which indicates that it is not affected by the proof command.[3] Therefore the proof step only transforms goals A and B, resulting in a new sub-goal D. This information is recorded for each *ProofStep*: $P(\mathit{inGoals}\colon [A, B], \mathit{outGoals}\colon [D])$. Goal D is discharged by a subsequent proof command $Q(\mathit{inGoals}\colon [D], \mathit{outGoals}\colon [\,])$, which leaves C unchanged. It is finally discharged by a proof command $R(\mathit{inGoals}\colon [C], \mathit{outGoals}\colon [\,])$. Figure 6.1 illustrates this scenario.

By tracking the affected goals, the system can identify the scope of each proof command. Afterwards, the *input* and *output* goals of two subsequent proof steps can be compared. A branching point is identified when some of the *output* goals are not "consumed" by the subsequent proof step (e.g. proof step K in the example

[2] To improve proof overview in proof script files, users sometimes perform manual indentation of proof commands to illustrate "nested" proof branches.

[3] Collisions (i.e. that exactly the same goal was the result of transforming some other goal) should be rare and are therefore ignored.



(a) Graphical representation.

```
...
apply K      // result goals: A, B, C
apply P        // result goals: D, C
apply Q          // result goals: C
apply R        // result goals: none
done
```

(b) Proof script fragment.

Figure 6.1: Branching by tracking affected goals.

above, because P only consumes two sub-goals). The chain of proof commands is then followed ahead and proof commands are assigned to the proof branch (e.g. proof step P starts the branch, proof step Q continues the same branch as it consumes *all* output goals of P). When a proof step consuming an earlier open goal is encountered, it starts a new proof branch (e.g. proof step R), parallel to the earlier recorded ones. Tracking goal changes can be done recursively for further branches within branches: the analysis can infer complex proof structures. A *merge* point is identified when input goals of a proof step come from different proof branches: i.e. they match output goals of last proof steps in different branches.

The example in Figure 6.1 features a proof command that "consumes" two sub-goals and produces a single one. As discussed in Section 4.3.7, without access to internal goal tracking in the prover, it is difficult to infer which of the goals were discharged completely, and which produced the residual sub-goal. However, recording two sub-goals being transformed into a single sub-goal aligns to how the user perceives the proof via the prover output. In the majority of cases, however, only a single sub-goal is affected and the number of proof branches is the same as the number of output goals. The approach presented here works well with both the straightforward and the complex structures.

The proposed approach of inferring proof structure by tracking goal changes is simple and can be used for different theorem provers. It does not need changes to the theorem prover internals or proof command definitions. While not entirely accurate in terms of how goals are transformed internally (i.e. when multiple goals are affected), it captures the structure as the user perceives it.

Other systems aim to follow the prover very closely when capturing the proof

structure. For example, the Tactician and HipCam tools are used to capture *hiproofs* in HOL Light proofs [OAA13]. This requires following how the sub-goals are transformed by proof tactics, extracting the proof structure along the way. Their functionality is invasive to the prover: Tactician requires modifying proof tactics used in the proof, HipCam modifies the kernel of the HOL Light theorem prover. Note, however, that they can capture the nesting of proof tactics as well (i.e. if a tactic utilises other tactics within), warranting a deeper integration.

Inferring the branching structure is not needed in some other theorem provers as the information is readily available. For example, when a case split is done in Z/EVES proofs, the prover outputs a "case number" indicating which proof branch is active. Nested proof branches are identified by a sequence of numbers, e.g. "case 2.1.3". This information can be utilised to identify proof branches in the captured ProofProcess structure. This is helpful to the user as well, since Z/EVES proof scripts are the usual sequences of proof commands. Other systems provide the proof structure information via add-ons,[4] or a tree representation is used as the default user interface.[5] The existing functionality can be reused when capturing the proof process structure in such cases.

Inferring proof *branching* structure can be done by tracking how proof goals change, or by querying the prover directly. Inferring proof step *grouping* structure (i.e. how high-level proof steps abstract over sequences of lower-level ones) is a more difficult problem. The high-level proof steps are marked by the expert, thus to infer them, it is necessary to look at how previous proofs were structured. The approach would be similar to inferring proof intent or features by comparing with the previously captured data, which is discussed in Section 6.6.

6.3 Recognising proof attempts

Proofs attempts are used to capture the full proof development: failed proof directions, successful alternative proofs, clean-ups of already-discovered proofs, and so on (Section 4.4). Each proof attempt is a self-contained proof process structure, so when a user backtracks in the proof and searches for a new direction, it needs to be recognised and captured as a new attempt in the system.

Identifying a new attempt requires comparing the captured proof process data of the current proof with previously captured proof attempts. Some considerations

[4] For example, the proof tree visualisation in Coq [Tew11], the PVS prover [ORS92], etc.
[5] For example, in the Ωmega theorem prover [SHB+99], the Rodin tool [ABH+10], etc.

in recognising whether an attempt is new are listed below:

- Proof re-runs do not create new attempts. Large developments span multiple days: a user would re-run the existing proofs upon resuming work. To avoid populating the captured proof process database with copies, re-runs do not spawn new attempts: existing attempts are used instead.

- Proof process data is used for attempt comparison. Re-run matching cannot rely on low-level proof script information such as command locations, order of steps, etc. A user may split the proof script into different files, re-order theorems and proofs in the proof script, add comments or delete text, etc. Such changes do not alter the proof or the proof process, thus the same attempt is recognised. The captured proof process provides an abstraction of the low-level proof and is used for attempt comparison.

- Order of branches is ignored. The *ProofParallel* structure captures branches as a set: from the abstract point of view, it is not important which goal was attempted first. Reordering of individual branches (i.e. lists of proof steps comprising the branches) does not spawn a new attempt.[6]

- An attempt is *extended* when the new proof wholly matches it but has additional steps. Instead of producing a new attempt, the existing one is extended with new proof commands. All new attempts are built up in this way: when a new proof step is submitted, the whole of the current proof is matched to an existing attempt which is extended with the new proof step.

- If the current proof is "shorter" than a previously captured attempt, it is not a new attempt yet. In many cases, this is an old proof being re-run. Sometimes the user may inadvertently start re-doing the proof in the same way as it was done before. By matching with a previous attempt, the system can inform the user about treading an old path.

- A new attempt is recognised if it diverges from all existing attempts on the same proof. It may match an existing one in the majority of the proof, but, for example, one branch employs a different proof step. This indicates the user has backtracked the proof in this branch and has taken a new direction,

[6] In the prototype ProofProcess system, commands that "do nothing" are ignored during proof capture. For example, the `prefer` and `defer` proof commands in Isabelle are used to reorder the sub-goal list. Since the branch order is not important in proof process capture, they are ignored.

constituting a different attempt. The proof attempts can be compared to identify an actual point where the proof diverges, but this information is not recorded explicitly in the ProofProcess model.

- When a new proof attempt is created, the appropriate high-level insight is copied from the previous one. The expert may have indicated the high-level proof steps and marked the proof features for the majority of the proof: this information is duplicated in the new attempt (where the attempts match).

- There can be cases when the user wishes to record an alternative set of high-level insight on the same proof: e.g. indicate different high-level steps, etc. This can be supported by allowing manual duplication of an attempt and adjustment of the information.[7]

- If the specification changes significantly (e.g. a function is redefined, etc.), even if the goal looks the same, a different thing is being proved. This results not just in a new attempt, but in a whole new proof (see Section 4.6.2).

The prototype ProofProcess system recognises new proof attempts and provides basic proof re-run matching. Graph-subgraph isomorphism is used to compare proof attempts and identify new ones as well as re-runs or extensions of previous attempts. Support for transferring high-level proof process information between the attempts is limited, however. Section 8.6.2 provides more details.

6.4 Inferring proof intent

Proof intent is a user's description of a high-level proof step, abstracting over low-level proof commands to convey the idea of what the proof step actually is intended to do (Section 4.1). Inferring this information requires looking at examples from previous proofs. The only proof intent candidates that could be inferred "from scratch" are high-level descriptions of proof commands or related mathematical concepts: e.g. whenever an `induct` tactic is used in Isabelle, the proof step could be tagged with **Induction** intent. Following it, the subsequent proof branches could be identified as **Induction base case** and **Induction step case**, etc.

The general problem is similar to inferring proof features by "learning" from the previously captured proof processes (discussed in Section 6.6). Nevertheless,

[7] Manual deletion of attempts is also useful, as the user may wish to discard attempts of little value (e.g. when unsure about which proof command needs to be used and trying all of them).

it may be easier to infer proof intents than features. Intent tags are parameterless, therefore they do not need to be matched to corresponding parts of the goal or the proof context. Furthermore, they are used to describe what the proof step *does*, suggesting a strong correlation between the proof command and the intent.

For example, if the proof command `apply (elim conjE)` (conjunction elimination) was tagged as **Cleanup** proof intent in a previous proof (see Section 11.2.1 for example), when the same command is used again, the same proof intent could also be suggested. The proof intent suggestions could be linked to tactics (e.g. the `induct` tactic discussed above) or to lemmas explicitly listed by the proof commands: e.g. the application of lemma *disjoint_union* is always used to **Split disjointness** in the heap case study (Section 11.2.2).

6.5 Inferring proof features

Proof features mark the important parts of the proof, highlighting the triggers, prerequisites and expected results of each proof step (Section 4.2). They provide flexible instruments to mark everything important to the proof. However, this flexibility means that inferring them automatically is difficult in general.

Furthermore, inferring proof features exhaustively can generate a lot of noise: only a small subset of all possible proof features are important to each step. For example, the *Top symbol* () proof features are easy to infer for each proof step. The number of proof steps triggered by the existence of a certain top symbol would be small, however. Selecting only the actually significant proof features is as important as precisely defining the proof features themselves.

The best selection of proof features can be inferred by looking at the examples of previously captured proof processes. This approach is discussed further in Section 6.6. This section discusses particular proof feature types that could be inferred automatically without previous data. These inferred proof features would be *suggested* to the user for confirmation to avoid polluting the captured data with unimportant ones. Some ideas on how to narrow down the set of suggested proof features by tracking the changes in goals are also explored.

6.5.1 Known goal feature types

Certain proof feature types (e.g. the *existential features* in Section 4.2.1), are simple to define and encode within the system. Their evaluation is deterministic on any

proof goal. For example, the *Top symbol* () feature would always match to the top symbol (operator, function definition, etc.) of the goal. The *Has symbol* () feature would yield a list of all symbols appearing in the goal. Such proof features can be inferred by evaluating all known (implemented) feature types on the current goal.

A massive list of suggested proof features, however, would not be helpful. For example, the system suggests every symbol in the goal as an important feature according to the *Has symbol* () feature type. This cannot be the case and the expert, instead of marking the important features manually, has to manually reject most of the suggestions. Such a scenario is counter-productive and the inferring of proof features brings no benefit. Instead, the list needs to be trimmed down with more accurate suggestions: e.g. only suggest the symbols within the changed parts of the goal (Section 6.5.3) or compare to what was done in previous proofs.

The approach to extract certain "known" proof features from the goal has been employed in other systems to learn strategies or match predefined strategies on the current goal (e.g. in [Hen06, GM13, KHG13, HKJM13]). The number of supported proof feature types, however, is normally very small or the proof features are very narrowly defined (e.g. type of the main argument, top symbol, etc). Nevertheless, the issue of noise from proof features is cited.

Shape proof features (Section 4.2.1) provide more flexibility but also introduce more complexity in inferring them. However, if a list of "known shapes" existed, inferring proof features with these shapes would be of similar complexity as with proof features of other known types. For example, the system could check if the *Has shape* ($?a \cap ?b$) feature matches the current goal and suggest it as important. The list of known shapes could be collected from previous proofs. However, the noise issues would be the same as earlier. Inferring new shapes is a very difficult problem and such proof features are better left for the expert to mark manually.

6.5.2 Proof context features

Proof context features capture that the expert's strategy is related to a particular type of conjecture (*origin* features), is domain-specific (*domain* features), depends on previous proof steps (*provenance* features), etc. (Section 4.2.1). Rather than *inferring* them from scratch, these proof features can be easily derived or replicated from a single selection. However, proof context features are subject to generating noise as well: they might be important only for a small number of proof steps.

Proof obligations (POs) are often generated automatically according to the used

formal method. The specific PO type can be recorded upon generation. When it comes to proof, this information can be used to record the *origin* proof features: the PO type remains the same for every proof step, e.g. *Origin* (Feasibility PO).

The *domain* of a proof step usually only needs to be indicated once. Then the whole proof and even the whole formal development would be done within the same domain and the proof feature would be repeated: e.g. *Domain* (Automotive).

The *provenance* proof features can be suggested by looking at the preceding proof steps. For example, if the expert has indicated the intent of the earlier proof step as **Expand definition**, the system could suggest the *Provenance* (**Expand definition**) proof feature for the current proof step.

6.5.3 Goal analysis for important terms

A good way to identify what comprises the important features of a particular proof step is to investigate what the proof step actually does in the proof. If there was a prover API to trace the internal execution of a proof command, this information could be collected. The prover knows which hypotheses or lemmas are used, what goal terms are transformed, which proof context objects are referenced, how proof tactics are executed, etc. However, this information is normally not exposed for external access. Furthermore, supporting such tracing would have a significant overhead, resulting in a prover slowdown and increased memory requirements.

Some information about "what a proof command does", however, can be approximated by analysing the proof goals, the proof command and the available proof context. For example, by comparing the input and output goals of a proof step, one can identify the differences (i.e. the parts of the goal that have changed). If a sub-term has been transformed by a proof step, it is a good candidate to bear important proof features. This approach can be particularly useful in large industrial-style proofs. The goals in these proofs can have a very large number of predicates (e.g. see Figure 12.5 for a "small" example), most of which are not used in a particular proof.

Comparing goals for differences, however, does not reveal proof step "dependencies". For example, a hypothesis can be instrumental to transforming the goal—yet it is unchanged between the input and output goals. Further steps are needed to identify such related parts of a goal. A similar problem is encountered in [Meh07], where hundreds of hypotheses in proof obligations are filtered to analyse which of them were used in the proof. The ones actually used are extracted

from the constructed proof tree. The knowledge is then used to check whether a proof reuse is possible (e.g. if only unused hypotheses have changed).

Isolation of the affected terms (i.e. either changed or used by the proof step) in the goal suggests locations of important proof features. This can be used to reduce the noise of automatically inferred proof features (Section 6.5.1). Furthermore, the information would be useful during the interactive proof as well: it would help the expert identify affected parts in a large goal. Basic comparison of goals to identify the changed parts is implemented in the prototype ProofProcess system. The user can "filter" the goals when marking the important proof features (Section 8.2.3).

6.5.4 Used lemmas

Used lemma proof features capture the fact that a lemma is needed for the proof step as well as its important properties (Section 4.2.1). Inferring such proof features entails tracing the use of lemmas in the prover and identifying the important ones.

Lemmas explicitly indicated in the proof commands are prime candidates to be marked as important: e.g. the lemma *disjoint_union* in an Isabelle proof command `apply (rule disjoint_union)`. The expert has selected the lemma particularly for the proof command.

Identifying which lemmas are important in automatic proof steps (e.g. `simp` tactic in Isabelle, `prove` tactic in Z/EVES, etc.) requires obtaining the used lemmas from the prover. Z/EVES theorem prover outputs all used lemmas with each proof step: a sample output is listed in Figure 12.7. In other systems, accessing this information may require additional functionality. For example, lemmas used by the simplifier in Isabelle could be parsed from the simplifier trace or extracted from the constructed *proof tree* representation.[8]

An automatic proof step can use a large number of lemmas during rewriting or simplification. Marking all of them as important is infeasible. Also, an expert's high-level insight rarely considers low-level lemmas used by the prover. Therefore, after tracing lemma use, it is important to filter them to a smaller number that would be suggested to the expert for confirmation. For example, user lemmas (i.e. belonging to the developed specification) would usually be preferred to system or library lemmas. Firstly, the user lemma may actually be written specifically for this proof (or this family of proofs). Secondly, the library lemmas would also be

[8]Obtaining the simplifier trace or the proof tree representation requires explicitly enabling these features in Isabelle and introduces an additional processing overhead.

available for similar proofs: missing one of them among the important ones may not actually hinder the reuse of the extracted strategy.

Tracing important used lemmas and suggesting them to the user can facilitate proof process capture. The information is helpful during the interactive proof as well: the user can inspect what the automatic proof tactic has done; whether a particular lemma has actually been used; etc.

6.6 Matching with previous proof processes

Inferring the proof process aims to improve the automation of *capturing* adequate details about the proof process. This chapter proposes approaches to infer certain proof process information via analysis of the incoming data or by querying the prover. Another approach to infer information about the current proof process is to compare it with the previously captured proof process data.

Ideally, as the expert is doing a "new" proof, the ProofProcess system could find that the chosen proof steps are similar to some previous proof. It might suggest *"You seem to be doing **Induction** similar to proof X"*. The expert would confirm whether the suggested proof intent is correct. Proof features could be inferred in a similar manner: the system would identify which proof features are marked for a previous proof, then automatically mark them for the current one.

The overall approach is almost identical to how proof strategies are generalised and replayed. Instead of learning proof strategies from the expert doing proof, the system learns how to infer high-level information from the expert doing proof. In both cases, the current goal would be matched against the previously captured and (possibly) generalised proof process data. Matching can be done against the whole proof, or against any sub-proof (sub-strategy) at any level of abstraction. If a match is found, the existing information is reused to either replay a strategy or to infer the intent, proof features, etc. Extracting and replaying strategies is discussed further in Chapter 7.

In the context of a full AI4FM system (i.e. with support for capturing, extracting and replaying strategies), proof strategies would pre-empt the expert doing a manual proof step. For every goal, the expert would be presented with a list of matching strategies (if available). Selecting a strategy would replay it, executing the underlying proof step. If successful, the new proof process would be marked according to the strategy description (Section 7.1). In this scenario, there is no

need to infer the proof process, as the strategy replay supersedes the functionality.

However, there are still opportunities for inferring proof process by analogy. For example, there may be no good suggestions of matching strategies; use of *custom* proof features will prevent automatic matching of strategies; or the expert may take a manual proof step without looking at strategies at all. Matching proof processes can be more accurate than matching strategies, because of the extra hindsight available: the expert has already done the proof step, thus proof features about the proof command and the resulting goal can be used for an accurate match.

The availability of the proof command is particularly useful. For example, when the same proof command is used, the same proof intent should be suggested and the same proof features should be attempted. If this fails, the expert may start marking the important features manually and as enough context is established, the ProofProcess system may recognise similarities with the previously captured data and suggest the remaining proof features.[9]

[9]For example, the Gmail email service from Google has a feature called "Don't forget Bob": if the user always messages Jane, William and Bob together but the current email only includes Jane and William as the addressees, the system suggests to also include Bob based on previous history.

CHAPTER 7

Proof strategies

Extracting and reusing general proof strategies is the ultimate goal of the AI4FM research project. A perfect AI4FM system would generalise the high-level ideas of a proof into a reusable strategy—and then apply it automatically where it matches the conjectures and goals. The overall process can be split into smaller parts: *capturing* the proof process, *extracting* the proof strategies and *replaying* them to discharge other proof goals. Each of these parts still represents a very complex research problem and requires significant effort on developing the theory and the involved concepts, implementing the approach including integration with theorem provers as well as running extensive experiments.

This thesis is concerned with the first part: *capturing* an expert's proof process, particularly the high-level ideas involved in finding the proof. The other chapters in this part propose an architecture to perform the capture, whereas the description of the prototype implementation and the case studies evaluating the approach follow in Parts III and IV, respectively. The topics of *extracting* and *replaying* the proof strategies are outside the scope of this thesis. Nevertheless, the overall thesis would be lacking if its description of proof process capture was to be presented without at least a brief overview on how the captured information is to be used.

This chapter aims to present the "strategies" side of the AI4FM research and how it links with the proof process capture. Section 7.1 outlines how the proposed AI4FM system interacts with the user and the theorem prover. Section 7.2 presents an abstract model specifying what a *strategy* is, its context and the overall environment involved in proof process reuse. Furthermore, *proof-strategy graphs*—a tool (with implementation) to encode proof strategies—is described in Section 7.3.

Thesis contributions

The research presented in this chapter has been done by other researchers within the AI4FM project. Aside from the explicitly mentioned shared work and contributions, this research is the work of other people and thus is not part of the PhD research. However, discussion of proof strategies here establishes how the captured proof process data is used to facilitate strategy extraction and replay.

The author of this thesis has contributed to the development of the AI4FM interaction approach (Section 7.1) as well as the abstract model of proof strategies and the wider AI4FM system (Section 7.2). An early (unpublished) abstract model of the "state" of proofs, proof context and proof strategy was developed by Jones, Freitas and Velykis (thesis author). Subsequent publications [FJV14, JFV13, FJVW13, FJVW14] evolved the initial model. The ProofProcess model of this thesis (Chapter 4) has evolved in another direction in order to accommodate proofs as they are encountered and perceived in proof assistants. Parallel to this, the author continued contributing to the evolution of the abstract model as well.

The development of *proof-strategy graphs* and the Tinker tool (Section 7.3) has been done by Grov, Kissinger and Lin [GKL13, GKL14]. The thesis author only has developed a basic integration between the ProofProcess system and these tools.

7.1 Replaying strategies: interaction

The AI4FM system has always been envisioned as an "assistant" to the theorem proving process. It is not intended to replace the theorem prover, but rather help the user with choosing a suitable high-level direction in the proof. Even in the cases where the AI4FM system would be able to find a strategy that completes the proof automatically, it would do so by "driving" the theorem prover *in the background* rather than fully constructing the proof.

The architecture of the proof process capture system outlines how it "listens" and tracks the interactions between an expert user and the theorem proving system (Section 3.2). The system is there to help the user, however it should not interfere with how the user does proof, neither should it enforce a new approach to theorem proving. The same approach is taken when using the captured and extracted information: strategy *replaying* is there as an additional functionality to assist the user but does not replace the familiar interaction with the theorem prover.

Figure 7.1 outlines how an AI4FM system interacts with the theorem proving

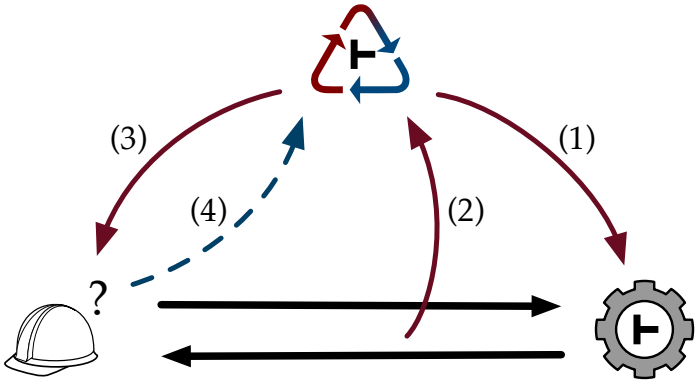


Figure 7.1: AI4FM proof strategy replay process.

process. The symbols, clockwise, depict the user (hardhat to emphasise support for an "engineer" level user), the proof replay system (AI4FM logo: *recycling deductions*) and the theorem proving system (cogwheel around the turnstile symbol). The basic interaction between the expert and the TP system is marked by the horizontal arrows. The AI4FM system "listens" to this interaction but can also "drive" the theorem prover and present relevant information to the user. The specific interactions comprise the following (numbered in Figure 7.1):

1. Adapt and replay proof strategies on the current goal by matching proof features of the strategy with the current goal and proof context.

2. Track prover output to match with available strategies; follow the application of proof strategies and adapt them or select alternatives in case of failure.

3. Present matching strategies and important features to the user. Automation may not always be possible and suggesting available strategies to the user may help find successful proofs manually.

4. The engineer might be able to assist if automatic attempts (just) fail; proof strategies may require certain proof features to be available: e.g. the strategy requires a lemma similar to one captured in the original proof process. The engineer may be able to provide one.

The following sections explore these interactions in more detail: how the strategies come in to existence, how the system interacts with the theorem prover and how the user interacts with the AI4FM system.

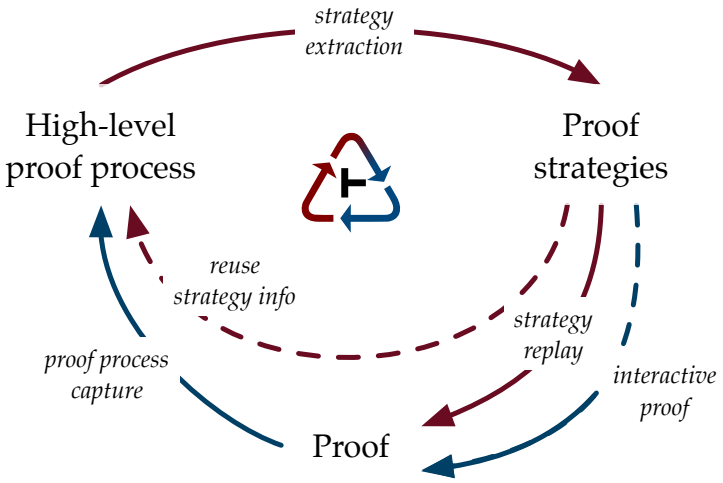


Figure 7.2: AI4FM proof capture, extraction and replay process.

7.1.1 Extracting strategies

Figure 7.1 does not depict strategy extraction, only their use. However, extraction is not portrayed in proof process capture either (see Figure 3.5). Learning from the captured proof process to produce reusable proof strategies could be seen as a third facet of the AI4FM approach: proof capture produces the data, whereas proof replay "consumes" strategies and is not concerned with extraction.

There can be different approaches to extracting proof strategies, from using machine learning to find patterns in the captured high-level data to generalising (or just straight using) the proof features of a single captured proof process (see Section 7.3.3 for a brief discussion on generalising a captured proof process). The immediacy of the extraction also depends on the approach: e.g. if the captured proof process data is extensive, it may be better to run machine learning later when the system is idle to avoid impact on theorem proving speed. Alternatively, if the strategy can be generalised quickly, such turnaround may allow reuse of the strategy immediately after it has been first employed.

The "modular" approach to capture, extraction and replay suits the proposed interaction model (see Figure 7.2 for illustration). Strategy replay advances the proof. The new parts of the proof can be captured by the proof process. The captured proof process is added to the existing pool of captured data—strategy extraction can use it to extract better strategies. The (possibly) new strategies are again available for replay.

In fact, during the proof capture of strategy application, the high-level proof

metadata can be reused from the strategy replay, but it is not technically a requirement (illustrated by a dashed arrow in Figure 7.2). The user could supply additional proof information to strategy application: indicating the specific differences, changing the proof intent, etc. Depending on these interactions, strategy extraction would create a new proof strategy, adapt or generalise the existing one (e.g. by adding alternative branches, new proof features, etc.), or leave the strategy as-is if it was replayed without changes.

7.1.2 Driving the theorem prover

Strategy replay does not aim to replace the theorem prover, instead the AI4FM system "drives" the theorem prover by sending proof commands, which represent the high-level proof ideas. Using strategies does not compromise the soundness of the proof, which still depends on the theorem prover verifying it.

The most non-intrusive way of using the strategies is generating a proof script for the user. During proof process capture, the low-level proof commands are recorded along with the high-level proof information. Thus when a strategy is extracted, it has information about which low-level proof commands represent the high-level proof steps. When the user plays a strategy, the corresponding proof commands could be inserted into the proof script and submitted to the prover.

Such an interaction model allows the user to easily switch between using the strategies and doing the familiar interactive proof process. For example, the user could query for strategies when stuck and replay one of them, which would insert one or more proof commands into the proof script. If this gets the user over the obstacle, he may choose to continue with interactive proof, or select a new strategy.

However, close interaction with the theorem prover is necessary for streamlined and automated work in the AI4FM system. The system needs to track the current goal, the proof context and other proof information to be able to suggest matching strategies. Furthermore, the strategy matching process may need theorem proving capabilities to evaluate proof features, perform unification, etc. Finally, the system may continue applying matching strategies automatically, in the background, to find a full proof of the current conjecture. In this case, the system may suggest to the user that a proof has been found and insert all proof commands into the proof script, completing the proof. A "batch mode" in an AI4FM system would find appropriate strategies and generate proof scripts for a set of problems at once.

By generating a proof script rather than using a new representation for strategy replay, the system positions itself as an assistant, rather than a primary theorem proving tool. The proofs would still be reproducible for other users without access to the proof process capture and replay systems.[1]

7.1.3 Role of the user

The user can use the strategies to help with the proof, but is also required to help strategy replay when it finds difficulty. The system would offer several ways to play the strategies: do a slow single-step strategy application, where a new strategy is searched for and manually selected by the user after each proof step; find the whole proof automatically and suggest the composed multi-step strategy to the user; run a "batch" mode on a set of conjectures and try to find full strategies for each one, inserting the corresponding proof scripts automatically. In case of failures, the system would construct a high-level proof of how much was achieved so that the user could quickly familiarise himself with how the proof was attempted. Also, the system could suggest the strategy it was attempting at the point of failure—the user may find a way to fix it or enter a new manual proof step, after which strategy application could be resumed. The heap case study in Chapter 11 provides some examples of manual intervention during strategy replay.

When selecting a strategy, several may be available that match the current goal. The AI4FM system would rank them for applicability, depending on the previously learned weights, the proof context, etc. However, the final decision would be up to the user in selecting the best strategy. The user's selection may influence the future weights of the strategy as the system would learn the best ones.

In certain cases of strategy replay, some strategy prerequisites may not be available. For example, if a strategy requires a lemma of certain shape, a corresponding one may be unavailable for similar proofs. In these cases, the system would notify the user that a lemma is needed for the strategy to continue and even suggest its important features. The user would formulate and prove the appropriate lemma and then continue with the strategy. Furthermore, there are tools to generate such lemmas automatically (see discussion on lemma discovery in Section 13.3.3) and the user might not actually be required.

[1] The current implementation of the Tinker tool (see Section 7.3) takes over theorem prover interaction when replaying proof strategies with its own graphical interface. However, underneath it still applies standard Isabelle proof tactics, thus generating a proof script from the result would not be difficult.

Furthermore, the architecture for proof process capture acknowledges that the high-level ideas about how the proof is advanced are not always automatable (e.g. *custom* features in Section 4.2.1). User input is needed to utilise the strategies in these cases. For example, the user manually tags the conjecture origin as "feasibility proof". This allows strategies designed for feasibility proofs (and thus requiring this proof feature in a conjecture) to match the current goal. Similarly, the user may mark the important proof features to narrow the strategy matching: e.g. highlight a certain operator and get strategies involving it to be suggested.

In case of a partial strategy match (e.g. when some proof features of a strategy are too restrictive), the user may still choose to replay it. However, if successful, the user would specify additional proof features that made the strategy successful for the current proof. These could be used to produce a better generalisation of the strategy during the extraction process. Alternatively, the user could derive a new strategy from applying the partially-matched one: its application would be given a new proof intent, new proof features, etc.

The user may disregard the automatic strategy replay facilities altogether. The suggested strategies could be used as a guide to the manual interactive proof process. The user would still benefit from the suggested high-level ideas on the current conjecture but may choose a new approach or just different proof commands to advance the proof. This interaction—learning from the high-level captured proof process—is illustrated in Figure 7.2 by a dashed half of the *interactive proof* arrow.

7.2 Abstract model of strategy replay

The proposed AI4FM system to learn and replay strategies is a complex one. In particular, the complexity arises from the lack of a clear notion of how people prove theorems, how the high-level proof ideas can be represented and what constitutes a strategy that can reuse these high-level ideas for similar proofs. Some narrower parts of the system have been identified to enable their design and prototype implementation: e.g. the proof capture system proposed and implemented in this thesis; or the mechanisation of proof strategies and replay (see *proof-strategy graphs* and the Tinker implementation in Section 7.3).

However, the design of the *overall* AI4FM system is developed by starting with a formal model of the system, which specifies its *state* (including identifying important parts of the proof, strategy, context, etc.) and the system functionality as

operations manipulating the state. The task of designing a system to deduce/re-use (high-level) proof strategies can be compared to that of designing a "programming language"—it is better to design a programming language from its state. This approach is similar to that taken when developing the *mural* theorem proving system [JJLM91]: after formalising the model of the system, its implementation was straightforward.

The full development of this abstract model is ongoing: refer to the publications [FJV14, JFV13, FJVW13, FJVW14] for further details about the model, including its evolution over the publication history. This section presents the main ideas about strategies from this formal model. The author of this thesis has contributed to the development of the overall model as part of this PhD research.

7.2.1 Anatomy of a strategy

At the core of an abstract model is the tandem of *Conjecture* and *Strategy*. A *Conjecture* represents a single goal and its associated context (including the high-level metadata). A *Strategy* in the model represents a "single-step"[2] abstract proof step that can match and transform a *Conjecture*. Typically, a strategy just decomposes a proof task to several (hopefully simpler) conjectures. At the high-level, *Conjecture* represents the proof side of the theorem proving process, whereas *Strategy* embodies the idea of when and how to advance it.

The strategy and its properties can thus be modelled as the following VDM record (see [FJVW14] for the overall model):

$$
\begin{array}{lll}
Strategy :: & intent & : [Why] \\
& by & : (ConjId \mid ToolIP) \\
& weightings & : MTerm \xrightarrow{m} \mathbb{N} \\
& mvars & : mvar^* \\
& specialises & : [StratId]
\end{array}
$$

The first component in a *Strategy* is the *intent*, modelled as a *Why* token. It has a dual purpose:

[2] "Single-step" strategy means that it represents a single high-level strategy step and does not contain other strategies (cf. "multi-step" strategies that consist of a tree/graph of other high-level strategies). "Single-step" strategies in the abstract model can actually involve multiple prover steps, advanced proof tools or smart use of conjectures to simulate multiple proof steps.

- The *intent* describes what the strategy does;
- It is added to the provenance of a conjecture, linking strategy definition with its instance (application).

Both the strategy intent (*Why* type) and the proof process step *Intent* (Section 4.1) are used for the same purpose: to capture a high-level description (a kind of *tag*) of an abstract proof step. During strategy extraction, the captured proof process intent would be used to populate the *intent* field of strategies, identifying individual high-level ideas.

Next, the *by* field models the *contents* of a proof strategy: i.e. *what* it does. The *ConjId* references other conjectures, representing the conjecture that is used to do natural deduction inference. This allows the modelling of natural deduction proof process and strategies within the abstract model. The *ToolIP* datatype represents various theorem proving tools available within modern theorem proving systems:

$$\begin{aligned} \mathit{ToolIP} \;::\; \mathit{name} \;&:\; \{\textsc{Sledgehammer}, \mathrm{SMT}, \textsc{Simplify}, \textsc{Tinker}, \cdots\} \\ \mathit{script} \;&:\; \textbf{token} \end{aligned}$$

A *ToolIP* entry in a strategy specifies that the strategy calls an external tool to advance the proof: its *name* and configuration (the *script* field) is recorded to enable the actual strategy *replay*. Playing a strategy means launching the proof method defined by the *by* field. The *script* field in *ToolIP* is yet unspecified, since different tools would require different configuration data.

The information required to extract the contents of the proof is captured in the ProofProcess architecture as the *ProofTrace* datatype (Section 4.6.3). It records the actual proof step in various theorem proving systems, including prover context and configuration details. During strategy extraction, this information would populate the content of the strategies.

The *weightings* and *mvar* fields of the abstract *Strategy* model are used to describe the matching parameters of a strategy. Datatype *MTerm* ("matching term") describes the features of the strategy, particularly when it should be used. The *mvars* ("matching variables") are bindings allowing to use matching information in strategy configuration. The *MTerm*s are used to match the strategy with a *Conjecture*, thus they are discussed in detail below, after establishing what constitutes required information about conjectures.

Finally, the *specialises* field in *Strategy* is used to specify relationships between strategies. By utilising this field, a "taxonomy" of strategies can be established: e.g. a **Peano $\mathbb{N}$ induction** specialises **Induction proof**.

7.2.2 (Meta-)information about conjectures

The concept of *Conjecture* in the abstract model represents each proof task (both top-level lemmas or theorems and sub-goals within their proofs). However, in addition to the goal predicate, the *Conjecture* captures various associated meta-information about the goal:

$$
\begin{array}{lll}
\mathit{Conjecture} :: & \mathit{what} & : \mathit{Judgement} \\
& \mathit{role} & : \{\textsc{Axiom}, \textsc{Trusted}, \textsc{Lemma}, \textsc{Subgoal}, \cdots\} \\
& \mathit{justifs} & : \mathit{JusId} \xrightarrow{m} \mathit{Justification} \\
& \mathit{provenance} & : (\mathit{Origin} \mid \mathit{Why})^* \\
& \mathit{emphTps} & : \mathit{TyId} \xrightarrow{m} \mathbb{N} \\
& \mathit{emphFns} & : \mathit{FnId} \xrightarrow{m} \mathbb{N} \\
& \mathit{other} & : \cdots
\end{array}
$$

In the *Conjecture* record, the proof goal itself is captured in the *what* field. Its abstract *Judgement* type allows for different goal representations in various theorem proving systems: e.g. the goal can be a sequent, equation, etc.

As the proof tasks can arise from different scenarios, the *role* field captures the role of each one: e.g. a top-level lemma would be tagged with Lemma role, whereas its proof sub-goals would have Subgoal roles. Furthermore, the model allows conjectures specified as axioms; or ones that do not need the proof (Trusted): e.g. if their correctness is well known or the proofs are available in textbooks.

There can be multiple attempts to prove the same conjecture: the *justifs* field captures the different *Justification*s. A *Justification* represents a single step in the proof, recording what proof method was used (*by* field) and the resulting new goals (*with* conjectures):

$$
\begin{array}{lll}
\mathit{Justification} :: & \mathit{by} & : (\mathit{ConjId} \mid \mathit{ToolOP}) \\
& \mathit{with} & : \mathit{ConjId}^*
\end{array}
$$

The details (and high-level meta-information) about each sub-goal is then captured within the corresponding *Conjecture*s, referenced using their *ConjID*s. By following these references, a full proof tree can be constructed. However, the justifications do not have to lead to a complete proof. For example, when a conjecture is first generated, there are no justifications for it. Also, the user may start one proof justification, leave it aside and try another, then come back and complete the first proof. Checking whether a proof is complete would require a transitive check following the justifications.

The proposed architecture for proof process capture (Chapter 3) implements and elaborates on the ideas in this abstract model of *Conjecture*. Some of the particular differences in modelling the proof process are historical, but both specifications can be seen as two very similar sides of the same model.

The *Judgement* data type corresponds to *in* goals in a *ProofStep* (Section 4.3.3). Multiple proof attempts on a top-level conjecture are modelled using the *Attempt* datatype; while the branching tree structure of the proof is captured via the *ProofParallel* element (Section 4.3). These parallels between the specifications confirm that the same concepts are modelled, however the focus is slightly different: the abstract model presented here aims to abstract away from theorem proving systems, proof scripts and how the proof is achieved in particular cases—instead modelling the proof process *in general*. The ProofProcess architecture described in this thesis aims to represent how the human expert *perceives* the proof: e.g. recording different attempts at the top-level, capturing proof steps not intermediate goals, etc. The information can be translated between the two models, thus using models appropriate for each situation is actually preferred (i.e. modelling a strategy replay system *vs* a proof process capture system).

The remaining fields of the *Conjecture* record try to explicitly identify the different points of important meta-information about the conjecture. The *provenance* field tracks how the conjecture came to be: i.e. what were the strategies applied that led to this particular goal. Provenance represents the *Origin* of the conjecture (e.g. it is a feasibility proof obligation) as well as the list of proof steps as *Why* intents, recording the path taken by the expert. This trace of intents requires the expert to identify each proof step with an appropriate strategy. The *emphTps* and *emphFns* ("emphasised types and functions", respectively) capture that the expert would explicitly mark certain parts of the goal with varying importance, hinting towards the strategy search to focus on these features. Other important proof feature types about the conjecture or the proof context (*known unknowns*) are also

anticipated, hence the *other* field in the *Conjecture* record.

The abstract model aims to *explicitly* identify the important features of a conjecture. The ProofProcess architecture enables marking *any* proof information as important by using the general concept of *proof features* (Section 4.2). The meta-information listed in the *Conjecture* record above, including the *role* field, could be recorded using proof features within the ProofProcess data. The open-ended way of modelling proof features as named predicates avoids changing the core model (and the corresponding implementation) to support new types of features.

7.2.3 Matching strategies

The rich meta-information about a *Conjecture* is modelled to enable strategy matching on these features. The matching properties of a strategy (i.e. *when* a strategy should be applied) are modelled using the *MTerm* ("matching term") constructs:

$$
\begin{aligned}
\mathit{Strategy} \;::\; & \cdots \\
& \mathit{weightings} \;:\; \mathit{MTerm} \xrightarrow{m} \mathbb{N} \\
& \mathit{mvars} \;:\; \mathit{mvar}^* \\
& \cdots
\end{aligned}
$$

The use of a *weightings* map recognises that a single strategy can have different levels of suitability for a given conjecture: the natural number describes the utility of the current strategy for progressing the proof, if that *MTerm* is satisfied.

A basic *MTerm* is a proposition built from negation ($\neg$), conjunction ($\wedge$) and an unbounded set of *atomic* (paramaterised) predicates. The *MTerm* is evaluated over the whole of *Conjecture* and the atomic predicates can refer to the properties of the goal (*Judgement*) as well as the high-level meta-information (e.g. provenance checks, existence of certain emphasised functions, etc.).

In the proposed ProofProcess architecture, the high-level information associated with each goal is captured as *proof features*. Furthermore, the strategy extraction process would generalise the proof features—they would constitute the *MTerms* of the strategy. These two lives of a proof feature enable strategy matching even for custom proof features: for example, a user highlights the *My-proof-feature(params)* custom proof feature during the proof capture. Strategy extraction would likely fail to generalise on an unknown proof feature, thus it could be copied verbatim

to a proof strategy: *My-proof-feature(params)* would now constitute part of the *MTerm* within a strategy. When doing a similar proof, the user highlights *My-proof-feature(params)* on a new conjecture, which triggers a match with an *MTerm* in the extracted strategy and leads to strategy replay, even though the proof feature is not interpreted by the system.

The *weightings* map for matching proof strategy enables modelling multiple (disjunctive) scenarios in which a strategy is applicable (including partial matching). For example, all permutations of the *MTerm* feature set could be included in the weightings map, with *MTerms* that have less features resulting in a weaker numerical weight. This allows a strategy to match the goal when some proof features do not exist or are not satisfied, however a low weight would indicate lower confidence in the success of replaying the current strategy.

The weightings are used to rank strategies by value for any given conjecture. Matching in a single strategy must therefore return the highest ranking *MTerm* (that matches the conjecture) in the *weightings* map. The actual numerical weights of strategy matching could be refined using machine learning techniques, based on success and failure of strategy applications.

In addition to specifying *MTerms* that determine when a strategy is applicable, the *mvars* ("matching variables") are used to parametrise the strategy application. The contents of the strategy could depend on specific parameters (e.g. the *MTerm* would match a goal that has an existentially quantified predicate, where a candidate witness is also available within the goal). The strategy would use the matched witness as a parameter to the proof command that instantiates the quantifier (e.g. `apply (rule_tac x = ?y in exI)` in Isabelle/HOL). The *mvars* are used to bind variables matched by the strategy for use in the strategy configuration. Several examples of *MTerms* matching and binding are listed in [FJVW14].

❦ ❦ ❦

The abstract model of the AI4FM system aims to identify what constitutes a strategy, how to describe when it should be used, what information is needed about the conjectures to facilitate reuse of high-level proof ideas and other related concepts. For full details, including structuring of proof process and strategy data into theory *bodies*, refer to recent AI4FM publications [FJVW13, FJVW14].

The proof process architecture proposed in this thesis aligns with the wider development of the strategy capture, learning and replay system: the specific links and corresponding data structures are identified in this section. The next section

presents a development of a concrete graph-based strategy replay system within the AI4FM project as well as how the ProofProcess system integrates with it.

7.3 Proof-strategy graphs

Proof-strategy graphs (PSGraphs) have been developed as part of AI4FM to provide a concrete representation for proof strategies and their replay functionality (strategy evaluation). At the basic level, proof-strategy graphs provide a robust representation for encoding composite proof tactics. The concepts and functionality, however, are general enough to describe high-level proof *strategies*.

This section provides a brief overview of proof-strategy graphs, their representation and evaluation. Implementation of PSGraphs is available in the Tinker tool, which also provides a user interface to inspect PSGraph evaluation and to construct new strategies within the system. For full details about the formalism and the implementation, refer to relevant publications [GKL13, GKL14].

Furthermore, this section highlights parallels between the captured ProofProcess data and the PSGraph strategy representation. Integration between the systems is discussed, including possibilities to generalise the captured data into strategies: i.e. how strategies can be extracted from the captured proof process and generalised further into highly-reusable forms.

7.3.1 Graphical strategy language

Proof-strategy graphs (PSGraphs) is a formalism that represents proof strategies as graphs, specifically *string diagrams*. String diagrams consists of *boxes* (graph nodes) that are connected using typed *wires* (graph edges), which need not be connected to a box at both ends. When encoding proof strategies, the boxes represent prover tactics (or nested PSGraphs—see below). The *wires* carry information about inputs and outputs of each tactic. This information, formalised as *goal types*, ensures that goals are "piped" correctly between tactics: i.e. that an output goal in one tactic matches the linked input goal of the following tactic. The open wires represent inputs and outputs of the overall strategy. A graphical representation of a proof strategy allows use of graph rewriting to formalise and perform strategy evaluation or transformation. It also provides a clear view of what the strategy does, particularly during a step-by-step evaluation. Full formal details about PSGraphs, including semantics of strategy evaluation are available in [GKL13].

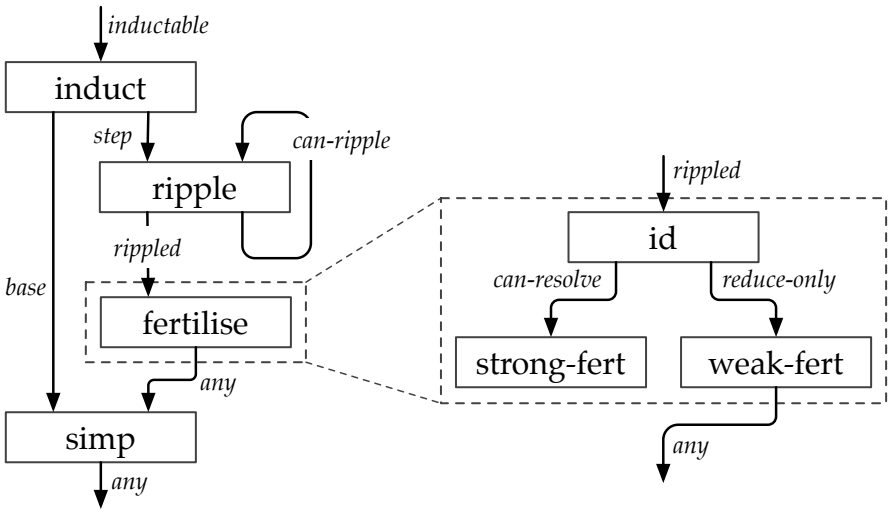


Figure 7.3: Proof-strategy graph for *rippling* strategy (see [GKL13] for details).

Goal types

Figure 7.3 depicts an example PSGraph representing the *rippling* [BBHI05] proof strategy. Each box represents either a low-level tactic in a theorem prover, or a high-level proof step, which in turn is represented as a nested PSGraph (e.g. the "fertilise" step). The wires in Figure 7.3 are labelled with *goal types*, each of them representing properties about the goal and serving as tactic inputs and outputs.

Goal types are specified as predicates over goals. A basic language to specify goal types is outlined in [GKL13], including a number of *atomic* goal types such as *top-symbol*$(x_1, \ldots, x_n)$ (top symbol of the goal is one of $x_1, \ldots, x_n$), *inductable* (structural induction is applicable), *hyp-embeds*, *measure-reducible*, etc. The goal types language allows more complex goal types to be constructed: e.g. the *can-ripple* goal type in Figure 7.3 is a conjunction of *hyp-embeds* and *measure-reducible*, meaning that it requires the hypothesis to embed in the goal and the rippling measure towards the hypothesis to be reducible.

Goal types represent the same matching information in strategies as *MTerm*s in the abstract model (Section 7.2.3). Furthermore, both representations are analogous to *proof features* in the proposed ProofProcess architecture, where proof features are used to capture high-level information about the proof steps. During strategy extraction, the proof features would be transferred and generalised to constitute *goal types* of corresponding proof strategy steps.

Atomic goal types are implemented within the system, thus must be known in advance. Composite goal types must be constructed out of them. Currently,

there is no support for custom (unknown) goal types (proof features). The basic goal types, as described in [GKL13], are evaluated over the proof state: i.e. the properties can describe the actual open goal term, available facts, shared meta-variables. Also, provision for relational properties is available, where goal types can link the parent goal and possible children goals (akin to the *out* proof features in the ProofProcess architecture). Furthermore, a more advanced *clausal* goal type is being developed that will generalise some mechanics of goal types as well as permitting expression of additional proof process properties, matching and binding variables (as required by *MTerms* and *mvars* in Section 7.2.3), and writing more general parametric proof strategies. The clausal goal type will also provide more support for passing, querying and constraining additional user-provided data, building up towards custom proof features.

Strategy evaluation

To prove a goal using a strategy encoded as a PSGraph, the goal is wrapped into a special *goal node* and added to a matching graph input wire. Then graph rewriting is used to transform the graph by "pushing" the goal node through wires with matching goal types and applying the tactics in boxes (see [GKL13] for details).

When a goal node reaches a box, the prover tactic represented by the box is applied and one or more output goals are produced. These goals are matched with the goal types of the output wires, determining the direction for each of them. Note that if an output goal does not match any of the output wires, the strategy fails. However, in contrast to standard tactic application, using PSGraphs allows pin-pointing the reason why the strategy evaluation has failed as well as what is expected of the goal after the tactic execution.

When the goals are matched with corresponding goal types, the process continues as the goal nodes are piped further through the graph. The goal type specifications on wires ensure that only the right "types" of goals are accepted.

Non-determinism is permitted in strategy evaluation. A tactic can perform non-deterministically (e.g. when alternative strategies are available in a *graph tactic*—see below), or a sub-goal can match more than one goal type. In these cases, the search space would expand as the evaluation branches to accommodate all possible outcomes. Note, however, that appropriate goal-type definitions with narrower predicates could help in minimising the non-determinism.

Strategy hierarchies and combinators

Strategy *hierarchies* (nested strategies) in PSGraphs are represented in the same way as prover tactics: as *boxes*. A special tactic—*graph tactic*—is used to embed nested PSGraphs into tactic boxes. These hierarchical nodes can be labelled explicitly. When extracting strategies from the captured ProofProcess data, proof process *intent* data can be used to label corresponding strategy steps. The hierarchical structure of PSGraphs allows replication of the different levels of abstraction in the captured proof process.

Furthermore, a hierarchical node can hold a number of alternative nested strategy graphs. The order in which they are evaluated can be given by specifying an *alternation style*: e.g. *OR* style would indicate non-deterministic choice of a nested strategy (actually branching the evaluation to check all possibilities), *ORELSE* style would evaluate the nested strategies in turn until the first success, etc.

The hierarchical nodes in PSGraphs allow specification of alternative proof strategy steps. The alternatives can be collected from the captured ProofProcess data, for example by grouping the different proof steps with the same proof intent. This works well when the same intent is used to describe the same *high-level* strategy, hence marking them as alternatives even when the low-level tactics differ.

Graph *combinators* are used to build new strategy graphs from existing ones. They can specify sequential strategies (*THEN* combinator), parallel partitioning of goals (*TENSOR* combinator), repetitive application of strategy ($REPEAT_{\alpha}$ combinator to repeat application while the output goal matches α goal type), etc. *THEN* and *TENSOR* combinators correspond to *ProofSeq* and *ProofParallel* data structures in the ProofProcess tree (Section 4.3), thus allowing direct mapping during strategy extraction. The repetition combinator, however, is not required when capturing the proof process. The captured data represents the actual proof process, where strategy repetition is recorded as one or more sequential instances of the actual strategy being applied. However, when extracting the strategy, this can be generalised into strategy repetition, improving reusability of the strategy.

By using graph combinators and hierarchical nodes, a high-level strategy can be described as a proof-strategy graph. The low-level proof tactic steps would be wrapped into hierarchical nodes with labels recording the high-level intent. These can be combined into larger multi-step proof strategies with branching, repetition and alternative proof steps. However, at all levels of abstraction, goal types would govern correct "piping" of goals during strategy evaluation. Furthermore, each

hierarchical strategy is justified by actual prover tactic steps at the lowest level.

7.3.2 Tinker: implementation of proof-strategy graphs

The first implementation of PSGraphs, the Tinker tool,[3] provides a generic, theorem prover independent framework to specify and evaluate PSGraphs. The Tinker tool is implemented in Poly/ML, which allows easier integration with a number of theorem prover systems. Currently, integration with Isabelle and ProofPower theorem provers is available, providing special proof methods/functions to evaluate a specific PSGraph for the open goal. Refer to [GKL14] for details on the architecture and implementation of the tool.

The integration with theorem provers is done as an extension, so users can still use other proof commands: the Tinker tool appears as yet another proof method. When used automatically, the selected PSGraph is evaluated as much as possible to transform the current goal. The proof is constructed within the theorem prover, ensuring its correctness. Furthermore, an interactive evaluation of the PSGraph can be launched. This provides a graphical user interface to step through the evaluation, useful for "debugging" the strategies. However, manually selecting the wires for sub-goals or otherwise influencing the execution is not supported at the moment. Finally, the graphical interface can also be used to create new PSGraph strategies (including hierarchical nodes), by specifying prover tactics and goal types from a given selection of supported ones.

Strategy search

Currently Tinker supports evaluating a single multi-step proof strategy, encoded as a PSGraph. When executing the appropriate proof method in the theorem prover, the user indicates the strategy that is to be used for the current goal.

Searching for a matching strategy is not supported at the moment. Furthermore, changing to a different strategy midway in the proof is also not possible, as the strategy execution continues until success or failure. However, the basic building blocks for the functionality are there. When sub-goals are directed to matching wires, goal-type matching is performed. This can be extended to run on the current goal (e.g. as if there is a single super-PSGraph that combines all known PSGraphs with the *TENSOR* combinator) and select an appropriate wire leading to a matching strategy. Furthermore, the evaluation is done with standard prover

[3] Tinker website and source code are available at `http://ggrov.github.io/tinker/`.

tactics, thus exporting the used proof commands or just the proof state would allow selection of a new strategy after terminating the execution of another.

The approach to non-determinism in Tinker means that alternative strategies would be searched exhaustively via branching (*OR*), or sequentially in some order until first success (*ORELSE*). The abstract description of proof strategy matching using *MTerms* in Section 7.2.3 proposes using weightings to guide the proof search when using strategies. These, however, are not supported at the moment in Tinker. Examining the different approaches to strategy replay (multi-step vs single-step and search, exhaustive search vs weighted matching, etc.) is among the future directions of AI4FM research.

Integration with proof process capture

The PSGraphs formalism and implementation in Tinker provide ways of encoding and replaying proof strategies. However, the issue of creating good reusable strategies still exists. The ProofProcess system can provide data on how the expert achieves the proof, which could be generalised into strategies that can be encoded and replayed as PSGraphs.

The previous section highlighted a number of parallels between the captured proof process data and the strategies. Proof intents can be used to label hierarchical strategies as well as group alternative ones. Important proof features, either inferred automatically or marked manually by the expert, can make up the goal type predicates and describe proof strategy matching. The branching structure of the captured proof can be translated to the strategy branches, etc.

A captured proof process attempt can directly become a strategy when converted to a PSGraph. To support that, a provisional interface between the ProofProcess system and the Tinker tool has been developed. It can export captured proof process data of each proof attempt, encoding the branching and grouping structure, tactics used, intents and proof features, etc. When imported into Tinker, this becomes a proof-strategy graph, representing the proof attempt as a *very specific* instance of a strategy. To improve reusability, this strategy should be generalised further (Section 7.3.3 elaborates more on generalisation). Furthermore, other integration features are also planned: e.g. exporting the proof attempts live, while doing the proof; using Tinker to generate proof feature suggestions, etc.

7.3.3 Generalising proof strategies

Proof strategy generalisation can be done via multiple vectors: generalising proof features and tactics, discovering patterns in the strategy, introducing alternative steps, etc. All these aim to make the strategy more reusable to handle *similar* proofs to the one that gave rise to the strategy. However, generalisation has an important property that any valid proofs within a strategy should also be valid within the generalised one. This preserves links between the captured or replayed proofs as *instances* of strategies: each proof can be seen as a very specific refinement of the general strategy and vice versa.

Automated strategy generalisation within the PSGraphs framework is explored by Grov and Maclean [GM13]. They start at the lowest level: with a single proof of a goal. At each proof step, the goal and proof state are lifted to derive their *goal types*. This is done by matching predefined goal type functions on the goal. However, this approach generates a lot of noise: not all proof features are important. The excess proof features (goal types) clutter the generalisation. Deferring to the expert doing the proof for identification of important proof features would limit the noise and highlight the genuinely important ones. The ProofProcess framework employs this approach to collect high-level proof process data, which will be passed to the strategy extraction.

With the proof abstracted using goal types, these can be used to generalise further or identify patterns in the strategies [GM13]. The authors propose several automatic generalisation techniques:

- Proof tactics with arguments (e.g. `subst` or `rule` in Isabelle/HOL) can be generalised by taking a union of lemmas used in the arguments.
- Alternatively, proof tactics could be wrapped into a graph tactic, making them alternative steps. However, the proof search would be widened as both alternatives would be explored. This approach can also be used for generalising high-level proof steps, by wrapping strategies with the same intent into a single graph tactic as alternatives.
- Goal types can be generalised by computing the most general type for two existing goal types (similar to *anti-unification* [Plo69]). There can be multiple variants of generalised goal types for the same existing ones.
- A single goal type can be weakened by making its description more general. Again, there can be multiple variants of weakening for the same goal type.

When an expert is using the ProofProcess framework to capture the proof process, he could mark the important proof features as already generalised. For example, the *shape* proof features use placeholders to describe an important shape, already generalised to use with different variables. These proof features could be used directly in strategy specification.

Goal types can also be used to discover loops in a strategy [GM13]. When the goal type on an output wire matches ("is a subtype of") the goal type on an input wire of the same tactic, a loop can be extracted. The loop is represented using a *REPEAT* combinator on the goal type. A subsequent non-matching (*orthogonal* in [GM13]) goal type would become the termination condition of the loop.

❦ ❦ ❦

Proof-strategy graphs and their implementation as the Tinker tool allow specification of proof strategies with high-level abstractions such as goal types. The strategies can be replayed on goals in the Isabelle or ProofPower theorem provers. The system is being developed to support more advanced goal types, better integration with proof process capture and reuse of user-provided high-level information.

Extracting general proof strategies is a difficult task. This section presents several approaches to generalising proof features, proof steps and identifying patterns. The approach is still young: implementation requires proof feature types to be predefined, all goals need to be carefully marked with proof features, etc.

The ProofProcess framework has been designed to allow the expert to provide any data he feels important to describe the proof process and the high-level insight. Collecting all this information helps in populating the list of proof feature types as well as giving ideas of how different proofs are constructed. The mechanisation of this information can follow. However, in the interim—and even long-term—the captured proof process data can still be used manually. The next section briefly explores the "manual" extraction and reuse of proof strategies.

7.4 Reuse by analogy

This chapter presented an overview of *proof strategies*: what comprises them, how to specify when strategies should be used, what are the approaches to strategy replay as well as how to extract strategies from the captured proof process information. The research done in the AI4FM project aims to automate the process of extracting and replaying proof strategies, however much still needs to be done.

Nevertheless, even without the machinery to extract and reuse strategies, the captured proof process information can help "engineer" users to tackle similar proofs. While difficult to mechanise, most of the proof features are easy for a human user to comprehend and manually adapt for similar proof steps. A human user can spot proof feature or tactic generalisations or even leapfrog a gap in the strategy—all of which may require advanced heuristics or massive datasets for machine learning to replicate automatically.

Strategy reuse by (manual) analogy—i.e. when the user manually creates a new proof, analogous to an existing one—is the most basic approach to strategy extraction and replay. When dealing with a new proof, the user would sift through the selection of existing proofs to find one with a similar goal. Then, by looking at how the previous proof was done, the user could replicate the approach with the new goal. The captured high-level proof process information would highlight the important properties and guide the user through the previously taken proof steps to discover the new proof. Strategy extraction would be done in the user's head via theoretical/intuitive generalisation, while replay would be him manually typing the adapted proof commands in the proof script.

This approach highlights the self-sufficiency of a proof process capture system. By providing means to mark high-level information, the system makes it easier to adapt and reuse the expert's insight. Even without subsequent strategy extraction or automatic replay functionality, new users can benefit from the captured information. However, the "repetition" effort for similar proofs is not really helped by this approach as the user has to manually retype the proof commands. Nevertheless, the benefit is obvious for other use cases: e.g. when an "engineer" needs to reuse the proof ideas of the "expert".

The main issue with manual reuse is finding the matching strategies. If the database of captured proof process data is large, it can take a long time to scour the list of attempts for a matching one. Furthermore, new searches for matching strategies may be needed midway through the proof or even after each step.

The list of similar previous proofs can be narrowed by matching proof features. Such functionality is also needed for "inferring" the proof process: e.g. to suggest proof features when a proof process is being captured (see Section 6.5). Furthermore, proof feature matching is a core part of automated strategy search and replay (this time with generalised proof features). Therefore matching proof features is among the principal goals when "learning proof from the expert".

The manual reuse of the captured proof process, as described here, is used in

the case studies listed in Chapters 11–12. The case studies describe how the proof process of some lemma is captured. The captured data is then reused to prove similar lemmas. The search part there is simplified, as the user already knows which proof is being reused and the overall list of captured strategies is small. The matching and generalisation are done intuitively. The case studies illustrate that the manual reuse is viable and successful as well as inform further research on automated techniques to replicate the process.

PART III

Implementation

CHAPTER 8

ProofProcess framework

The architecture of the proposed proof process capture system is described in Part II, covering how high-level proof insight can be captured by abstracting the interactive proof process as well as how such a capture system would work. A prototype implementation of the core ideas outlined in the architecture is available as the ProofProcess framework,[1] described in this part. The implementation includes a generic core that focuses on data representation and manipulation, storage and proof process analysis as well as generic user interface components. Prototype integrations with Isabelle [NPW02] and Z/EVES [Saa97] theorem provers are available, described in Chapters 9 and 10, respectively. They "wire-tap" the provers and provide prover-specific data representation and analysis functionality.

The proposed architecture describes various aspects of proof process capture, including streamlined user interactions for data input or advanced automation facilities to reduce manual effort. A perfect ProofProcess system would be mostly invisible, crunching the proof process data from the prover in the background, while the user is doing interactive proof. The captured data would be presented conveniently, seamlessly intertwined with what the user is doing in the theorem prover, but also enabling the user to explore the captured details or the historical data in a natural manner. When manual input is required, the user would be able to easily query the prover for information about the proof context, link the proof data with the captured proof process, mark and manipulate the proof features and other high-level proof process information. Furthermore, the system would

[1] The ProofProcess framework and theorem prover integrations are open-source. All source code is available at `http://github.com/andriusvelykis/proofprocess`.

always be one step ahead, suggesting possible proof features or other abstractions. Finally, the captured proof process would yield proof strategies, which the user could reuse immediately.

Unfortunately, implementing such a *perfect* ProofProcess system is an enormous engineering effort. Polishing the user interaction, designing automation facilities and implementing extraction and replay of proof strategies requires a large development team and significant resources. Lacking that, the prototype system focuses on data representation and capture with basic manual input functionality. However, the system is designed to be generic and extensible, where support for additional features leading towards the *perfect* system can be added in a modular manner. Some of these features (e.g. the graph view of the proof process data) are already implemented as very basic proofs of concept.

In general, the implementation follows the ideas outlined in the architecture (Part II) quite closely. Therefore, rather than repeating them, this chapter describes the details of challenges and solutions in realising these ideas. These include how the data is represented and stored in the system, how file history is recorded as well as how a generic proof capture system is built and organised. Chapters 9–10 describe how the core framework is integrated with the theorem provers, how proof data is intercepted and analysed. They also cover additional infrastructure development to support the integration, namely the Isabelle/Eclipse and CZT+Z/EVES systems. Before delving into the details, the next sections present an overview of the current prototype system and how it is used.

8.1 Implementation overview

Size and system requirements

The prototype implementation is not large: the core functionality together with both theorem prover integrations totals approximately 15k (15,000) Scala source lines of code (sloc), implemented in almost 900 commits. This number does not include the implementation of the data structures, which are generated automatically from an EMF model and total ~30k sloc.

Supporting the ProofProcess framework implementation, the Isabelle/Eclipse prover IDE is of a similar size (~12k Scala sloc, 600 commits, see Section 9.4). Quantifying the contributions to CZT (Section 10.3) is difficult, but the project itself is much larger (~300k Java sloc, developed since 2002 by a number of contributors).

When capturing proof process data for the case studies presented in this thesis, the systems were given 1GB of memory. The underlying platform components (Eclipse and Scala as well as CZT and Isabelle) have quite high system requirements and more memory would be recommended if needed or if faced with large formal specifications and proofs.

Early versions of the prototype system had difficulty handling larger amounts of captured proof process data. As reported in Section 8.5, after an intensive day of work, the system would crash with an out-of-memory error even when allocated 8GB of memory or more. This has been addressed in later versions by using a database with dynamic loading and unloading, thus memory requirements are sensible (e.g. 1GB of memory is enough for the case studies).

The system tracks the expert doing interactive proof and captures a lot of data about it, including all proof attempts and details about the goals, etc. All this information is stored in a database, which size thus depends on the size of the project, the number of attempts, the size of the goals, etc. When using early versions of the system, captured proof process data in databases reached sizes of 20GB or more after a day of work. However, this has also been solved using data compression (see Section 8.5.4): captured data from the *separation kernel* Z specification (see [Vel09] and Chapter 10) totalling ~8k sloc (1300 Z/EVES proof steps) requires database size of approximately 60MB (single attempt on all proofs).

The computation overhead of capturing the proof process data is difficult to quantify. However, because of low-priority processing, it should not affect the user's work with the prover. Nevertheless, a user of early versions of the system reported perceived slowdown of the user interface after extended use, which has not been fully investigated.

Usage

The ProofProcess system has been trialled by the author and another researcher within AI4FM (Leo Freitas) during its development. The main proof developments it was used with include the heap case study (~5k sloc in Z/EVES, ~6k sloc in Isabelle/HOL, see [FJVW13]), the separation kernel case study (~8k sloc in Z/EVES, see [Vel09]), and several other developments of similar size. The prototype system captured the low-level proof process data, identified different attempts, recorded proof history, etc. Since marking of high-level proof insight was not supported in earlier versions, the majority of captured data is low-level. Furthermore, the

aforementioned scalability issues were identified and rectified.

These case studies are *industrial-style* formal developments, however their size is small in comparison to the ones done in industry. No large-scale evaluation of the system with actual *industrial-size* formal developments has been done.

8.2 Using the system

The development of the ProofProcess system is driven by the need to test the model and ideas about capturing an expert's interactive proof process: both the high-level insight and the associated low-level proof data. When dealing with industrial-style proofs (Section 2.1), instantiating a model as a whiteboard or pen-and-paper exercise becomes infeasible. Even replicating the proof process in an electronic document is difficult, particularly when it comes to linking with low-level proof data. Tool support with links to a theorem prover is necessary when capturing large proof processes. Furthermore, data from a number of earlier industrial-style formal developments is available (e.g. the mechanisation of the Mondex smart card specification using Z/EVES in [FW08]) and running the proofs through a tool could yield good examples of proof processes.

The focus of the prototype implementation is, therefore, on data capture, representation and manual manipulation. The first priority is provision of data for strategy extraction. The tool enables the testing of the accuracy and descriptiveness of the proof process model, how such data could be generalised into strategies. The necessary abstractions would be provided manually. Improving the automation to infer the proof process (Chapter 6) is only a subsequent step.

However, even though some of the current functionality of the system is at a prototype level, the design anticipates it growing into a fully fledged proof capture system. Hence it is a ProofProcess *framework*: a solid generic base system for proof process representation and capture, which can be extended by other systems.

8.2.1 Recording proof data

The ProofProcess system aims to eventually become a non-intrusive proof assistant. It would assist users with capturing and reusing the proof process, but would not force them to work in a prescribed way. Figure 8.1 presents a screenshot of an Isabelle/Eclipse application with the ProofProcess system running alongside and capturing the proofs. Some parts of the ProofProcess user interface can be

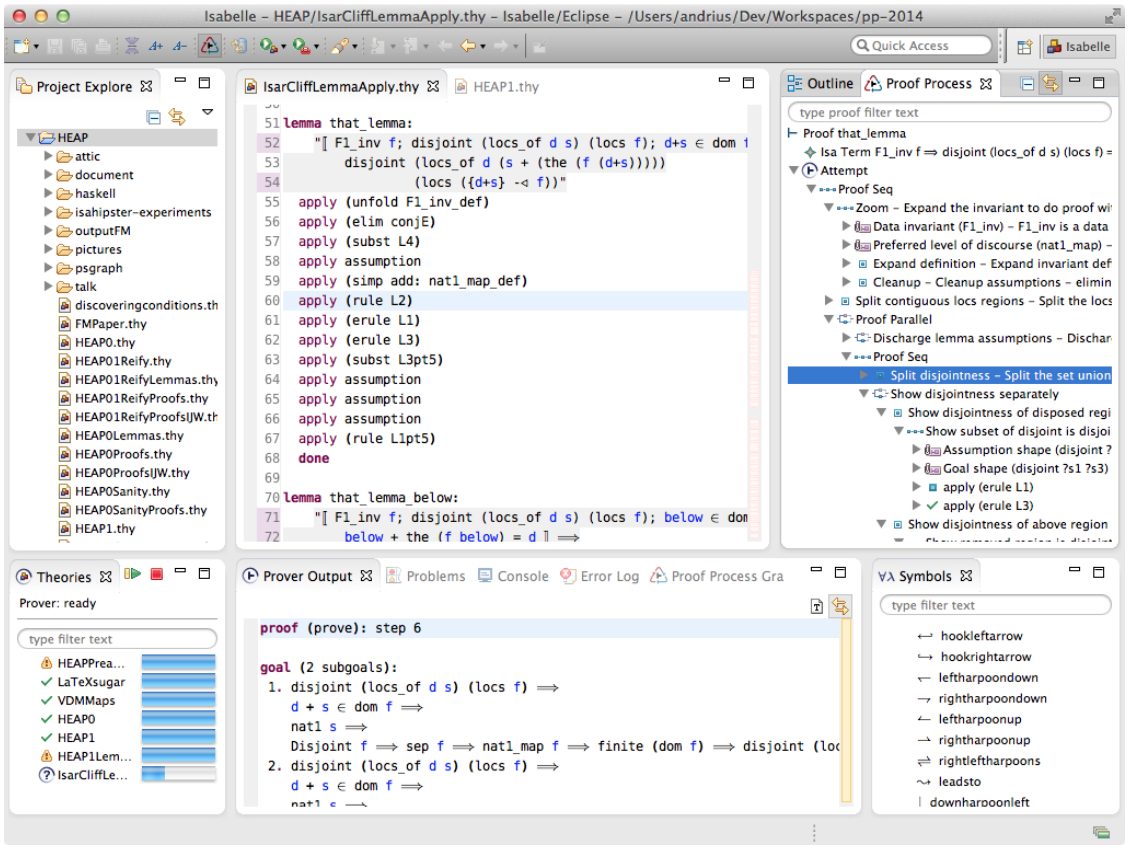


Figure 8.1: Screenshot of Isabelle/Eclipse with the ProofProcess system.

seen (e.g. particularly the *Proof Process* view in the top-right corner), however a significant part of the application (and especially the generic framework) is used in the background and does not have much user interaction.

The majority of the "screen estate" of the application is kept for the theorem prover functionality.[2] The user does interactive proof with the familiar tools and the ProofProcess system does not aim to take over that. For example, in Isabelle/Eclipse, the formal specification, lemmas and the proof commands are entered using the main proof script editor. These are processed automatically by the theorem prover: its results are displayed in the editor as well as in the *Prover Output* view for the highlighted command. Other proof assistants work in a similar manner.

The ProofProcess system subscribes to proof change events using the *observer* pattern: i.e. "wire-taps" the prover communication. Upon each notification, the change events are scheduled to be analysed. By using parallel processing techniques, the analysis work is offloaded to a lower-priority job, thus the user can

[2]More precisely, to the functionality of the *proof assistant* that provides user interface to the theorem prover: e.g. Figure 8.1 shows the Isabelle/Eclipse prover IDE for the Isabelle theorem prover.

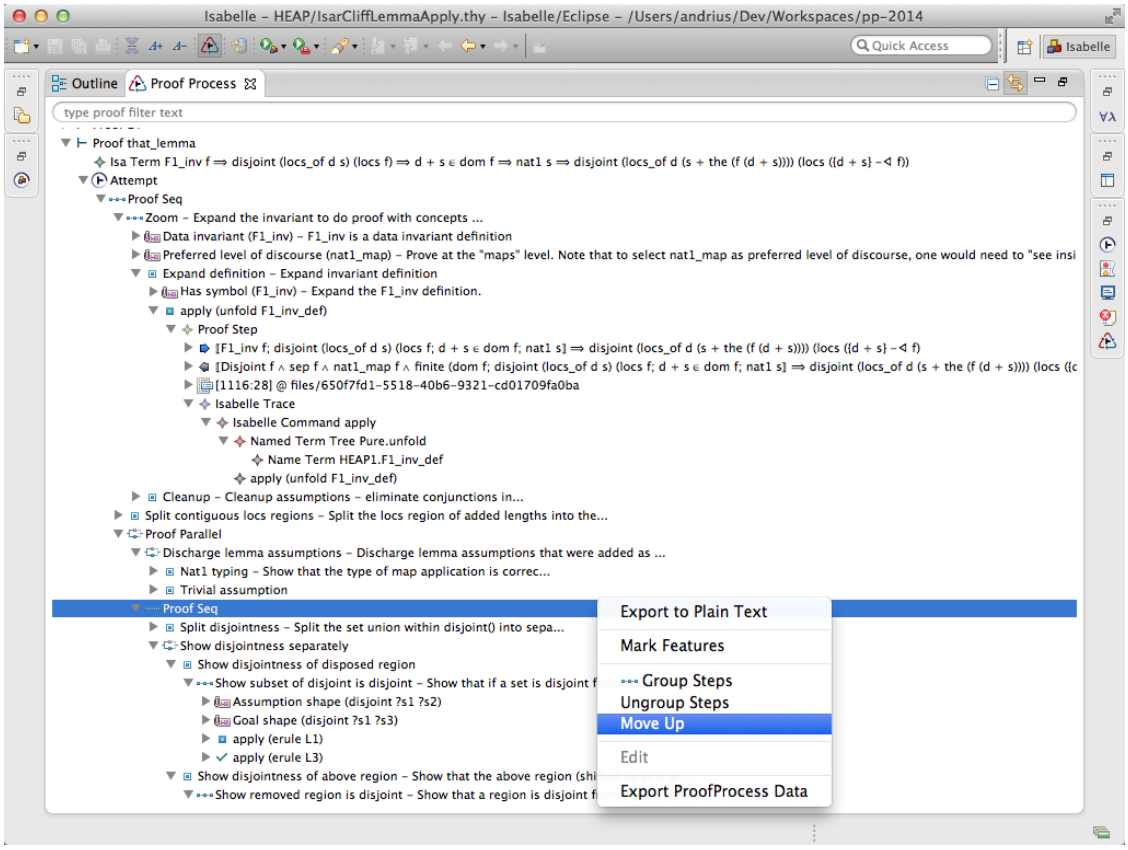


Figure 8.2: Screenshot of the *Proof Process* view with a ProofProcess data tree.

focus on the theorem proving with little effect on processing speed. Proof process tracking can be disabled temporarily using a toggle button: the user can work with the prover as if the add-on was not there. This only turns off the new data tracking, the user can still interact with the captured data.

8.2.2 Viewing the captured data

Currently the main point of interaction with the captured proof process data is the *Proof Process* view.[3] Figure 8.2 presents a screenshot of this view with some captured data listed. The *Proof Process* view displays all captured data in a tree structure. At the root of the tree is a *ProofStore*, which contains *Proof*s. Each *Proof* can have multiple *Attempt*s with individual proof trees within, etc. Refer to Section 8.4 for more details about how proof process data is structured. As this dataset can get large, a search field allows for filtering of the displayed proof data.

[3] *Views* in Eclipse are application parts (represented as window tabs) used to display information related to open *editors*.

As proof process data is captured, the *Proof Process* view is constantly updated and can track the "latest" proof element. For example, as the user is submitting the proof commands to the prover, the last-submitted command is highlighted in the tree with a small delay. This improves the awareness of how the captured data corresponds to what the user is doing. Furthermore, the tree presentation is tuned to improve readability: e.g. intermediate elements are omitted or flattened, data is visualised using labels and icons, etc.

Some proof process information is inferred immediately after capture: e.g. recognising the proof branching structure or matching the attempt (Sections 6.2–6.3). New data is presented to the user assigned to a correct attempt and with a basic structure. However, the user has to supply other proof process abstractions.

The *Proof Process* view provides actions to manipulate the proof process structure as well as to assign proof intent and mark proof features. They are available in the context menu, as shown in Figure 8.2. Users can group or ungroup proof steps: grouping introduces higher-level proof steps, e.g. a sequence of steps can be grouped into one. A single proof step can be "grouped" to provide a *decoration* step to assign additional meta-information. Furthermore, proof branches can be reordered within a *ProofParallel* element: they belong to a set and the order is not important, but providing a particular order in the tree can help with readability.

8.2.3 Marking proof insight

Proof insight (proof *intent* and proof *features*) can be marked for every proof step, at any level of abstraction (see Section 4.2.3). The *Mark Features* dialog is used to view and enter this data (Figure 8.3). The proof intent can be selected from the list of existing ones (e.g. if the same high-level step is used) or a new intent name can be provided. A convenient dialog (Figure 8.4) combines searching and entering a new name—it is used to select proof intents or proof feature names.

The *Mark Features* dialog provides an overview of the proof step: it lists the in/out goals and proof features, i.e. the *before-* and *after-state* of a proof step. For high-level proof steps, in/out goals are collected from their underlying low-level steps by "flattening" the proof tree (Section 4.3.5). Furthermore, the goal presentation can be *filtered* to show only the changed parts. The in/out goals of a proof step are compared and the parts of the goal that are the same are hidden. Then only the parts that have changed are left on display, pinpointing the important terms in the goal (see Section 6.5.3). Figure 8.5 shows how lemma insertion is emphasised

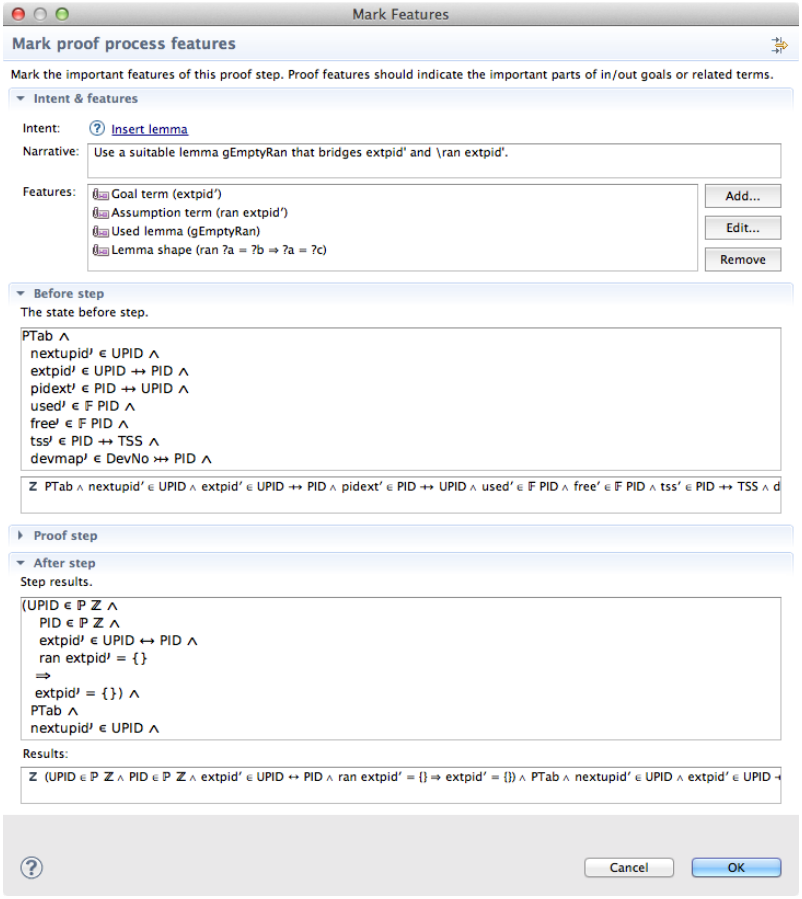


Figure 8.3: Screenshot of the *Mark Features* dialog (Z/EVES proof process data).

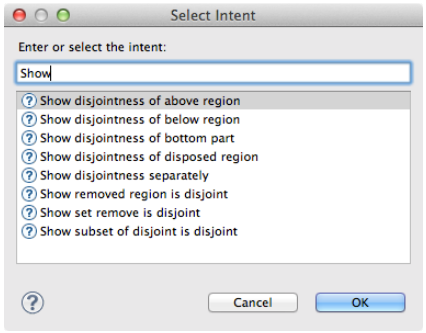


Figure 8.4: Screenshot of proof intent selection.

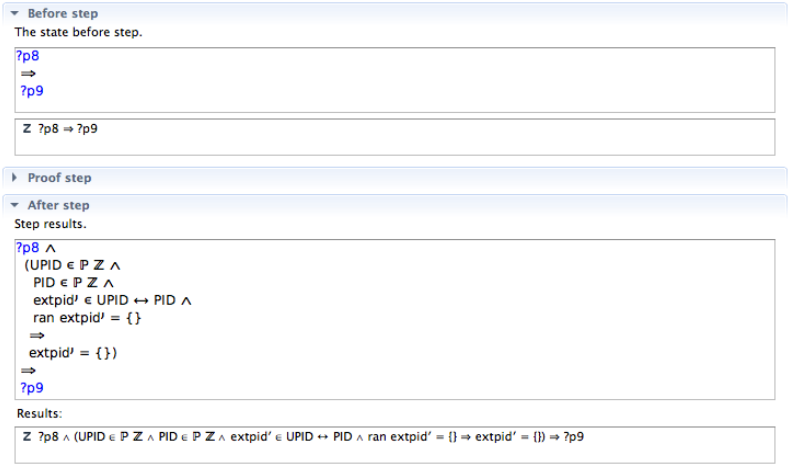


Figure 8.5: Filtered view of goals in the *Mark Features* dialog (cf. Figure 8.3).

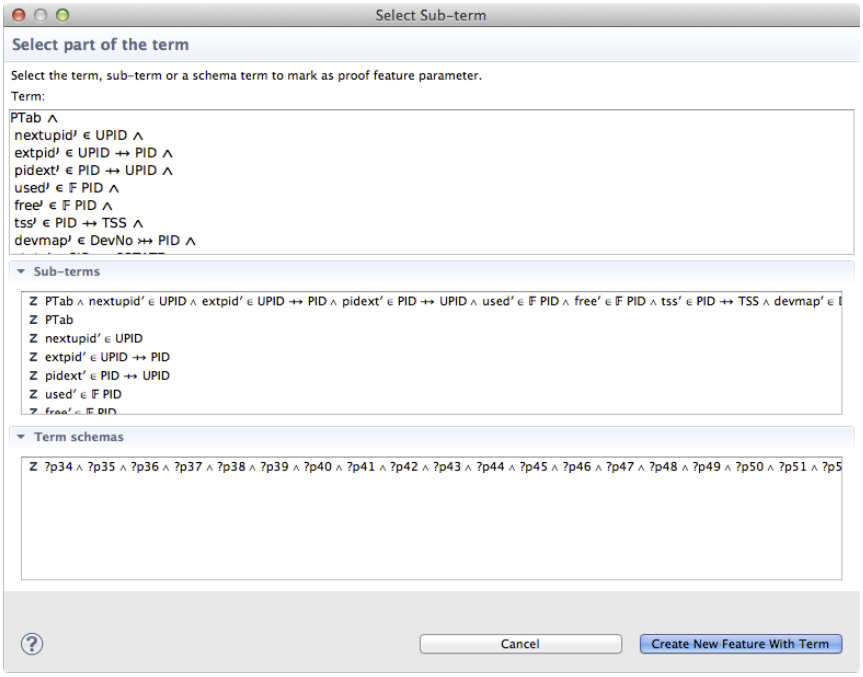


Figure 8.6: Screenshot of sub-term selector.

using goal filtering—compare it with the unfiltered goals in Figure 8.3.

Current facilities to mark proof features are quite basic. A proof feature is recorded by selecting a proof feature definition (in the same way as intents are selected) and adding parameter terms. When terms within the goal are used as proof feature parameters, the system facilitates the sub-term selection by breaking down the goal term and allowing the user to "dig" into the term to mark the important sub-terms (see a screenshot in Figure 8.6). A "generic" version of a sub-term

(*schema*-term, term with placeholders) is also generated for selection by simply replacing the terms with placeholder variables. The sub-term and *schema*-term selection is provided by the prover-specific extensions: e.g. Z/EVES ProofProcess knows how to break down Z predicates into constituent predicates and further.

More sophisticated user interface solutions can be imagined (and wished for): e.g. a user would select sub-terms directly in the proof assistant; the placeholders would have more variability or allow to generalisation on types, sub-terms, etc. However, extra choices would add overhead to marking the proof features—and would not necessarily improve their quality; or waste effort on generalising them when specific ones would have sufficed for solving the proof family. Alternatively, the expert could be faster by entering details as text, which could be parsed against the proof context and goals to resolve the terms and disambiguate the text input.

Parameters for other proof feature types (Section 4.2.1) are not supported at the moment. As a workaround, a temporary *StringTerm* parameter is available to enter parameters as text.

Using the available tools in the prototype ProofProcess system, an expert can capture the interactive proof and manually provide the high-level insight. The prototype system supports the proposed expressiveness of proof intent and proof features when describing the proof process. The expert (and the AI4FM researchers) can explore the best approaches to describing a proof process and extracting reusable strategies from it.

8.2.4 Supporting functionality

A significant part of the provided functionality in the ProofProcess system is not directly visible to the user. The captured data is collected and stored within a database, preserving data on various previous proof attempts. Furthermore, full proof history is recorded and versions of proof scripts are saved to enable future functionality in "animating" the expert by replaying the proof scripts. These are discussed in more detail within this chapter.

The captured data can have many uses: some examples in addition to strategy extraction are explored in Section 13.4. The prototype implementation also provides a couple of options to export the data. Exporting the captured proof process as *plain text* prints out a large text document with the selected attempts and their proof trees. The high-level insight information is included for each proof step. Furthermore, Isabelle ProofProcess supports the exporting of proof trees in a format

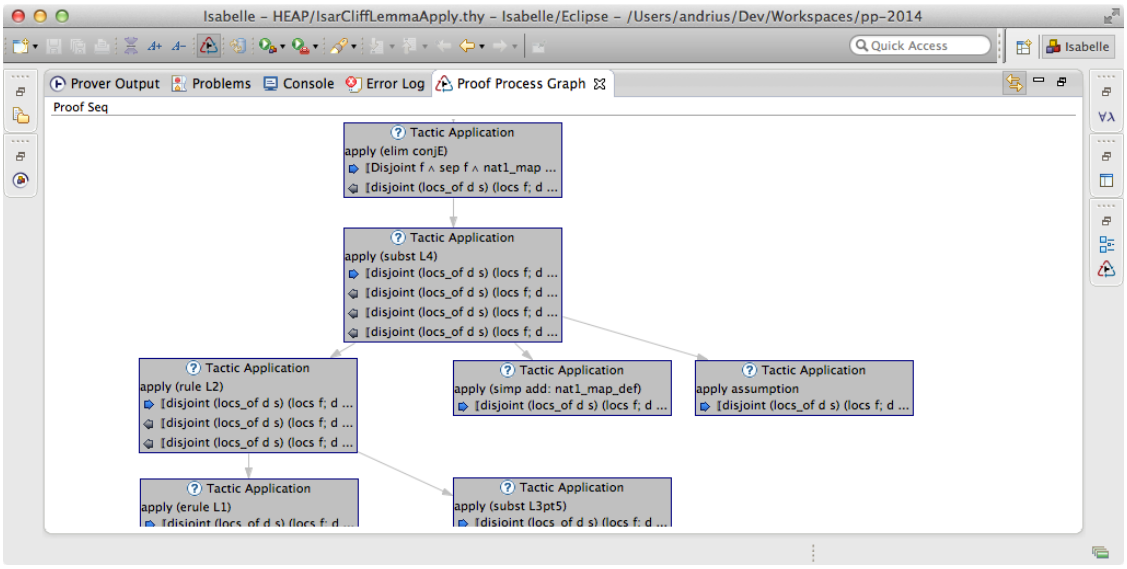


Figure 8.7: Screenshot of the *Proof Process Graph* view.

that can be imported into the Tinker tool [GKL14] as a step towards providing data for strategy generalisation. In general, the captured data can be accessed and manipulated using standard EMF tools and the available APIs (see Section 8.3.2).

The *Proof Process Graph* view is a proof-of-concept on rendering the captured proof process data as a graph structure (Figure 8.7). It uses the Eclipse Zest visualisation toolkit to render the graph representation of data (Section 8.6). The view helps in identifying the relationships between the low-level proof steps and proof branches in a complex structure. Higher-level abstractions are not supported in the current version. Eventually, the aim is to support both the tree and graph views of proof process data, where graph representation would be similar to Figure 11.5 or to the *hiproofs/higraphs* rendering style (e.g. in IsaPlanner [DF03]).

8.3 Structure and design

This implementation of the proof capture system focuses on building a generic platform rather than hacking together a quick system to test the model. Producing a generic, prover-independent framework is an objective of this PhD research. The prototype proof process capture support for two theorem provers, Isabelle and Z/EVES, shows how the approach and the generic core functionality can be reused.

This section outlines the overall structure of the system: the main components, their dependencies and the modularity of the design. Furthermore, the software

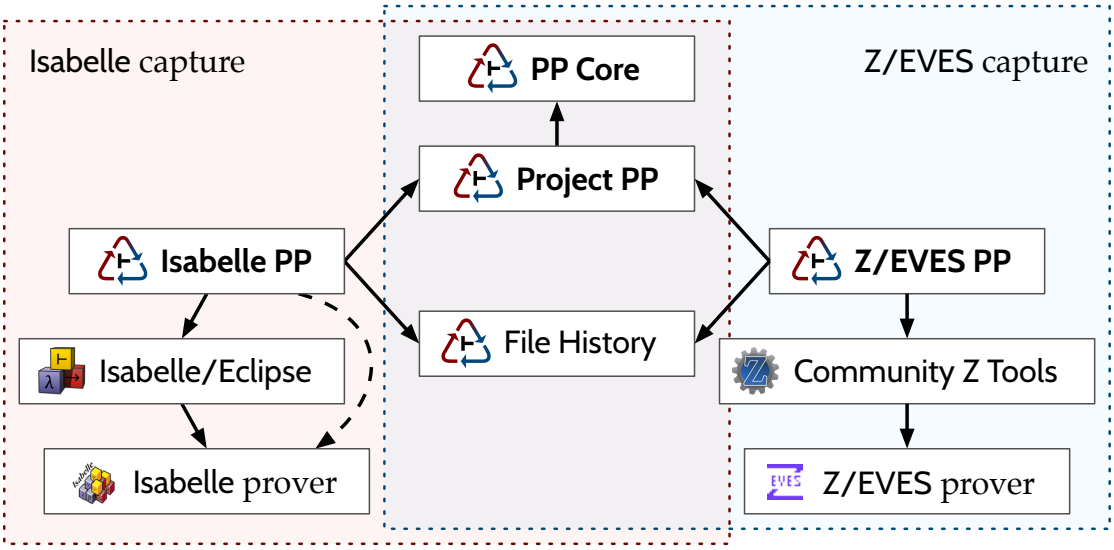


Figure 8.8: Structure of the ProofProcess (PP) system.

platform on which the prototype system is implemented is presented.

8.3.1 Component systems

The aim to reuse the proof process capture functionality with different theorem provers leads to a modular design of the ProofProcess system. The system features a generic core together with (currently) two prover-specific extensions integrating with Isabelle and Z/EVES theorem proving systems. Figure 8.8 provides an overview of the ProofProcess system structure. The figure lists the main components and identifies their dependencies. The arrows between components indicate their dependencies: e.g. Project ProofProcess depends on ProofProcess Core.

Such a structuring of the components enables identification of *subsystems* along the lines of the dependencies. The components comprising the generic base of the ProofProcess *framework* occupy the central region of Figure 8.8: they are independent of any theorem prover. Furthermore, Isabelle/Eclipse or CZT+Z/EVES extensions, which have also been developed during this PhD research, can be used standalone (without the ProofProcess add-ons) as general-purpose tools for formal specification development and theorem proving. Two main applications, Isabelle ProofProcess and Z/EVES ProofProcess, collect their respective components into full proof process capture systems.[4]

[4]In general, separate systems would be used to capture proofs from different theorem provers, e.g. Isabelle ProofProcess and Z/EVES ProofProcess would be two distinct applications. However,

The main components listed in Figure 8.8 provide a convenient structure to discuss the whole ProofProcess framework. This structure is preserved at all levels of design and implementation: data structures, core functionality, user interface, etc. It is convenient to present the implementation details of the ProofProcess system in this chapter along similar lines: this chapter covers the generic *framework*, whereas prover integration details are discussed in Chapters 9–10. Before delving into the details of some of the ProofProcess system features, the following paragraphs present an overview of the main components.

ProofProcess Core

The core modules provide the main data structures to represent a generic proof process with the accompanying functionality to manipulate and present it:

- Core data structures to represent a generic proof process. The data structures closely follow the ProofProcess model proposed in Chapter 4.
- Proof attempt recognition: proof re-runs, extending with new steps and diverging attempts (Section 6.3).
- Alternative *graph* representation of ProofProcess proof tree structures and conversion facilities between the two representations (Section 8.6).
- Goal-based proof structure analysis (Section 6.2).
- All UI components to view and manipulate the captured proof process data.
- Proof feature identification and marking (Section 8.2.3).

Project ProofProcess

The "project" component hosts further generic functionality: a *ProofStore* implementation to provide separate storage of proof process data for each Eclipse project, as well as a *ProofLog* to record the history of proof activities. This functionality extends the main core but takes certain implementation decisions, hence it lives outside of ProofProcess Core to allow for different implementations in the future. Section 8.4 provides more details.

the implementation allows packaging both within the same IDE that runs two different theorem provers and shares the ProofProcess components.

File History

The file history component provides storing and tracking of the history of proof script file changes. It is used to reference the actual proof script text from the captured proof process. Rather than capturing every minor edit, the file history is optimised for linear proof scripts and only records significant changes. The functionality is only concerned with text files, thus making it a generic standalone component without any dependencies on the ProofProcess components (therefore it does not carry the ProofProcess name). Section 8.7 provides further details on the proof history design and implementation.

Isabelle ProofProcess and Z/EVES ProofProcess

The integration components provide prover-specific extensions to the base ProofProcess framework. These include representation of terms, details about the actual proof steps, etc. Furthermore, these components provide "wire-tapping" (Section 3.2.1) of the theorem proving system to query, parse and record the ongoing proof details. Further details are available in Chapters 9–10.

Isabelle/Eclipse

Isabelle/Eclipse provides an Eclipse-based prover IDE (PIDE) for the Isabelle theorem prover. It supports interactive proof document authoring with asynchronous proof checking. Isabelle ProofProcess currently builds upon Isabelle/Eclipse to "wire-tap" the prover, however the main dependencies are on Isabelle directly (particularly the Isabelle/Scala API). The dependency is highlighted using a dashed arrow in Figure 8.8. This means that integration via other Isabelle PIDEs (e.g. Isabelle/jEdit) would not be difficult. Isabelle/Eclipse has been developed as part of this PhD research and is discussed in Section 9.4.

Community Z Tools + Z/EVES

Community Z Tools (CZT) provides a set of tools to develop formal specifications in Z notation. During this PhD research,[5] CZT (particularly CZT Eclipse) have been significantly improved. The major upgrades include support for Z/EVES proofs within Z specifications, linking with the Z/EVES theorem prover to check the proofs, close

[5]The work on CZT and Z/EVES theorem prover has been done together with Leo Freitas.

UI integration to facilitate the theorem proving process and many more improvements. The Z/EVES integration provides an API to communicate with the prover as well as tracking and displaying the prover results in an Eclipse-based IDE. The Z/EVES ProofProcess integration utilises this link with Z/EVES via CZT to capture the proofs. Section 10.3 provides further details on the improvements to CZT and the Z/EVES integration.

❦ ❦ ❦

The clear dependencies between the components aim to support other extensions in the future or different implementation decisions. Furthermore, the development of proof analysis, user interface, proof process data manipulation and other functionality does not depend on particular theorem provers and provides extension points for prover-specific data. A major part of the implementation code therefore can be shared between proof capture systems for different provers.

8.3.2 Software platform

The ProofProcess system is developed using Scala and Java programming languages on top of the Eclipse platform [Eclb]. The data structures are realised using the Eclipse Modeling Framework (EMF) [SBPM08]. Eclipse provides an excellent extensible application platform, which sees increasing adoption in industry and academia as the platform of choice for industry-grade tool development.

Eclipse platform

The software platform choice for developing a generic proof process capture framework is influenced by opportunities to adapt it to the different existing theorem provers. A platform with widespread usage can help avoid porting the system onto different platforms. The theorem provers and formal tools are developed using different languages, such as Lisp (ACL2, PVS, Z/EVES), Standard ML (Isabelle, HOL), Java (Rodin tool), etc. Nevertheless, most agree when it comes to providing user interfaces to the theorem provers. Emacs used to be the default platform, to which most of the theorem provers provided interfaces. For the next generation user interfaces, most provide Eclipse platform plug-ins as interface to the prover (e.g. Proof General Eclipse, Rodin tool, ACL2s, CZT Eclipse, alloy4eclipse, etc). The ProofProcess system is developed as a set of Eclipse plug-ins with the aim of making future integrations with the available proof assistants easier.

Eclipse provides an extensible application development platform. It makes developing quality applications easier. The plug-in model simplifies adding new components to supplement the existing functionality. Developers can focus on implementing new functionality or new user interface components rather than figuring out how to make systems coexist. Furthermore, the platform provides convenient facilities for parallel processing, which are utilised to analyse the captured proof data. Finally, the Eclipse platform has a myriad of extensions for various domains. Reusable components or even whole frameworks built on top of Eclipse can be reused in new applications: e.g. the Eclipse Modeling Framework facilitates model-based software development.

EMF data structures

The proposed ProofProcess model (Chapter 4) is realised within the system using the Eclipse Modeling Framework (EMF) [SBPM08]. EMF provides code generation facilities, various serialisation and scalability options (from files to databases), as well as close integration with Eclipse platform UI components.

EMF is the core around which a large number of modelling projects have grown within the Eclipse community. Model-driven development effort is very large, with many industrial applications using EMF-based data.[6] This results in a good availability of various EMF-based tools and reusable components. Furthermore, EMF can be used standalone if a non-Eclipse implementation is required.

The existing support for model-based development as well as opportunities for future extensions using the standard EMF APIs allows focusing on the model itself during the research. However, Section 8.5 describes some of the data-related issues that still need solving when capturing and storing proof process data.

The ProofProcess model is translated to a corresponding EMF model that closely matches the proposed representation. EMF code generation is used to produce Java source code for these data structures (both the *interfaces* providing data encapsulation as well as the implementation classes). EMF supports modular design of data structures via model extensions: e.g. the prover-specific data models for Isabelle and Z/EVES extend the EMF model of the core proof process data structures.

Mapping all prover-specific data to custom EMF models is not feasible and native representations are used where available. For example, both Z/EVES and

[6] The Rodin toolset is an example in the area of formal development and proof: the Event-B specifications and other data structures are implemented using EMF.

Isabelle integrations provide XML serialisation facilities for their terms. The native term data structures are used together with the EMF data structures seamlessly.

Plug-in organisation

Eclipse platform and EMF provide good modularity support via extensible data models and clear Eclipse plug-in dependencies. Thus the ProofProcess system is implemented as a modular application (currently consisting of 26 plug-ins[7]) and achieves clear separation of concepts. The high-level dependencies between logical components of the ProofProcess systems are illustrated in Figure 8.8. The actual dependencies between the plug-ins are more fine-grained. Such dependencies allow for a very modular structure, where data structures are independent of the algorithms to manipulate them, which in turn do not depend on the user interface plug-ins that invoke them.

The plug-in names indicate the component they belong to and follow the general pattern of `org.ai4fm.proofprocess[.componentname][.type]`, where `componentname` has the following values for the main components:

- *omitted* for ProofProcess Core, i.e. simply `org.ai4fm.proofprocess[.type]`;
- `project` for Project ProofProcess;
- `isabelle` for Isabelle ProofProcess;
- `zeves` for Z/EVES ProofProcess.

The File History framework, because of its independent nature, skips the `proofprocess` designation for its plug-ins: `org.ai4fm.filehistory`.

Each component can consist of four plug-ins, which allow partitioning of the component's code according to its function and use: the data structures, generated code to manipulate them, core algorithms and user interface code. Such a partitioning into plug-ins allows achieving fine-grained dependencies that lead to a modular system implementation. The plug-in types are reflected in the plug-in names (the `[.type]` qualifier) and are the following:

- *omitted* (e.g. `org.ai4fm.proofprocess.isabelle`): pure data structures.

  The data structure plug-in contains the EMF models of components' data structures. The generated Java source code for each EMF model is also placed into the same plug-in. The model may extend some other EMF model (e.g.

[7] Source code is available at `http://github.com/andriusvelykis/proofprocess`.

ProofProcess Core) and thus would depend on the plug-in containing that model. In addition to the data structure source code, the plug-in contains serialisation facilities for custom EMF data types (e.g. the serialisation of Isabelle terms). Otherwise any other code to generate or manipulate the data structures goes into the `*.core` plug-in.

- `*.core`: source code for UI-independent functionality.

  The `core` plug-in contains source code for the main algorithms, definitions and general functionality that can involve generation and manipulation of the data structures, proof process analysis, proof parsing, theorem prover communication, etc. This plug-in is independent of (neither contains nor references) any user interface source code. Having UI-independent plug-ins aims to allow the reuse of their functionality in applications built using different UI toolkits. Furthermore, the analysis functionality contained within such plug-ins could, for example, be running on some remote server without any user interface capabilities (so-called *headless* execution). Plug-in dependencies include the associated data structures plug-in as well as data/core plug-ins from *component* dependencies.

- `*.edit`: editing and viewing support for the generated data structures.

  EMF models provide a number of code generation options. In addition to generating the classes of defined data structures (as described above), code generation can be used to create support classes for editing and viewing these data structures. This code is closely aligned to the Eclipse UI frameworks and therefore is placed in a separate `*.edit` plug-in.

- `*.ui`: user interface code.

  Separating the user interface from the core functionality gives clearer code structure as well as allowing possible code reuse in applications using different UI toolkits. The user interface code builds upon the UI frameworks available in the Eclipse platform. The actions defined at the UI level normally call the functionality defined in the `*.core` plug-ins and visualise the results.

Coding and release engineering

The source code of the prototype system mostly uses the Scala programming language [OSV08]. Scala is an object-functional programming language and has

strict type system as well as functional programming concepts well suited for data manipulation. Furthermore, it interfaces easily with Java libraries and code, such as the generated EMF data structure classes or the Eclipse components.

The build process of the ProofProcess system is streamlined and uses Maven project management and build tools. They enable nightly builds of the applications, which are released automatically. The applications, Isabelle ProofProcess or Z/EVES ProofProcess, can be used standalone or installed as add-on components using the standard Eclipse software installation and update mechanism. An online software update system is available for the ProofProcess plug-ins.

8.4 Data representation

The ProofProcess model proposes data points to describe an interactive proof process and capture the high-level insight. In the prototype implementation, these data structures are realised using EMF. This section lists the particulars of how the data is represented as well as some of the implementation decisions taken.

8.4.1 Core data structures

The core data structures follow the proposed abstract model (Chapter 4) quite closely. The abstract model is translated into a corresponding EMF model, from which actual Java classes and interfaces representing the data structures are generated. Figure 8.9 shows the UML class diagram of the core data structures.

The core data structures focus only on the main concepts of proof process capture, providing a generic kernel for the ProofProcess framework. Systems implementing this base framework need to provide their extensions to the data model. These would include the required data specialisations (e.g. prover-specific implementations of *Term*, *Trace*, etc.) as well as additional data representation and functionality, if needed.

The prototype ProofProcess system extends the base model in several directions (Figure 8.10). Some of the more generic[8] extensions, related to how the core data is embedded and accessed in the system, form the Project ProofProcess component. Figure 8.11 lists the data structure extensions provided by the component. The

[8] *Generic* here is taken to mean that the extensions are applicable to both Isabelle and Z/EVES prototype proof capture systems (and possibly some other future implementations). The prover integrations themselves provide further independent extensions, representing their own terms and proof steps (e.g. see Sections 9.2 and 10.2).

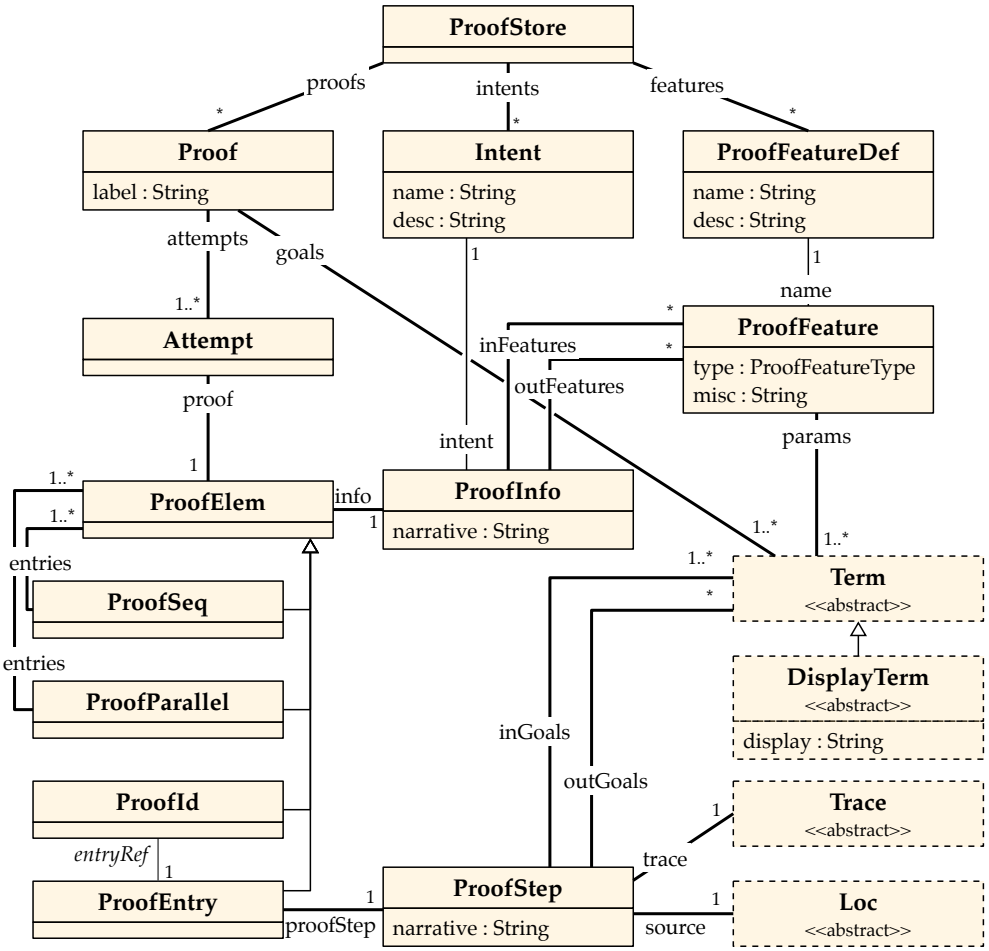


Figure 8.9: UML class diagram of ProofProcess Core data structures.

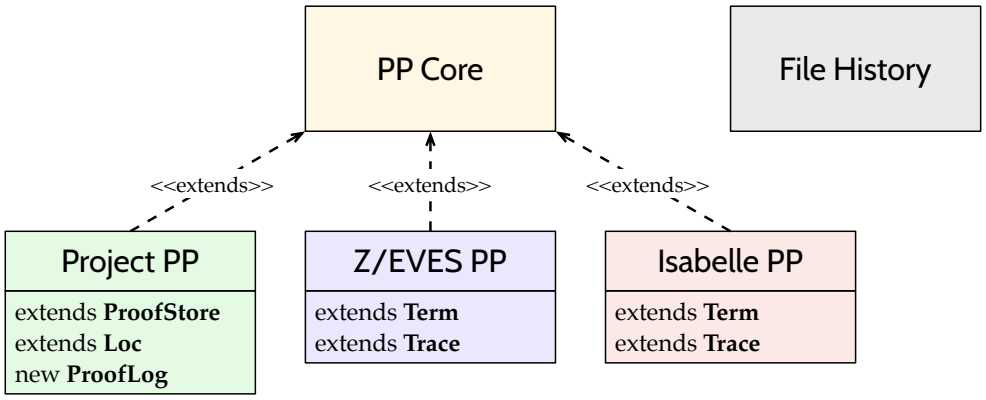


Figure 8.10: Relationships of ProofProcess (PP) data structure modules.

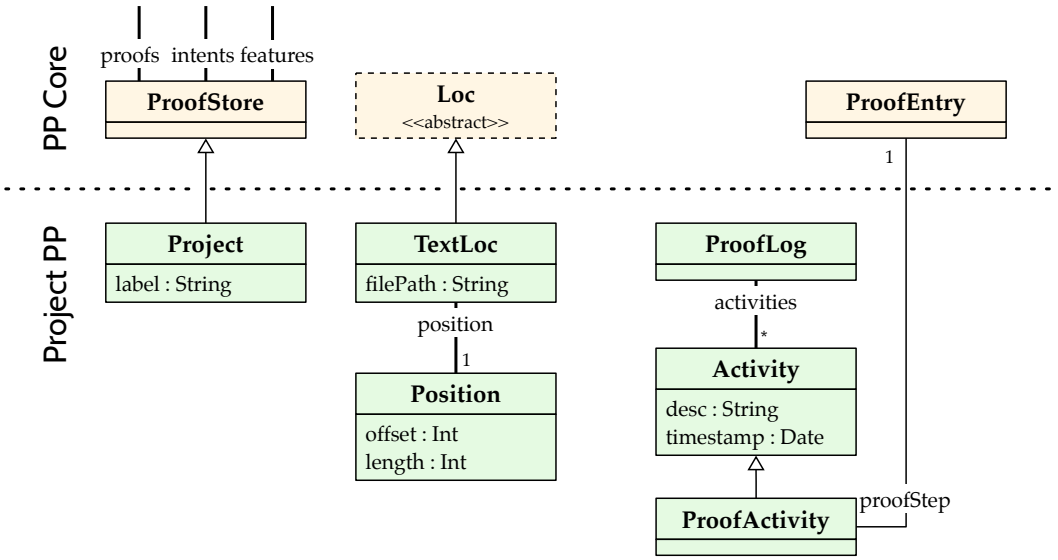


Figure 8.11: UML class diagram of Project ProofProcess data structures.

following sections discuss some of the extensions: the specialisation of *ProofStore* and the add-on of *ProofLog*.

8.4.2 Project proof store

The core data structures listed in Figure 8.9 form a hierarchical structure along the *containment* relationships (thick lines). A containment reference indicates that the referenced objects are contained within the parent one. The containment hierarchy controls the life-cycle of EMF data and is needed to store it. Data objects exist as long as their container does: so a proof tree cannot be stored without a proof attempt to which it belongs.

The *ProofStore* serves as the root container for the proof process data and records the top-level objects: *Proof*s, *Intent*s and *Feature*s. Section 4.5 describes the abstract model of *ProofStore* and argues that the collection of captured proof process data could be structured and partitioned in different ways, depending on the implementing system. The Project ProofProcess component provides one such implementation, by storing proof process data on a *per-project* basis.[9]

Projects in Eclipse IDE provide a logical way of structuring resources (source files, configuration, etc.) within the IDE. A project often corresponds to some folder in the file system that contains all associated files. Both Isabelle/Eclipse and

[9]During data analysis and manipulation, an abstract *ProofStore* interface is used, allowing different implementations to be swapped in without changing the analysis code.

Z/EVES Eclipse adhere to this Eclipse convention and use projects for structuring the formal specification files and proof scripts. The Project ProofProcess provides a simple approach to storing the proof process data captured from proofs: each project gets allocated a corresponding proof process data repository that is used for that project only. This approach is better suited as a prototype solution, as it makes inspecting the captured data easier and the analysis scope smaller. Furthermore, one of the goals of AI4FM is to work with *families* of proofs, which are expected to be found within the same formal development (i.e. the same project).

The usage of *per-project* proof store is recorded by providing an extension to the ProofProcess Core model. The *Project* data type extends the *ProofStore* (Figure 8.11). The relationships to *Proof*s, *Intent*s and *Feature*s are established within *ProofStore* (Figure 8.9, which provides the default implementation of using lists for each data collection). However, because the Java *interface* of *ProofStore* is used everywhere to access the data, other future extensions can disregard the default implementation and provide more sophisticated partitioning and management of proof stores.

Accessing proof store data

The proof store provides both storage and access to the proof process data. It is used when the proof process is being captured (e.g. to register new proof attempts) as well as to view or query the captured data. The current implementation accesses the stored data directly, by traversing the objects contained within the root *ProofStore* (in this case—the *Project* extension). For example, when a new proof command is captured, the data is traversed in reverse to find the latest matching *Attempt* within a matching *Proof* in a *ProofStore*. The current support for *viewing* the data mainly involves displaying the captured data in all its entirety: a full record of the *ProofStore* contents (Section 8.2.2). Developing a suitable proof query API (rather than exhaustive traversal) is left for future work (Section 13.3.4).

Because of the modular design of the ProofProcess system components, the root *ProofStore* is accessed from different places: e.g. data capture functionality is independent from data viewing, but both need to work on the same proof store. The Project ProofProcess component provides an API to access the project instance of the *ProofStore*. Furthermore, in the case of a new project, a new proof store for it is created on the first *write* of the data to avoid empty repositories. This functionality is conveniently hidden behind the API, simplifying the *ProofStore* usage. In the current implementation, *ProofStore*s are stored in individual repositories within an

embedded database on the user's local computer. Refer to Section 8.5 for details about data persistence.

8.4.3 Proof activity log

In addition to the *Project* proof store, the Project ProofProcess component also houses several other generic extensions to the core modules. One of them is the proof activity history, collected in a simple sequential log. Section 5.1.1 proposes the *ProofLog* structure as a basic way of recording the time dimension of the proof process. The log would consist of a sequential account of different activities related to the development of formal specification and proofs.

Project ProofProcess provides a rudimentary implementation of such a proof log. Figure 8.11 lists the proof logging classes provided by Project ProofProcess, namely the *ProofLog* and its list of *Activities*. A special activity representing a proof step is also provided: *ProofActivity*. It references the actual proof step (*ProofEntry*) in the proof attempt. Further extensions could provide other *Activities* to the proof log: e.g. to record specification changes, new definitions, etc.

The *ProofLog* is normally populated by the prover "wire-tapping" code, which can use it to register the events happening in the prover. Each activity is also timestamped, which may prove useful for calculating metrics about the proof process (see Section 13.4.3).

ProofLog is a *root* data structure that is constructed alongside the core proof process data. It is closely related to the *ProofStore*, whose activities it represents. Therefore certain implementation details, related to *ProofLog* storage and loading, are almost the same as with the *Project*/*ProofStore* described above. Project ProofProcess ensures a single *ProofLog* per project and provides an API to access and populate it. In fact, *ProofLog* and *ProofStore* are loaded in pair to ensure consistency of data between the two structures.

Proof history logging supplements the static view of the proof process defined in ProofProcess Core. It is designed as a separate component add-on to the core data structures. The aim is to minimise the core structures for implementations that do not require proof history logs. *ProofLog* is technically independent of the project proof store and could eventually be split off into a separate component, representing a smaller reusable module (for implementations that need the log but utilise a different *ProofStore* implementation). Currently it is hosted in Project ProofProcess for convenience and in order to avoid too many small modules.

8.5 Data persistence

Captured proof process data covers the whole duration of a formal proof development. It needs to persist between proof sessions and be available afterwards for querying. The captured data can stay useful for a long time: even after extracting proof strategies, it may be needed for further analysis, or, in case of proof maintenance (Section 13.4.2), may be accessed after a prolonged period of time. Furthermore, the data needs to "live" outside the actual proof development resources. The proof scripts can change, be discarded and cleaned up. The captured proof process data is an account of all this development and must be separated from the changing artefacts.

The choice to represent ProofProcess data structures using EMF conveniently presents several good solutions to data persistence. This section presents some of the options explored in the implementation of the prototype ProofProcess system as well as several issues that have been overcome.

8.5.1 Initial XML file storage

EMF includes a powerful framework to store its data structures [SBPM08, Chapter 15]. The framework provides a flexible API that allows different implementations of model storage. XML serialisation is the default implementation and is supported out of the box. This approach was selected as the initial persistence option for the captured proof process data within the ProofProcess framework.

The main "storage" item within EMF persistence is a *resource*. In this case, a resource represents a single XML file. Within this resource, EMF objects are stored. A resource can have one or more EMF objects as its *root* elements. When an EMF data object is saved (serialised to XML in this case), all of its contents are also saved. If there are any references to EMF objects that are not serialised, these dangling references are dropped, resulting in a possibly inconsistent model. The ProofProcess data structures are modelled in a way that all data is contained within some parent container, with *ProofStore* being the *root* container that is added to the persistence resource (see Section 8.4.2). The other *roots* of proof process data, *ProofLog* (Section 8.4.3) and *FileHistoryProject* (Section 8.7), also get allocated their own individual *resources* (XML files).

The API available to use XML serialisation for EMF models is simple and straightforward. To implement a *per-project* proof store (see Section 8.4.2), a special

XML file is allocated within each Eclipse project. The file is used to store and load proof process data. The XML representations of EMF data structures are derived from the defined EMF model, thus no additional storage schemes need to be devised for the prototype ProofProcess system.

Unfortunately, due to the large amount of data and its throughput speed in capturing proof process, the *XML file* persistence had major issues within the prototype ProofProcess system. The biggest issue was system performance in loading the captured data from the files. To access the data, the whole file needs to be loaded into memory. This results in very long loading times and massive memory usage. In just a day of normal use, the captured data would grow to such a size that the application was crashing during file loading with an out-of-memory error, even when multiple gigabytes of memory were allocated to its use. Additionally, writing and reading very large proof process data files started causing intermittent data corruption errors. If a file became corrupted, the default EMF persistence implementation had issues recovering the data, since the XML encoding became invalid. This usually meant that the file was no longer usable and the data it contained was lost.

The issues plagued early users of the prototype ProofProcess system, with them either losing the data to corruption or having their applications crash when loading the large data files. The default *XML file* persistence solution was lacking for this use case, thus a different persistence framework was sought.

8.5.2 CDO with database

Storing large EMF data models is a common problem within the Eclipse community. A number of frameworks are available that utilise databases for saving and dynamically loading the data. Connected Data Objects (CDO) [Ecla] is one of the most established frameworks. It provides both repository and persistence functionality: i.e. CDO governs how the data is structured and stored within a database, as well as how it is loaded within the application. A large variety of back-end databases with different configurations are supported: the storage can be distributed, located in a remote central server or locally as an *embedded* database within the application.

The most beneficial CDO feature for capturing and inspecting proof processes is the *dynamic loading* of saved data. The storage procedures of the generated EMF data structures are changed to become *lazy*: i.e. references to other data objects are

only resolved upon request. When a data object is loaded, its references to other objects are not resolved—they are loaded dynamically upon access. For example, the initial loading of *ProofStore* constructs the data object itself, but its *Proofs* are loaded into memory only when requested at some later time. This approach requires minimal memory expenditure when the scope of data inspection is not exhaustive. For example, if some proof is not of interest to the analysis code, its contents (e.g. attempts) do not need to be loaded into memory. Furthermore, the dynamic loading is coupled with dynamic *unloading* of EMF data: i.e. data that is no longer used can be unloaded to reduce the memory consumption. The dynamic loading of EMF data significantly improved use of the prototype ProofProcess systems. The allocated memory requirements to capture and inspect proof process data are significantly lower and the application no longer crashes when dealing with large data sets. Even with smaller sizes of allocated memory, the data would be swapped in and out during access to avoid out-of-memory crashes.

Switching the persistence solution from XML files to CDO database within the prototype ProofProcess system was quite quick and straightforward. The data structures inherit the CDO functionality and APIs by simply introducing a new parent class to all of them. The actual data storage implementation uses the H2 database [H2]. The H2 database provides good integration with CDO and features a small and efficient Java-based engine, which makes it easy to use as an *embedded* system. The H2 engine is packaged within the ProofProcess system and provides an embedded database solution. This simplifies the usage of the system, since users do not need to establish or configure any database provision—they can just start using the system. Furthermore, an embedded system removes database communication overhead, especially when compared with a remote database solution, resulting in a fast solution to data storage. The physical storage for the embedded H2 database is allocated locally within the user's workspace and is shared between all projects. As mentioned in Section 8.4.2, each project gets allocated its own *repository* within this database, partitioning the proof process data *per-project*. Within the repository, the data organisation follows the standard EMF structure as with XML files: there can be several EMF *resources* in a repository, each of which can contain one or more EMF objects.

When using CDO, the regular data manipulation does not change. Data structures retain all EMF properties and APIs. Therefore no data capture or analysis code needs changing, as it operates directly on the data structures and is not concerned with persistence. However, when it comes to synchronising concurrent

data editing (e.g. manually manipulating the proof process data while new proof steps are being analysed), CDO provides a rich API that supports *transactions*. Within the ProofProcess system, each viewing or proof tracking component has its own transaction. When one of these commits new data, the notifications are propagated via CDO, ensuring data consistency and avoiding data races.

Furthermore, a transaction can be *rolled back* completely or to some save point. These are beneficial within user interface code: e.g. when the user edits the data, the changes are applied directly to the data structures; but if the user decides to cancel these changes, a transaction can be rolled back without worrying about a complex *undo* implementation. A user is able to see live how the changes affect the overall model and discard the changes if not satisfied.

The transactional nature of CDO persistence also helps with ensuring data consistency. CDO can verify and repair if data corruption occurs within the database: e.g. if the application was not closed gracefully, etc. This seems to solve the data corruption issue that plagued the proof process data saved in XML files.

8.5.3 Data evolution

Data evolution is concerned with changes in the data models themselves. Data models evolve to account for changes in the modelled domain, when new features are needed, etc. During development of the ProofProcess model and the prototype system, the evolution was mostly related to representing prover-specific data. The model was extended incrementally to accommodate new types of data that is being captured (e.g. the specific details of prover command configurations), to generalise (or specialise) existing representations of the captured data, etc. In a number of cases, the evolution was supplemental: i.e. it added something to the model without major changes to the existing relationships.

When the model changes, instances of that model (the already captured proof process data) need to be brought up to date with the new model. This is not a trivial exercise, even for minor changes in the model. The data models are used to configure the persistence of the modelled data. For example, loading an XML file with old instances of the data may fail because the de-serialisation process cannot find values of some attributes which have been newly added to the model. With XML files, the persistence framework can be configured to either ignore such missing values or provide default ones, thus giving a basic approach to "upgrading" the old data. For more complicated cases, there exist EMF model

evolution frameworks that support defining migration rules during data upgrade.

Unfortunately, the issue of model mismatch is more serious with CDO and its underlying database. The data models are used to define the storage schemes within CDO, i.e. how the data objects are mapped into the external database. This creates appropriate data tables in a relational database, where table colums represent data attributes, etc. Thus a change in an EMF model must be matched with corresponding changes to the underlying database schemes. This is a difficult general problem: no programmatic solutions currently exist for EMF and CDO.

To support evolution of the ProofProcess model, a simple solution has been implemented within the prototype ProofProcess system. It takes advantage of the *reflection* API available for EMF models and allows simple data migration between different versions of the model.

The ProofProcess system carries the full history of versioned ProofProcess EMF models. They can be used to load legacy data, even when the *current* version of the model is not compatible with the saved data. Loading the data in such a manner produces a *dynamic* instance of the model that is inspected and manipulated using the *reflection* API.[10] The *dynamic* EMF is a powerful part of the framework that allows construction and instantiation of the models during runtime. It enables manipulation of the data without having the generated classes.

The dynamic models enable the loading of legacy data. Before use, however, the data needs to be converted to instantiate the current version of the model and its generated classes. Using the reflection API, the matching classes in the current version of the model are found for the legacy data structure and the data is converted. Finally, the issue of upgrading the tables in the underlying database is solved by "brute-force": a fresh new repository is created with the updated model definitions and the old one is dropped altogether. The converted data is placed in the new repository and used henceforth until the next model evolution.

8.5.4 Data compression

Capturing an expert's proof process with all required details can result in large amounts of data. Therefore, implications of the storage space occupied by this data need to be considered. The data is normally serialised into XML form, which is quite verbose in representation. This is, however, not a significant issue for

[10] For example, instead of calling `Proof.getAttempts()`, the `EObject.eGet(attemptsRef)` is used by passing the resolved "attempts" attribute reference.

the proof process data structures themselves. They are quite small and occupy a reasonable amount of space when stored. The main issues with excess size were encountered when recording prover-specific terms: e.g. goal predicates, etc.

Both Isabelle ProofProcess and Z/EVES ProofProcess integrations encode terms using XML(-like) *native* representations. Isabelle provides encoding of its terms into YXML format [WB], whereas Z/EVES ProofProcess utilises CZT facilities to serialise terms into ZML (an XML markup for Z specifications) [UTS+03]. These save the ProofProcess system from re-implementing proof representation by utilising the available serialisation facilities provided by the theorem proving systems.

The terms, however, can be of a very large size, especially for industrial-style proofs involving complex data structures. When serialised with all their details into XML-based representation, such terms produce text strings of substantial sizes. Since the terms make up the majority of the captured proof data, they immensely inflate the overall size of the stored proof process data. Early users of the prototype ProofProcess system reported their proof process databases easily reaching sizes of 20GB and more.

Data size explosion is avoided by using ZLIB [DG96] compression during term serialisation. The algorithm is available as part of the Java libraries. The algorithm is used to compress the produced XML text representing the term into a byte array. The byte array is then converted to a text string using the *Base64* encoding [IET06], which is written to the database as the final representation of the term data. A reverse algorithm is followed to decompress this data.

Compressing the terms achieves size reductions of 90-95% for each term serialisation. The only downside is a small performance penalty when accessing or writing the native term representation. Because of the *dynamic* loading of CDO data, the representation is accessed only when needed, therefore reducing the number of times decompression is invoked.

8.6 Graph representation for proofs

The captured proofs are recorded using tree structures (Section 4.3). This representation provides a convenient way to introduce hierarchy and high-level insight in the proof, identifying higher-level proof steps that encapsulate the low-level ones. However, for certain analysis functionality that deals with relationships between low-level proof steps, the tree structure can be unwieldy. The difficulty

particularly arises for complex proof structures: e.g. merging of proof branches, Isabelle/Isar declarative proofs (Section 13.3.5), etc. To accommodate this, the ProofProcess system supports converting proof trees to and from *graph* representation.

8.6.1 Constructing proof graphs

The system supports directed acyclic graphs that consist of proof steps (*ProofEntry* data structures) as graph nodes. The graph edges between proof steps normally capture the proof structure: e.g. proof branches, dependencies between proof steps, etc. The high-level proof information is retained on the side: e.g. as a mapping for each *ProofEntry* to its high-level proof insight. The proof graph implementation uses *Graph for Scala* libraries [Emp].

Proof graphs are constructed during the initial proof process capture. For example, when capturing Isabelle proofs, the linear proof script is analysed and transformed into a basic graph structure by tracking goal changes (Section 6.2). When performing this analysis, it is clearer to use a graph structure to link proof steps according to how goals are "consumed". Afterwards, when the whole proof structure has been processed, the graph can be converted to a ProofProcess tree. Using graphs is even more beneficial when constructing a proof structure for an Isabelle/Isar proof, where assumption introductions are linked to their use later in the proof, producing very complex structures (see Section 13.3.5).

The bi-directional conversion between the Scala graph and a ProofProcess tree structure is available. The algorithms actually abstract the tree structure: e.g. they can be used with non-EMF implementations of proof process trees. Conversion from a graph to a tree uses *ProofParallel* elements to encode multiple outgoing edges from a proof step: each edge corresponds to a separate proof branch. Merge points (Section 4.3.7) are realised in the tree structure by leaving unfinished branches within a *ProofParallel* element. However, to ensure deterministic resolution of graph edges afterwards and to improve the efficiency of the conversion, *ProofId* elements (Section 4.3.7) are used to encode merge point graph edges. Each unfinished proof branch ends with a *ProofId* element that contains a reference to the *ProofEntry* step that "continues" the branch.

8.6.2 Attempt matching

Captured proof commands are evaluated against the existing proof attempts to identify whether they constitute a new attempt, a proof re-run or an extension of

a previous attempt (see Section 6.3). This analysis compares low-level proof steps while ignoring the order of proof branches and some other structural information.

The prototype implementation utilises *graph-subgraph isomorphism* checks to match proof attempts. For example, a new proof is captured as a proof graph G_{new}. Then it is compared to existing proof attempts for the proof. The comparison is done in reverse: i.e. by first comparing with the *latest* attempt.

During the comparison each proof attempt is converted into a corresponding proof graph $G_{k \in 1..n}$. If subgraph isomorphism is found (e.g. G_k is an isomorphic subgraph to G_{new}), the new attempt extends the existing one. This happens when additional proof steps are added to the current proof. In this case, the attempt is extended with new steps and care is taken to preserve all proof meta-information (e.g. marked proof features). If the subgraph isomorphism check fails, the same comparison is performed in another direction. For example, if G_{new} is an isomorphic subgraph to G_k, the new attempt has already been tried before and the user is re-running an old proof.

When subgraph isomorphism cannot be established, a new attempt is logged. This occurs when the user backtracks and attempts a new proof direction. The diverged attempt can happen at the very start of the proof (completely new attempt) or by backtracking several steps. In the latter case, there is a need to preserve existing proof meta-information already available on the older (backtracked) attempt. To match entries in the new attempt with the old one, a *maximum common subgraph isomorphism* problem must be solved.

The subgraph isomorphism resolution is implemented using a fast VF2 algorithm [CFSV04]. No existing implementations of the algorithm for Scala graphs were available, so a reusable implementation was done by the thesis author. However, no algorithms are currently available for Scala graphs to solve the maximum common subgraph isomorphism. Therefore preserving high-level information when deriving a new attempt is limited in the prototype implementation.

8.7 Recording file history

This section presents the File History framework for recording changes in text-based proof script files in a prover-aware manner. It is optimised for linear proof script files that are "submitted" to the prover for processing: the framework aims to efficiently capture a minimal set of proof script versions while preserving validity

of proof commands referencing particular versions. Linear proof scripts that are checked by the prover are standard among a number of popular theorem prover systems, including Isabelle and Z/EVES, whose integration is provided by the prototype implementation of the ProofProcess system.

The need to record the version history of proof scripts is part of the overall problem of capturing proof history (Chapter 5): the proof steps are linked with the contents of actual proof scripts, which have to be consistent when proofs change. Proof history preserves old versions of proof attempts, thus corresponding versions of proof scripts also need to be recorded. The File History framework serves this need by providing an approach to track proof script versions efficiently.

The framework is quite generic, as it can be used for recording the history of different text-based proof scripts from different theorem proving systems. This section describes the architecture of the framework as well as details of the important algorithms. The particulars of how the File History framework is realised within the prototype ProofProcess system implementation are highlighted. Furthermore, alternative solutions to proof script history capture are also explored.

8.7.1 File versions

The history of each file is recorded on a per-project basis. A *project* here represents some arbitrary collection of files:

$$FileHistoryProject :: files : FileId \xrightarrow{m} FileEntry$$

Implementations could employ different approaches to reference files within a *FileHistoryProject*. For example, the *FileId* could represent different *file paths* within the project, identifying files by their location. This approach is used by the prototype ProofProcess system implementations: file history is tracked for each file location with the separate file history mechanism for each Eclipse-project (see Section 8.7.5). Different requirements would warrant other choices: e.g. if renamed files are expected to preserve version history, path-based identification would not be a suitable choice.

Each file history is represented as a sequence of different file versions.

$$
\begin{array}{l}
\mathit{FileEntry} :: \mathit{versions} : \mathit{FileVersion}^* \\
\textbf{where} \\
\mathit{inv\text{-}FileEntry}(\mathit{mk\text{-}FileEntry}(\mathit{versions})) \triangleq \\
\quad \forall i, j \in \textbf{inds}\ \mathit{versions} \cdot \\
\quad\quad i < j \Rightarrow \mathit{versions}(i).\mathit{timestamp} \leq \mathit{versions}(j).\mathit{timestamp}
\end{array}
$$

The last version in the sequence always represents the *latest* file version: i.e. if the latest version is the same as some previous version but different from the current-last one, it becomes the new-last version in the sequence. Thus a full history of document evolution is recorded, not just a set of distinct versions. The invariant specifies this requirement for incremental version history.

Each file version records the file contents and the version's save-time.

$$
\begin{array}{lll}
\mathit{FileVersion} :: & \mathit{contents} & : \mathit{File} \\
& \mathit{timestamp} & : \mathit{Timestamp} \\
& \ldots &
\end{array}
$$

The way of storing the contents of a file version needs to suit each implementation. For example, the file system or a database solution can be used. The *File* reference would thus represent a file system path or a database access location, respectively. When using file system storage, different options can be chosen to prevent file name clashes: e.g. the prototype ProofProcess system uses universally unique identifier (UUID) [ISO08] names for each file version; other systems may choose incremental naming based on the original file name, etc.

The reference of file version *contents* is the same as that to be used in *TextLoc.file* when recording the location of a proof step source file (Section 5.1.2). The different *File* implementations mentioned above also apply to resolving the source of the proof step. The *File* abstraction acts as a link between the File History framework and the proof process capture. However, proof process capture does not have a direct dependence on the File History: the *File* reference can be provided in other ways; and the *FileHistoryProject* is just one possible independent implementation that can be used alongside the *ProofStore* and *ProofLog* structures.

8.7.2 File version synchronisation

During proof development, new commands are added to the proof script file, or the existing commands are changed and even deleted. To avoid capturing *every* minor edit of the file, the recording of file versions is done by tracking how the proof commands are "submitted" to the theorem proving system. A concept of *synchronisation point* is used to mark how much of the file has been processed successfully by the prover. The *syncPoint* is the end position of the last proof command that has been "submitted" to the prover and produced results.

$$
\begin{aligned}
&\mathit{FileVersion} :: \ldots \\
&\qquad \mathit{syncPoint} : \mathbb{N} \\
&\qquad \ldots
\end{aligned}
$$

The analysis and comparison of synchronisation points between the new and the last version allow the recording of only the important version changes and thus minimise the total number of file versions being saved. The following example illustrates the algorithm to determine which file version to save when the proof script is changed.

Consider a proof script version F_1 with a number of commands submitted up to a synchronisation point s_1. Now the user changes some parts of the file and submits the new commands or re-submits the changed ones. F_2 denotes the new file with the new synchronisation point s_2 after the last submitted commands.

Figure 8.12 outlines the algorithm that compares the old file version (F_1, s_1) with the "current" version (F_2, s_2) and either updates and uses the old version reference or creates a new version. The goal is to avoid creating new versions unnecessarily. However, old file version references must remain valid.

The algorithm checks whether the old version can be used without saving a new version: e.g. if the file contents have not actually changed, only the "submit" (synchronisation) point has. This happens when more commands are submitted, or a (part of) proof is re-submitted. In this case, the old version is reused with the (possibly) larger synchronisation point now saved (line 2). The consistency of old references is preserved because the version contents have not changed.

The synchronisation points come into use from line 3 in Figure 8.12. If the contents in both versions match up to the synchronisation point (i.e. the beginning

```
1 if F1 = F2                                          // if the same file
2 then use (F1, max(s1, s2))          // keep the last version with the later sync point
3 else if F1(s1) = F2(s1)                  // if the last sync point content matches
4     then if s2 ≤ s1                                 // if submitted within the last sync
5         then use (F1, s1)                                  // keep the last version
6         else replace (F2, s2)      // replace the last version with new contents
7     else new (F2, s2)                                    // create new version
```

Figure 8.12: File version synchronisation algorithm.

of the file is the same), then the references to the old file version are not in danger, because the file has changed after the point on which previous versions depend. If the new synchronisation point is smaller than the previous one, it indicates resubmission of some of the commands: the old version reference is kept (line 5). If the synchronisation point is larger, the contents can be replaced with the new file (line 6). However, all previous references are still valid with the new contents, as their text is the same. Finally, if the file is changed within the previously-submitted part, this signals backtracking and a new version is created (line 7).

The use of synchronisation points reduces the number of versions saved. It "squashes" different versions of a linear incremental development of the proof script: i.e. if the new commands are added and submitted sequentially, the last version is constantly replaced, resulting in a single *final* version with all the commands having been saved eventually. The incremental positions of proof step traces would remain valid within the contents of the final file version.

Synchronisation optimises the file history of linear proof scripts and records only significant changes (e.g. backtracking). The approach can also work for automated parallel proof script checking.[11] For example, proof branches may be processed independently and in parallel. This can cause later branches to be "submitted" and processed faster than ones typeset earlier. The linearity of proof version synchronisation points could be preserved by inspecting all commands of the proof: the synchronisation point would be the last command that has everything before it processed successfully, e.g. stop on error or "unfinished" results.

[11] Automated parallel proof checking is supported in the newest versions of Isabelle/jEdit and Isabelle/Eclipse proof assistants.

8.7.3 Improving synchronisation performance

Version synchronisation algorithm outlined in Figure 8.12 can require two comparisons of file contents: whether the whole file is the same (line 1) and whether the synchronisation area is the same (line 3). To compare contents, files must be loaded and then compared character-by-character in their entirety to ensure equality. File I/O operations are very slow compared to in-memory calculations, thus avoiding them where possible would improve performance significantly. Furthermore, the files may be of a large size and the performance of character-based comparison slows down in linear time for the file size. The "current" contents of the proof script file are normally already available in the memory via the proof assistant API, therefore loading issues affect the last saved file version.

To avoid loading the file version for comparisons, a *checksum* representing the contents can be calculated for the compared text. The *checksum* provides a fixed-size bit string for the text with a *practically-non-existent* probability of collision (i.e. when the same checksum is calculated for different inputs). Instead of comparing the whole text, only checksums are compared: if they are equal, the text is assumed to be equal as well. The checksums for both the whole file (*checksum* field) and for the synchronisation area (*syncChecksum*) are stored.

```
FileVersion :: ...
               checksum     : Checksum
               syncChecksum : Checksum
```

SHA-256 [NIS12] cryptographic hash function, for example, can efficiently calculate a checksum (hash) of the file contents with (practically) no collisions. Many implementations are available: a default Java implementation of SHA-256 has been used in the prototype implementation of the File History framework within the ProofProcess system. The checksums are calculated when first saving the new file versions and then compared during synchronisation. The benefit of using checksums is even more significant because the last version is preserved during re-submission of previous proofs and the same checksum is used many times.

8.7.4 Alternative solutions

The File History framework is a generic way to automatically record minimal file histories that capture all important changes of the proof scripts. It is used to provide distinct file versions to be referenced from proof step traces. However, other approaches could be employed by different implementations.

A file versioning system could be discarded altogether: e.g. if the proof trace locations are only used to highlight captured proof process within the current proof script. The proof steps would point to locations in the actual proof script files, but old proof step locations would be invalid. However, this would not support re-running proofs (Section 5.2).

A very basic solution would be to save the then-current proof script file copy for each proof step. This would avoid the necessity of synchronising different versions, but would pollute the storage with numerous versions of the proof script carrying minimal differences. Proofs could still be re-run to some extent, but following the version history would be difficult.

Finally, some general-purpose version control systems could be employed to provide file versioning. For example, the Git distributed version control system [Git] provides a fast source file management solution. Integration with such a system, however, may pose more difficulties. Furthermore, such systems are more tuned to manual version management, whereas support for "on-the-fly" versioning (e.g. when the user does not save the proof script but submits parts of it to the prover) may be more limited. If a version control system is used, the *File* references in proof trace would need to indicate both the file path and the revision to be queried within the system.

8.7.5 Implementing proof history tracking

The file versions recorded by the File History framework are used to link the captured proof commands with their locations in the corresponding proof scripts. The prototype implementation currently utilises the file system to store the different versions of files for each project. Each file version is assigned a random UUID name: e.g. a new version of a file `ProcessTableProofs.zed` is saved as `f835f495-7d53-4d56-89fa-553d2d772e54` (no extension). This provides an efficient structureless way of storing new versions of various files. Paths to the particular file versions are stored within a corresponding *TextLoc*, linking proof steps with the actual proof script.

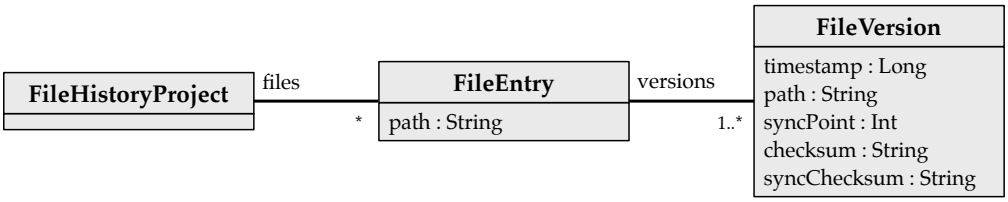


Figure 8.13: UML class diagram of File History data structures.

Implementation of the high-level *proof history* tracking closely follows the proposed architecture (Chapter 5). The prototype system employs a basic implementation of a *ProofLog* that records *ProofActivity* entries for each proof step (see Section 8.4.3). A separate *ProofLog* is established for each Eclipse-project, mimicking the organisation of the proof scripts into such projects.

The proposed architecture of the File History framework is also replicated in the implementation. Figure 8.13 lists the data structures provided by the prototype implementation. A *FileHistoryProject* structure is established for each Eclipse-project to track all formal development files within the project. The files are identified by their paths (locations); each location tracks a list of file versions. This structure provides a link between a file path, which can encode some folder structure in the project, and the UUID-named file version that is stored in a flat structure.

File history is tracked for every submitted proof command. During standard interactive proof, the commands are written one-by-one and submitted to the prover. At each point, the proof script is compared with the proof history and a new (or updated) version of the proof script may be saved. If a large number of proof commands are submitted at once (e.g. re-submitting the proof, or running in a "batch" mode), the behaviour of the prototype system remains the same for simplicity: the proof script file is synchronised for every submitted proof command. Optimisations could be done to accommodate a "batch" mode, but the file history synchronisation is efficient (Section 8.7.3) and processing in the theorem prover can take a long time anyway compared to the file history checks.

8.7.6 Re-running proof history

Section 5.2 discusses how the captured proof history can be used to re-run the proofs, to "animate" the expert. However, the prototype ProofProcess system does not provide re-running functionality at the moment due to limited PhD research resources and time. Nevertheless, the proof history is collected alongside high-level

proof processes as described, therefore the data and the expert effort is available for future implementations.

One of the requirements for re-running full proof development identified in Section 5.2 is ensuring that correct versions of associated proof scripts are used. For example, if a particular version of a proof script depends on others, it is important to load their correct versions to preserve consistency of the overall development. The correct versions of associated files can be located via the *FileHistoryProject* structure. An easy approach would be to use all files within the *FileHistoryProject* that have versions with the same (or older) timestamp as the proof script in question. This would recreate the full project structure as it was during the original recording and should be resolved correctly by the prover, given that the file locations are adhered to. Since the imported files are normally processed before the current one, the correct versions of related files would have been captured by the File History framework.

8.8 Integrating with theorem provers

The ProofProcess framework is designed as a platform for building proof process capture functionality for a theorem proving system and is not a standalone application. Therefore, rather than adapting theorem provers to suit the ProofProcess system, one would reuse generic framework components and build a new proof capture add-on to the prover. An overview of tool components in Section 8.3 shows that the end-user applications are Isabelle ProofProcess and Z/EVES ProofProcess, which reuse and extend the framework functionality. This approach is similar, for example, to the Eclipse platform: developers of new IDEs can select and adapt relevant generic components as well as supplement them with custom functionality to create the final product.

Requirements for future ProofProcess integrations are discussed in various places within this thesis when describing the details of the ProofProcess framework architecture and implementation. Some key guidelines on how to build a new ProofProcess integration are reiterated here:

- Provide prover-specific implementations of *Term* and *ProofTrace* data structures, encapsulating details about prover terms and commands (proof steps), respectively. These abstract data structures serve as extension points of the

core ProofProcess model. The core data structures and their manipulation functionality can be reused within integrations.

- Provide prover-specific implementation of *Loc* (locations of proof commands within proof scripts). If the prover uses text-based representation, the File History framework can be reused (Section 8.7).
- Select to use the existing *per-project* proof store (see Section 8.4.2) or develop an alternative proof store representation (as discussed in Section 4.5). The same considerations would be used for proof activity logging.
- Select to reuse the underlying database functionality: a convenient API is available for instantiating the embedded database. If prover terms have large representations, consider providing a compressed serialisation option when saving to the database.
- Implement prover "wire-tap": track and record user and prover activities. It is convenient to implement this using an *observer* pattern on the prover: e.g. when the prover processes new commands, record them together with the corresponding results. Parsing the prover data into the ProofProcess data structures as well as the *Term* and *ProofTrace* implementations is needed. All this functionality is specific to each prover.
- If applicable, reuse the functionality to infer proof process structure and recognise proof attempts (Sections 6.2 and 6.3) as well as other analysis and proof inferring functionality within the ProofProcess framework. In other cases, custom implementations would be used: e.g. the Z/EVES ProofProcess integration employs prover-specific proof structure analysis, based on proof branch indices reported by the prover (see Section 10.2.2).
- If building upon the Eclipse platform, generic user interface components can be reused. They work with the core data structures and thus can be used directly. Prover-specific extensions need to be provided for manipulation of terms and proof traces: e.g. to break down a term into sub-terms (see Section 8.2.3).

8.8.1 Prover requirements

The ProofProcess framework aims to accommodate different theorem provers by providing generic, reusable components and functionality. The framework uses

a *progressive enhancement* approach to supported functionality: if more and better data is available from the prover, the analysis and inferring capabilities of the framework capture better proof process information.

To support good proof process capture, theorem provers should have the following functionality (the details about these requirements are discussed in corresponding architecture and implementation sections of this thesis):

- Provide details about entered proof commands, including mapping them to the proof script, capturing command parameters, etc. Furthermore, help with handling the expressiveness of the proof language (syntactic sugar, alternative ways of stating the same commands, proof script styling, etc.).
- If applicable, capture other proof assistant user interactions such as querying for theorems, inspecting the proof context, other user interface actions, etc.
- Provide details about proof command results, such as the full list of goals. Isabelle truncates the list of goals by default if it is too large.
- Track detailed goal changes in order to avoid non-determinism in inspecting the effects of proof commands. This is relevant when proof commands can affect multiple goals.
- Provide a canonical representation for prover terms (e.g. goals and their subterms, etc.); or provide a matcher for different representations to check when two terms are equivalent.
- Retrieve details about prover terms: variable types, function definitions, etc.
- Inspect and query the proof context, prover configuration, available lemmas and definitions for each proof command.
- Track used lemmas, definitions and concepts within proof commands.
- Provide basic manipulation functionality for the captured terms: e.g. extract sub-terms, perform pattern matching or unification, etc.
- If needed for advanced analysis, provide basic automated prover functionality at any proof command (e.g. some proof feature implementations may utilise the prover).
- Support executing queries and possibly "drive" the prover in the background without affecting the user doing the interactive proof.

The described functionality is often available within the theorem prover, but an appropriate API is needed to access it from the proof process capture components. For example, one could perform most of these functions in Isabelle/ML, however the functionality is not available in the Isabelle/Scala API that is used in Isabelle ProofProcess (see Section 9.1.2).

CHAPTER 9

Integration with Isabelle

Integrating the generic ProofProcess framework with the Isabelle [NPW02] theorem prover enables the capture of a user's interactive proof when using this popular theorem proving system. The core framework (Chapter 8) is prover-independent, therefore extensions are needed to link it with theorem provers. The Isabelle ProofProcess extension provides support for prover-specific data when describing the interactive proof process: terms, proof command information, etc. Furthermore, it "wire-taps" the prover communication and tracks the user's interactive proof. The system automatically populates the low-level proof process details, performs further analysis such as inferring the proof structure, and so on.

Isabelle is a powerful and popular[1] LCF-style theorem prover. Isabelle/HOL, its most widespread instance, provides a higher-order logic theorem proving environment that is used for a variety of applications. Isabelle comes with a large theory library of formally verified mathematics that can be used in formal specifications and proofs. Furthermore, the *Archive of Formal Proofs* [KNP] contains a vast collection of formal verification developments spanning various domains. The proofs from the base library and the extended collection could be considered for capture and generalisation to high-level proof processes. However, the majority of these examples are quite mathematical and do not demonstrate the properties of industrial-style proofs (Section 2.1), which are the target of this research. Nevertheless, recent formal developments using Isabelle/HOL, such as the verification of the seL4 microkernel [KEH+09], provide examples of industrial-style proofs.

[1] Prior experience with Isabelle among the AI4FM project researchers has been a factor in determining the initial provers to use within the project.

Chapter 11 presents a case study of capturing and reusing an expert's high-level proof process for a family of lemmas. It is part of the formal development of a heap memory manager specification using Isabelle/HOL.

Furthermore, other tools are available for Isabelle for *replaying* proof strategies, which could be extracted from the captured proof process data. IsaPlanner [DF03] is a proof planning tool. Tinker [GKL14] is another tool from the AI4FM project that encodes and replays proof strategies using proof-strategy graphs[2] (Section 7.3).

The ProofProcess system integration with Isabelle and proof data tracking are built on Isabelle/Scala API. Section 9.1 discusses how the link between the prover and the ProofProcess system is implemented. The proof capture system is currently built on the standalone Isabelle/Eclipse proof assistant, which has been developed during this research. It is presented in Section 9.4. However, adding support for other Isabelle proof assistants such as Isabelle/jEdit [Wen12] would not be difficult.

9.1 Recording interactive proof

The Isabelle integration utilises the recent Isabelle/Scala layer to implement the tracking and capture of the interactive proof. The Isabelle/Scala layer is now part of the default interaction with the Isabelle prover. It is developed to facilitate new ways of constructing a proof interactively: e.g. as an asynchronously checked *proof document* instead of the classic read-eval-print interaction [Wen12].

Isabelle/Scala provides a strongly-typed API to access prover output and is being updated with every new Isabelle release. However, it focuses on supporting the functionality of graphical proof assistants rather than providing a universal API to the theorem prover. Section 9.1.2 discusses how using such an API affects the implementation of proof process capture. Before that, the next section reviews how Isabelle proof tracking is implemented in the Isabelle ProofProcess: particularly, how proofs are captured from an asynchronously processed proof document.

9.1.1 Capturing asynchronous proof

The *proof document* approach to Isabelle interaction provides powerful and user-friendly proof checking capabilities, particularly in the area of *parallel* processing of proof theories and proofs. The prover interaction is a move forward from the classic read-eval-print approach, allowing the user to develop a proof document

[2]The captured Isabelle proof process data can be exported to Tinker from the ProofProcess system.

that is being checked asynchronously rather than focusing on a single command. Furthermore, the interaction model allows parallel processing of the proofs to reap the benefits of multi-core processors available in modern computers. The opportunities for parallelism are plenty: the branches of the theory dependency graph can be processed in parallel; individual proofs can be delegated to separate processes (later proofs assume that their dependencies hold—the overall consistency is ensured only at the end); parts of structured Isabelle/Isar proof can be checked independently; etc [MW10]. Processing the eligible parts in parallel can speed up the overall evaluation significantly.[3]

From the implementation perspective, Isabelle/Scala provides a simple API of a *proof document model* for prover interaction, hiding the internal communication between the Isabelle/Scala and Isabelle/ML (the theorem prover implementation) layers [Wen12]. As the user edits the proof document within the proof assistant (e.g. Isabelle/Eclipse), the document model is updated accordingly, triggering the submission of the new proof commands to the prover. When the commands are processed, their results are returned to the Isabelle/Scala layer and populate the document model. The graphical proof assistant receives a notification to update the UI according to these results.

The ProofProcess system integration utilises the document model to "wire-tap" the prover communication. It registers to receive notifications when the commands are processed. As soon as a notification about changed commands is received, the system schedules the analysis of these commands as a low-priority task to avoid slowing down the actual proof processing. Each analysis request carries the current *snapshot* of the document model. Snapshot data is immutable, thus a delayed analysis is not affected by subsequent changes to the document model. The snapshot data is queried to record the proof process details as it carries proof command results, goal terms and other proof information.

When capturing proof process data, the asynchronous processing presents some issues in recognising what the expert is actually doing. First, the order of proof evaluation is frequently non-deterministic: i.e. a proof branch appearing *later* in the proof can finish evaluation before the results of an *earlier* calculation become available. Furthermore, the processing jumps over incomplete or erroneous proof commands and continues processing the remainder of the document (after a short delay), even if the subsequent proof commands belong to another proof. Such

[3]Matthews and Wenzel [MW10] report processing speedup of 5-6 times for sample Isabelle theories when using 8 processor cores.

processing produces a non-linear stream of data about the proof process, which does not reflect what the expert is actually doing.

To normalise the proof process capture, Isabelle prover events are filtered. The Isabelle/Scala API reports results of every processed command as they become available. This allows the prover IDE to render proof results "on the fly" within the proof document so that the user is aware of how the proof progresses. For performance reasons, the results are batched to some extent, but each batch does not necessarily represent a continuous sequence of commands. Upon receiving a notification about a new batch of results from processed proof commands, Isabelle ProofProcess performs the following filtering steps:

1. Identify all proofs containing the changed (reported) commands.

2. Collect full proof scripts (all proof commands) of each of these proofs. The proof scripts are collected from the user's proof document and may contain commands that are as yet unprocessed.

3. Select the minimal continuous sequence of proof steps with valid, processed commands: i.e. start from the beginning of the proof and select only the proof steps that have already been processed by Isabelle (their results are available) and are not erroneous.

4. Use the minimal valid proof attempt for proof process analysis.

Such an approach prevents the pollution of proof process data with interim proof results. For example, when a user is editing a proof command, it may be erroneous or fail to produce valid results. The asynchronous processing nevertheless continues processing the subsequent commands, which rarely advance the proof in the expected direction. Thus by taking the minimal valid sequence of commands, the interim results of the eager proof processing are filtered out.

Furthermore, the minimal sequence approach addresses the parallel processing issues. For example, if a command later in the proof is reported before the earlier ones are finished, these results will be ignored initially, since the earlier unfinished commands will break the minimal valid sequence of commands. However, when these are eventually processed, the whole proof will be analysed again, including the results of the previous batches. This will ensure that the proof is processed in its entirety and closely matches the expert's proof process, rather than the order of parallel processing.

Each minimal valid proof attempt is analysed by Isabelle ProofProcess to infer its proof structure. For Isabelle proofs, goal-change analysis is done to infer proof branching and merge points (Section 6.2). The current implementation of proof process capture mainly supports procedural, *apply-style* Isabelle proofs. Such proofs are used in industrial-style formal developments: e.g. the verification of the seL4 microkernel [Tru14].[4] However, Isabelle proofs can switch between the procedural and declarative approaches mid-proof, thus rudimentary support for processing declarative Isabelle/Isar proofs and even inferring the proof structure is available in the prototype implementation. Section 13.3.5 provides more details and establishes future work on providing full support for declarative proofs.

Inferring the proof structure of Isabelle proofs produces a low-level proof graph (Section 8.6) that is used to match against existing proof attempts in order to identify whether it is a new attempt, an extension of one, or just a re-run of a previous proof (Section 8.6.2). If the processed proof commands result in a new or extended proof attempt, the captured data is added to the proof process database and presented to the user, who can mark the high-level proof process insight.

9.1.2 Working with limited API

The Isabelle ProofProcess integration offers different parsing capabilities for Isabelle data, which depend on how much information is provided by the theorem proving system. The current API provided by Isabelle/Scala is quite limited[5] in regards to the actual proof query and manipulation. Isabelle/Scala has been developed first and foremost to support user interfaces for proof assistants, notably Isabelle/jEdit and afterwards Isabelle/Eclipse. Because of this focus, the API is concerned with *rendering* the proofs and their results, rather than supporting querying and manipulation of the actual proof search and construction. The latter functionality is mostly available via the Isabelle/ML APIs, which have a different interaction model for the user code. Figure 9.1 provides a rough overview of these Isabelle APIs.

The difference between Isabelle/ML and Isabelle/Scala APIs means that different results can be achieved depending on which APIs are used to query and record the proof process data. Isabelle/Scala allows easy interception of the prover

[4] During personal communication, the authors of the seL4 microkernel verification shared that it is infeasible to use declarative Isabelle/Isar proofs for formal development on such a scale. For example, the goals are too large to be useful in a declarative approach.

[5] Isabelle ProofProcess integration currently supports and requires Isabelle2013. The missing Isabelle/Scala API as discussed in this thesis may appear in future versions of Isabelle.

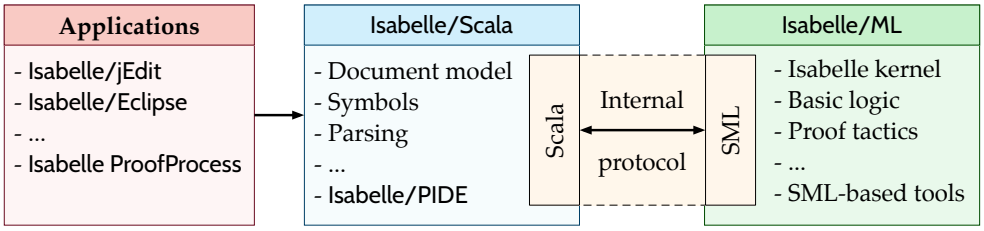


Figure 9.1: Isabelle APIs.

communication via the *observer* pattern. The data that is being sent to/from the prover can be parsed, structured and recorded. However, this data mostly represents instructions on how to render particular facets of the current proof. Significant details about the actual proof objects and what happens in the prover are omitted from the output. For example, the actual goal terms as used within the Isabelle prover get "printed" to a structure that is easy to render in a human-readable form. However, they no longer possess the original structure. While some of the information is still available within the *rendered* term (e.g. fully qualified names of functions or types of variables—see Section 9.2), the original data structures are lost in Isabelle/Scala APIs. These and other missing details are important to proof process analysis, therefore their absence limits the options of what can be done with the proof process recorded in this manner.

Isabelle/ML APIs, on the other hand, allow querying the *internal* representations of the proof process as well as accessing other details about the proof context. The data includes all necessary details to construct the proofs and hence provides a much richer account of what is happening in the prover.[6]

Unfortunately, because of the way Isabelle/Scala API works, retrieving the internal representations is not straightforward. The current API does not allow for querying of the proof information at arbitrary places within the proof document. The execution is *stateless* and asynchronous within the prover and only the proof output is stored within the prover IDE for *stateful* access. Therefore, to access certain data for each proof command (e.g. the internal representation of goals), this data needs to be part of the prover *output*.

Currently this is achieved within the ProofProcess system by *patching* the user's Isabelle installation. The changed code supplements the prover output with data of interest: e.g. when the goals are output for display, their internal representations

[6]Even using Isabelle/ML APIs is not enough to access all information in certain cases. For example, accessing *proof terms* (low-level λ-structure of proofs [BN00]) requires that they are explicitly enabled and uses HOL-Proofs as the base logic rather than just HOL.

are also sent via the *tracing* mechanism. This way the user does not see the additional output if tracing is turned off within the prover IDE, but the ProofProcess system is able to capture it. Extra development effort has gone into streamlining and automating the *patching* process within the ProofProcess system: the user only needs to confirm the action, then the selected Isabelle installation is patched and re-built automatically.

Such a workaround is not ideal as users do not appreciate changing the prover itself. The patches are only concerned with outputting data and do not introduce additional functionality. However, it would be preferable to avoid such intervention altogether. This is also a very brittle solution as patches need to be adjusted when new Isabelle versions are released.

A perfect solution would see improvements in the Isabelle/Scala API so that arbitrary queries could be executed at any point of the proof document. This would allow the ProofProcess system to query for term details "in the background" without patching the system. Simulating such interaction by inserting necessary commands directly into the proof document is complicated: the new commands invalidate the subsequent proof state and all subsequent proof commands need to be re-evaluated. This would severely slow down the proof processing.

The development of Isabelle/Eclipse, Isabelle ProofProcess and the usage of Isabelle APIs involved communication and collaboration with key Isabelle developers. The work has led to improved APIs in some Isabelle releases. Following discussions, APIs to query internal proof states are expected to be available in future versions of Isabelle. This would provide an "official" way to access internal data from within ProofProcess system without the need to patch the prover.

Isabelle/Scala APIs for third-party applications such as the ProofProcess system are still young and keep changing in each release. The development of Isabelle/Eclipse and the ProofProcess system required significant effort to be spent on chasing a moving target of new Isabelle releases.[7] Working with the developers on more mature APIs and designing more robust parsing algorithms is necessary to achieve better and easier integration with future versions of Isabelle.

The majority of Isabelle ProofProcess functionality is built using the Isabelle/Scala APIs. It follows the approach of the *least intrusive* capture of proof process. Patching the Isabelle installation is an intrusive operation, but provides richer data. To support both options, the ProofProcess system follows a "progressive enhancement"

[7] During this PhD research, a number of new Isabelle releases have been published: Isabelle2011→2011-1→2012→2013→2013-1→2013-2→2014.

approach: basic functionality is available using just the Isabelle/Scala APIs, but richer analysis can be done if Isabelle/ML patches are allowed.

An argument could be raised for developing the proof process capture system using the Standard ML language and integrating directly with the Isabelle/ML APIs. However, building a *generic* proof process capture framework is one of the objectives of this research. The current integration solution has been chosen because of the need to also integrate with the Z/EVES theorem prover (Chapter 10), requirements for data persistence, low-priority processing, non-intrusive integration with the proof assistants and good UI to manipulate the captured data, as well as the author's experience with Eclipse and the Java programming language. Furthermore, the Isabelle/Scala API has emerged as the default Isabelle interface for proof assistants; it is being actively developed, thus the required functionality is expected to appear in the future.

9.2 Recording terms

Isabelle ProofProcess provides prover-specific representation for Isabelle's terms and proof commands as well as encoding this information within the proof process data during proof capture. Because of the limitations of the Isabelle/Scala API as discussed earlier, two representations for Isabelle's terms[8] are available:

- *MarkupTerm*: *rendered* terms;
- *IsaTerm*: *internal* representation as used by the Isabelle prover.

A class diagram listing the term implementations is presented in Figure 9.2.

Both term representations wrap the native data objects, provided via the Isabelle/Scala API. Furthermore, textual representation is recorded in both cases, to facilitate the human inspection of the data. For example, decrypting the internal structure with fully qualified names of functions can be difficult for humans, while reading $P \Longrightarrow Q$ is straightforward. The text representation of the term is recorded in the *display* field provided by the *DisplayTerm* class that both term implementations extend. The following sections explore each representation in more detail. For illustrations of the different representations, refer to Figures 9.3 and 9.4

[8] *Terms* in this context are taken to mean expressions, predicates, variables, their sub-terms and other constructs written in Isabelle's logic. This should not be confused with *proof terms* [BN00], which represent the low-level λ-structure of Isabelle proofs.

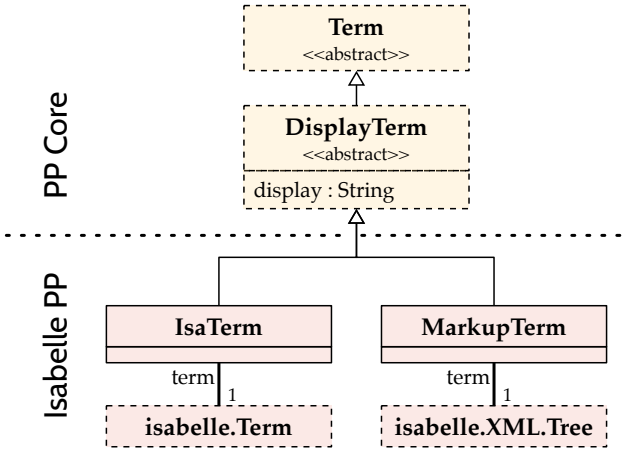


Figure 9.2: UML class diagram of Isabelle ProofProcess basic terms.

for *MarkupTerm* and *IsaTerm*, respectively. Each figure presents the corresponding representation of a simple lemma: $(x::nat) \in S \Longrightarrow S \neq \{\}$.

9.2.1 Rendered term (*MarkupTerm*)

The *rendered* term is the initial representation of Isabelle terms within the ProofProcess data structures. Their contents are parsed from the default prover communication using the Isabelle/Scala API. To capture Isabelle terms using this representation, no alterations to the theorem prover are needed. The information that comprises rendered terms is produced by the prover and is used by the proof assistant (e.g. Isabelle/Eclipse) to display the terms in rich, human-readable form. The Isabelle ProofProcess integration parses the terms from this prover output and uses them in recording the proof process.

Figure 9.3 provides an example of the *rendered* term content. The representation is mainly the term output string that is annotated with rendering and cross-referencing information. The information is structured into an XML document, which links the associated information with the corresponding parts of the term. The XML structure follows the printed output: e.g. the lemma $x \in S \Longrightarrow S \neq \{\}$ is encoded sequentially within the XML in Figure 9.3 at lines `10` (x), `14` ($\in$), `22` (S), `27` ($\Longrightarrow$), `36` (S), `40` ($\neq$), `43` ($\{\}$).

The details about variables include their typing information, whether they are fixed, etc. The types, function symbols and other constants also have their fully qualified names recorded: e.g. $\neq$ is defined as `not_equal` in the `HOL` theory (see `HOL.not_equal` at line `39` in Figure 9.3). The representation also carries some

```
<term>
  <block indent="0">
    <xml_elem xml_name="typing">
      <xml_body>
        <entity def_end_offset="854" def_file="~~/src/HOL/Nat.thy"
                def_line="36" def_offset="851"
                kind="type_name" name="Nat.nat" ref="383441">nat</entity>
      </xml_body>
      <fixed name="x">
        <free>x</free>
      </fixed>
    </xml_elem>
    <entity def_file="~~/src/HOL/Set.thy" kind="constant" name="Set.member"...>
      <delimiter>∈</delimiter>
    </entity>
    <xml_elem xml_name="typing">
      <xml_body>
        <entity kind="type_name" name="Nat.nat"...>nat</entity>
        <entity kind="type_name" name="Set.set"...>set</entity>
      </xml_body>
      <fixed name="S">
        <free>S</free>
      </fixed>
    </xml_elem>
  </block>
  <entity def_file="pure_thy.ML" kind="constant" name="==&gt;" ref="137">
    <delimiter>⟹</delimiter>
  </entity>
  <block indent="0">
    <xml_elem xml_name="typing">
      <xml_body>
        <entity kind="type_name" name="Nat.nat"...>nat</entity>
        <entity kind="type_name" name="Set.set"...>set</entity>
      </xml_body>
      <fixed name="S">
        <free>S</free>
      </fixed>
    </xml_elem>
    <entity def_file="~~/src/HOL/HOL.thy" kind="constant" name="HOL.not_equal"...>
      <delimiter>≠</delimiter>
    </entity>
    <entity def_file="~~/src/HOL/Set.thy" kind="constant" name="Set.empty"...>
      <delimiter>{}</delimiter>
    </entity>
  </block>
</term>
```

Figure 9.3: *MarkupTerm* representation of $(x::nat) \in S \Longrightarrow S \neq \{\}$.
Note: the XML presented here has been cleaned up for readability and space saving reasons. The removed elements are `<block>`, `<break>` and other similar layout elements. Attributes within some `<entity>` elements have also been trimmed.

layout information: e.g. which parts of the term are to be grouped when the terms span multiple lines to display. This information is represented using `<block>` and `<break>` tags, which have been mostly trimmed from Figure 9.3.

The *rendered* term does carry some important details about Isabelle terms, particularly the types of variables and the fully qualified names of function symbols and other constants. This representation is easily accessible by "wire-tapping" the communication between the prover and its IDE.

Unfortunately, the actual structure of the term is not captured by the *rendered* representation, making it difficult to decompose such terms (e.g. to separate assumptions and goals) or to perform even the simplest manipulations. Any such operations would require parsing the text and inferring the term structure. The actual representation of the term (i.e. the one used by *IsaTerm*) could be queried for the *MarkupTerm* from the prover. The rendered term could be "parsed" *within* the prover using the associated proof context. However, doing this with the current Isabelle/Scala API[9] would require workarounds such as temporarily modifying the current proof document to access the proof context at the desired location.

Goal-change analysis that is used to infer Isabelle proof structure (Section 6.2) can utilise the *MarkupTerm* representation in the majority of cases. Terms can be compared for equality, because the rendered representations of equal terms are equal themselves. They can be used to follow goal changes across proof steps. This allows inferring the basic proof structure within the procedural, *apply-style* Isabelle proofs. However, using them to infer the structure of a *declarative* Isabelle/Isar proof is more difficult, as simple term manipulation is required there (see Section 9.2.3).

The *MarkupTerm* representation is used to display goals within the prover output and to annotate processed proof commands. For example, when a proof command is processed by the prover, the command results (e.g. outstanding goals) are displayed as the prover "output". These goals are rendered using the described XML structure and can be captured as *MarkupTerm*s by the ProofProcess system. Furthermore, the processed command and its parameters are also "marked up": i.e. the terms are enriched with type and cross-reference details as explained above. This allows capture of *MarkupTerm* representation for proof command details.

However, since *MarkupTerm*s are captured from the "display" information, they are susceptible to user configuration on how results are visualised. For example, by default Isabelle limits the number of goals presented to the user to ten. Therefore,

[9] As of Isabelle2013.

if more goals are produced as the result of a proof command, some of them are trimmed and are not captured by the ProofProcess system if running without in-depth prover querying (i.e. without the *IsaTerm* parsing).

9.2.2 Internal term (*IsaTerm*)

The *internal* representation of Isabelle terms supports logic operations and can be manipulated more easily. Furthermore, it is more robust: i.e. this representation is not affected by rendering changes between different versions of the Isabelle prover. However, retrieving this representation for goal and command parameter terms is more difficult and requires adjustment of the Isabelle installation.

Figure 9.4 provides an example of the *internal* term representation for the same simple lemma as with the *rendered* term (cf. Figure 9.3). This representation is used to denote terms within the base Isabelle/Pure framework. The Pure logic encodes a simply typed λ-calculus and provides logic operations to manipulate such terms. Other Isabelle logics build upon the base Isabelle/Pure framework. The base term encoding and supporting implementation of logic operations as well as other functionality are available within the Isabelle/ML API. The Isabelle/Scala API mirrors the term encoding,[10] however the libraries providing logic operations or other functionality are not yet available within Isabelle/Scala.[11]

Like the *rendered* representation presented above, the *internal* representation records fully qualified names of the included constants and provides the full type information. However, its structure is superior in representing sub-terms: i.e. it captures function applications; expands the abbreviations to provide a normalised representation (e.g. $\neq$ is encoded as `HOL.Not` wrapping a `HOL.eq`); etc. The structure makes it easy to extract sub-terms: e.g. by taking the contents of the function application (`App`) arguments. Sub-term identification is needed for ProofProcess analysis: e.g. for sub-term selection in proof features, etc.

To capture *IsaTerm* representations, the Isabelle system needs to be patched (see Section 9.1.2). For goal terms, the Isabelle/ML code that outputs the goal after processing the proof command is supplemented: for every result goal it also outputs its *internal* term representation via the *tracing* mechanism. When the output is parsed by Isabelle ProofProcess during analysis, this internal representation is used to record the proof step goals.

[10] Defined in `isabelle.Term` class in Isabelle/Scala.
[11] As of Isabelle2013.

```
App(
  App(
    Const(==>,
          Type(fun, [Type(prop),
                     Type(fun, [Type(prop), Type(prop)])])),
    App(
      Const(HOL.Trueprop,
            Type(fun, [Type(HOL.bool), Type(prop)])),
      App(
        App(
          Const(Set.member,
                Type(fun, [Type(Nat.nat),
                           Type(fun, [Type(Set.set, [Type(Nat.nat)]),
                                      Type(HOL.bool)])])),
          Free(x,
               Type(Nat.nat))),
        Free(S,
             Type(Set.set, [Type(Nat.nat)])))))),
  App(
    Const(HOL.Trueprop,
          Type(fun, [Type(HOL.bool), Type(prop)])),
    App(
      Const(HOL.Not,
            Type(fun, [Type(HOL.bool), Type(HOL.bool)])),
      App(
        App(
          Const(HOL.eq,
                Type(fun, [Type(Set.set, [Type(Nat.nat)]),
                           Type(fun, [Type(Set.set, [Type(Nat.nat)]),
                                      Type(HOL.bool)])])),
          Free(S,
               Type(Set.set, [Type(Nat.nat)]))),
        Const(Orderings.bot_class.bot,
              Type(Set.set, [Type(Nat.nat)])))))))
```

Figure 9.4: *IsaTerm* representation of $(x::nat) \in S \Longrightarrow S \neq \{\}$.
Note: the structure in the example is a print-out of Scala datatypes. To improve readability, the `List(elem1, elem2)` structure is represented as `[elem1, elem2]`, or omitted altogether if empty: e.g. `Type(Nat.nat)` is actually `Type(Nat.nat, List())`.

❦ ❦ ❦

The duality of term representation within Isabelle ProofProcess appears because of restrictions within the Isabelle APIs. Currently, a "progressive enhancement" approach is used during proof capture, where functionality is improved if better data is available. For example, *MarkupTerm*s have been used during initial development and are still the default term representation. However, capturing *IsaTerm*s enhances the functionality and enables better analysis techniques. Comparison and analysis involving mixed data (both *MarkupTerm* and *IsaTerm* data) is not possible at the moment. Eventually, if the Isabelle APIs provide the necessary functionality, *IsaTerm*s will become the default representation of Isabelle terms and will not require patching the prover.

9.2.3 Normalising goals in declarative proof

Formal proofs in Isabelle can be very expressive, particularly when constructed using the Isabelle/Isar [Wen02] proof environment. It enables authoring of declarative and structured proofs: i.e. where assumptions and goals used in the proof are declared explicitly. This approach is very well suited for mathematical proofs and produces human-readable proof documents. Furthermore, Isabelle/Isar is more suitable for a *forward* proof style, where facts are posited and then used to prove subsequent ones. The procedural style is more aligned with the *backward* proof style, where proof tactics transform the open goal and produce sub-goals.

The current ProofProcess system focuses on capturing the procedural proofs. However, the flexibility of declarative proof and opportunities to control the proof context tempt users to switch to Isabelle/Isar proofs in certain cases, even if the other proof is done in procedural, *apply-style*. To avoid system failure when such a proof style is encountered (sometimes mid-proof), rudimentary support for recording Isabelle/Isar proofs is implemented within the prototype system. Refer to Section 13.3.5 for more details.

A particular issue encountered when parsing Isabelle/Isar proof goals is the need to normalise the goal representations. The explicit declaration of assumptions within the Isar declarative proofs allows them to be chosen manually to build up the proof context within proof steps. These assumptions are used to prove the goal, but are presented to the user separately. An example of prover output for a declarative proof step in Isabelle/HOL is listed in Figure 9.5(a). The assumptions

```
proof (prove): step 18

using this:
  ss ≠ X
  ∀s∈p. ∀t∈p. s = t ∨ s ∩ t = {}
  ss ∈ p
  X ∈ p

goal (1 subgoal):
 1. ss ∩ X = {}
```

(a) Declarative Isabelle/Isar proof step.

```
ss ≠ X ⟹
∀s∈p. ∀t∈p. s = t ∨ s ∩ t = {} ⟹
ss ∈ p ⟹
X ∈ p ⟹
ss ∩ X = {}
```

(b) Full goal.

Figure 9.5: Sample output of a declarative Isabelle/Isar proof step.

(listed under *using this:*) have been declared (and possibly proved) previously. They are selected explicitly to prove the listed goal `ss ∩ X = {}`. Figure 9.5(b) displays the exact same goal but with assumptions embedded within.

When recording proof process results of a declarative Isabelle/Isar proof, the captured goal terms are normalised. Furthermore, the explicit assumptions are recorded. The assumption selection (and especially their declaration) is an important feature of the expert's proof process: e.g. when replaying such strategy, it may be important to establish and select *similar* assumptions.

Within declarative proof steps as illustrated by Figure 9.5, the assumptions and goals can be separated. This means that the ProofProcess capture records them as separate terms, using either *MarkupTerm* or *IsaTerm* representations for each one. To normalise the goal terms as well as to mark the assumptions, Isabelle ProofProcess uses additional term representations: *JudgementTerm* and *AssumptionTerm*, respectively. They wrap the captured base term representations. Figure 9.6 provides a class diagram depicting these data structures.

The *JudgementTerm* structure encodes the goal as a list of assumptions with a single goal term. During normalisation, the explicitly declared assumptions are added to the ones already existing within the goal. Any duplicate assumptions are trimmed. The *AssumptionTerm* is used to wrap the actual terms of the explicitly declared assumptions. Both *JudgementTerm* and *AssumptionTerm* are used to record goal terms within *in/outGoals* fields of the *ProofStep* structure (Section 4.3.3). The recording of assumptions allows the inference of a richer proof structure, namely the dependency of each proof step on previous steps that declare the required assumptions. Refer to Section 13.3.5 for more information on how proof structure of such proofs can be captured, analysed and represented.

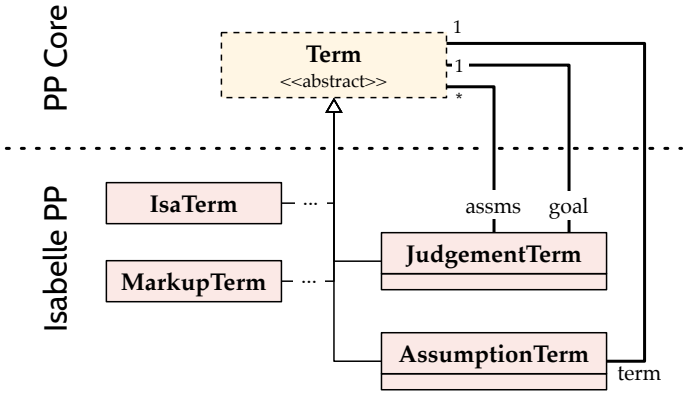


Figure 9.6: UML class diagram of Isabelle ProofProcess goal terms.

9.3 Recording proof steps

Capturing the Isabelle proof step chosen by the user requires the parsing of the details entered in the proof document. The proof command can consist of the chosen proof method (e.g. tactic) as well as its configuration, e.g:

```
apply (induct_tac k rule: nat_less_induct)
```

Here, the user has chosen to use the *induction* tactic `HOL.induct_tac` on some variable `k` using the induction rule `Nat.nat_less_induct`. The proof commands can be of varying complexity and different tactics utilise different configuration options. To capture the user's proof step, it is important to parse and record the proof command itself as well as all of its configuration, which normally contains the important parts of the selected proof strategy.

When a proof command is processed by the prover, it gets annotated with additional details: e.g. the referenced objects are identified with fully qualified names, the parameter terms are given structure (see term parsing above), certain parts of the proof command are marked up as well. This information is normally available via the Isabelle/Scala API to enhance the proof command rendering within the prover IDE. Isabelle ProofProcess uses it to parse the command details.

The data structures used to represent Isabelle proof traces and capture proof command details are listed in Figure 9.7. The large number of proof tactics available within Isabelle with different configuration options have led to using a generic approach to representing a proof command: the *NamedTermTree* structure. The structure represents a hierarchy of named lists of terms. This allows the encoding

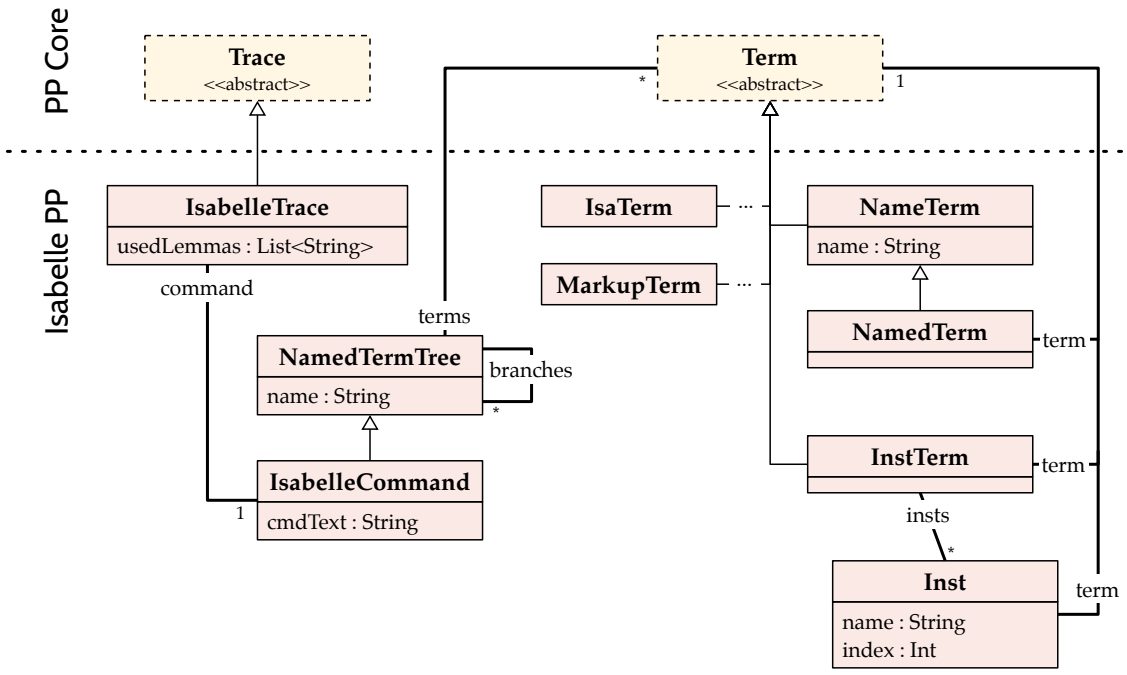


Figure 9.7: UML class diagram of Isabelle ProofProcess proof command trace.

```
NamedTermTree(apply, [], [
  NamedTermTree(HOL.induct_tac, [IsaTerm(k)], [
    NamedTermTree(rule, [NameTerm(Nat.nat_less_induct)], [])])])
```

(a) `apply (induct_tac k rule: nat_less_induct)`

```
NamedTermTree(by, [], [
  NamedTermTree(HOL.simp, [], [
    NamedTermTree(add, [NameTerm(Big_Operators.setsum_Un_disjoint)], []),
    NamedTermTree(del, [NameTerm(Set.Un_Diff_cancel)], [])])])
```

(b) `by (simp add: setsum_Un_disjoint del: Un_Diff_cancel)`

Figure 9.8: Proof command encoding using *NamedTermTree*(*name*, *terms*, *branches*) structures. Lists are represented using square brackets `[]`.

of different proof command configurations within the same structure rather than creating explicit data types for each proof method.

The proof command itself is a *NamedTermTree*, where each nested branch represents a particular proof command. The proof command, in turn, records the named command parameter lists as its branches, and so on. Figure 9.8 lists examples of such proof command representations.

Furthermore, to represent certain proof command arguments, new term data structures are defined within Isabelle ProofProcess (see Figure 9.7):

- *NameTerm*: denotes a named lemma or proof fact within a command. For example, when lemmas are explicitly indicated within a proof command, their names are recorded as *NameTerm*s (see the encoding of induction lemma `Nat.nat_less_induct` in Figure 9.8).

- *NamedTerm*: records introduction of named facts in to the proof context in Isabelle/Isar declarative proofs. For example, an assumption command `assume conj: "A ∧ B"` introduces the named fact that can be referenced in the proof later. A *NamedTerm* records both the introduced term (e.g. "`A ∧ B`" here as *MarkupTerm* or *IsaTerm*) as well as the name given to it (e.g. '`conj`').

- *InstTerm*: records term instantiations:[12] e.g. in rule application `apply (rule_tac x="a - b" in exI).`

Some of the data structures presented here are still in development and may change eventually when the ProofProcess functionality evolves: e.g. instead of capturing just the lemma name (*NameTerm*), its full representation should be recorded to account for possible definition changes (see Section 4.6.2).

❦ ❦ ❦

Parsing of Isabelle terms and proof commands provides basic capture of low-level proof process data with opportunities for future analysis. Implementing more advanced proof process analysis functionality, however, requires use of the prover functionality in many cases. The limitations of the current Isabelle/Scala APIs hinder such implementations.

Nevertheless, the core ProofProcess functionality is supported by Isabelle ProofProcess: the expert's interactive proof is recorded; the proof structure is inferred automatically and new attempts are recognised; afterwards the expert can mark the high-level insight on the captured data by indicating high-level proof steps, proof intent and marking the appropriate proof features.

[12] To access the instantiation information, Isabelle/ML APIs are used and require patching of the Isabelle system.

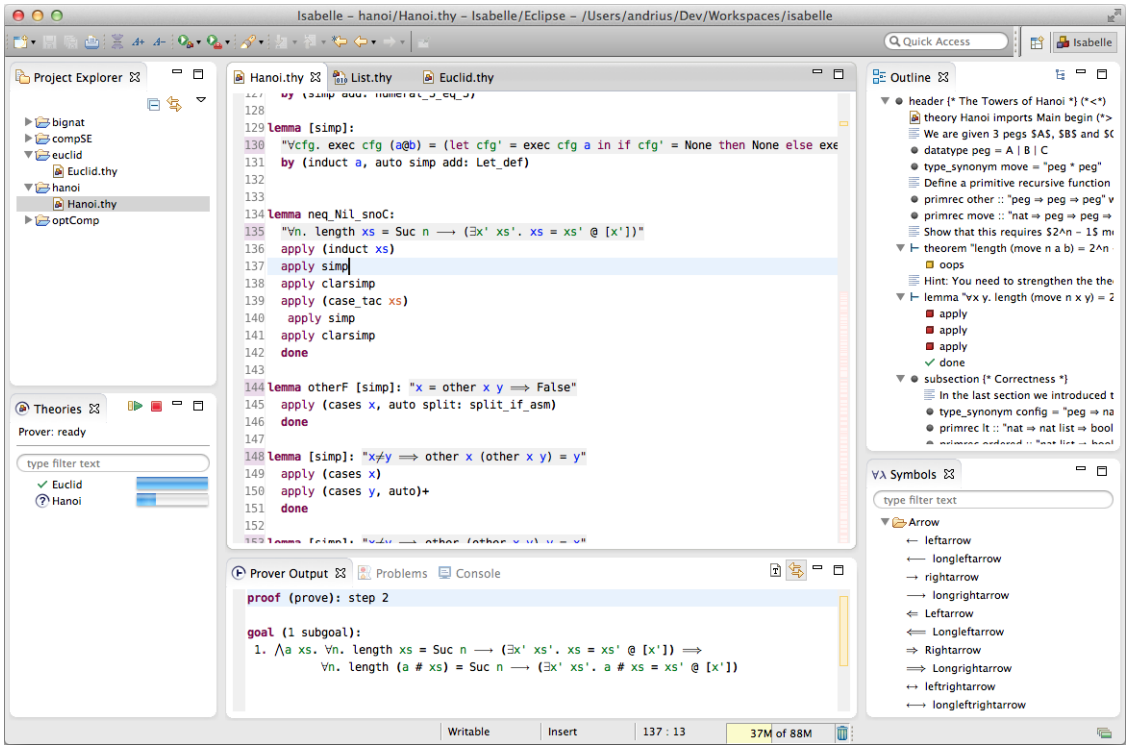


Figure 9.9: Screenshot of Isabelle/Eclipse.

9.4 Isabelle/Eclipse prover IDE

Isabelle/Eclipse[13] is a prover IDE for the Isabelle theorem prover. A screenshot of the system is presented in Figure 9.9. Isabelle/Eclipse has been developed to support the functionality and usability of the ProofProcess system. It is part of the infrastructure of the ProofProcess system, and although it does not directly solve the problem of capturing the proof process, part of the PhD research time has been spent on developing such infrastructure.[14]

The ProofProcess system works as an add-on to proof assistants. It "wire-taps" the communication between the user actions and the theorem proving system. Isabelle/Eclipse provides the said proof assistant functionality for doing interactive proof, which is then captured by the ProofProcess system. Alternative proof assistant implementations exist for Isabelle, however, but have not been selected:

- Isabelle/jEdit [Wen12] has recently become the default prover IDE for Isabelle.

[13]Released and available at `http://andriusvelykis.github.io/isabelle-eclipse/`. The latest release is for Isabelle2013.

[14]In a similar manner, the improvements to the Community Z Tools and the development of Z/EVES Eclipse constitute the infrastructure developments for Z/EVES theorem prover integration (see Section 10.3).

However, when Isabelle/Eclipse development started, Isabelle/jEdit was still in its infancy. Furthermore, the prototype ProofProcess system is developed on the Eclipse platform (Section 8.3.2) and integrating it with jEdit would require reimplementing the user interface components. Nevertheless, the Isabelle ProofProcess integration mostly depends on Isabelle/Scala and, apart from porting the UI, extra development to support Isabelle/jEdit would be small.

- Proof General/Eclipse [WAL05] provides a generic Eclipse-based interface to Isabelle. Unfortunately, it is no longer actively developed and the communications protocol it uses to interact with Isabelle is being phased out.

- Proof General/Emacs was the default proof assistant interface for Isabelle until Isabelle/jEdit made it obsolete. As with Proof General/Eclipse, its communications protocol is being phased out. Furthermore, its dependency on Emacs, LISP programming language requirement, limited user interface extensibility and other development obstacles make it infeasible to use as the basis for the ProofProcess system.

Isabelle/Eclipse is similar to Isabelle/jEdit in terms of functionality. It builds on the same *proof document* interaction model as well as other proof assistant support functionality provided by the Isabelle/Scala layer. However, as the name suggests, the overall prover IDE functionality is built on the Eclipse platform and adheres to the platform's conventions and user interface paradigms. Isabelle/Eclipse enables the user to author formal specifications and proofs and supports convenient typesetting with text completion suggestions, selector for mathematical symbols and syntax colouring. It provides inspection of proof results and proof progress, links to definitions, the overview of the specification, etc. More information about the Isabelle/Eclipse prover IDE and its features is available on its website [Vel]. Links to download the tool or access the source code are also listed there.

The initial prototype of the Isabelle/Eclipse prover IDE was developed early in the PhD research. A full-featured final version supporting Isabelle2013 has been released to users with the aim to maintain support for newer Isabelle versions in parity with the Isabelle/jEdit prover IDE. The application can be used standalone (without the ProofProcess add-on) and has received interest from a number of researchers: e.g. it is used within the COMPASS project, to provide Isabelle integration to the Symphony IDE tools [CCL+14].

CHAPTER 10

Integration with Z/EVES

This research focuses on capturing proofs from industrial-style formal developments, where proofs often fall into *families* according to common high-level proof ideas. Integration of the generic ProofProcess framework with the Z/EVES [Saa97] theorem prover enables capture of the proofs from industrial-style formal specifications developed using the Z notation and verified using Z/EVES.

The Z notation [Spi92, WD96] is a formal specification language, aimed at mathematically describing and modelling computer and other systems. It is used in the industry to formalise systems that require good assurance: e.g. the CICS transaction processing system from IBM [HK91], the Mondex smart card [FW08], the Tokeneer ID station for the NSA [CB08], etc. There are not many theorem proving systems that support the Z notation because of its expressiveness. The Z/EVES theorem prover is one: it has been used successfully to mechanise and verify formal industrial-style Z specifications (e.g. [FW08, FWF09, BFW09, VF10]). The availability of a large corpus of such industrial-style proofs as well as the prior experience with using Z/EVES among the AI4FM project researchers has led to selecting Z/EVES for capturing proof processes and learning the experts' strategies.

The Z/EVES theorem prover provides a small number[1] of tactics, resulting in a small proof vocabulary when capturing proof processes. Z/EVES proof scripts are linear and proof is mainly constructed in a *backward* proof style. The scope of what a user can do with a prover is quite small, allowing easier ProofProcess integration, since the different prover-specific details that need to be captured are

[1] Compared with Isabelle, for example.

limited. However, the limitations in expressivity often cause the user to construct proof workarounds: e.g. adjusting the definitions to a form that is better supported by the prover, or just "fighting the prover" in general. The case study in capturing an expert's proof process using Z/EVES (Chapter 12) features such workarounds.

The Z/EVES ProofProcess integration functionality is built on the Community Z Tools (CZT) [MU05], which provide a development environment for Z specifications and the integration with the Z/EVES prover. Section 10.1 describes how the ProofProcess system "wire-taps" the Z/EVES prover communication, whereas Section 10.2 presents the prover-specific data structures used to encode the low-level proof information. To support the proof process capture as well as the general usability of Z/EVES, extra development effort has gone into improving CZT and developing an integration to the Z/EVES theorem prover via a modern IDE. Section 10.3 provides an overview of these infrastructure developments.

Thesis contributions

The development of the Z/EVES ProofProcess integration, which description comprises the majority of this chapter, is sole work of the thesis author. The integration enables capturing proof process information from formal developments that use Z notation and the Z/EVES theorem prover.

Section 10.3 outlines improvements to CZT that are equal parts joint work together with Leo Freitas. These developments comprise necessary infrastructure to facilitate the aforementioned proof process capture developments.

10.1 Recording interactive proof

To capture interactive proof, the Z/EVES ProofProcess utilises the API available as part of CZT integration with the Z/EVES theorem prover. This integration provides a document model abstraction (similar to that in Isabelle/Scala) that can be queried for proof results and provides notifications when new proof commands are processed. Furthermore, a modern Z/EVES Eclipse prover IDE is used to develop the specifications and proofs, submit them to Z/EVES and inspect the results. Section 10.3 describes the CZT and Z/EVES integration in more detail.

User interaction with the prover is linear, with a manually controlled proof process (cf. automatic asynchronous proof checking in Isabelle):

1. User enters new proof commands (or formal specification definitions);

2. CZT parses and type-checks the proof script and the specification;

3. If the entered proof commands are valid, the user submits them to the prover;

4. Z/EVES processes the proof command and responds with a new proof goal and proof step details. If the command fails, an error message is returned;

5. The prover response is added to the document model and presented within the user interface;

6. Upon editing the proof script, the proof is backtracked to the edit location.

Overall, the Z/EVES ProofProcess integration is similar to that of Isabelle ProofProcess (see Section 9.1), stemming from the similarities in the prover interfaces. The Z/EVES ProofProcess utilises the document model and prover notifications to "wire-tap" the communication and record details about the interactive proof process. When commands are processed, the results are scheduled for analysis as a low-priority task. To support the analysis, a *snapshot* of the specification and its results is taken: this data is cloned and therefore does not change during analysis.

The integration with CZT provides the necessary tools to implement the proof process data analysis. For example, when CZT parses the specification or the proof commands, it provides annotations about the types of parameters, links to function or schema definitions, etc. All "internal" representations as well as additional analysis, editing or logic manipulation functionality are available to the extending tools such as the ProofProcess system. Furthermore, the Z/EVES responses are also parsed by CZT, providing the same support for extracting, analysing and manipulating the proof step details.

Z/EVES responses consist of new proof goals and additional details about the proof step: a list of lemmas used by proof commands, proof case numbers, etc. These are captured as part of the Z/EVES proof step *trace* (Section 10.2.2). Proof case numbers are used to infer the proof structure: proof steps with different case numbers belong to different proof branches. The resulting low-level proof graph (Section 8.6) is matched against the existing proof attempts and added to the captured proof process database. The expert can then mark the high-level insight for the captured proof process.

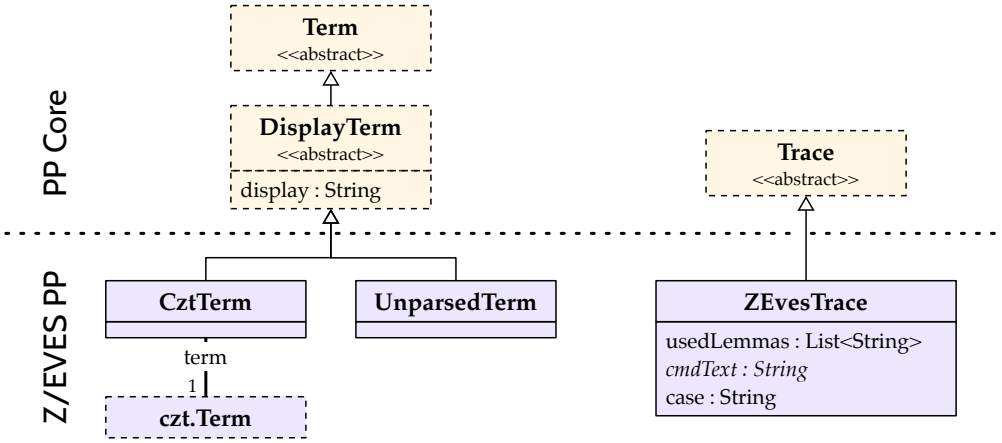


Figure 10.1: UML class diagram of Z/EVES ProofProcess data structures.

10.2 Recording prover-specific details

When recording the interactive proof process, the Z/EVES ProofProcess provides prover-specific representation for Z/EVES terms and proof commands. Native CZT representation is used for terms (goal predicates, expressions, parameters, etc.).

10.2.1 CZT terms

At the core of CZT is the XML markup for Z specifications (ZML) [UTS+03], specified as XML Schemas for different dialects. ZML provides a structure to Z specifications and can be used as an interchange format for Z tools [MU05]. The data structure carries all necessary details to reconstruct the Z specification and can be supplemented by further annotations, such as typechecking information, etc. Furthermore, CZT provides a programmatic representation (Java interfaces and classes) for these data structures as an *annotated syntax tree (AST)*. The AST is generated automatically from the ZML definitions.

Z/EVES ProofProcess uses the AST representation to manipulate and analyse Z and Z/EVES terms. Furthermore, when the captured proof process data is saved into the database, the terms are serialised to their ZML representation for storage. Figure 10.1 lists the data structures provided by Z/EVES ProofProcess integration: the AST representation is listed as `czt.Term`. The *CztTerm* class wraps the native CZT term for embedding within the ProofProcess data structures. Furthermore, the *CztTerm* extends *DisplayTerm*, which is used to provide a human-readable representation of the contained term. It is used for human inspection only.

Z/EVES ProofProcess also uses *UnparsedTerm*s (Figure 10.1) as a fall-back solution for integration limitations. In certain cases, Z/EVES responds with an "unprintable predicate" message. This signals a conversion problem between Z/EVES and the underlying EVES prover engine: prover results cannot be rendered as a Z predicate. Such a situation can happen when, for example, the goal contains multiple existentially-quantified variables and only some of them are instantiated. Often a follow-up rewrite command transforms the goal to something that can again be rendered as a Z predicate. Furthermore, *UnparsedTerm*s are used when CZT fails to parse Z/EVES results: such a situation, however, should be reported so that it can be fixed in new CZT versions. *UnparsedTerm*s are artefacts of the limitations of the prover integration. They disrupt the proof process analysis as such goals cannot be part of it. Although unavoidable in general, such cases are rare.

Having the full power of CZT as part of Z/EVES ProofProcess enables the analysis and manipulation of the captured proof process data. For example, proof-of-concept implementations are available to break down the goal term into sub-terms or to provide *schema* terms for marking proof features (Section 8.2.3). However, proof analysis involving prover functionality (rather than term analysis) is more difficult to implement because the Z/EVES prover is outside CZT and to do anything, appropriate queries and proof context need to be sent to the prover.

10.2.2 Proof step trace

Current Z/EVES proof command representation is straightforward but limited.[2] Figure 10.1 presents the implementation of the *ZEvesTrace* class that encodes proof step details. The proof command is not decomposed and instead captured using its textual representation in the *cmdText* field. However, adding support for proof command details would not be difficult as this information is available within the parsed and typechecked specification in CZT.

With results of every proof command, Z/EVES outputs the lemmas that have been used by the proof step. Z/EVES supports three types of lemmas for automated use: *assumption rules*, *forward rules* and *rewrite rules*. The first two are used to introduce temporary assumptions to the proof context, whereas *rewrite rules* are

[2]During this PhD research, the focus shifted from the Z/EVES prover to Isabelle. Support for capturing Isabelle proof details, including an attempt on capturing declarative Isabelle/Isar proof (Section 13.3.5), has been a priority in order to facilitate strategy extraction and integration with the Tinker tool [GKL14] within AI4FM. Therefore fully developing the Z/EVES ProofProcess integration is left for future work.

used to transform the goal by rewriting [MS97]. An example listing of lemma use within a proof step is available in Figure 12.7. Z/EVES ProofProcess currently collects the names of all lemmas used by the proof step and records them within *ZEvesTrace*. This information can be presented to the user or used to infer *used lemma* proof features (Section 6.5.4).

Finally, the *ZEvesTrace* structure records the proof case number. These numbers are output by Z/EVES to identify proof branches when case split occurs. For example, a conjoined goal can be proved by showing that each conjunct holds. The case split can be done in Z/EVES using the `cases` proof command. Afterwards, only a single goal with one of the conjuncts is presented to the user. However, the proof case "1" is indicated to show that this is part of a proof branch. The user can switch to the next proof branch goal using the `next` proof command: the proof case number changes accordingly. Furthermore, if nested case splits are encountered, the case number is extended to indicate this: e.g. case "1.2".

Proof case information is used by the Z/EVES ProofProcess to infer the low-level proof structure. Proof commands from different proof cases are separated into individual branches (Section 6.2). The user can "complete" a case split even if the proof branches are unfinished: the remaining goals are combined. This is represented as a *merge point* in the captured ProofProcess structure (Section 4.3.7).

❦ ❦ ❦

The current Z/EVES ProofProcess integration supports the core ProofProcess functionality: the expert's interactive proof is recorded; the proof structure is inferred automatically and new attempts are recognised; afterwards the expert can mark the high-level insight on the captured data by indicating high-level proof steps, proof intent and marking the appropriate proof features. Furthermore, the availability of the CZT API allows implementation of advanced proof process analysis features in the future and improves the automation of capturing the proof process.

10.3 Community Z Tools

The Community Z Tools (CZT) project [MU05] is building a set of tools for editing, typechecking and animating formal specifications written in the Z specification language, with some support for Z extensions such as Object-Z, Circus, etc. These tools are all built using the CZT Java framework for Z tools. As part of the integration with the ProofProcess system, the available CZT infrastructure was improved

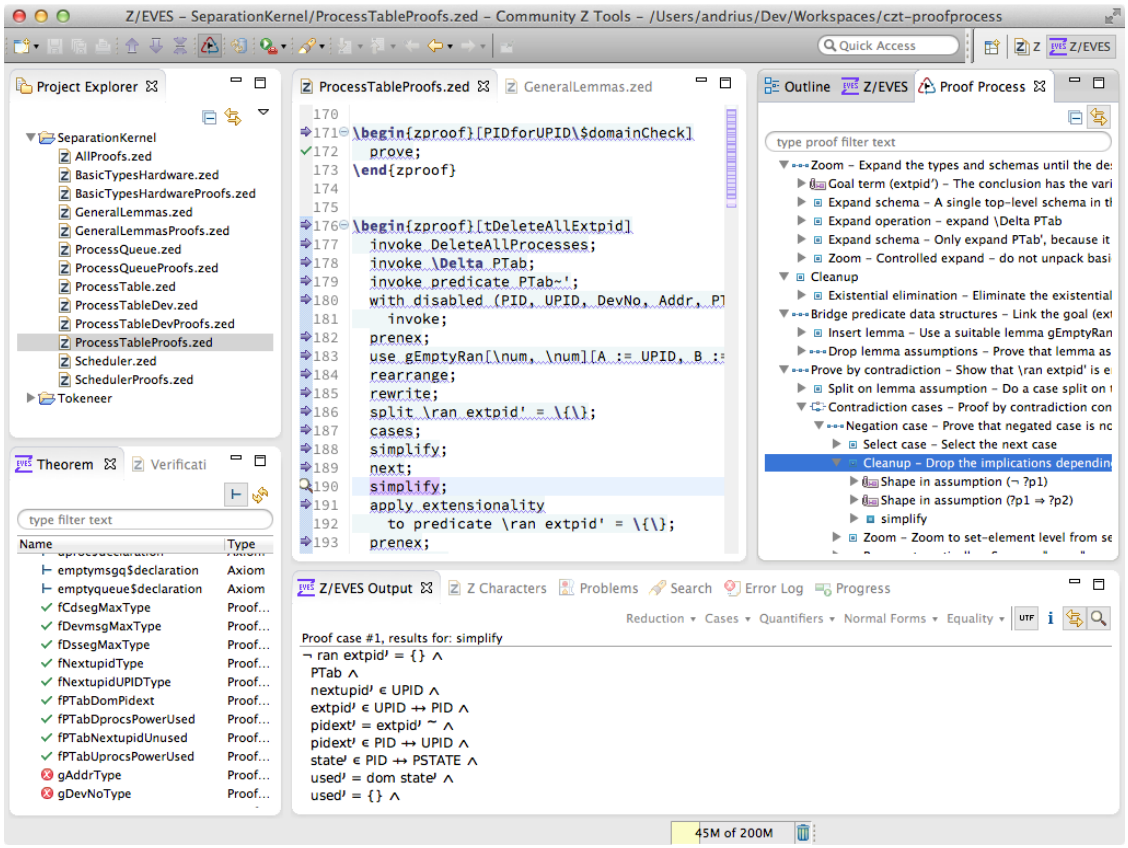


Figure 10.2: Screenshot of Z/EVES Eclipse with the ProofProcess system.

and extended to provide an interface to the Z/EVES theorem prover. The resulting modern CZT IDE with Z/EVES integration is presented in Figure 10.2. The infrastructure works have been done together with Leo Freitas, a colleague in the AI4FM project and a CZT contributor. This section highlights the main features.

10.3.1 Integrating the Z/EVES prover

The standard user interfaces to the Z/EVES theorem prover are old and limited: there is a command-line read-eval-loop style interaction as well as a Python-based graphical user interface. The command-line interface is cumbersome to use interactively and is most useful for batch-processing Z specifications. The Python UI utilises an internal socket-based protocol that uses XML messages to communicate with the theorem prover. However, the user interface is not extensible and crashes when used in modern operating systems.[3]

[3] Z/EVES is no longer developed, thus issues arising as platforms evolve remain unaddressed. However, there are talks with the original developers about providing an alternative open-source theorem prover with similar functionality and communications protocol.

To circumvent the user interface limitations and support extensibility, an interface to the Z/EVES prover has been implemented as part of Community Z Tools. This integration is based on the reverse-engineered XML communications protocol and provides a Java interface via CZT to interact with the prover.

A new "zeves" dialect has been developed for CZT to support typesetting of Z/EVES specifications and proof scripts. Z/EVES uses an older, pre-standardisation version of the Z notation, which is slightly different than the ISO-Z [ISO02] notation used by default in CZT. Furthermore, the dialect extends the Z notation to support writing proof scripts using all Z/EVES proof commands. With the new dialect, Z/EVES proof scripts can be parsed and typechecked using CZT: e.g. via the CZT command-line, CZT plug-ins for jEdit or the modern Eclipse-based CZT IDE.

The parsed specification can be submitted to the prover by encoding the commands in XML-based messages. The communications protocol has been reverse-engineered from the Z/EVES Python UI. A translator has been implemented from the AST to the Z/EVES XML encoding.

When developing the prover interface, ideas have been drawn from the implementation of the Isabelle/Scala layer [Wen12] for the Isabelle prover. The Z/EVES communications protocol is encapsulated by a strongly-typed Java API. This API is used to establish socket-based communication to the prover process, send prover messages and receive proof results. The results (e.g. new proof goals or error messages) are collected in a *document model* abstraction. When the results are received, user interface components are notified: they use the information in the document model to present the fresh results to the user.

An API has also been developed to control the proof script submission to the prover. New proof commands can be submitted "up to a point" or individually. Furthermore, the proof can be backtracked to an arbitrary earlier location. The submission to the prover is done in a linear manner, though: the API encapsulates the underlying read-eval-loop interaction.

With the API in place, additional extensions have been developed to improve theorem proving in Z/EVES. Advanced proof tactics can be specified by combining the basic proof commands in Z/EVES. Furthermore, support for section management has been added. Sections in the Z notation can be used to split the formal specification into parts (e.g. into different files). Each section can specify on which other sections it depends, etc. The concept is similar to *theories* in Isabelle. Unfortunately, Z/EVES does not support them and expects a single specification file containing all the definitions and proofs. With the CZT integration, section

dependencies are resolved before submitting to the prover and "flattened" for submission to the prover, simulating a single specification.

In addition to providing an API to parse and process proofs in Z/EVES, new user interface components have been developed as part of CZT Eclipse to provide a *prover IDE*. They include displaying the progress and the results within the proof script editor, providing a prover output view to inspect proof results, actions to easily select proof commands (including lemma application suggestions for selected terms), controlling what is submitted to the prover, displaying a list of proved and unproved theorems, etc. Some of these features are seen in Figure 10.2.

10.3.2 Improvements to CZT

Some of the other work on CZT has been done to improve the general infrastructure, usability and extensibility of the tools, thus indirectly benefiting the ProofProcess system that builds on top of it:

- Improvements to the Eclipse-based CZT IDE. The existing Z specification environment has been upgraded and streamlined: better document outline, more consistent file management, fine-tuning of the specification editor and Z characters selection, etc. Furthermore, the user interface for the Z/EVES prover integration is available as Eclipse plug-ins. The ProofProcess system also builds upon this.

- Rework of the overall CZT build process and module organization. Module dependencies have been fine-tuned to enable a minimal set of libraries for each application. The build process has become easier and almost fully automatic. This enables provision of continuous integration, nightly builds, downloads and updates. Users no longer need to wait for the next release and can easily use nightly builds. The improvements were also needed to facilitate the inclusion of CZT libraries within the ProofProcess system.

- New proof obligation (PO) generator that is integrated in the CZT IDE. It facilitates verification of industrial-style formal specifications.

- Rework of the *section manager* to enforce *transactional* updates. The section manager is at the core of CZT: it manages dependencies between sections and stores the data for the current one as well as all its parents, ensuring that all data relationships are accounted for. This work has been to streamline the

section manager so that section reuse ultimately becomes possible. Currently each editor re-checks all of its parent sections. The section data cannot be shared because it can be polluted by subsequent sections. With transactional updates, dependencies between data are clearer and easier to manage.

PART IV

Evaluation and use

CHAPTER 11

Capturing proof about memory deallocation

CASE STUDY

In order to evaluate the approach of describing and capturing high-level proof processes, as proposed in this thesis, examples are required. A formal development of a heap memory manager specification [JS90, Chapter 7] has provided a good case study within the AI4FM project [FJVW13]. The problem consists of modelling a heap memory storage as well as memory allocation and deallocation operations. Its formal development includes several layers of refinement. Formal verification effort involves proving data refinement, operations feasibility, well-formedness and other properties. Refer to [FJVW13] for further details as well as mechanisations in both Isabelle/HOL and Z/EVES provers.

This case study presents proofs of three similar lemmas arising within the formal development of the heap specification using Isabelle/HOL.[1] One of the lemmas is proved interactively and its proof is captured using the ProofProcess system with high-level descriptions and abstractions as proposed in this thesis. This information represents the expert's high-level insight and could then be reused to prove the remaining "sibling" lemmas.

[1] The standard Isabelle/HOL libraries have been supplemented with some VDM data structures and operators, e.g. definitions and lemmas for domain subtraction $\lhd$ operator for maps.

A holistic evaluation of the ProofProcess framework would have strategies extracted from the captured data, which would then be used to prove similar lemmas automatically. Unfortunately, such tool chains (as well as the necessary techniques) are not available yet within AI4FM. Therefore, the evaluation presented here and in Chapter 12 illustrates how this process *would* happen. Sections 11.3–11.4 reuse the captured proof process information manually as if strategies would have been extracted. The success of the approach is repeated in the subsequent case study of separation kernel verification using Z/EVES in Chapter 12.

The overall heap memory manager case study and the lemmas presented in this chapter are of smaller scale than full industrial formal developments and proofs. These particular lemmas and their proofs have been selected because they can illustrate different facets of the proof process capture approach and other ideas presented in this thesis. Examples from real industrial proofs would require significantly more background to establish, would be much larger to present and would likely fail to demonstrate more than several interesting abstractions or proof features. Furthermore, the full-scale evaluation of the ProofProcess system and the approach with real industrial proofs has not been done. Nevertheless, these case studies are representative of *industrial-style* formal developments (see also Section 2.1). The approach, ideas and tools should scale to industrial-size proofs.

11.1 Modelling the heap and memory deallocation

The core of the heap manager specification consists of a heap memory storage data structure as well as two main operations: memory allocation (*NEW*) and deallocation (*DISPOSE*).[2] The AI4FM attempt builds upon the original specification in [JS90, Chapter 7]. The intention is to preserve the original specification where possible and avoid changing the model just to make the proofs easier. The AI4FM approach tries to tackle proof complexity via strategies rather than model changes. Thus model changes of the heap specification mostly comprise error correction and clearer abstractions [FJVW13].

The proofs described in this case study (Sections 11.2–11.4) are about memory deallocation in refined specification—the *DISPOSE*1 operation. This section provides a brief overview of the concepts involved (see [FJVW13] for full details).

[2]The words *deallocate* and *dispose* are used interchangeably. *Disposing* a previously allocated region of heap memory returns its locations back to the pool of *free* (available) memory.

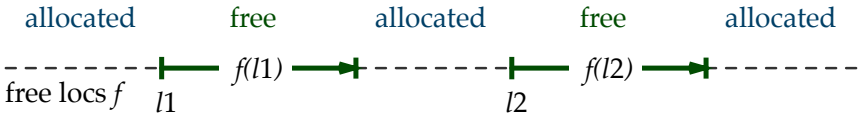


Figure 11.1: Map f : *Free*1 of free heap locations.

The main heap data structure is modelled as a map $Free1 = \mathbb{N} \xrightarrow{m} \mathbb{N}_1$[3] of free (unallocated) memory regions (illustrated by Figure 11.1). Each free memory region is represented as a mapping from the start location to the non-empty size of the region. The regions must be separate (i.e. there cannot be two abutting free regions—they must be joined into a single contiguous region) and cannot overlap. The restrictions are specified as invariants on *Free*1.

Memory deallocation operation $DISPOSE1(d, s)$ returns a previously allocated memory region (location d, size s) to the free memory pool. The operation has a precondition that the disposed memory region is currently allocated, i.e. it is *disjoint* from (does not overlap with) the free regions map f. The postcondition, however, is more complicated due to the non-abutting requirement on the resulting free regions map f'. If the disposed region abuts existing free regions from either side of it, all these regions must be joined together into a single contiguous region within f'. The four cases are: (1) the region $d \mapsto s$[4] does not abut any existing free regions; (2) abuts from either *above*; or (3) *below*; or (4) from both directions. These cases are illustrated in Figures 11.2, 11.3, 11.16, and 11.20, respectively.

The feasibility proof of *DISPOSE*1 posits that given the operation preconditions, its postconditions are satisfiable and all after-state invariants hold. The heap map f' gets updated with the new disposed (and possibly merged) free region; and the previous abutting (now merged) region is removed from it. One needs to show that the updated map f' is still disjoint and separated.

This case study focuses on a family of lemmas used in the *DISPOSE*1 proof. They show that the updated heap map f' is *disjoint*: i.e. none of the regions are overlapping. Such a conjecture appears in each of the *DISPOSE*1 cases.

The non-abutting case of $DISPOSE1(d, s)$ (Figure 11.2) is straightforward: new mapping is added directly without any merging and removals: $f' = f \cup \{d \mapsto s\}$. The goal that f' is still disjoint is then trivially shown from the operation precondi-

[3]The number '1' indicates the first level of refinement—the full development [FJVW13] has two levels of refinement: abstract '0' and more concrete '1'.

[4]The heap regions in this section will be presented as a mapping $start\_loc \mapsto size$.

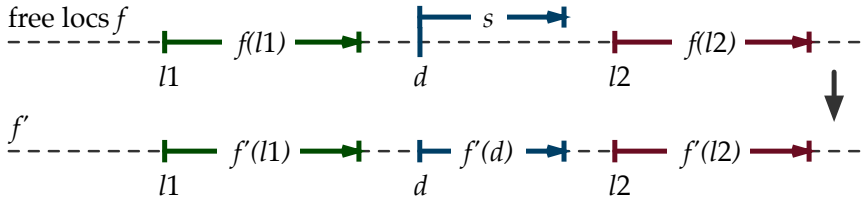


Figure 11.2: $DISPOSE1(d, s)$: no free regions abut the disposed area.

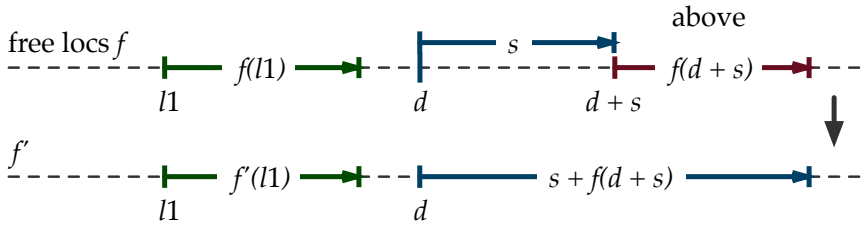


Figure 11.3: $DISPOSE1(d, s)$: free region abuts *above* the disposed area only.

tion, which requires the region $d \mapsto s$ to be disjoint from f (i.e. can only deallocate a non-free region).

Lemmas of the remaining three cases—"above", "below" and "both"—are presented in full below. Their proofs are captured using the ProofProcess framework. The captured information includes the proof scripts in Isabelle/HOL enriched with proof process metadata.

11.2 Proof of disjointness in "above" case

The "above" case of $DISPOSE1(d, s)$ describes memory deallocation when an existing free memory region abuts the disposed region $d \mapsto s$ from *above* (illustrated by Figure 11.3). The *above* region hence starts at the location $d + s$ and its size can be queried from the heap map via $f(d + s)$. The $DISPOSE1$ operation makes the following changes:

1. The original entry about the *above* region is removed from f: specified using the domain subtraction operator $\{d + s\} \lhd f$.

2. The disposed region is merged with the *above* and added to the updated map f. The new region starts at d and is of the combined length of both regions. Thus the updated map $f' = (\{d + s\} \lhd f) \cup \{d \mapsto (s + f(d + s))\}$.

```
lemma dispose1_disjoint_above:
 "F1_inv f ⟹
  disjoint (locs_of d s) (locs f) ⟹
  d + s ∈ dom f ⟹
  nat1 s ⟹
  disjoint (locs_of d (s + the (f (d + s)))) (locs ({d + s} ⩤ f))"

apply (unfold F1_inv_def)
apply (elim conjE)
apply (subst locs_add_size_union)
apply assumption
apply (simp add: nat1_map_def)
apply (rule disjoint_union)
apply (erule disjoint_subset)
apply (erule locs_ar_subset)
apply (subst locs_region_remove)
apply assumption
apply assumption
apply assumption
apply (rule disjoint_diff)
done
```

Figure 11.4: Lemma *dispose1_disjoint_above* with proof.

Proving feasibility of the *DISPOSE1* operation involves showing that the result heap f' satisfies the invariants of *Free1*. One of these is the *disjointness* invariant: i.e. showing that the heap regions in the result map are disjoint from one another. After simplification,[5] this goal is collapsed to show that the disposed region of the *heap* is disjoint from the other *free* heap locations:

```
disjoint (locs_of d (s + the (f (d + s)))) (locs ({d + s} ⩤ f))
```

The first argument describes the disposed-and-merged region $d \mapsto s + f(d + s)$ (see Figure 11.3). The set of locations comprising this merged region is specified using the `locs_of(start, size)` function. The second argument represents the set of all unchanged locations in the original heap map f, i.e. minus the replaced *above* region. The `locs(map)` function collects the set of all remaining free heap locations from the mapping. This goal is extracted as lemma *dispose1_disjoint_above* (Figure 11.4). The lemma assumptions come from *DISPOSE1* preconditions (e.g. the disposed region $d \mapsto s$ is disjoint from the original map f) and the before-state invariant on f. The lemma is proved using Isabelle/HOL: its apply-style proof is listed in Figure 11.4.

[5]The lemmas discussed in this section form the critical parts of corresponding proofs. The full feasibility proofs in Isabelle/HOL including steps leading to the lemmas can be viewed in [FJVW13].

The presented proof uses lower-level, deterministic proof tactics, rather than the more automated `auto`, `blast`, etc. This approach gives a better illustration of the proof strategy for the purposes of this thesis. It should be noted that neither of the standard automatic tools in Isabelle (including Sledgehammer) can find a proof for this lemma initially. However, the automatic theorem provers used by Sledgehammer are successful after several initial manual steps (e.g. after using the *locs_add_size_union* lemma).

Figure 11.5 shows an overview of the *dispose1_disjoint_above* proof as captured by the ProofProcess framework. The actual proof script commands (listed in Figure 11.4) are parsed into a structural proof, recognising branches in the proof that discharge independent goals. The user provides additional proof metadata by indicating proof intents at several levels of abstraction. Furthermore, the user marks the important parts of the proof as *proof features* (not pictured in Figure 11.5). Finally, proof steps record the whole proof transformation: the starting goals of each step, details about the proof commands, as well as the resulting goals after tactic execution. Figure 11.6 shows a screenshot of the Isabelle ProofProcess tool with a fragment of the captured proof process data displayed. The rest of the section presents the ProofProcess data and its capture step-by-step.

11.2.1 Appropriate level of discourse in proof

The proof of lemma *dispose1_disjoint_above* depends on certain properties of the data structure (the map `f` of free heap regions). These properties are specified as part of the data type invariant on `f`. The invariant predicate is available among the assumptions of lemma: `F1_inv f`. However, the invariant is not expanded by default and thus the properties are not directly available for use in the proof. The initial step is to unfold the invariant definition and introduce the predicates contained within as assumptions to the proof.

Abstract "zoom" step

Industrial-style formal developments usually involve modelling of complex data structures and their properties as invariants on data types. Such invariants often consist of (e.g. a conjunction of) smaller invariant predicates, mirroring the development of composite data structures from smaller components. To avoid unnecessary explosion of proof detail, invariant definitions are expanded (unfolded)

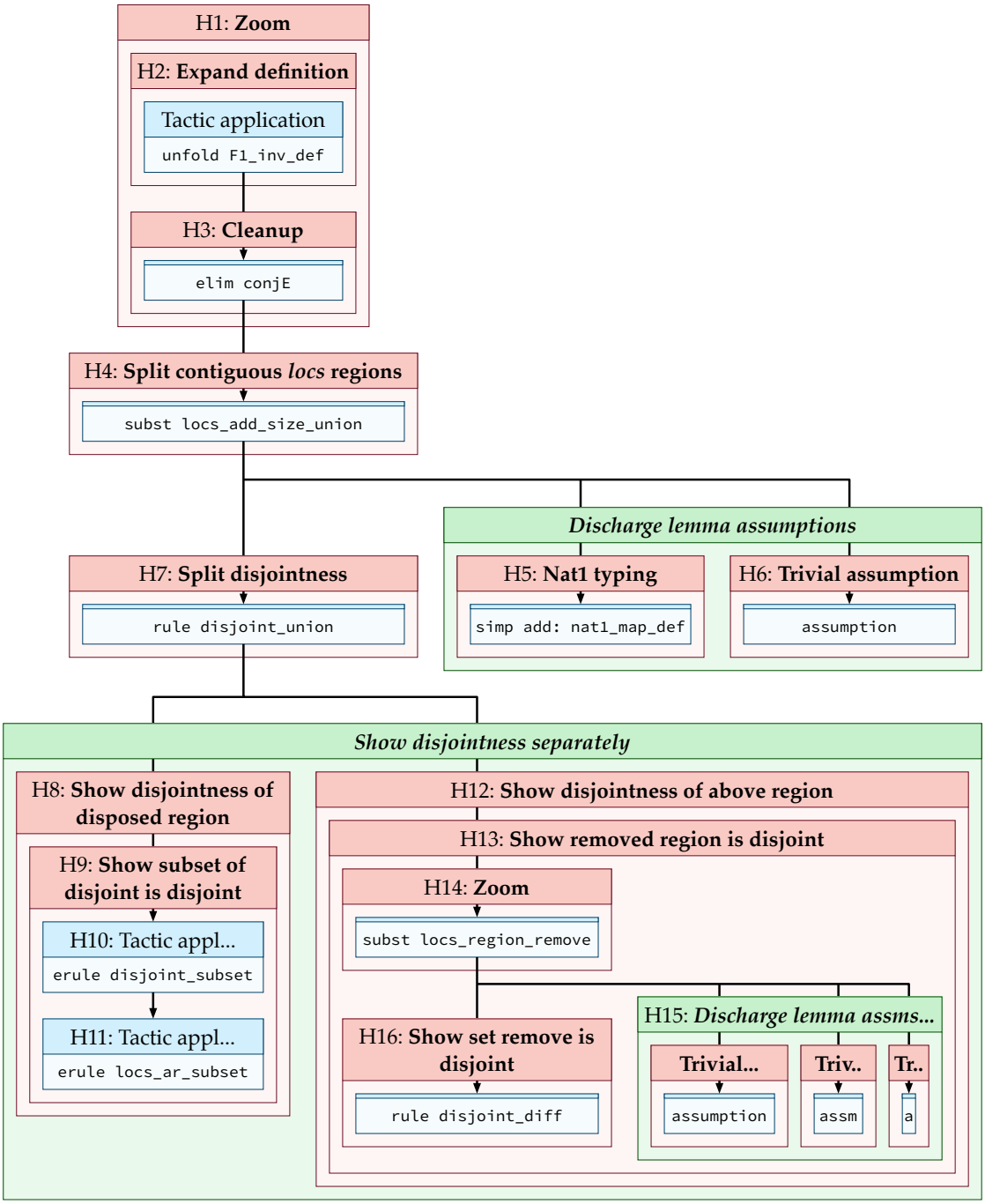


Figure 11.5: Simplified[6] ProofProcess tree of lemma *dispose1_disjoint_above*. Legend:

- 'Tactic application' (blue box) : prover commands as *ProofEntry* elements. The `apply()` text in the commands is skipped for legibility.
- **Intent** (bold, red box) : *ProofSeq* elements.
- ***Intent*** (bold italic, green box) : *ProofParallel* elements.

[6]Proof structure and intents are shown. Details are listed in corresponding ProofProcess steps (e.g. H16). Some tree elements without ProofProcess metadata have been suppressed for legibility.

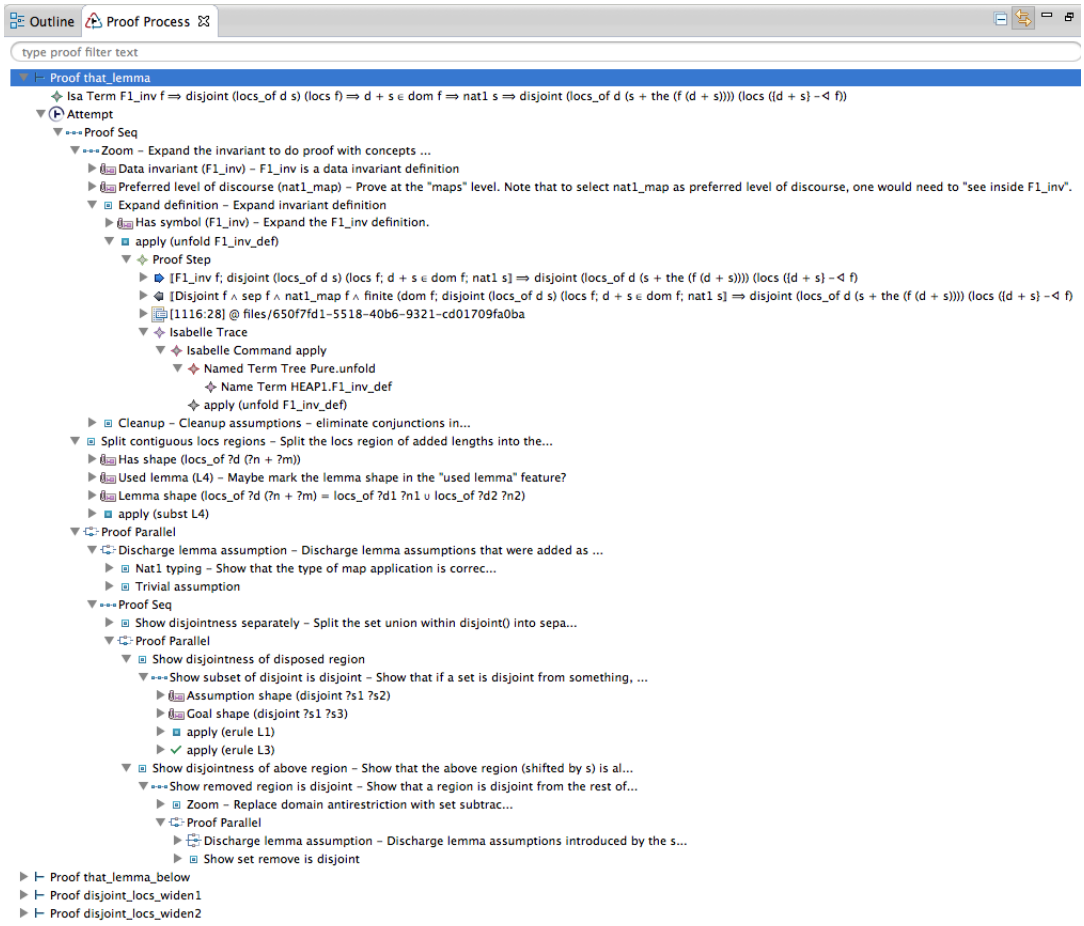


Figure 11.6: Screenshot of Isabelle ProofProcess showing fragments of the captured proof process of *dispose1_disjoint_above*.

gradually to obtain the necessary predicates at appropriate levels of proof. This "zooming" of definitions is the initial step of the *dispose1_disjoint_above* proof.

Step H1[7] lists the details about the initial **Zoom** step of the proof. The user marks the proof to indicate that the first activity is to "zoom" the definitions to an appropriate level, introducing necessary assumptions. The **Zoom** step is an abstract one: in the current proof, the actual "zooming" requires two proof steps in Isabelle/HOL. Therefore, step H1 groups these child steps (H2–H3) using a *ProofSeq* structure. They are examined later in this section.

The ProofProcess framework allows marking the proof at different layers of abstraction. The **Zoom** step groups[8] the actual proof commands required to achieve

[7]The ProofProcess step details are presented in a structured format. The letter 'H' in step numbering is for "heap" proof process steps.

[8]Figure 11.5 provides a visual overview of grouping, different levels of abstraction and other relationships between proof steps.

Intent: Zoom *ProofSeq*

Narrative: Expand the invariant to do proof with concepts it defines.

In features:

- *Data invariant* (`F1_inv`) — `F1_inv` is a data invariant definition.
- *Preferred level of discourse* (`nat1_map`) — Prove at the "maps" level. To select `nat1_map` as preferred level of discourse, one would need to "see inside `F1_inv`".

Out features: *(none)*[a]

Children:

- (H2) *ProofSeq*: **Expand definition** ...
- (H3) *ProofSeq*: **Cleanup** ...

In goals (flattened):

```
1. F1_inv f ⟹
   disjoint (locs_of d s) (locs f) ⟹
   d + s ∈ dom f ⟹
   nat1 s ⟹
   disjoint (locs_of d (s + the (f (d + s)))) (locs ({d + s} ⩤ f))
```

Out goals (flattened):

```
1. disjoint (locs_of d s) (locs f) ⟹
   d + s ∈ dom f ⟹
   nat1 s ⟹
   Disjoint f ⟹
   sep f ⟹
   nat1_map f ⟹
   finite (dom f) ⟹
   disjoint (locs_of d (s + the (f (d + s)))) (locs ({d + s} ⩤ f))
```

[a] In subsequent ProofProcess step details, empty sets of features will be omitted altogether.

Step H1: **Zoom**

the desired proof result. Such abstraction provides a higher-level overview of the proof: hiding the actual proof steps but communicating the user's intent. Furthermore, it allows for several "implementations" of the proof strategy. There are numerous different proof commands in Isabelle/HOL to perform unfolding of an invariant definition. Each of them can be recorded as "implementing" the same zooming intent. When extracting a strategy, these different steps would constitute alternatives on how to zoom in a proof.

When grouping several proof steps using a *ProofSeq*, the ProofProcess system provides a "flattened" view of the actual proof in regards to the higher-level step. For example, step H1 lists the *in/out* goals of the proof step. The goals are derived

by flattening the inner structure of a proof step (Section 4.3.5): the *in* goals come from the first proof step and the *out* goals from the last step within a *ProofSeq*.

The flattened *in/out* goals of H1 show the goal transformation by an abstract **Zoom** step: the invariant predicate `F1_inv f` is replaced by its constituent predicates in the assumptions. This presentation omits any intermediate proof states and provides a "bigger picture" of how the proof advances. Furthermore, it allows recording *out features*[9] over the whole abstract step.

Capturing zooming proof features

The **Zoom** step was chosen as the initial proof step to avoid expanding the invariant later for every case split. The predicates within `F1_inv` definition, namely the type of the heap map `nat1_map f`, are hidden by the definition. This indicates that the invariant predicate is at a level too high for the proof. One of the *proof features* in H1 marks that the *preferred level of discourse* is at the level of `nat1_map` definition: i.e. the expert wants to do proof at the "maps" level, not at the invariant level. This proof feature could be inferred by matching the assumption predicates (and their contents) with the definitions used by the goal: e.g. definitions that appear unexpanded in the goal might not need expanding. The user could mark this feature by "drilling down" into the invariant via the user interface to see what level of expansion would suit the goal definition.

Another proof feature in H1 marks the `F1_inv` definition as a data invariant. This aims to emphasise the different role that the function plays in comparison to other similar function definitions.[10] For example, dealing with data invariants in proofs frequently involves expanding them to access predicates within. On the other hand, general functions (e.g. `locs`) are supplemented with various lemmas about their properties to avoid "zooming" into function definitions. Thus dealing with a data invariant often requires a corresponding **Zoom** proof step.

No *out features* are marked for the H1 step. They are useful when automated proof tactics are used, e.g. `auto` in Isabelle/HOL. Such tactics can perform multiple proof steps and their outcome depends on which lemmas are available. The out features can be used to mark the expected result of a strategy involving an automated proof step. During strategy replay, matching just the *in features* is not enough: the strategy needs to be executed for *out feature* matching on the results.

[9] *Out* proof features (Section 4.2.3) record the expected results of a proof step. Not used in H1.

[10] The data invariants as well as most general functions in the heap example are encoded as Isabelle/HOL *constant definitions*.

The multi-thread proof checking in Isabelle would serve well for this use case, allowing matching of the *out features* "in the background". On the other hand, deterministic proof strategies transform the goal in a predetermined way and during replay would produce the expected result, if successful. Since matching on *out* features requires executing the proof commands, it is preferable to describe deterministic strategies using *in* proof features. The **Zoom** steps (H1)—as well as the majority of the *dispose1_disjoint_above* proof—use deterministic proof tactics, thus *out features* are omitted from the captured proof process.

Proof features as strategy hints

The two proof features recorded in step H1 give a "hint" on when to use the proof strategy. They suggest that zooming is used because there is a data invariant definition (`F1_inv f`) in the conjecture that is not at the preferred level of discourse for the proof. Such a hint would be enough to extract into an initial strategy: try zooming by unfolding the data invariants when they are above the preferred level of discourse for the proof. Such a strategy is sufficient for reuse in the "sibling" lemmas *dispose1_disjoint_below* and *dispose1_disjoint_both* (see Sections 11.3–11.4). However, in more complex proofs such a strategy would be too unrestricted: e.g. it would unfold all data invariants, when a more fine-grained control of zooming is needed. Upon encountering this, the expert would add additional proof features to indicate the important features of the fine-grained zooming. This information would be used to refine the **Zoom** strategy.

The amount and precision of proof features specified when capturing the proof process affects the quality of extracted strategies. However, marking the proof features manually is a significant overhead to the proof process. The initial goal of the ProofProcess system is to provide a framework for capturing the proof process information at the level that the expert deems appropriate—the user marks the important parts of the proof. For manual strategy recall and reuse, marking a minimal number of particularly important features can be enough. Furthermore, advanced proof analysis algorithms may correctly suggest common proof features, thus reducing the overhead of marking them manually (e.g. compare the function definitions in the goal and the assumptions to derive the *preferred level of discourse*). Rather than generalising to strategies immediately, the user just tags the actual proof as it is. The data can then be collected to try machine learning, sophisticated generalisation algorithms or other approaches to extract good strategies (see

Chapter 7). It is not necessary to consider the general cases for the intents and proof features immediately. On the first occasion when the strategy does not apply correctly, the user may adjust the proof feature set to a better fitting one and let the strategy extraction make sense of these additional examples.

Expanding invariant definition

The abstract **Zoom** step in the current proof is realised by two lower-level proof steps: **Expand definition** and **Cleanup**. Each corresponds to a single proof command used by the expert in Isabelle/HOL. The overview of the captured ProofProcess data for each step is listed respectively in steps H2 and H3.

Isabelle ProofProcess captures every proof command as a *ProofEntry* element and populates the *intent* automatically: each entry is tagged as a **Tactic application**. The user provides high-level descriptions for each command by "wrapping" the *ProofEntry* steps into *ProofSeq* elements and indicating the additional proof intents and features (see Figure 11.5 for illustration). Alternatively, the user could replace the **Tactic application** intent with other metadata directly on the *ProofEntry* data element. The differences between these approaches are insignificant and this section collapses the ProofProcess details of the decorating *ProofSeq* and the underlying *ProofEntry*/**Tactic application** into a single presentation (e.g. in step H2).

The first *actual* proof step of *dispose*1*_disjoint_above* is expanding the definition of `F1_inv` to access predicates within. The proof uses `unfold F1_inv_def` tactic to achieve this. Step H2 lists the captured ProofProcess data for this proof step. The proof step is decorated with a *ProofSeq* containing a slightly higher-level description, namely the **Expand definition** intent and an associated proof feature.

The higher level description serves several purposes. It gives an at-a-glance view of what the proof command is trying to achieve. Some proof commands are self-explanatory (e.g. the `unfold` tactic)—others can be cryptic. A human-readable explanation can tell a better story of how the proof is achieved. Furthermore, the high-level description allows recording alternative proof commands to achieve the same goal. For example, here the proof command `apply (unfold F1_inv_def)` can be replaced with `apply (simp only:F1_inv_def)` yielding the same result. A proof expert can use different proof commands to represent similar actions. The same *intent* marks them as alternatives for strategy extraction.

The single proof feature of step H2 marks `F1_inv` for expansion. A trivial extracted strategy would be that `F1_inv` invariant definition can always be expanded.

Intent: Expand definition *ProofSeq* (as decoration)[a]

Narrative: Expand invariant definition.

In features:

- *Has symbol* (`F1_inv`) — Expand the `F1_inv` definition.

Children:

- *ProofEntry*: **Tactic application** `apply (unfold F1_inv_def)`—expanded as "goals" and "proof step" next.

In goals:

```
1. F1_inv f ⟹
   disjoint (locs_of d s) (locs f) ⟹
   d + s ∈ dom f ⟹
   nat1 s ⟹
   disjoint (locs_of d (s + the (f (d + s)))) (locs ({d + s} ⩤ f))
```

Proof step:

Command: `apply (unfold F1_inv_def)`

Tactic: *Pure.unfold*

Used lemmas: *HEAP1.F1_inv_def*

Source: Offset 1116, length 28 in

file `files/650f7fd1-5518-40b6-9321-cd01709fa0ba`

Out goals:

```
1. Disjoint f ∧ sep f ∧ nat1_map f ∧ finite (dom f) ⟹
   disjoint (locs_of d s) (locs f) ⟹
   d + s ∈ dom f ⟹
   nat1 s ⟹
   disjoint (locs_of d (s + the (f (d + s)))) (locs ({d + s} ⩤ f))
```

[a] A *ProofSeq* that contains a single proof step can be thought of as *decoration*, as it serves to add metadata rather than to group its steps.

Step H2: **Expand definition**

This simplistic approach actually works well for "sibling" lemmas: i.e. the `F1_inv` invariant definition always needs to be expanded there. When dealing with proofs involving different datatypes, however, the trivial strategy would no longer apply. The expert could manually select the new datatypes to be expanded in the proofs. That strategy would then apply for *its* "siblings". Alternatively, strategy could be generalised to apply to different datatypes. Proof analysis algorithms could try to infer additional properties about the more general strategy: e.g. that the `F1_inv` is a *constant definition* in Isabelle/HOL; that there is a definitional lemma `F1_inv_def`

that can expand the predicates within; etc. Strategy extraction via generalisation could use these proof features to produce a generic strategy that suggests expansion of any constant definitions in Isabelle/HOL. The proof features on a **Zoom** step (H1) could provide further hints to strategy extraction: e.g. restricting these expansions to datatype invariant functions only. Note that the proof process capture and strategy extraction should be viewed as separate applications: there can be different approaches to extract appropriate strategies using the captured data.

Each proof step captured by the ProofProcess framework is self-contained. It records both *in* and *out* goals: i.e. the proof state before and after the proof command execution. Goals in step H2 record the unfolding of `F1_inv` definition: the predicate `F1_inv f` is replaced by its contents:

```
Disjoint f ∧ sep f ∧ nat1_map f ∧ finite (dom f).
```

Recording proof commands

Proof command details are also recorded within the *ProofEntry* structure. The Isabelle ProofProcess system captures fully qualified configuration details of the proof command: i.e. the tactics used, their parameters, lemma instantiations, etc. Some of these are listed for the **Expand definition** step H2. In addition to parsing the proof command, the system could capture further information about the proof environment and other hidden information: e.g. the lemmas that a simplifier tactic has access to (*simpsets*)—and actually uses; the maximum search depth of automatic tactics; etc. This information links the higher-level proof process description with its "implementation" in the prover. It will be used to drive the prover during strategy replay. The information can also be used to infer additional proof features: e.g. by scanning the *used lemmas* of a proof command to mark the particularly important ones as proof features.

Furthermore, each step records the *source* of a proof command: i.e. its location within a proof script, as part of the *proof history* capture (see Chapter 5). The proof scripts are versioned and their historical change is collected. Linking the proof history with the captured proof process data enables opportunities to *animate* the expert doing this proof in the future (see Section 5.2).

Goal cleanup

The next proof step—**Cleanup**—breaks down the unfolded invariant predicate into separate assumptions by eliminating conjunctions. Step H3 provides an

Intent: Cleanup *ProofSeq* (as decoration)

Narrative: Cleanup assumptions—eliminate conjunctions in the assumptions.

In features:

- *Assumption shape* (`?p1 ∧ ?p2`) — Match a conjunction in one of the assumptions.

In goals (filtered):

```
1. Disjoint f ∧ sep f ∧ nat1_map f ∧ finite (dom f) ⟹
   ?p1 ⟹ ?p2
```

Proof step: `apply (elim conjE)...`

Out goals (filtered):

```
1. ?p1 ⟹
   Disjoint f ⟹
   sep f ⟹
   nat1_map f ⟹
   finite (dom f) ⟹
   ?p2
```

Step H3: **Cleanup**

overview of the proof process data captured for it. Conjunction elimination is a general and frequently used proof step; it splits a conjoined predicate piecewise, allowing the individual use of its parts. This is particularly useful when dealing with complex invariants, when the proof requires specific properties (parts of the invariant) rather than the whole thing.

The **Cleanup** intent in H3 signals that the conjunction elimination "cleans up" the results of an expansion of an invariant definition. The proof step can easily be used generally, without invariant definition expansion preceding it. Then it can be viewed as a normalisation of assumptions in a conjecture—still a groundwork, "cleanup" step. There would be more such general "cleanup" steps: e.g. elimination of existential quantifiers in assumptions, etc. During strategy extraction, they would become alternatives in the **Cleanup** strategy. However, the alternatives are not mutually exclusive. All applicable "cleanup" steps should be executed as long as they match the goal. For example, both conjunction elimination and quantifier elimination should be performed if applicable. The repetition may continue: a quantifier elimination may produce more predicates that can be split up further using conjunction elimination. Thus the extracted **Cleanup** strategy would contain a *loop* that would keep applying various **Cleanup** steps as long as they match.

Conjunction elimination is performed because one of the assumptions contains conjoined predicates. This is marked as an important proof feature in H3. The

Assumption shape specifies the *shape* of a particular assumption term: i.e. that one of the assumptions matches the term `?p1 ∧ ?p2`, where `?p1` and `?p2` are term placeholders. This basically specifies that there must be an assumption with a top-level conjunction. Alternatively, this could be specified using a *Top symbol* feature, requiring that the top symbol in an assumption is a conjunction. See Section 4.2 for further discussion on different *proof feature* types. The `apply (elim conjE)` command will eliminate all top conjunctions in all assumptions. As a trigger, it is enough to mark that at least a single top conjunction symbol exists.

The goals in step H3 are presented in a *filtered* view. In such presentation, only the changed parts of large goals are presented to the user. The ProofProcess system compares the *in* and *out* goals and calculates the differences. This can be used by proof analysis to infer the important proof features: the changed goal parts make very good candidates! Furthermore, the user can have a more focused view to inspect the goal changes. The large number of predicates in industrial-size proofs can make the goals unmanageable. The *filtered* view shows only the affected goal terms and thus is a more accurate representation of "what has happened", i.e. the proof process. Other proof step details in H3 (and in the majority of the subsequent proof steps) are omitted to avoid unnecessary clutter in the thesis. They are captured in the same manner as described earlier.

The combination of **Expand definition** and **Cleanup** steps achieves the proof state expected by the abstract **Zoom** step: the invariant `F1_inv f` is expanded and its properties (particularly `nat1_map f`) are available as assumptions. This brings all parts of the goal to the preferred level of discourse: maps.

11.2.2 Partitioning the problem

The expert proves lemma *dispose*1*_disjoint_above* by partitioning the problem into two parts. The merged region $d \mapsto s + f(d + s)$ is split back into its constituent parts: i.e. regions $d \mapsto s$ and $d + s \mapsto f(d + s)$. The disjointness of each region is afterwards proved separately. Two main proof steps—**Split contiguous *locs* regions** (H4) and **Split disjointness** (H7)—perform the partitioning of the goal within the proof. Both of these steps use additional lemmas, which are assumed to be already available to the proof.[11]

[11] Additional lemmas used in this proof have been developed during the initial hand-proof of the problem (see [FJVW13, Section 3.2.3]) and are assumed to be available. Discovering needed lemmas is a different problem outside the focus of this thesis (see Section 13.3.3 for a brief discussion).

Intent: Split contiguous *locs* regions *ProofSeq* (as decoration)

Narrative: Split the *locs* region of added lengths into their constituents.

In features:

- *Has shape* (`locs_of ?d (?n + ?m)`)
- *Used lemma* (`locs_add_size_union`)
- *Lemma shape* (
  `locs_of ?d (?n + ?m) = locs_of ?d1 ?n1 ∪ locs_of ?d2 ?n2`)

In goals:

```
1. disjoint (locs_of d s) (locs f) ⟹
   d + s ∈ dom f ⟹
   nat1 s ⟹
   Disjoint f ⟹
   sep f ⟹
   nat1_map f ⟹
   finite (dom f) ⟹
   disjoint (locs_of d (s + the (f (d + s)))) (locs ({d + s} ⩤ f))
```

Proof step: `apply (subst locs_add_size_union) ...`

Out goals (filtered)[a]**:**

```
1. ?assms ⟹ nat1 s
2. ?assms ⟹ nat1 (the (f (d + s)))
3. ?assms ⟹ disjoint (locs_of d s ∪ locs_of (d + s) (the (f (d + s))))
                     (locs ({d + s} ⩤ f))
```

[a] `?assms` variable denotes the same assumptions as *in* goals.

Step H4: **Split contiguous *locs* regions**

Splitting heap regions

An overview of the captured proof process structure in Figure 11.5 illustrates how the proof is advanced. The **Split contiguous *locs* regions** step transforms the main goal but also produces additional sub-goals. These are side-conditions to the application of lemma *locs_add_size_union* to split the merged region. The side-conditions, however, are simple and quickly proved as individual proof branches.

The approach of this proof step, as described by its intent, is to split the locations of the merged region into two sets. The region was merged from two abutting regions, thus it can be split again to apply different proof strategies for each region. The partitioning uses *locs_add_size_union* (Figure 11.7) to do the location arithmetic: a region of added lengths is split at the addition point to a union of two constituent sets of locations. The side-conditions (region lengths are non-zero: $n, m \in \mathbb{N}_1$) are used to prevent empty regions of heap memory in the model. Refer

```
lemma locs_add_size_union:
 "nat1 n ⟹ nat1 m ⟹
  locs_of d (n + m) = locs_of d n ∪ locs_of (d + n) m"
```

Figure 11.7: Lemma *locs_add_size_union*.

to [FJVW13, Section 3.2.3] for further discussion on using $\mathbb{N}_1$ sizes in the model.

The proof features of H4 mark the addition of sizes within the location calculation as well as the usage of the appropriate lemma. These proof features are very model-specific, they capture important properties involving the `locs_of` function. The expert's usage of *locs_add_size_union* lemma is recorded in two ways (with *Used lemma* and *Lemma shape* features), illustrating how lemma usage could be marked within the ProofProcess framework in general.

Capturing lemma use

The *Used lemma* proof feature records just the *fact* that the particular lemma has been used. This simple feature could be inferred automatically in a number of cases: e.g. by parsing the command parameters, which indicate lemmas used by the tactic, extracting from the simplifier trace, etc.

Automatic tactics (e.g. `simp`, Sledgehammer) may use a larger number of lemmas in a single step. Some filtering may be necessary when automatically inferring the *important* used lemmas from the used lemma set. Model-specific used lemmas may be better candidates for selecting automatically rather than general ones from the Isabelle/HOL library. Furthermore, the overhead of simply selecting an important lemma manually is low. An expert could quickly select these features, especially given a pre-filtered list of lemmas used in the proof step.

Basic recording of used lemma names reduces the initial proof capture overhead, but defers the analysis of why and how the lemmas were used. There are a number of opportunities to automate this analysis, e.g. try to infer the important properties of the lemma by machine-learning all of its uses, analyse differences when disparate lemmas are used for similar goals, etc. Furthermore, if a manual strategy replay *by analogy* is opted for, a user would see that a particular lemma was previously used with the strategy. He may quickly realise whether the lemma can be reused, adapted or a similar lemma generated for the strategy replay.

The *Lemma shape* proof feature describes the particulars of the used lemma that were important to the proof step. In H4 this proof feature records that a

lemma is available to partition the set of the region's locations. It is deliberately vague: a region of a certain added size (to match the goal) is equal to a union of some other regions. It marks the *general* idea of this proof step: partitioning a region into its constituents. When reusing this strategy, the required lemma may not match exactly and the partitioning may need to be adjusted for the proof to succeed. Having a quite general lemma shape allows the proof strategy to search for matching lemmas or even generate new ones according to the captured shape.

Marking the *Lemma shape* proof features can incur a significant overhead to proof process capture. The expert needs to identify how the lemma represents the general idea, what shape may be the most useful. A lemma shape that is too specific (i.e. "we need exactly this lemma") is not useful: *Used lemma* proof features should be used to signal that the same lemma is needed. *Lemma shape* features are better suited when *similar* lemmas are anticipated for strategy reuse.

The *Lemma shape* features are important for automatic strategy reuse, especially when *similar* (*vs.* exactly the same) lemmas are needed in similar proofs. They can be used to guide lemma discovery to accommodate different datatypes (e.g. the different representation of heap regions may require a different way of partitioning the locations set). The important lemma shapes can be marked manually by the user or be the result of additional automatic analysis of the *Used lemma* proof features as described earlier.

From the proof features and intent in H4, the expert's strategy could be described as follows: "if there is a heap region of a compounded size, use a lemma to split its set of locations into two". The size shape requirement could be generalised: e.g. it may be enough to record that if there is a lemma that can split a region into two, use it. However, there may be too many lemmas matching this general description and the strategy would match the goal too frequently.

Proving lemma side-conditions

The proof command in H4 applies the lemma *locs_add_size_union* using the `subst` tactic. The tactic replaces the merged region with the union of *disposed* and *above* region locations. However, `subst` tactic does not simplify the lemma assumptions automatically: each becomes a new sub-goal after the proof step. Thus there are three separate *out* goals after executing the proof step. Since the proof assumptions do not change and each sub-goal gets the same original ones, step H4 uses an `?assms` variable instead for legibility.

Intent: **Nat1 typing** *ProofSeq* (as decoration)

Narrative: Show that the type of map application is correct, particularly that it is `nat1` (not supported directly by Isabelle).

In features:

- *Goal shape* (`nat1 (the (?f ?e))`)
- *Assumption shape* (`nat1_map ?f`)
- *Assumption shape* (`?e ∈ dom f`)
- *Used lemma* (`nat1_map_def`)

In goals:

```
1. disjoint (locs_of d s) (locs f) ⟹
   d + s ∈ dom f ⟹
   nat1 s ⟹
   Disjoint f ⟹
   sep f ⟹
   nat1_map f ⟹
   finite (dom f) ⟹
   nat1 (the (f (d + s)))
```

Proof step: `apply (simp add: nat1_map_def)...`

Out goals: ✓ *(none)*

Step H5: **Nat1 typing**

Each sub-goal is proved separately: lemma side-conditions are finished by single proof steps H5 and H6; whereas the main proof continues in a separate branch H7 (see Figure 11.5 for illustration). Even though the Isabelle proof script handles the goals in a certain order, the ProofProcess framework discards the order of the proof branches. Reordering of proof steps that affect different sub-goals does not change the tree representation.

The **Nat1 typing** proof step (H5) proves the side condition of applying lemma *locs_add_size_union*. It requires that the *after* region's length is of correct type, i.e. $f(d+s) \in \mathbb{N}_1$.[12] This can be readily inferred from the definition of free locations map $f : Loc \xrightarrow{m} \mathbb{N}_1$,[13] which ensures that all range elements in this map are of $\mathbb{N}_1$ type. Since the *above* region already belongs to the map (`d + s ∈ dom f`), the goal follows. However, automatic unfolding of `nat1_map` definition is disabled to prevent the prover from applying it too eagerly. Thus the proof step uses the simplifier tactic with `nat1_map_def` added explicitly, discharging the goal completely.

[12] This is the second sub-goal of **Split contiguous *locs* regions** (H4) proof step.

[13] The $Loc \xrightarrow{m} \mathbb{N}_1$ map of free heap locations is encoded as the `nat1_map` function.

Intent: Trivial assumption *ProofSeq* (as decoration)

In features:

- *Goal shape* (`?g`)
- *Assumption shape* (`?g`) — The goal is among the assumptions.

In goals (filtered):

```
1. ?p1 ⟹ nat1 s ⟹ ?p2 ⟹ nat1 s
```

Proof step: `apply assumption ...`

Out goals: ✓ *(none)*

Step H6: **Trivial assumption**

Since this proof branch is complete, the proof step has no *out* goals.

The proof features in H5 capture the expert's strategy of proving the type of map application. In the specific case of $\mathbb{N}_1$ typing, a direct strategy would prove that map application is of the `nat1` type by using the lemma `nat1_map_def`. To narrow the strategy matching, the proof features also require the goal element to belong to the `nat1_map` map already. A more sophisticated strategy extraction could generalise on the types and lemmas used to provide a generic "type of map application" strategy. For example, the `nat1`, `nat1_map` and `nat1_map_def` definitions could be replaced with corresponding ones of some *placeholder* type, to be instantiated to a concrete type during strategy replay.

The other side-condition of lemma application is trivial.[14] The goal (to show that the *disposed* region's size is `nat1 s`) is already given among the assumptions. This goal is discharged by the **Trivial assumption** proof step (H6), using the `assumption` tactic in Isabelle/HOL. This tactic is generic and the proof features in H6 capture its requirements: the goal must appear among its assumptions.

Partitioning disjointness proof into regions

With lemma side-conditions dealt in separate branches, the main proof continues by partitioning the problem into two subgoals, one for each region (*disposed* and *above*). The previous proof step has taken apart the locations of each region into a union-set. Now the **Split disjointness** proof step can partition the disjointness proof of this union into proving disjointness of each region separately. The captured proof process details are listed in step H7.

[14] This is the first sub-goal of the **Split contiguous *locs* regions** (H4) proof step.

Intent: Split disjointness *ProofSeq* (as decoration)

Narrative: Split the set union within `disjoint()` into separate cases.

In features:

- *Goal shape* (`disjoint (?s1 ∪ ?s2) ?s3`) — One of the disjoint arguments is a set union.
- *Used lemma* (`disjoint_union`)
- *Lemma shape* (

  `?P1(?s1) ⟹ ?P2(?s2) ⟹ disjoint (?s1 ∪ ?s2) ?s3`)

In goals:

```
1. disjoint (locs_of d s) (locs f) ⟹
   d + s ∈ dom f ⟹
   nat1 s ⟹
   Disjoint f ⟹
   sep f ⟹
   nat1_map f ⟹
   finite (dom f) ⟹
   disjoint (locs_of d s ∪ locs_of (d + s) (the (f (d + s))))
            (locs ({d + s} ⩤ f))
```

Proof step: `apply (rule disjoint_union)` ...

Out goals (filtered)[a]**:**

```
1. ?assms ⟹ disjoint (locs_of d s) (locs ({d + s} ⩤ f))
2. ?assms ⟹
     disjoint (locs_of (d + s) (the (f (d + s)))) (locs ({d + s} ⩤ f))
```

[a] `?assms` variable denotes the same assumptions as *in* goals.

Step H7: **Split disjointness**

```
lemma disjoint_union:
 "disjoint s1 s3 ⟹ disjoint s2 s3 ⟹ disjoint (s1 ∪ s2) s3"
```

Figure 11.8: Lemma *disjoint_union*.

The partitioning of disjointness proof is performed using the *disjoint_union* lemma (Figure 11.8). It states that if two sets are individually disjoint from the third one, their union is also disjoint. The **Split disjointness** proof step applies this lemma as a backward proof step using the `rule` tactic. Each lemma assumption becomes an individual sub-goal. Proving each sub-goal is not trivial: Figure 11.5 shows captured proof sub-trees of each sub-goal branch.

The captured proof process details of this proof step are analogous to the previous lemma application proof step H4. The proof features include the shape of the

goal[15] as well as the used lemma features. The *Lemma shape* proof features tries to generalise the shape to allow for different assumptions, while still narrowing where the strategy is to be applied: i.e. on disjointness of set-union.

11.2.3 Proof process branch structuring

The branching proof structure is represented using *ProofParallel* tree elements, which group independent proof branches. The grouping allows satisfaction of a ProofProcess tree invariant (Section 4.3.6), which requires that all *out* goals of a proof step are matched by the *in* goals of the following step. *ProofParallel* ensures the correct "plumbing" by collecting the *in* goals of its branches to match the multiple *out* goals of the previous proof step.

All sub-goals encountered in the proof of *dispose*1*_disjoint_above* are proved independently,[16] their proofs comprising separate branches. Sibling branches can be grouped further for descriptive purposes using nested *ProofParallel* elements. For example, the '*ProofParallel*: **Discharge lemma assumptions**' element groups the side-condition branches together. This separates the "main goal" from the lemma side-conditions in the proof process tree.

Figure 11.5 illustrates the proof process branches graphically, but omits some purely structural *ProofSeq* and *ProofParallel* elements for legibility. An accurate structural overview of the captured ProofProcess tree elements is instead available in Figure 11.9. The figure uses parentheses to provide a compact representation for *ProofEntry* elements nested within their parent *ProofSeq* elements.

Structural elements without metadata

Some of the captured proof process tree elements are purely structural, necessary to construct a well-formed ProofProcess tree structure. For example, a sequence of proof steps is always contained within a *ProofSeq* element; whereas independent branches handling different sub-goals are collected within a *ProofParallel* element. These elements may have empty descriptions with no proof intent or other meta-data (marked as '**-**' in Figure 11.9).

[15] Application of `rule` tactic requires the goal to match at the top level, while the earlier `subst` tactic could perform substitutions *within* goal terms, hence the difference between *Goal shape* feature here and *Has shape* earlier.

[16] In general, the ProofProcess structure allows a proof step to transform ("handle") multiple goals at once. For example, an application of `auto` tactic in Isabelle/HOL transforms all open goals.

ProofSeq: -

- *ProofSeq*: Zoom
  - *ProofSeq*: Expand definition (*ProofEntry*: `unfold F1_inv_def`)
  - *ProofSeq*: Cleanup (*ProofEntry*: `elim conjE`)
- *ProofSeq*: Split contiguous *locs* regions (*ProofEntry*: `subst locs_add_size_union`)
- *ProofParallel*: -
  - *ProofParallel*: Discharge lemma assumptions
    - *ProofSeq*: Nat1 typing (*ProofEntry*: `simp add: nat1_map_def`)
    - *ProofSeq*: Trivial assumption (*ProofEntry*: `assumption`)
  - *ProofSeq*: -
    - *ProofSeq*: Split disjointness (*ProofEntry*: `rule disjoint_union`)
    - *ProofParallel*: Show disjointness separately
      - *ProofSeq*: Show disjointness of disposed region ...
      - *ProofSeq*: Show disjointness of above region ...

Figure 11.9: Partial ProofProcess tree structure of *dispose1_disjoint_above*.

Tagging a proof process structure with proof *intents* and *features* assists with strategy extraction, provides appropriate description, etc. However, doing this can create a significant overhead (*naming things* is a very hard problem!) and thus should be used where beneficial, rather than exhaustively. For example, the root '*ProofSeq*: **-**' element in Figure 11.9 has empty metadata. The expert chose not to provide an abstraction for the whole proof, as he had deemed it to be too general (e.g. along the lines of **Prove this lemma**). Such intent is not very useful: even at the most abstract level it is easier to consider the proof as a sequence of steps. On the other hand, one could argue that a root-level intent could tag the overall strategy of this proof: e.g. **Prove by partitioning heap regions**. If such a strategy appeared within some larger proof, it may be useful to have a name for it.

A similar approach is taken when annotating *ProofParallel* elements: proof metadata is recorded if the branching itself constitutes an important proof step.

Otherwise, it is better to tag its branches directly. For example, '*ProofParallel*: **-**' after the **Split contiguous *locs* regions** step (Figure 11.9) diverts the interest to its branches, which have detailed descriptions. A suitable proof intent here would go along the lines of **Discharge side-conditions and continue partitioning**, which is not very useful. On the other hand, '*ProofParallel*: **Show disjointness separately**' provides an appropriate high-level name for the last part of the proof.

Exporting branches out of a *ProofParallel* element

The basic ProofProcess tree structure uses self-contained proof steps (trees). A proof tree element at any level is by itself a full proof tree, as if its goals were initial goals in the proof. However, such a structure can sometimes interfere with marking high-level proof steps. For example, the proof of *dispose*1*_disjoint_above* could be described as a sequence of the following high-level steps:

1. **Zoom**
2. **Partition into regions**
3. **Show disjointness separately**

The **Partition into regions** proof step would encompass the lower-level **Split contiguous *locs* regions** and **Split disjointness** proof steps. Unfortunately, the captured proof process structure as depicted in Figure 11.9 would not allow such grouping. The **Split disjointness** step is nested *within* a *ProofParallel* element: grouping it would include its branches wholly within the abstract step (see Figure 11.10 for illustration).

To circumvent the *ProofParallel* nesting when grouping proof steps (and support advanced proof process structures), the branch goals can be "exported" outside the *ProofParallel* element. Any *out* goals of an unfinished branch in a *ProofParallel* are treated as *out* goals of the whole *ProofParallel* element. This can then be followed by the next proof step, handling the exported goals.

Figure 11.11 outlines this alternative structure of the proof: the *ProofParallel* of lemma side-conditions (H5 and H6) is now treated as part of the **Split contiguous *locs* regions** proof step. The "main goal" branch, however, is not finished within the *ProofParallel* element and is therefore exported outside it. The **Split disjointness** step can follow the *ProofParallel* and allow the desired grouping.

To support complex ProofProcess tree (graph) structures, the exported branches must end with a *ProofId* element. It records the actual *ProofEntry* element that

ProofSeq: -

- *ProofSeq*: Zoom …
- *ProofSeq*: **Partition into regions**
  - *ProofSeq*: **Split contiguous *locs* regions** …
  - *ProofParallel*: -
    - *ProofParallel*: Discharge lemma assumptions …
    - *ProofSeq*: -
      - *ProofSeq*: **Split disjointness** …
      - *ProofParallel*: Show disjointness separately …

Figure 11.10: Grouping proof steps within *ProofParallel*.

ProofSeq: Prove by partitioning heap regions

- *ProofSeq*: Zoom …
- *ProofSeq*: Partition into regions
  - *ProofSeq*: Split contiguous *locs* regions
    - *ProofEntry*: `subst locs_add_size_union`
    - *ProofParallel*: Discharge lemma assumptions
      - *ProofSeq*: Nat1 typing (*ProofEntry*: `simp add: nat1_map_def`)
      - *ProofSeq*: Trivial assumption (*ProofEntry*: `assumption`)
      - *ProofId*: ⇝ *ProofEntry*: `rule disjoint_union`
  - *ProofSeq*: Split disjointness
    (*ProofEntry*: `rule disjoint_union`)
- *ProofParallel*: Show disjointness separately
  - *ProofSeq*: Show disjointness of disposed region …
  - *ProofSeq*: Show disjointness of above region …

Figure 11.11: Alternative ProofProcess tree structure of *dispose1_disjoint_above* with exported branch and *ProofId* element to accommodate advanced grouping.

Intent: **Show disjointness of disposed region** *ProofSeq* (as decoration)
Narrative: Show that the disposed region is disjoint from the untouched heap regions.
Children:

- (H9) *ProofSeq*: **Show subset of disjoint is disjoint** ...

In goals: Same as in step H9.
Out goals: ✓ *(none)*

Step H8: **Show disjointness of disposed region**

"continues" the branch outside the *ProofParallel*. In contrast to other proof tree elements, the *ProofId* only references the *ProofEntry* but does not "contain" it. The referenced element is contained within the next proof step, be it the *ProofEntry* itself, *ProofSeq* or a new *ProofParallel*. This breaks the strict tree structure, but enables complex proof process structures, including merges of separate branches, arbitrary handling of multiple sub-goals, etc. The ProofProcess system utilises *ProofId* elements to support conversion between a tree structure and directed graph representation used for attempt matching (see Sections 4.3.7 and 8.6 for details).

11.2.4 Finishing the proof for each region

After partitioning the disjointness goal, the proof of *dispose*1*_disjoint_above* continues in two separate branches, each proving disjointness of one of the regions: *disposed* and *above*. The goal for each region requires a different strategy: a quick overview of the captured proof process information for these remaining proof steps is presented below.

Different names for the same proof step

The overall abstract proof step encapsulating both proof branches records its intent as **Show disjointness separately** (see Figure 11.9). It is a *ProofParallel* step, containing both branches as high-level proof steps: **Show disjointness of disposed region** (step H8) and **Show disjointness of above region** (step H12). This provides a high-level model-specific description of the proof process. The proof process capture approach proposed in this thesis encourages recording both the generic and the model-specific proof strategies. The former represent general approaches to doing interactive proof and involve mathematical concepts, whereas the latter describe how particular problems about the specification are tackled.

Intent: Show subset of disjoint is disjoint *ProofSeq*

Narrative: Show that if a set is disjoint from something, its subset is also disjoint.

In features:

- *Assumption shape* (`disjoint ?s1 ?s2`)
- *Goal shape* (`disjoint ?s1 ?s3`)

Children:

- (H10) *ProofEntry*: `erule disjoint_subset ...`
- (H11) *ProofEntry*: `erule locs_ar_subset ...`

In goals (flattened):

```
1. disjoint (locs_of d s) (locs f) ⟹
   d + s ∈ dom f ⟹
   nat1 s ⟹
   Disjoint f ⟹
   sep f ⟹
   nat1_map f ⟹
   finite (dom f) ⟹
   disjoint (locs_of d s) (locs ({d + s} ⩤ f))
```

Out goals: ✓ *(none)*

Step H9: **Show subset of disjoint is disjoint**

However, the strategies of proving disjointness of each region are quite general (they are reused for proofs of "sibling" lemmas in Sections 11.3–11.4). Therefore, each proof branch is tagged with additional proof *intent* and *features*, this time focusing on the names and properties of the *generic* strategy.

The proof of **Show disjointness of disposed region** makes use of the precondition that the removed region is disjoint from the original heap map. The strategy proves that even when some region is removed from the heap map (the map becomes a subset of the original), the disjointness still holds. The expert names this strategy **Show subset of disjoint is disjoint**: its details are listed in step H9.

This abstract strategy proof step is "wrapped" by the **Show disjointness of disposed region** element (see step H8), which thus becomes its *decoration*. By stacking multiple *ProofSeq* elements as *decorations* (each *ProofSeq* has only a single child), the structure allows providing different proof process information for the same proof step at any level of abstraction. Thus one can simultaneously tag a proof tree with proof-specific description as well as generic strategy metadata. The goal *flattening* means that all these elements would share the *in* and *out* goals.

Under-specified proof features

The proof features in the abstract proof step **Show subset of disjoint is disjoint** (H9) are quite vague: they link disjointness of a set in the assumptions and the goal. The proof features do not talk about `?s3` being a subset of `?s2`, which is the actual requirement of the proof strategy. In general, proving that a set is a subset of another is a proof task and it should not be part of the proof feature matching. It may require multiple proof steps, additional lemmas, etc. For this reason, the proof features under-specify the matching, only requiring disjointness with the same set `?s1`. The AI4FM system is not aiming to be a theorem prover, thus the strategy suggestion does not always have to be correct. Proof feature matching is intended to narrow the search space in automatic replay—or suggest *possible* strategies when the user selects suggested strategies manually. The AI4FM system relies on the underlying theorem prover to verify strategy application. For example, if a user selects a strategy that fails, he may need to adjust the strategy, add missing lemmas or select a different strategy altogether.

However, if the expert requires the marking of the subset relationship as an important feature, the ProofProcess framework allows doing that with *custom proof features* (Section 4.2.1). For example, the proof feature could be *Subset* (`?s3`, `?s2`), indicating that a subset relation should exist between the sets, i.e. `?s3` $\subseteq$ `?s2`. However, using this proof feature automatically for strategy replay would require that the ProofProcess system supports such a proof feature type: i.e. there is some implementation of *Subset()* feature checking. Otherwise (which is the general case for custom proof features), such a proof feature is uninterpreted and its name, parameters and the parameter order are arbitrary. Nevertheless, the custom features would still be useful for manual replay: the user would see that the strategy required a "subset" relation between the sets in the previous proof. He would able to comprehend that a subset relation is required and may already know whether it holds and hence whether the strategy is applicable.

Proving a disjoint subset

The actual proof of **Show subset of disjoint is disjoint** (H9) consists of two steps in Isabelle/HOL. They are no longer wrapped into individual *ProofSeq* elements to indicate some higher-level steps. The expert has chosen **Show subset of disjoint is disjoint** as the lower-level abstract proof step, which is realised in two Isabelle steps (captured as *ProofEntry* elements). The proof steps and their goal transformation

Intent: Tactic application *ProofEntry*

In goals (filtered):

```
1. disjoint (locs_of d s) (locs f) ⟹ ?p1 ⟹
   disjoint (locs_of d s) (locs ({d + s} ⩤ f))
```

Proof step: `apply (erule disjoint_subset) ...`

Out goals (filtered):

```
1. ?p1 ⟹ locs ({d + s} ⩤ f) ⊆ locs f
```

Step H10: **Tactic application**: `erule disjoint_subset`.

```
lemma disjoint_subset:
 "disjoint s1 s2 ⟹ s3 ⊆ s2 ⟹ disjoint s1 s3"
```

Figure 11.12: Lemma *disjoint_subset*.

Intent: Tactic application *ProofEntry*

In goals (filtered):

```
1. ?p1 ⟹
   nat1_map f ⟹ ?p2 ⟹
   locs ({d + s} ⩤ f) ⊆ locs f
```

Proof step: `apply (erule locs_ar_subset) ...`

Out goals: ✓ *(none)*

Step H11: **Tactic application**: `erule locs_ar_subset`.

are listed as H10 and H11.

The first step (H10) applies lemma *disjoint_subset* (Figure 11.12) as a backwards proof step. The lemma states the main fact of this proof strategy: if a set is disjoint from another, it is disjoint from its subset as well. The *in* goal of step H10 is to show that the disposed region $d \mapsto s$ is disjoint from the map of "untouched" heap regions (i.e. with the *above* region removed). Among the assumptions is the precondition of the *DISPOSE*1 operation: the disposed region is disjoint from the original heap map. By applying the lemma backwards, the goal holds if it can be proven that the "untouched" heap map is a subset of the whole heap map (stated as the *out* goal of step H10). The `erule` tactic in addition to applying the lemma also consumes its matching assumptions: they are removed from the goal.

The last step H11 of the proof branch shows the subset relation between the heap states. Domain subtraction $\lhd$ removes a region from the map but keeps the

```
lemma locs_ar_subset:
 "nat1_map f ⟹ locs (S ⩤ f) ⊆ locs f"
```

Figure 11.13: Lemma *locs_ar_subset*.

Intent: Show disjointness of above region *ProofSeq* (as decoration)
Narrative: Show that the above region (shifted by s) is also disjoint.
Children:

- (H13) *ProofSeq*: **Show removed region is disjoint** ...

In goals: Same as in step H13.
Out goals: ✓ *(none)*

Step H12: **Show disjointness of above region**

rest of the mappings the same. Therefore it removes the region's locations but the other locations are preserved. Hence the set of locations with the region removed is a subset of the full heap map. This goal is actually stated as lemma *locs_ar_subset* (Figure 11.13), which is used to finish the proof in step H11.

Multiple proof intents of this proof branch suggest that an extracted proof strategy could be used in different situations. For example, the user may turn to using it when dealing with similar proofs involving the *disposed* region. On the other hand, the more general strategy is suggested for dealing with disjointness of arbitrary sets. The vague proof features allow the strategy to match whenever set disjointness appears among assumptions and in the goal. In contrast to the previous proof steps, this strategy would be "implemented" by two consecutive Isabelle/HOL proof commands.

Proving disjointness of the *above* region

The proof branch showing disjointness of the *above* region is captured in a similar fashion. The whole branch is tagged with two intents: **Show disjointness of above region** (step H12) and **Show removed region is disjoint** (step H13). As previously, the first intent is used as a description of this part of the proof (also it suggests a strategy when dealing with the *above* region in similar proofs), whereas the second intent marks a more general strategy for proving disjointness.

The goal of this proof branch (see step H13) is to show that the *above* region is disjoint from the rest of the heap (i.e. the original heap locations map with the *above* region removed). The main proof idea here is that there is a map where all

Intent: Show removed region is disjoint *ProofSeq*

Narrative: Show that a removed region is disjoint from the rest of the heap.

In features:

- *Goal shape* (`disjoint (locs_of ?l ?s) (locs ({?l} ⩤ ?f))`)

Children:

- (H14) *ProofSeq*: **Zoom** …
- *ProofParallel*: **-**
  - (H15) *ProofParallel*: **Discharge lemma assumptions**
  - (H16) *ProofSeq*: **Show set remove is disjoint** …

In goals (flattened):

```
1. disjoint (locs_of d s) (locs f) ⟹
   d + s ∈ dom f ⟹
   nat1 s ⟹
   Disjoint f ⟹
   sep f ⟹
   nat1_map f ⟹
   finite (dom f) ⟹
   disjoint (locs_of (d + s) (the (f (d + s)))) (locs ({d + s} ⩤ f))
```

Out goals: ✓ *(none)*

Step H13: **Show removed region is disjoint**

regions are disjoint from one another (there are no overlapping regions). Thus when one region is removed from this map, the locations of remaining regions are still disjoint from the removed region. This idea is captured with the intent **Show removed region is disjoint** and proof features in step H13.

The proof branch consists mainly of two proof steps: **Zoom** (step H14) and **Show set remove is disjoint** (step H16). However lemma application in the **Zoom** step has several side-conditions to prove, resulting in a more complex proof process structure to accommodate these additional proof branches (see Figure 11.5 for an overview illustration).

Zooming to set-level reasoning

The "zooming" step here is quite different from unfolding invariant definitions in the first step of the proof (see step H1). However, the general idea remains the same: the goal is taken to a lower level of discourse, replacing higher level concepts with lower level counterparts. The **Zoom** step H14 moves from map operators (particularly domain subtraction ⩤) to the set-theoretical level. To achieve this,

Intent: Zoom *ProofSeq* (as decoration)

Narrative: Replace domain anti-restriction with set subtraction operation.

In features:

- *Has shape* (`locs (?s1 ⩤ ?f)`
- *Used lemma* (`locs_region_remove`)

Out features:

- *Has shape* (`locs ?f - locs_of ?s1 ?l1`) — Result uses set-theoretical rather than map operations (domain subtraction).
- *No symbol* (⩤)

In goals (filtered)[a]**:**

```
1. ?assms ⟹ disjoint (locs_of (d + s) (the (f (d + s))))
                     (locs ({d + s} ⩤ f))
```

Proof step: `apply (subst locs_region_remove)` ...

Out goals (filtered):

```
1. ?assms ⟹ d + s ∈ dom f
2. ?assms ⟹ Disjoint f
3. ?assms ⟹ nat1_map f
4. ?assms ⟹ disjoint (locs_of (d + s) (the (f (d + s))))
                     (locs f - locs_of (d + s) (the (f (d + s))))
```

[a] `?assms` variable denotes the same assumptions as in step H13.

Step H14: **Zoom**

```
lemma locs_region_remove:
 "s ∈ dom f ⟹
  Disjoint f ⟹ nat1_map f ⟹
  locs ({s} ⩤ f) = locs f - locs_of s (the (f s))"
```

Figure 11.14: Lemma *locs_region_remove*.

lemma *locs_region_remove* (Figure 11.14) is used. It replaces the domain subtraction with set removal: the set of locations of a map with a region removed is the same as if the locations of the region were removed from all locations of the map. The lemma has several assumptions: e.g. that the regions do not overlap (are disjoint). All necessary assumptions are already available in the main goal. Set-theoretical formulation of the goal allows using more general lemmas about set disjointness in the last step of this proof (see step H16).

The proof features in step H14 capture that domain subtraction is used when calculating locations as well as the use of lemma *locs_region_remove*. Furthermore,

```
lemma disjoint_diff: "disjoint s2 (s1 - s2)"
```

Figure 11.15: Lemma *disjoint_diff*.

this proof step illustrates one usage of *out* features. They require that after applying this proof step, a set difference operator is used instead of the domain subtraction, which is no longer part of the goal (*No symbol()* feature). To match this strategy, both sets of features should be evaluated: first, the *in* features are matched and if they are successful, a "lookahead" of lemma application and the *out* feature matching needs to be performed.

This proof step can match the goal quite frequently. For example, it would apply to the majority of steps within the proof of lemma *dispose1_disjoint_above*. However, early "zooming" to set-theoretical level may not be desirable, especially where lemmas about higher-level concepts are available. To avoid applying this strategy too often, it could have a low ranking in *score* (see Section 4.2.3). Thus other, more specific, strategies would have a higher ranking and hence priority over this one. Alternatively, if the strategy applies too frequently, the expert may choose to narrow it down. This would require identifying other important features of the goal that indicate such a strategy is needed: e.g. suitable lower-level lemmas being available, manual marking of *Preferred level of discourse* features, etc.

As in step H4, the `subst` tactic applies the lemma without handling its assumptions automatically. These become additional sub-goals in the proof step (see *out* goals in step H14). They are trivial because all assumptions are already available. To differentiate from the main goal, the side-conditions are grouped into a *ProofParallel* element indicating their intent: **Discharge lemma assumptions**. The proof commands are analogous to step H6: `apply assumption` in each case.

Show set remove is disjoint (H16) is the final step in this proof branch. It uses the lemma *disjoint_diff* (Figure 11.15) to prove a simple property about sets: set subtraction partitions the original set, thus the subtracted part and the remainder are disjoint. This lemma matches the goal in full and does not have any assumptions. Applying it with the `rule` tactic completes the goal. This strategy is quite general and can be used when proving disjointness of sets in various proofs.

Intent: Discharge lemma assumptions *ProofParallel*

Narrative: Discharge lemma assumptions introduced by the `subst` tactic.

Children:

- *ProofSeq*: **Trivial assumption** ...
- *ProofSeq*: **Trivial assumption** ...
- *ProofSeq*: **Trivial assumption** ...

In goals (filtered)[a]**:**

```
1. ?assms ⟹ d + s ∈ dom f
2. ?assms ⟹ Disjoint f
3. ?assms ⟹ nat1_map f
```

Out goals: ✓ *(none)*

[a] `?assms` variable denotes the same assumptions as in step H13.

Step H15: **Discharge lemma assumptions**

Intent: Show set remove is disjoint *ProofSeq* (as decoration)

In features:

- *Goal shape* (`disjoint ?s1 (?f - ?s1)`)
- *Used lemma* (`disjoint_diff`)

In goals (filtered):

```
1. ?assms ⟹ disjoint (locs_of (d + s) (the (f (d + s))))
                     (locs f - locs_of (d + s) (the (f (d + s))))
```

Proof step: `apply (rule disjoint_diff)` ...

Out goals: ✓ *(none)*

Step H16: **Show set remove is disjoint**

11.2.5 Reviewing and reusing the captured proof process

This section presents the proof of lemma *dispose1_disjoint_above* and how a description of its proof process is captured and represented. The case study illustrates a number of features of the ProofProcess framework and the ideas presented in this thesis. The proof structure and high-level description serve as guides to how the proof was achieved, removing the requirement of parsing the proof commands and running the theorem prover to traverse the results. A quick glance reveals that the proof is done by first "zooming" into the definitions; then partitioning the problem into proofs about individual regions; and finally proving disjointness of each region (*above* and *disposed*) separately. Furthermore, one can expand

high-level proof steps to take a more detailed view: e.g. to figure out the expert's ideas on proving the disjointness of each region. The marking of important proof features as well as the *filtered* view of the goals cuts through the clutter of industrial style proof and helps in identifying the particulars of the strategy taken to complete the proof. Finally, full recording of goals for each step enables inspection of proof sub-trees as each proof process tree element is self-contained.

The main use of the captured proof process data is for extracting reusable proof strategies—the principal goal of the AI4FM research project. Unfortunately, strategy extraction and replay functionality is not yet available from AI4FM. Thus this case study takes a simplified approach to strategy extraction and replay: it assumes that the user performs a manual interactive proof but utilises the captured proof process data. Thus the strategies are more akin to "the way it was done previously". The user manually generalises or adapts them for similar proofs.

The whole captured proof process can be thought of as a single multi-step strategy. However, its proof steps (including the lower-level ones) can also be considered to be separate strategies. In the case of multi-step strategies, the user could select one and follow its proof step structure as long as the proof features match and the strategy succeeds. For example, when dealing with a lemma similar to *dispose*1*_disjoint_above*, the user could follow the same approach: i.e. "zooming", partitioning the problem into regions, etc. If stuck, he would query for new matching strategies and select another one to follow. Or, if a new proof step is needed, he could resume the earlier strategy after doing this proof (Section 11.3.3 shows how an additional proof step is used in a strategy). A strategy is resumed by matching its remainder as a sub-strategy. Alternatively, strategy matching could be re-evaluated after every step. For example, after doing an initial step of one strategy, another strategy may be a better match. Identifying the best strategy replay approach is among the future research topics within AI4FM.

Although some of the proof steps captured in this case study are somewhat generic, the overall proof process, as expected, is quite specific to the problem. Therefore, the strategies are also quite problem-specific. However, the main idea within AI4FM is to assist with proving *proof families*. A single captured proof process could be enough to reuse in similar proofs within the same family. Such proof families often share the data structures, problem facets and proof approaches. Therefore strategies being too problem-specific is not a disadvantage of the AI4FM approach. To illustrate this, the captured proof process information and strategies are used to prove "sibling" lemmas of *dispose*1*_disjoint_above*. The following

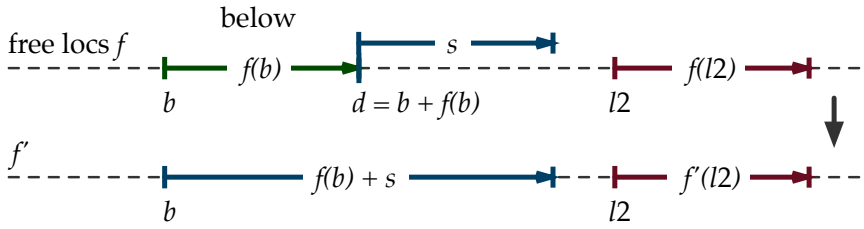


Figure 11.16: *DISPOSE*1(d, s): free region abuts *below* the disposed area only.

Sections 11.3 and 11.4 present these lemmas; and how their proofs are discovered by using the strategies captured in proving *dispose*1*_disjoint_above*.

11.3 Proof process reuse for "below" case

The "below" case of heap memory deallocation operation *DISPOSE*1(d, s) is to some extent a mirror image of the "above" case. It describes the scenario when an existing memory region abuts the disposed region $d \mapsto s$ from *below* (illustrated by Figure 11.16). Using location d as a reference point, the location b of the *below* region's start can be expressed by the formula $b + f(b) = d$: i.e. the *below* region starts at b, has the size $f(b)$ (as queried from the heap map) and ends at location d, where the *disposed* region abuts.

Analogous to the "above" case, the *DISPOSE*1 operation removes the *below* region from the original map f of free heap regions, merges the *below* and *disposed* regions, then adds the merged region back to the heap map. This produces an updated map $f' = (\{b\} \lhd f) \cup \{b \mapsto f(b) + s\}$. Proving the *disjointness* part of the invariant entails showing that the regions within this updated heap map are not overlapping. The proof boils down to demonstrating that the new disposed-and-merged region $b \mapsto f(b) + s$ is disjoint from the "untouched" regions of the original heap map (i.e. f minus the replaced *below* region):

```
disjoint (locs_of b (the (f b) + s)) (locs ({b} ⩤ f))
```

This goal is extracted as lemma *dispose*1*_disjoint_below* (Figure 11.17). The assumptions of this lemma include the definition of the *below* region's location (`b + the (f b) = d`) as well as preconditions of *DISPOSE*1 and the before-state invariant on f. Figure 11.17 also lists the proof script in Isabelle/HOL. This section describes how this proof was discovered via reuse of proof process information

```
lemma dispose1_disjoint_below:
 "F1_inv f ⟹
  disjoint (locs_of d s) (locs f) ⟹
  b ∈ dom f ⟹
  nat1 s ⟹
  b + the (f b) = d ⟹
  disjoint (locs_of b (the (f b) + s)) (locs ({b} ⩤ f))"
```

```
1  apply (unfold F1_inv_def)
2  apply (elim conjE)
3  apply (subst locs_add_size_union)
4  apply (simp add: nat1_map_def)
5  apply assumption
6  apply (rule disjoint_union)
7  apply (subst locs_region_remove)
8  apply assumption
9  apply assumption
10 apply assumption
11 apply (rule disjoint_diff)
12 apply (erule ssubst)
13 apply (erule disjoint_subset)
14 apply (erule locs_ar_subset)
15 done
```

Figure 11.17: Lemma *dispose1_disjoint_below* with proof.

and strategies captured when proving the "above" case (Section 11.4).[17] The final ProofProcess tree structure of the "below" proof is visualised in Figure 11.19. It can be helpful to use it as the guide for the proof description in this section.

11.3.1 Selecting initial strategy

The "mirrored" memory deallocation case involving the *below* region means that the proof goal and the proof itself are very similar to the "above" case. However, small differences and the "mirrored" effect means that a blunt "copy-paste approach" to proof reuse does not work in this case. Instead, the higher-level idea and matching proof branches need to be followed to complete the proof.

Consider that lemma *dispose1_disjoint_below* is being attempted by a user who wishes to use the captured proof process (and possibly the extracted strategies) to help with proving this lemma. This user may be a different one from the expert who has done lemma *dispose1_disjoint_above*. One of the use cases for the AI4FM project is that the "engineers" use the strategies captured from an "expert's" proof process. Thus the user may be unfamiliar with how the proof was done originally.

[17] The presentation of the new proof process is less verbose than in Section 11.4 as there are a lot of similarities.

Furthermore, adapting the proof script to the current lemma (even rearranging appropriate branches) may be not straightforward. Similarly, the original user may have encountered this "sibling" lemma after a significant period of time and cannot easily recall how the proof was done (or easily parse the original proof). Finally, even if the user does recognise the similarities and would know how to complete the proof, using the strategies may be faster (e.g. click to select and run a strategy), involve more automation or simply allow operating at a higher level of abstraction. Whatever the situation, consider that the user seeks assistance from the previously recorded high-level ideas right from the start of the proof.

When faced with a proof goal, the user would select an appropriate strategy from a selection of *matching* ones. While the eventual goal of the AI4FM system is to select and run the strategies automatically, for this exercise consider that the user selects a strategy manually (and adapts it to suit the problem). The selection of matching strategies is provided for the user based on the state of the goal and additionally inferred (or user-marked) proof features.

Zooming as previously

The initial proof step **Zoom** (H1) from the previous proof process technically does not match the bare-bones initial goal of *dispose1_disjoint_below*. The "zooming" strategy relies on meta-information about the goal, namely that `F1_inv` is an invariant function and the preferred level of discourse is lower than the level of datatype invariants. This information is not available in the "vanilla" goal, hence the **Zoom** strategy would not match. To enable matching, the user may manually mark these proof features. However, an inexperienced user may not do that as an initial goal, especially if other proof strategies match outright (see below). Nevertheless, as mentioned in the capture of the previous proof process, some of these features could be inferred automatically. The preferred level of discourse could be decided by comparing definitions in the assumptions and the goal; the tagging of `F1_inv` as an invariant definition could be inherited from the previous proof (as it is the same function) or be done when the specification is constructed.

With the *level of discourse* proof features inferred, the **Zoom** (H1) strategy would match the initial goal. However, the later step **Split contiguous *locs* regions** (H4)[18] or the lower-level "zooming" step **Expand definition** (H2) would also match.

[18] **Zoom** and **Split contiguous *locs* regions** work on different parts of the goal, the former on assumptions, the latter on the goal. In the proof, either could go first: the initial "zooming" is done to avoid expanding definitions on demand for each proof branch.

Selecting any of these proof steps would advance the proof. However, in general, an ordering of these strategies would help the user by suggesting "better" strategies first. The ordering could be machine learned from previous examples or determined using some heuristics. One approach would be to follow the structure of the previously captured proof process.

The priority of **Zoom** over **Expand definition** is suggested by the abstraction relation: "zooming" is the more abstract step, whereas expanding definitions is its constituent. Thus, if a more abstract step is available, it should be taken. If an abstract step fails midway (or altogether), the inner steps could be tried independently to adjust the more abstract strategy or introduce intermediate steps.

Similarly, the captured proof process suggests a precedence relationship between **Zoom** and **Split contiguous *locs* regions**: the latter follows the former in a previous proof. This precedence would be reflected in the matching strategy order: i.e. "**Zoom** should go first because it went first previously".

In addition to matching strategies of proof steps, one could check if a "whole proof strategy" matches: i.e. if there is a previous proof that matches the current one. When capturing the proof of *dispose*1*_disjoint_above*, the expert chose not to name the overall proof strategy or mark any of its features (see Figure 11.9 and the related discussion). However, some of it can be done automatically, e.g. by assigning the intent **Proof of dispose1_disjoint_above**. When the user reuses such a strategy, he is literally "doing the proof as previously". Furthermore, the proof features of this abstract strategy would come via *flattening* of its inner proof steps: it would get the discourse features from the "zooming" and the required goal shape features from region splitting steps. In the end, this strategy would have more matching proof features for *dispose*1*_disjoint_below* than the individual proof steps, making it the most suitable for the initial selection. *Using* this strategy would involve applying the inner proof steps until an actual Isabelle/HOL proof command is resolved. Thus it would start by doing a **Zoom** step, particularly its "implementation": **Expand definition** step and the associated `apply (unfold F1_inv_def)` proof command, followed by the **Cleanup** step (H3).

Capturing strategy application

Both the *dispose*1*_disjoint_above* and *dispose*1*_disjoint_below* lemmas are very similar and have the same `F1_inv f` assumption, thus the high-level **Zoom** proof step succeeds and produces exactly the same results in this proof, i.e. the expansion of

invariant assumptions:

```
1. disjoint (locs_of d s) (locs f) ⟹
   b ∈ dom f ⟹
   nat1 s ⟹
   b + the (f b) = d ⟹
   Disjoint f ⟹
   sep f ⟹
   nat1_map f ⟹
   finite (dom f) ⟹
   disjoint (locs_of b (the (f b) + s)) (locs ({b} ⩤ f))
```

The application of proof strategy and the resulting proof development are again captured by the ProofProcess system.[19] It is recorded as a proof attempt with an appropriate ProofProcess tree structure. However, since the system knows about the strategy being used, it can group proof steps into abstract ones automatically, tag the correct intents and mark the matching proof features. The resulting proof process metadata links the strategies with their *instances* (i.e. the proofs generated by applying the strategies). Furthermore, it improves the descriptive quality of the captured proof process: another user inspecting the proof in the future would benefit from the additional information. Also, the current user gets feedback on how the strategy progresses and is able to review the proof changes easily.

If the "whole proof strategy" was selected initially, the replay would continue by following the proof steps automatically. The correct branches would be resolved by matching corresponding proof features. The automatic strategy replay would try to advance the proof as far as possible. While doing this, it would also recreate the higher-level proof process structure, thus allowing the user to easily orient himself if the automatic strategy application gets stuck. However, for the purposes of describing a more detailed strategy replay in this thesis, consider that the user reins in the automatic application and reviews each larger step.

11.3.2 Following the strategy branches

After the **Zoom** step is applied, only a single proof strategy (from the previous proof) matches the goal: **Split contiguous *locs* regions** (H4). Even if there were more matching proof steps, this strategy would have priority as it directly follows the previously applied proof step (**Zoom**) in an earlier proof.

The *Has shape* (`locs_of ?d (?n + ?m)`) proof feature in step H4 matches the current goal: there is a region with a summed size, where `?n = the (f b)` and

[19] The strategy replay in this section is actually a thought exercise, i.e. a description of scenarios of how such a system *would* work.

`?m = s`. The addition is important as it allows splitting the region in two, as in the previous proof. The *used lemma* proof features also match, since the same lemma *locs_add_size_union* suits the current goal.

Applying the **Split contiguous *locs* regions** strategy splits the merged region into the set-union of each region as previously (see step H4 for details):

```
1. ?assms ⟹ nat1 (the (f b))
2. ?assms ⟹ nat1 s
3. ?assms ⟹ disjoint (locs_of b (the (f b)) ∪ locs_of (b + the (f b)) s)
                     (locs ({b} ⩤ f))
```

The order of side-conditions differs from the previous proof, because the order of operands in the merged region's size is flipped (cf. step H4). Regardless, the branching structure of the captured proof process abstracts over the proof step order and allows correct matching of the branches. Each branch records its own goals and proof features, thus giving rise to self-contained strategies. Furthermore, the *ProofParallel* element collects all branches into a *set*, disregarding the order.

During an automatic strategy replay, a *ProofParallel* element produces a very narrow strategy search scope: if the strategy matches, it is enough to match the branches with the subgoals, otherwise other matching strategies from other proofs could be queried. In a manual setting, the user would be presented with all matching strategies, but the ones corresponding to proof branches within the followed strategy would have priority: e.g. "here is a matching strategy for this branch from the overall previous proof you have been following".

The search within parallel branches for matching strategies produces a correct order of proof steps that Isabelle/HOL expects. The current open sub-goals are handled by doing the **Nat1 typing** (H5) first, followed by **Trivial assumption** (H6) and finally continuing the main proof with the **Split disjointness** (H7) proof branch. The splitting strategy again matches the goal: the goal is concerned with showing disjointness of two sets of region locations joined via set union. The exact same lemma *disjoint_union* as in step H7 applies to this goal thus the proof strategy is replayed successfully, resulting in the following goals:

```
1. ?assms ⟹ disjoint (locs_of b (the (f b))) (locs ({b} ⩤ f))
2. ?assms ⟹ disjoint (locs_of (b + the (f b)) s) (locs ({b} ⩤ f))
```

Up to this point, the replayed strategy has been applied successfully, producing a proof process very similar to the previous one: first zooming into the definitions, then partitioning the problem by splitting adjacent regions into separate goals.

Proving disjointness of each region individually is where the differences between this and the previous proof appear.

11.3.3 Adapting the strategy

Following the strategy, after splitting the disjointness, it forks into separate proofs of disjointness of each region: i.e. **Show disjointness of disposed region** (step H8) and **Show disjointness of above region** (step H12). Even if the user has not been following the strategy application closely, he can use this information to draw a high-level picture of the *current* proof: the remaining goals require showing disjointness of the *below* and the *disposed* regions. The *below* region is easily identifiable in the goal: $b \mapsto f(b)$. The *disposed* region, however, is expressed in the terms of the *below* region's location: $b + f(b) \mapsto s$. This will require an additional proof step (discussed later). Regardless of the different variables and expressions, a high-level description of the previous proof helps the user infer what the current goal predicates represent.

Proving disjointness of the *below* region

Now the user is facing each goal separately. The first goal is concerned with showing disjointness of the *below* region:

```
1. disjoint (locs_of d s) (locs f) ⟹
   b ∈ dom f ⟹
   nat1 s ⟹
   b + the (f b) = d ⟹
   Disjoint f ⟹
   sep f ⟹
   nat1_map f ⟹
   finite (dom f) ⟹
   disjoint (locs_of b (the (f b))) (locs ({b} ⩤ f))
```

The user chooses to continue with the overall strategy of the proof of lemma *dispose1_disjoint_above* and a matching strategy is applied: **Show disjointness of above region** (H12). This strategy matches perfectly: the goal shape is as required, i.e. showing disjointness of a region from a map where the same region is removed (see step H13 for proof feature details). As the strategy application advances the proof, its proof process is captured and inherits details from the strategy: the overall step is tagged **Show disjointness of above region**, with a **Show removed region is disjoint** (H13) step nested within. A small issue is that the strategy

replay names this proof branch "above". The user corrects the discrepancy by manually changing the proof intent to **Show disjointness of below region**.

The proof intents in the ProofProcess framework do not carry semantic information that can be easily understood by the strategy replay algorithms. Tagging the branches "above" or "below" is helpful for human reading, but in general they are just arbitrary names. Strategy replay would just copy the intent, because *that is* the strategy that is actually being replayed. However, after the user corrects the intent, there are now two strategies with the same proof features but different names (intents). To differentiate, the user could add additional proof features: e.g. *Has goal term* (`d + s`) for the "above" case and *Has goal term* (`b`) for the "below" one. These proof features could also be learned afterwards: e.g. by trying to identify differences in goals between the same strategies with different intents. The additional proof features would impact matching if some other region appeared that requires the same strategy (neither "above" nor "below"). However, these strategies could still be suggested as "good enough" due to partial matching.[20]

The remainder of this proof branch is proved by applying the inner proof steps of **Show removed region is disjoint** (H13). The domain subtraction involving the *below* region is replaced by set subtraction in the **Zoom** step (H14), then the lemma side-conditions are discharged trivially and finally **Show set remove is disjoint** (H16) completes the proof branch. The proof details are *exactly the same* as in the previous proof, only the *above* region's location `d + s` is replaced with the *below* region's counterpart `b`. Refer to steps H13–H16 and their descriptions for details.

Identifying strategy misalignment

The second proof branch requires showing disjointness of the *disposed* region:

```
2. disjoint (locs_of d s) (locs f) ⟹
   b ∈ dom f ⟹
   nat1 s ⟹
   b + the (f b) = d ⟹
   Disjoint f ⟹
   sep f ⟹
   nat1_map f ⟹
   finite (dom f) ⟹
   disjoint (locs_of (b + the (f b)) s) (locs ({b} ⩤ f))
```

Unfortunately, the proof strategy extracted from the previous proof does not match the current goal and the strategy replay gets stuck. The proof features in step **Show**

[20] The inner **Show removed region is disjoint** strategy would still match perfectly and the user would be able to finish the proof even if losing the reuse of higher-level description.

subset of disjoint is disjoint (H9) require that the first disjointness argument in the goal (`locs_of (b + the (f b)) s`) has an assumption about its disjointness with another set (in this case a larger set of map locations). However, the assumption here refers to the *disposed* set by using its start location expressed as `d` (i.e. `disjoint (locs_of d s) (locs f)`).

Unfortunately, no other extracted strategies match the goal, thus the user has to perform a manual proof step to continue the proof. The high-level proof information from the previous proof hints at what needs to be done. First, the replay of the overall strategy suggests that this proof branch should be handling the *disposed* region. Thus the user is not puzzled too much by encountering a region with start location of `b + the (f b)` and size `s`: he can recognise it as a different way of defining the *disposed* region. Furthermore, he can inspect what was done in the previous proof and what were the important proof features that enabled the **Show subset of disjoint is disjoint** strategy. He can identify that the obstacle lies with different ways of denoting the *disposed* region: the strategy will match as soon as the start locations `d` and `b + the (f b)` in the assumption and the goal are made the same. The recorded proof features narrow down the work for the user and hint at where the proof could be fixed.

New proof step to align with the strategy

To align the disjointness goal with the assumption, the user notices that there is already the fact `b + the (f b) = d` among the assumptions. Term substitution can be performed to replace `b + the (f b)` with `d` in the goal. This is done in Isabelle/HOL using the `apply (erule ssubst)` proof command, which performs a substitution from the assumptions and "consumes" the assumption fact.

The manual proof step is captured by the ProofProcess framework and the user marks additional metadata to indicate the high level idea. Some of the captured details are presented in steps H17 and H18. First, the user's intent is tagged as **Use below definition** (H17): i.e. he uses the definition of the *below* region (`b + the (f b) = d`) to obtain the disposed location `d`. Furthermore, he notices that the substitution strategy is generic and marks an additional intent and proof features as **Substitute assumption equality** (H18). This way both problem-specific and generic proof step intents are captured. Proof features in step H18 describe the generic strategy: use substitution when the goal contains an expression for which there is an equality fact among the assumptions.

Intent: **Use below definition** *ProofSeq* (as decoration)

Narrative: Use the definition of below (`b`) to get to the disposed location `d`.

Children:

- (H18) *ProofSeq*: **Substitute assumption equality** ...

In goals: Same as in step H18.

Out goals: Same as in step H18.

Step H17: **Use below definition**

Intent: **Substitute assumption equality** *ProofSeq* (as decoration)

Narrative: Substitute equality in assumptions to sub-term in the goal.

In features:

- *Assumption shape* (`?t1 = ?t2`)
- *Has shape* (`?t1`)

In goals (filtered):

```
1. ?p1 ⟹
   b + the (f b) = d ⟹
   ?p2 ⟹
   disjoint (locs_of (b + the (f b)) s) (locs ({b} ⩤ f))
```

Proof step: `apply (erule ssubst)` ...

Out goals (filtered):

```
1. ?p1 ⟹ ?p2 ⟹
   disjoint (locs_of d s) (locs ({b} ⩤ f))
```

Step H18: **Substitute assumption equality**

Continuing the strategy

After performing the substitution, the user gets a familiar goal:

```
1. disjoint (locs_of d s) (locs f) ⟹
   b ∈ dom f ⟹
   nat1 s ⟹
   Disjoint f ⟹
   sep f ⟹
   nat1_map f ⟹
   finite (dom f) ⟹
   disjoint (locs_of d s) (locs ({b} ⩤ f))
```

The goal involves showing disjointness of a subset (locations of `{b} ⩤ f`), when disjointness of a larger set (locations of the full heap `f`) is assumed. This goal is matched by the **Show subset of disjoint is disjoint** (step H9) strategy. Thus the overall strategy of the previous proof is resumed and the proof is completed.

...

ProofParallel: Show disjointness separately

- *ProofSeq*: Show disjointness of below region
  - *ProofSeq*: Show removed region is disjoint ...
- *ProofSeq*: Show disjointness of disposed region
  - *ProofSeq*: Use below definition
    - *ProofSeq*: Substitute assumption equality (*ProofEntry*: `erule ssubst`)
  - *ProofSeq*: Show subset of disjoint is disjoint
    - *ProofEntry*: `erule disjoint_subset`
    - *ProofEntry*: `erule locs_ar_subset`

Figure 11.18: Partial ProofProcess tree structure of *dispose1_disjoint_below*.

On strategy resume, both the problem-specific **Show disjointness of disposed region** (H8) strategy and its nested generic **Show subset of disjoint is disjoint** (step H9) strategy match (they are different names for the same proof branch). The new proof step in the current proof, however, is not generic: the substitution of *below* definition is part of the proof about the heap region. To represent that, the user adjusts the grouping in the ProofProcess structure of this proof: the substitution step precedes the generic step (see Figures 11.18 and 11.19 for illustration).

By adding a new proof step, the user has adapted the proof strategy of showing disjointness of the *disposed* region. The same proof intent indicates that **Show disjointness of disposed region**, as encountered in the current proof, contributes an alternative strategy. Both versions could be generalised into a single strategy with alternative steps: i.e. **Show disjointness of disposed region** can *optionally* perform a substitution of the *below* definition, but always uses the **Show subset of disjoint is disjoint** step to finish. The optional step contributes alternative proof features to the matching and thus the strategy should match both when substitution is needed and not.

Reviewing proof reuse

Reuse of strategies for the proof of lemma *dispose1_disjoint_below* has been highly successful. However, the previous lemma *dispose1_disjoint_above* is extremely

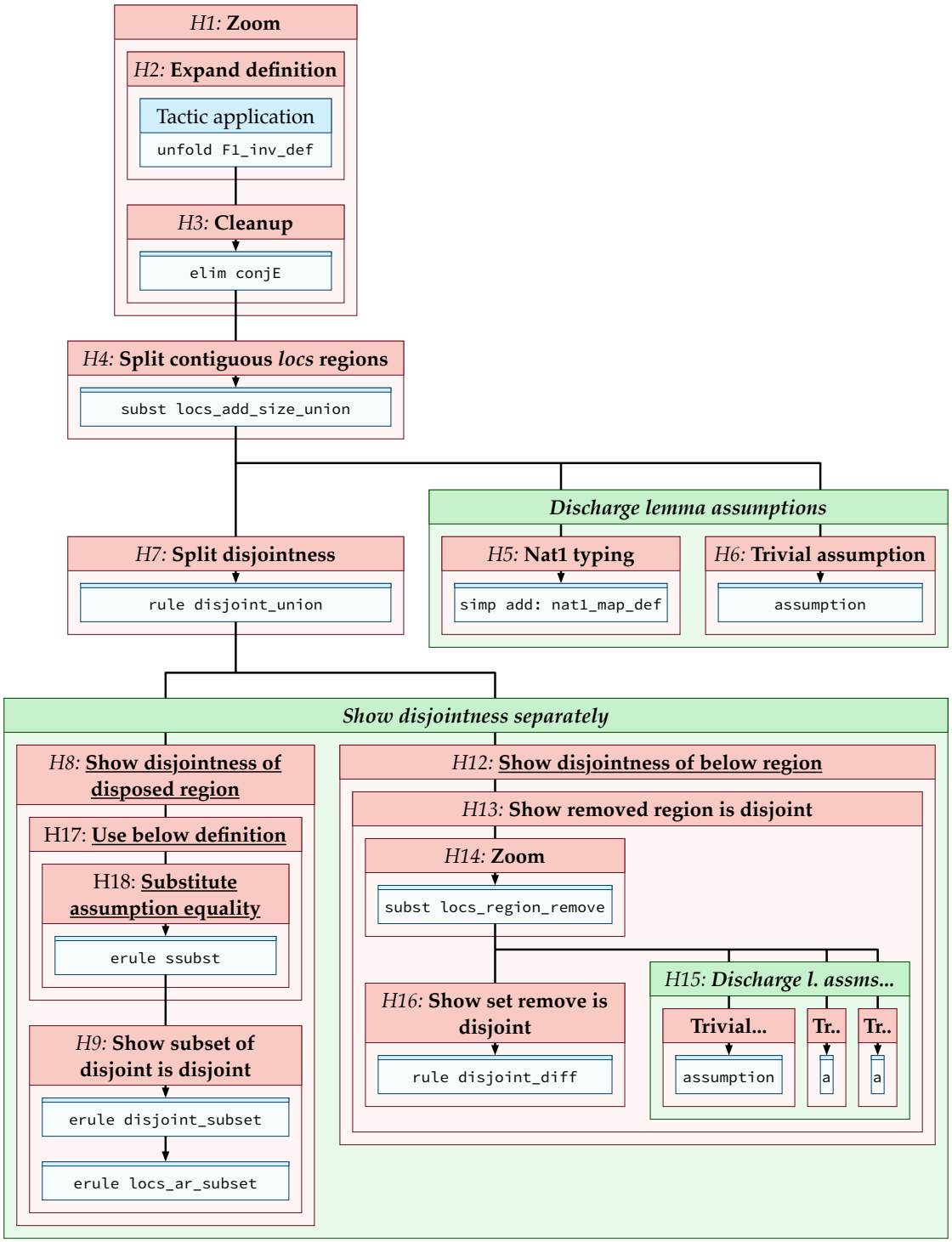


Figure 11.19: Simplified ProofProcess tree of lemma *dispose1_disjoint_below*. The legend is available in Figure 11.5. Underlined intents mark the new/changed proof strategies cf. Figure 11.5, *italic* step numbers (e.g. *H7*) indicate ProofProcess step reuse.

similar, thus a lot of similarity in the proof is expected. Yet this is the goal of AI4FM: help with *similar* proofs by reusing high-level strategies. Figure 11.19 visualises the captured ProofProcess structure of the current proof. In the majority of the proof, due to the fact that proof process branches are unordered, it is almost an exact replica of the previous proof (cf. Figure 11.5). The underlined proof tree elements highlight the changed parts, particularly the additional proof step **Use below definition** within the **Show disjointness of disposed region** proof branch.

The large degree of similarity between the proofs raises the question of whether lemmas could be used to generalise specific parts of the proof. For example, if an expert notices he is repeating proof steps a lot, maybe there is a general lemma that can be introduced to complete the repeating proofs. Producing general lemmas of good quality leads to a better proof library, but doing this is a significant effort and requires good theorem proving and abstraction skills. Therefore the expert would need to spend extra effort to discover the initial proof as well as generalise it for future problems. Furthermore, in industrial settings, the quality of a proof library is not as important as getting the proof done: duplicate applications of proof strategies is an acceptable solution. Finally, the differences in proofs may be minimal but significant enough (and deep in the proof) to prevent reusable general lemmas at the top level. The AI4FM project focuses on reusing the high-level ideas: strategy replay as in the proof of *dispose1_disjoint_below* is an example of the success of this approach.

The proof of *dispose1_disjoint_below* reuses the extracted proof strategies with minimal additional intervention by the user. Its proof is again captured by the ProofProcess framework and contributes additional instances of the strategies as well as generalises some of the existing strategies. The successful proof steps contribute back to the pool of available strategies and can be reused immediately. The following section continues reuse of the proof strategies within the same family of proofs and tackles the "both" case of heap memory deallocation.

11.4 Generalising strategies in "both" case

The final "both" case of $DISPOSE1(d, s)$ describes a scenario when an isolated region of heap memory is wholly deallocated. In this case, free memory regions are both *below* and *above* the disposed region $d \mapsto s$. As a result of the operation, the regions are merged and a single region spanning the widths of all constituent regions

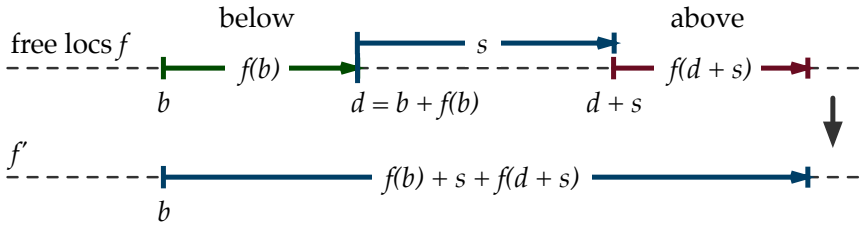


Figure 11.20: $DISPOSE1(d, s)$: free regions abut *below* and *above* the disposed.

replaces them within the map of free heap locations f (illustrated by Figure 11.20). As in the "below" case, the starting location b of the *below* region is expressed using the location d as a reference point: $b + f(b) = d$. Furthermore, the location and size of the *above* region are calculated as $d + s$ and $f(d + s)$, as previously. The merged region starts at location b and spans all three regions: $f(b) + s + f(d + s)$. Finally, the resulting heap map of the $DISPOSE1$ operation in the "both" case is $f' = (\{b, d + s\} \lhd f) \cup \{b \mapsto f(b) + s + f(d + s)\}$: i.e. both affected existing regions (*below* and *above*) are replaced by the merged region.

As previously, this case study focuses on proving disjointness of the resulting map. The crux of this problem is specified as lemma *dispose1_disjoint_both*: showing that the new disposed-and-merged region $b \mapsto f(b) + s + f(d + s)$ is disjoint from the "untouched" regions of the original heap map (f minus the replaced *below* and *above* regions). The lemma and its proof are listed in Figure 11.21.

Dealing with two abutting regions in the lemma results in extra proof effort: the proof is significantly longer than the previous ones (cf. Figures 11.4 and 11.17). This section highlights some facets of how this proof was discovered via reuse of previously captured proof process information. Furthermore, the differences from previous proofs, additional manual proof steps, and how they influence extracted proof strategies, are discussed. Figure 11.22 (and its extensions in Figures 11.24 and 11.25) provides an overview of the final ProofProcess tree structure of this proof and can be helpful when following the discussion in this section.

11.4.1 Partitioning the problem

Lemma *dispose1_disjoint_both* belongs to the same proof family as its "siblings" covering the "above" and "below" cases. This means that the high-level proof ideas from one proof can be reused in another within the same family. Section 11.3 has shown how the proof strategies from the "above" case are reused in proving the

```
lemma dispose1_disjoint_both:
 "F1_inv f ⟹
  disjoint (locs_of d s) (locs f) ⟹
  b ∈ dom f ⟹
  d + s ∈ dom f ⟹
  nat1 s ⟹
  b + the (f b) = d ⟹
  disjoint (locs_of b (the (f b) + s + the (f (d + s))))
           (locs ({b, d + s} ⩤ f))"
```

```
1  apply (unfold F1_inv_def)
2  apply (elim conjE)
3  apply (subst locs_add_size_union)
4  apply (simp add: nat1_map_def)
5  apply (simp add: nat1_map_def)
6  apply (rule disjoint_union)
7  apply (subst locs_add_size_union)
8  apply (simp add: nat1_map_def)
9  apply assumption
10 apply (rule disjoint_union)
11 apply (rule disjoint_locs_widen1)
12 apply assumption
13 apply (subst locs_region_remove)
14 apply assumption
15 apply assumption
16 apply assumption
17 apply (rule disjoint_diff)
18 apply (elim ssubst)
19 apply (erule disjoint_subset)
20 apply (erule locs_ar_subset)
21 apply (subst nat_add_assoc[symmetric])
22 apply (erule ssubst)
23 apply (rule disjoint_locs_widen2)
24 apply assumption
25 apply (subst locs_region_remove)
26 apply assumption
27 apply assumption
28 apply assumption
29 apply (rule disjoint_diff)
30 done
```

Figure 11.21: Lemma *dispose1_disjoint_both* with proof.

"below" case. The strategies have matched well, particularly because both lemmas are almost mirror images of each other. The same approach is taken when proving the current "both" case of disjointness: proof features from extracted strategies are matched with the current goal, then an appropriate strategy is selected from the matching ones. The selection is based on the progress within the overall strategy, the order of strategies in the previous proof, etc.

Initial zooming

The presence of two replaced regions in the "both" case requires additional measures later in the proof (compared with previous proofs). However, the *initial* proof steps follow the same high-level ideas: the extracted strategies match well. The **Zoom** proof step (H1) again starts the proof as it is the initial proof step of both previous proofs (Section 11.3.1 discusses how the initial strategy is selected). This proof step unpacks the assumptions within the `F1_inv` invariant definition.

During the selection of **Zoom** as the initial proof step, the user would see both "above" and "below" overall proofs as matching strategies, among others. However, the actual strategies of each proof are almost exactly the same. To consolidate this information, the user could give them both the same top-level intent: e.g. **Prove disjointness by partitioning regions**, to indicate that it is actually the same strategy. The only difference—names of proof branches **Show disjointness of above region** and **Show disjointness of below region**—would create an alternative choice within the strategy: e.g. after splitting there would be two branches, one for the *disposed* region and one for either the *above* or *below* region. Ascertaining the right balance between ever-generalising proof strategies and recording separate strategies is part of the future work of the AI4FM project.

First split of regions

The next move within the strategy is to partition a region of added lengths into sub-regions. This is realised via two high-level proof steps: **Split contiguous *locs* regions** (step H4) and **Split disjointness** (step H7). The first step matches the disjointness goal of a region with a summed size. It partitions the region at that size point and sets up the proof features for the second step: **Split disjointness**. This step then splits the partitioned region into two sub-goals, each of which requires proof of disjointness of an individual sub-region.

The goal of *dispose1_disjoint_both*, however, involves three regions. The splitting steps partition the full merged region ($b \mapsto f(b) + s + f(d + s)$) into two, i.e. the "bottom" ($b \mapsto f(b) + s$) and "top" ($b + f(b) + s \mapsto f(d + s)$) parts:

```
1. ?assms ⟹ disjoint (locs_of b (the (f b) + s)) (locs ({b, d + s} ⩤ f))
2. ?assms ⟹ disjoint (locs_of (b + (the (f b) + s)) (the (f (d + s))))
                     (locs ({b, d + s} ⩤ f))
```

At this point, however, the overall strategy can no longer be followed. Previously, after partitioning the regions, the branches were identified as the *disposed* region and the other (e.g. *above* or *below*) region. However, in this case, neither strategy matches the goal and the branches cannot be identified automatically. For the *disposed* branch, the strategy would expect a matching assumption for the first disjoint argument (see step H9); or at least that the start location references the disposed region's start d via the definition of *below* (as generalised from step H17). Neither of these match the goals—actually the *disposed* region has not been isolated yet. Furthermore, the *above*/*below* branch does not match either: the heap map removes two regions at once, whereas the strategy **Show removed region is disjoint** (H13) expects just one.

Second split of regions

Nevertheless, even though the overall strategy is stuck, the user can continue by querying for other matching strategies for this goal. The first goal is matched by the **Split contiguous *locs* regions** (step H4) strategy. The goal again contains a region with a summed size that can be partitioned into sub-regions. The user recognises that this whole proof branch still involves a merged region consisting of the *below* and *disposed* regions. Thus the whole proof branch is tagged as **Show disjointness of bottom part** and starts off by partitioning the bottom part's regions. Figure 11.22 shows how two instances of region splitting are performed (with slightly different yet trivial side conditions) to fully partition the problem.

The double splitting in this proof is required to partition three regions into individual branches. In general, this can be extrapolated for even more regions, requiring continued splitting of the merged region until it is fully partitioned. The proof steps involved in the actual splitting represent the same strategy, thus at a high level this can be treated as a *repetition* of the said strategy. The repetitive nature could be captured during strategy extraction, depending on the approach taken. For example, the proof-strategy graphs (PSGraphs) support repetition in strategy definitions (Section 7.3). This would make it possible to explicitly specify

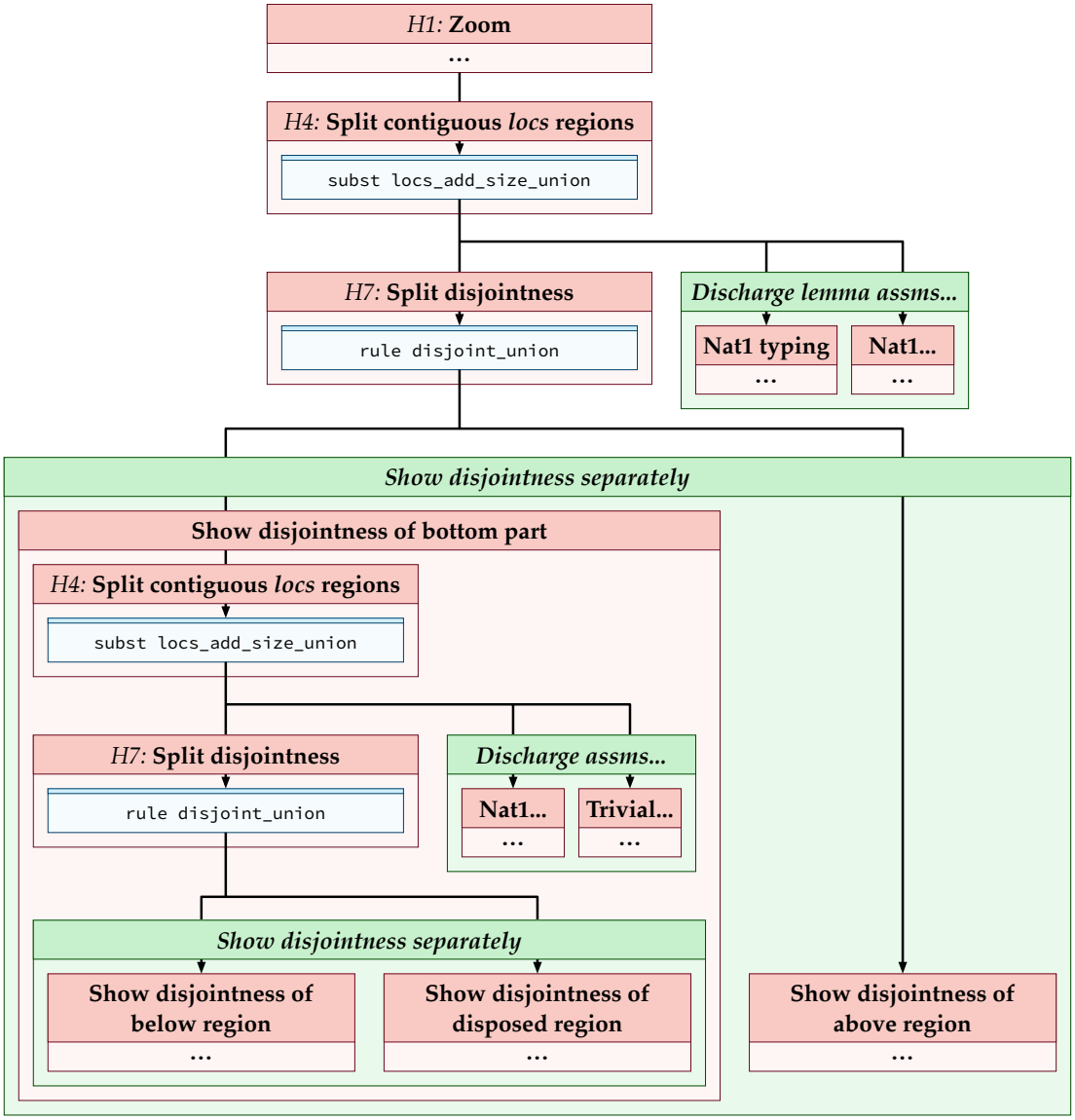


Figure 11.22: Partial ProofProcess tree of lemma *dispose1_disjoint_both*.

This figure shows the top level of the tree, branches are presented separately:

- **Show disjointness of below region** continues in Figure 11.24.
- **Show disjointness of disposed region** is exactly the same as in Figure 11.19.
- **Show disjointness of above region** continues in Figure 11.25.

The legend is available in Figure 11.5.

that a strategy should be tried multiple times: e.g. until the regions can no longer be partitioned. However, this would increase the complexity of proof strategies, especially if the repeating part is difficult to isolate. Thus, alternatively, the approach employed here is viable. The user follows a non-repeating strategy until it gets stuck. Then a new matching strategy is sought, resulting in the same strategy being found and applied again from the start. The separation of concerns between data capture by the ProofProcess framework and its "applications" of strategy extraction and replay allows for different approaches to be explored and implemented.

11.4.2 Extending region strategies

After splitting the bottom part, the user is left with three goals (two new ones and one from the previous split):

```
1. ?assms ⟹ disjoint (locs_of b (the (f b))) (locs ({b, d + s} ◁ f))
2. ?assms ⟹ disjoint (locs_of (b + the (f b)) s) (locs ({b, d + s} ◁ f))
3. ?assms ⟹ disjoint (locs_of (b + (the (f b) + s)) (the (f (d + s))))
                     (locs ({b, d + s} ◁ f))
```

The last goal (from the previous split) has not been affected by the last split thus it is treated as a branch separate from the whole **Show disjointness of bottom part** proof (see Figure 11.22 for illustration).

Reusing proof for the *disposed* region

An existing strategy now matches the second goal: the generalised **Show disjointness of disposed region** strategy can substitute the $b + f(b) = d$ definition using the **Use below definition** strategy (H17) and subsequently prove disjointness of the disposed region $d \mapsto s$ using the **Show subset of disjoint is disjoint** strategy (H9). In fact, the second goal is exactly the same for the purposes of this strategy as the *disposed* region branch in the "below" case (see Section 11.3.3 and Figure 11.19). By selecting the **Show disjointness of disposed region** strategy, the user completes this proof branch and tags it appropriately.

Goal shape mismatch for the *below* region

The other goals are not matched by any existing strategy and the user has to take a manual proof step. For assistance with finding a correct way forward, the user can check what strategy *should* go here compared to previous proofs. Previously, after splitting the regions (and discarding the *disposed* region branch), the remaining

strategy was one of **Show disjointness of below/above region**. A quick glance at the first goal shows that it involves the *below* region $b \mapsto f(b)$. Thus even though the strategy does not match, the user can inspect the proof features of the **Show disjointness of below region** strategy to see why it does not apply.

The proof features (actually specified on the nested **Show removed region is disjoint** (H13) strategy) do not match the current goal because they have been specified too strictly. The required goal shape allows only a single region to be removed from the heap map in the second disjointness argument:

Goal shape (`disjoint (locs_of ?l ?s) (locs ({?l} ⩤ ?f))`).

In the current goal, however, two regions are removed (`{b, d + s} ⩤ f`). Unfortunately, the goal *shape* features in Isabelle/HOL are limited to simple placeholders and use unification to perform matching. It is not possible to describe the condition of "a location must be *among* the regions removed from the heap map" as needed here. Theorem proving would need to be used to show that the location is among the removed regions.

Regardless, even if proof feature expression was not a limiting factor here, handling multiple regions requires additional proof steps within the strategy. Thus as the user adds these manual steps, they are recorded as strategy alternatives and would match subsequent encounters of this new case. Again, the argument of aiming for the most general strategy versus having specific strategies for different cases presents itself. Finding the best approach is left for future research.

Dropping goal argument to align with strategy

By examining the expected proof features, the user can identify that he needs to drop one of the regions from the second disjointness argument. Then he would have a goal that matches the strategy, making it possible to continue the proof as previously. Furthermore, a future AI4FM system could be able to suggest how to perform this by attempting automatic lemma discovery (Section 13.3.3): it is known what shape the goal should be from the proof features as well as what it is at the moment; lemma discovery functionality could generate various permutations and check for counterexamples and proof automatically.

The lemma needed here is *disjoint_locs_widen*1 (Figure 11.23). It shows that the disjointness property holds if an additional region is dropped from the heap map: i.e. if a set is disjoint from the locations of a map containing more regions, it is still

Intent: Widen disjoint *ProofSeq* (as decoration)

Narrative: Widen a disjoint argument by dropping a domain subtraction argument.

In features:

- *Goal shape* (`disjoint ?s (locs ({?p, ?q} ⩤ ?f))`)
- *Subterm has shape* (`?s, ?p`)
- *Used lemma* (`disjoint_locs_widen1`)
- *Lemma shape* (`disjoint ?s (locs ({?p} ⩤ ?f)) ⟹ disjoint ?s (locs ({?p, ?q} ⩤ ?f))`)

In goals (filtered):

```
1. ?assms ⟹ disjoint (locs_of b (the (f b))) (locs ({b, d + s} ⩤ f))
```

Proof step: `apply (rule disjoint_locs_widen1)` ...

Out goals (filtered):

```
1. ?assms ⟹ nat1_map f
2. ?assms ⟹ disjoint (locs_of b (the (f b))) (locs ({b} ⩤ f))
```

Step H19: **Widen disjoint**

```
lemma disjoint_locs_widen1:
 "nat1_map f ⟹
  disjoint s (locs ({p} ⩤ f)) ⟹ disjoint s (locs ({p, q} ⩤ f))"
```

Figure 11.23: Lemma *disjoint_locs_widen1*.

disjoint if some of the regions are removed. The only side condition (`nat1_map f`) describes the type of map `f` and is required for `locs` function to be defined. The number 1 in the lemma name signals that the *first* element in the removed set is retained when applying the lemma backwards.

The user can compare the current goal and the proof features of the expected strategy for hints when creating lemma *disjoint_locs_widen1*. Furthermore, the proof of this lemma can be found automatically using the Sledgehammer tool in Isabelle. Thus the user has a good chance of discovering the necessary manual step and advancing the proof without employing the expert. The application of lemma *disjoint_locs_widen1* in a backward proof step is captured in step H19. The proof step replaces the current goal with the assumptions in the lemma, essentially "dropping" the unnecessary second region from the goal. The side condition is trivial as the assumption `nat1_map f` is already available.

The user tags this proof step as **Widen disjoint** and marks the important proof

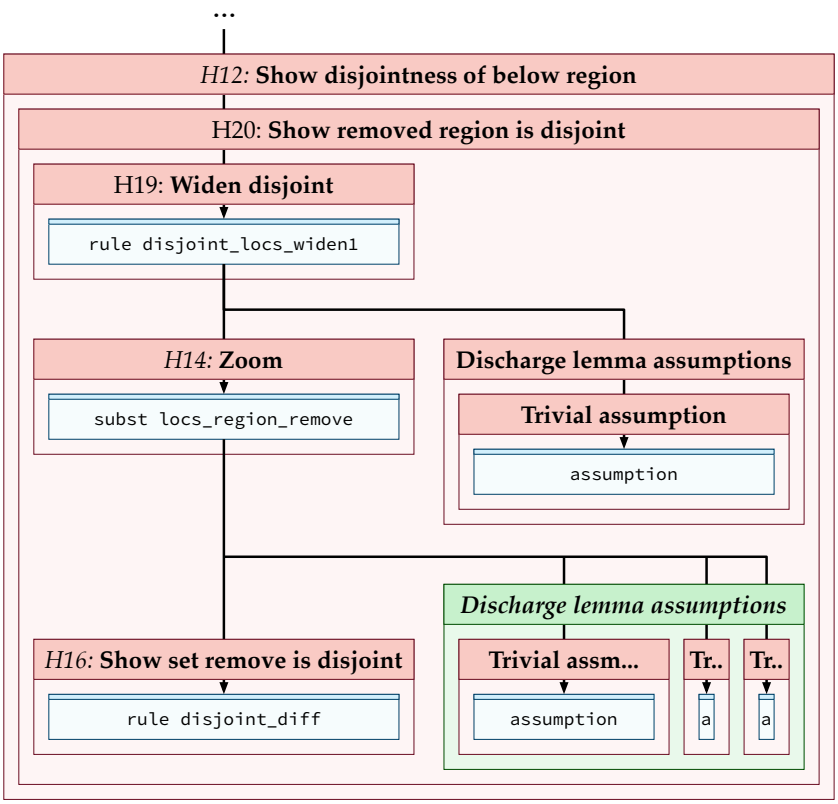


Figure 11.24: ProofProcess tree of "below" branch of lemma *dispose1_disjoint_both*. Continues the main tree from Figure 11.22.

features: i.e. that the goal involves a domain subtraction operation with a two-element set, as well as that the first set element (`?p`) is the important one. The latter is captured by the *Subterm has shape()* proof feature, which requires that `?p` is found within `?s` (the first disjointness argument). Furthermore, the used lemma is also captured, including the important shape.

After dropping the unnecessary domain subtraction argument, the widened disjointness goal matches the **Show removed region is disjoint** (step H13) strategy and the previous **Show disjointness of below region** strategy can be resumed. The reuse of **Show removed region is disjoint** completes the proof.

Adjusting the strategy

Widen disjoint (H19) is a quite generic proof step. The user rearranges the grouping of the captured proof process structure to include it within the generic **Show removed region is disjoint** step (step H20, also see Figure 11.24). The proof features of the extended **Show removed region is disjoint** step are also changed to mark

Intent: Show removed region is disjoint *ProofSeq*

Narrative: Show that a region is disjoint from the rest of the heap with the region itself removed.

In features:

- *Goal shape* (`disjoint (locs_of ?l1 ?s) (locs ({?l1, ?l2} ⩤ ?f))`)

Children:

- (H19) *ProofSeq*: **Widen disjoint** ...
- *ProofParallel*: **-**
  - *ProofSeq*: **Discharge lemma assumptions**
    - *ProofSeq*: **Trivial assumption** ...
  - (Analogous to H13) *ProofSeq*: **-** ...

In goals (flattened):

```
1. disjoint (locs_of d s) (locs f) ⟹
   b ∈ dom f ⟹
   d + s ∈ dom f ⟹
   nat1 s ⟹
   b + the (f b) = d ⟹
   Disjoint f ⟹
   sep f ⟹
   nat1_map f ⟹
   finite (dom f) ⟹
   disjoint (locs_of b (the (f b))) (locs ({b, d + s} ⩤ f))
```

Out goals: ✓ *(none)*

Step H20: **Show removed region is disjoint** (with widening). See Figure 11.24 for illustration. Apart from step H19 and its side-condition proof, the rest of the inner proof steps are analogous to H13.

this specific instance of the strategy: i.e. the removed region is the first one in a two-region removal from the heap map.

During strategy extraction, the new proof feature and an additional **Widen disjoint** step would specify an alternative matching and replay path for the strategy, thus generalising it with an additional case. Addition of more matching cases is not the nicest way of generalising strategies, however, as discussed previously, some proof cases are difficult or impossible to generalise without involving theorem proving. Furthermore, handling several cases may be enough to cover strategy

reuse within the entire family of proofs, thus achieving the initial goal without extracting "perfect" strategies.

By reusing a strategy for the *disposed* region and extending one for the *below* region, the user finishes the proof of each region's branch and therefore the overall branch of the "bottom part" (see Figure 11.22 for illustration).

Aligning parentheses for the *above* region

The remaining proof goal originated in the initial region splitting:

```
1. ?assms ⟹ disjoint (locs_of (b + (the (f b) + s)) (the (f (d + s))))
                     (locs ({b, d + s} ⩤ f))
```

By the process of elimination, the user can deduce that this proof branch is concerned with the *above* region, since disjointness of both *below* and *disposed* regions have been proved. However, its start location is defined in terms of the *below* region ($b + f(b) + s$), which complicates the proof and requires manual proof steps.

Substituting part of this expression with the assumption $b + f(b) = d$ has already been done in the "below" case (see step **Use below definition** (H17)), however the strategy does not match here immediately due to misaligned parentheses. To enable substitution, the parentheses around the addition in the start location expression ($b + (f(b) + s)$) need to be changed to match the expression in the assumption (i.e. to $(b + f(b)) + s$). This is performed by using the associativity of the addition operator: in Isabelle/HOL the addition associativity of natural numbers is available as lemma `nat_add_assoc[symmetric]`.[21] The captured proof process details about this step are listed in H21.

The proof features of step H21 capture the mismatch between the goal and the assumption. This allows stricter control on where the strategy would match (i.e. "only perform associativity substitution if there is an assumption that can be utilised after the associativity is performed") rather than marking a generic associativity proof step that would apply too frequently and unnecessarily. The specific aim to match the assumption is also captured by the intent **Get assumption shape**. The fact that associativity is actually used is captured by the *Lemma shape()* proof feature, which specifies the associative nature of the addition operator.

Strategy extraction could generate different reusable strategies from this proof step. The application of associativity of addition on natural numbers can be extracted directly from this problem. However, if the extraction substituted the

[21] The `[symmetric]` attribute gives a symmetric form of the associativity lemma, which is needed for the desired addition shape.

Intent: Get assumption shape *ProofSeq* (as decoration)

Narrative: Transform the goal to match the shape in assumption.

In features:

- *Assumption shape* (`?m + ?n = ?x`)
- *Has shape* (`?m + (?n + ?k)`)
- *Lemma shape* (`?m + (?n + ?k) = (?m + ?n) + ?k`)

In goals (filtered):

```
1. ?p1 ⟹ b + the (f b) = d ⟹ ?p2 ⟹
   disjoint (locs_of (b + (the (f b) + s)) (the (f (d + s))))
            (locs ({b, d + s} ⩤ f))
```

Proof step: `apply (subst nat_add_assoc[symmetric]) ...`

Out goals (filtered):

```
1. ?p1 ⟹ b + the (f b) = d ⟹ ?p2 ⟹
   disjoint (locs_of (b + the (f b) + s) (the (f (d + s))))
            (locs ({b, d + s} ⩤ f))
```

Step H21: **Get assumption shape**

addition operator with a different one, this would result in a different strategy. For example, by replacing addition with multiplication, this strategy could be reused in a quite different proof, where multiplication associativity is required.

11.4.3 Adapting strategy from the same proof

After correcting the placement of parentheses, the **Use below definition** (H17) strategy matches the remaining goal. After replaying this strategy, the *above* region becomes defined in the familiar manner, using the $d + s$ start location:

```
1. ?assms ⟹ disjoint (locs_of (d + s) (the (f (d + s))))
                     (locs ({b, d + s} ⩤ f))
```

The rest of the proof goes analogously to the **Show removed region is disjoint** (H20) strategy. It first drops the unnecessary region from the heap map's domain subtraction (thus widening the argument). Then the proof continues to show that this region is disjoint from the other regions. Figure 11.25 illustrates the captured ProofProcess structure of the *above* region's branch.

However, unlike step H20 (particularly its inner step **Widen disjoint** (H19)), the important region to retain is now the *second* element in the domain subtraction set. The user needs to adjust the strategy to match the current goal. As a straightforward approach, the user can inspect how the proof was done previously and try

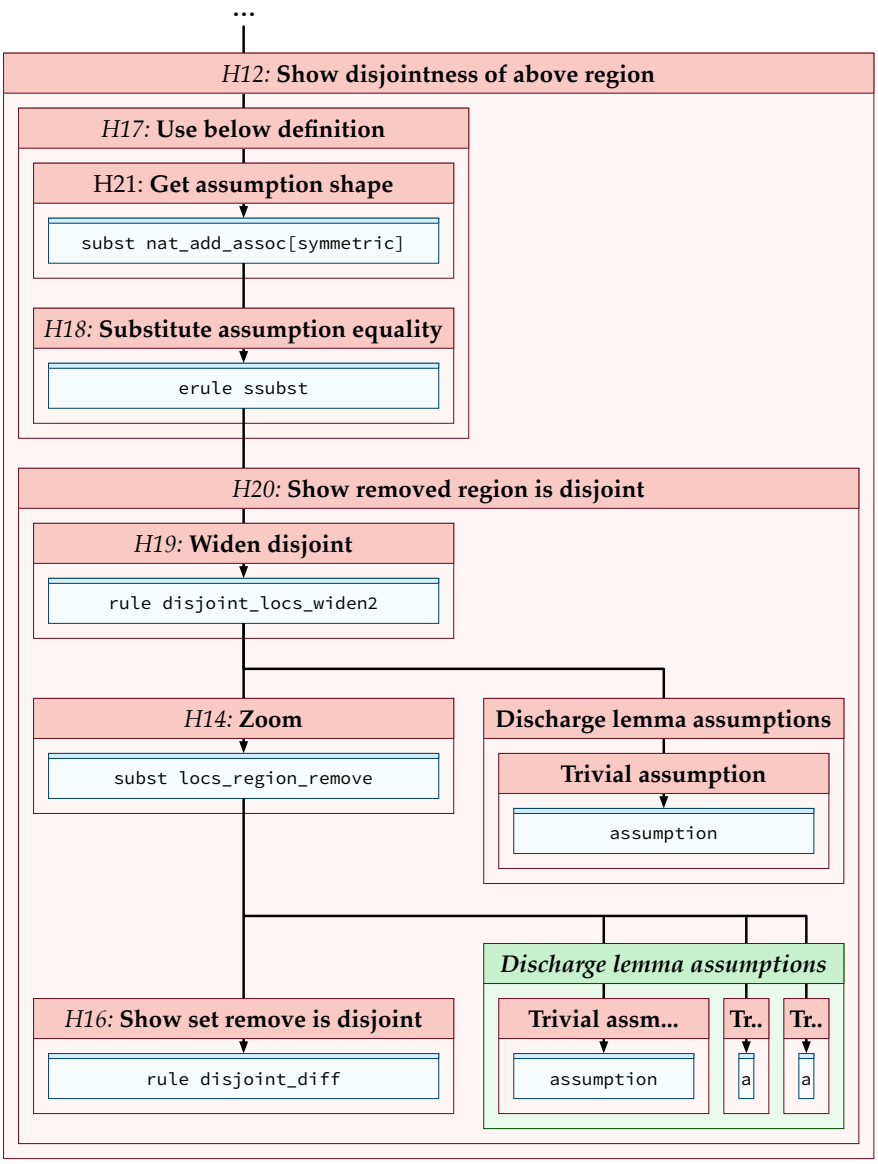


Figure 11.25: ProofProcess tree of "above" branch of lemma *dispose1_disjoint_both*. Continues the main tree from Figure 11.22.

```
lemma disjoint_locs_widen2:
 "nat1_map f ⟹
  disjoint s (locs ({q} ⩤ f)) ⟹ disjoint s (locs ({p, q} ⩤ f))"
```

Figure 11.26: Lemma *disjoint_locs_widen2*.

to replicate the idea: i.e. introduce a new lemma that retains the second argument. The lemma is copied and adapted trivially as *disjoint_locs_widen2* (Figure 11.26); its proof can be found automatically using Sledgehammer.

Alternatively, the user could follow the approach used previously: see what proof features are expected and try to bridge towards them rather than adapting the strategy and the proof features. In this case, he could use the `insert_commute` lemma from the Isabelle/HOL library to flip the set elements and continue with the strategy, having made $d + s$ the first element in the set.

Furthermore, the user is adapting or reusing a strategy that was introduced earlier within the same proof. The proof process capture is expected to work "live" along with the expert doing either manual proof or strategy replay. AI4FM tools could run in the background analysing data as it is captured and extracting proof strategies. Thus a strategy used earlier in the proof could be available for reuse later within the same proof, assisting the user as soon as possible.

The small adjustments to the **Show removed region is disjoint** strategy in the *above* region's branch contribute to yet another alternative case of this strategy. They now cover cases when the removed region is removed by itself, as well as with another region in any order. Otherwise the strategy is replayed without any changes and completes the proof branch as well as the whole proof.

❦ ❦ ❦

The case study presented in this chapter illustrates how the ProofProcess framework captures proof process data. More generally, it gives examples of how interactive proof can be described and structured at an abstract level, how to capture key features of proof steps, etc. Furthermore, the captured information is reused to assist with similar proofs, extract reusable strategies and extend them with additional instances of strategy application. This case study in Isabelle/HOL uses deterministic and basic proof tactics, enabling a clearer description of the proof process. The next case study uses the Z/EVES theorem prover with more automated tactics and a less "tidy" proof. The success of the approach is investigated when dealing with awkward proof strategies.

CHAPTER 12

Capturing proof about kernel properties

CASE STUDY

This case study presents the capture of proof processes from a formal development of a separation kernel specification using the Z/EVES theorem prover. The previous case study in Chapter 11 used an expressive Isabelle/HOL prover to capture and reuse a quite detailed and well guided proof. The proofs presented here use more automation and are less controlled. However, the case study shows that even awkward proofs can provide reusable strategies.

The integration of the prototype ProofProcess system with the Z/EVES theorem prover aims at supporting "industrial style" formal developments and evaluating proof process capture and strategy reuse in such scenarios. Furthermore, the integration provides access to an existing corpus (and experience within the AI4FM project) of industrial-type proofs (see Chapter 10 for more details).

The Z/EVES theorem prover [Saa97] is simpler in its scope of available tactics than Isabelle, thus part of the proof is "fighting" the prover: limited expressivity gives rise to strategies that work around the prover's shortcomings. The Z notation [WD96] can be used to construct specifications of complex models, however proving lemmas about them involves strategies for managing the complexity and scope of the proof and guiding the prover. The case study of the heap specification

development in Isabelle/HOL (Chapter 11) gives a step-by-step description of how the ProofProcess framework is used to capture the proof process. Then it shows how the information can be reused as strategies to prove similar lemmas. The case study presented in this chapter does not go into as much detail. Instead it peeks into the Z/EVES integration and the capture of an industrial-type proof process with its struggles and strategies.

This case study describes a proof of a theorem from a formal development of a *separation kernel* specification. It comes from an earlier work by the author of this thesis: full details (specification, proof scripts, justifications, etc.) are available in [Vel09, VF10]. The specification and the proof are not "tidy" or "nice": the case study aims to present a capture of a formal proof that is more real-world than one polished for an academic paper. The proof has been done by an "engineer-level" user (i.e. with limited theorem-proving experience) and without fine-tuned lemmas that could have improved the automation. Because of all this, the proof is quite clumsy and involves a number of proof steps working around the limitations of the prover as well as the complexity of the data structure rather than it being an actually difficult proof. Nevertheless, such properties are frequent among industrial style formal developments yet the proof strategies are useful, reusable and necessary to help with similar proofs. Like in the previous case study, the examples are of smaller scale than those in real industrial formal developments, but serve well to illustrate proof process capture, reuse of awkward proof strategies and the Z/EVES integration.

12.1 Modelling a process table

The formal development and verification of a separation kernel specification undertaken in [Vel09] covers several main components of the kernel: a process table, a process queue and a scheduler. For full details about the model, separating properties and all justifications, please refer to the Master's thesis [Vel09] as well as the original development by Craig [Cra07]. This case study focuses on the process table *PTab* and an operation to delete all processes stored there *DeleteAllProcesses*. Formal verification includes checking that this operation clears associated data structures correctly, specified as theorem *tDeleteAllExtpid* (Figure 12.3). Its proof is captured and analysed in this case study.

A process table in an operating system kernel stores a registry of processes and

$$
\begin{array}{|l}
\textit{PTab} \\
\hline
\textit{nextupid} : \textit{UPID} \\
\textit{extpid} : \textit{UPID} \pfun \textit{PID} \\
\textit{pidext} : \textit{PID} \pfun \textit{UPID} \\
\textit{used}, \textit{free} : \mathbb{F}\ \textit{PID} \\
\textit{tss} : \textit{PID} \pfun \textit{TSS} \\
\textit{devmap} : \textit{DevNo} \pinj \textit{PID} \\
\textit{state} : \textit{PID} \pfun \textit{PSTATE} \\
\textit{ptype} : \textit{PID} \pfun \textit{PTYPE} \\
\textit{msgq} : \textit{PID} \pfun \textit{MsgQ} \\
\textit{devmsg} : \textit{PID} \pfun (\textit{GPID} \times \textit{MSG}) \\
\textit{devrpy} : \textit{PID} \pfun \textit{MSG} \\
\textit{cdseg} : \textit{PID} \pfun \textit{SDesc} \\
\textit{dsseg} : \textit{PID} \pfun \textit{SDesc} \\
\hline
\textit{free} = \textit{PID} \setminus \textit{used} \\
\textit{used} = \mathrm{dom}\ \textit{state} = \mathrm{dom}\ \textit{ptype} = \mathrm{dom}\ \textit{tss} \\
\exists\, \textit{dprocs}, \textit{uprocs} : \mathbb{F}\ \textit{PID} \mid \\
\qquad \textit{dprocs} = \textit{ptype}^{\sim}(\!|\{\textit{dproc}\}|\!) \land \\
\qquad \textit{uprocs} = \textit{ptype}^{\sim}(\!|\{\textit{uproc}\}|\!) \bullet \\
\quad \textit{dprocs} = \mathrm{ran}\ \textit{devmap} = \mathrm{dom}\ \textit{devmsg} = \mathrm{dom}\ \textit{devrpy} \land \\
\quad \textit{uprocs} = \mathrm{dom}\ \textit{cdseg} = \mathrm{dom}\ \textit{dsseg} = \mathrm{dom}\ \textit{msgq} = \mathrm{ran}\ \textit{extpid} \\
\textit{pidext} = \textit{extpid}^{\sim} \\
\forall\, u : \textit{UPID} \mid u \geq \textit{nextupid} \bullet u \notin \mathrm{dom}\ \textit{extpid} \\
\hline
\end{array}
$$

Figure 12.1: Process table schema *PTab*.

their associated attributes. The process table is modelled as a Z schema[1] *PTab* and presented in Figure 12.1. At the core of a process table are process identifiers *PID*: the specification partitions them into the *used* and *free* registers (modelled as finite sets $\mathbb{F}\ \textit{PID}$). Furthermore, the process table supports translation between the externally visible "user" process identifiers *UPID* and the internal *PID*s by storing the *extpid* mapping (modelled as a partial function $\textit{extpid} : \textit{UPID} \pfun \textit{PID}$ and its mirrored mapping *pidext*). Various process attributes are recorded using individual maps: e.g. $\textit{state} : \textit{PID} \pfun \textit{PSTATE}$ records the state of each process.

The kernel distinguishes between *user* and *device* processes, which have some different attributes. The type of a process can be either *uproc* (user) or *dproc* (device), recorded in the *ptype* mapping. Then, based on this mapping, the process table ensures that specialised attributes are recorded for exactly the appropriate processes: e.g. *devmap* only includes *device* processes, while *extpid* only includes *user* processes. All of the described conditions are recorded as invariants within

[1] This case study assumes that the reader is familiar with the Z notation [WD96], though the majority of Z concepts utilised here are quite straightforward and/or explained briefly.

DeleteAllProcesses

$\Delta PTab$

$used' = \varnothing$

Figure 12.2: *DeleteAllProcesses* process table operation.

the *PTab* schema (Figure 12.1, refer to [Vel09] for full details). The attribute partitioning is achieved by using a *relational image* of an inverse of the *ptype* mapping: e.g. $uprocs = ptype^{\sim}(\!|\{uproc\}|\!)$ gives a set of all user process identifiers.

This case study focuses on the *DeleteAllProcesses* operation, which clears the data in a process table by deleting all *used* processes. This operation is specified as the *DeleteAllProcesses* schema (Figure 12.2). The operation schema describes a relation between the *before* and *after* states of the data structure (schema *PTab*). The $\Delta PTab$ statement includes the variables of the *before* state of *PTab* and its *after* state $PTab'$ (all *after* state variables are decorated with a prime $'$ symbol). The operation does not restrict any variables here except for ensuring that the set of used processes after the operation is empty: $used' = \varnothing$. In general, an operation should avoid erroneous underspecification by specifying the relations between the *before* and *after* states for *all* involved variables. However, in this particular case, the invariant of *PTab* ensures that all other variables are also cleared if $used'$ is empty.[2] By specifying only the minimal set of restrictions, future proofs are simplified. However, to achieve assurance in the overall specification, the correct deletion of the process table data needs to be verified separately.

12.2 Proof of correct process data deletion

To verify that the associated process data is cleared when all processes are deleted, a theorem stating the property of interest is conjectured and proved. For example, theorem *tDeleteAllExtpid* states the property that after executing *DeleteAllProcesses*, the *after* state of the external process identifier mapping is also empty: $extpid' = \varnothing$. This theorem and its proof in Z/EVES are presented in Figure 12.3. By inspecting the invariants in *PTab*, this fact is quite easy to verify: if $used'$ is empty, so is the domain (and thus the mapping itself) of *ptype*; then the *uprocs* set is also

[2]The next available *UPID* variable *nextupid* would still be underspecified. However, after clearing the process table, *nextupid* can be chosen arbitrarily, therefore this underspecification is good and allows the implementation to choose a preferred allocation scheme.

```
theorem tDeleteAllExtpid
  DeleteAllProcesses ⇒ extpid' = ∅

proof tDeleteAllExtpid
  invoke DeleteAllProcesses;
  invoke ΔPTab;
  invoke predicate PTab';
  prenex;
  use gEmptyRan[ℤ, ℤ][A := UPID, B := PID, P := extpid'];
  rearrange;
  rewrite;
  split ran extpid' = {};
  cases;
  simplify;
  next;
  simplify;
  apply extensionality to predicate ran extpid' = {};
  prove;
  next;
```

Figure 12.3: Theorem *tDeleteAllExtpid* and its proof in Z/EVES.

empty, since there are no more processes at all registered within *ptype*; from the empty *uprocs* set, the range (and thus the mapping itself) of the *extpid* must also be empty, concluding the proof. However, guiding the Z/EVES theorem prover to this conclusion requires a larger number of proof steps, as illustrated in Figure 12.3.

The remainder of this section describes how this proof is achieved and emphasises several strategies taken by the user. The proof process is captured using the Z/EVES ProofProcess system and annotated by the user with high-level proof information. Figure 12.4 shows an overview of the captured proof as a ProofProcess tree data structure, highlighting the high-level *proof intents* and their relationships as well as listing the actual proof commands. This section discusses the important proof steps, captured high-level strategies and other proof information. Details of the captured proof process data (e.g. *proof features*) are listed in individual "boxes" representing the ProofProcess steps (e.g. see step K1). Furthermore, the case study highlights how this data is captured and represented in the framework, particularly focusing on the Z/EVES integration.

12.2.1 Expanding definitions

Industrial-style specifications model and describe complex systems, which consist of multiple interacting components. Modularity is an important feature of

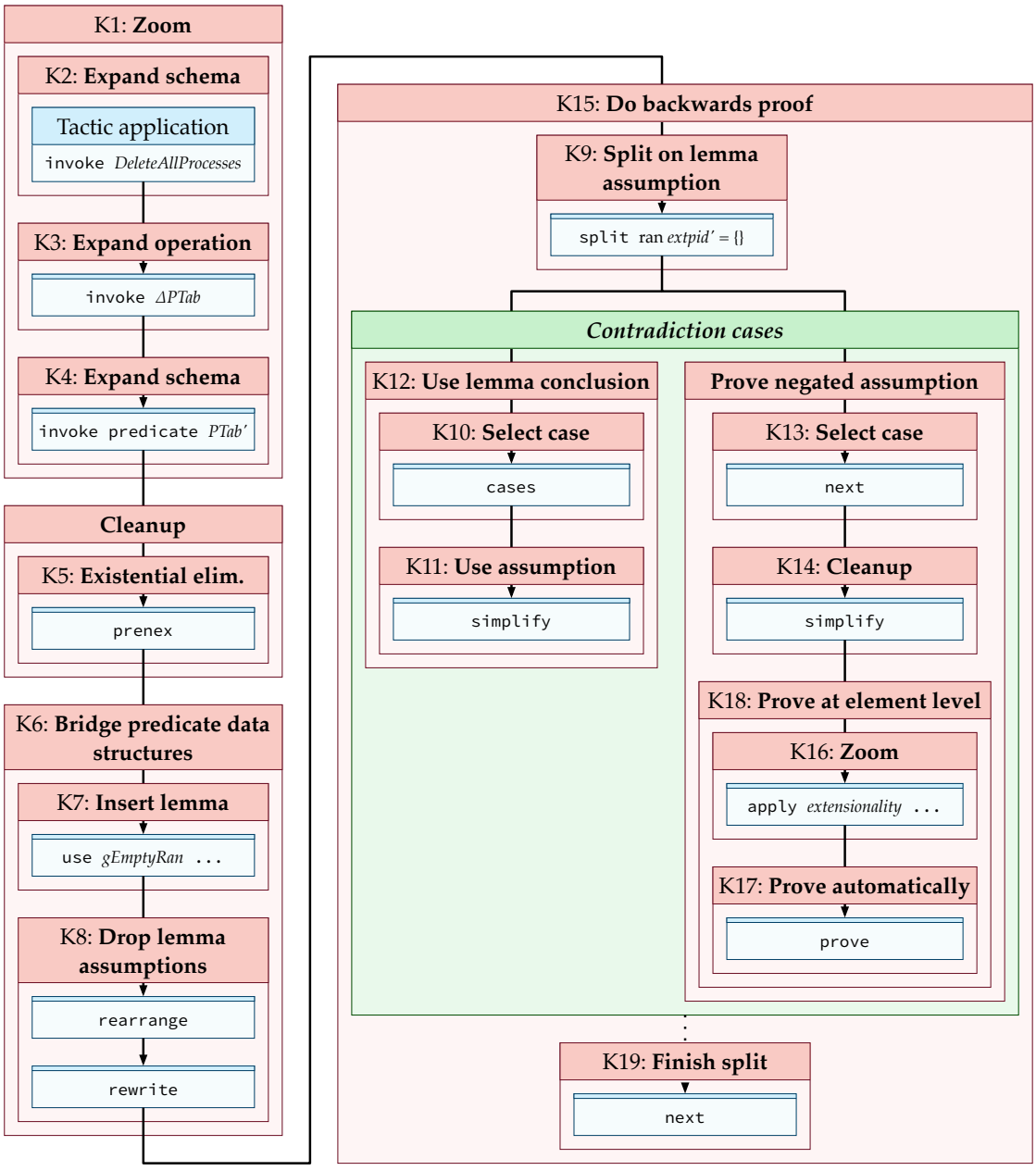


Figure 12.4: Simplified ProofProcess tree of theorem *tDeleteAllExtpid*.
The legend is available in Figure 11.5.

such specifications, as it separates concepts and allows different people to work on separate parts of the specification. Component properties are encapsulated; components themselves are bundled into larger systems with higher-level properties spanning the constituents. Accessing properties of lower-level components in a proof requires expansion of appropriate definitions.

In Z notation, *schemas* can be used to model such components: low-level components would be specified using schemas that define important variables and properties about them. Another schema can then include lower-level schemas and specify properties using their variables. This approach is used when specifying operations in Z notation: the *before* and *after* states of a data structure schema can be viewed as two sub-components of an operation schema, where an operation describes how variables in these schemas are related. For example, the operation *DeleteAllProcesses* includes $\Delta PTab$, which is a shorthand for including both *PTab* and $PTab'$, then specifies a relation[3] describing the properties of the operation: i.e. clearing the $used'$ variable (see Figure 12.2).

As sketched earlier, the proof of theorem *tDeleteAllExtpid* echoes the order of the clauses between variables of the process table. These relations are specified as invariant predicates on the $PTab'$ schema. The primed $PTab'$ schema has the same set of predicates as *PTab*, only with primed *after-state* variables. To access these predicates in the proof, encapsulating definitions need to be expanded.

A simple strategy is to "expand everything". Unfortunately, such an approach is not feasible in industrial-style proofs due to the large number of variables and predicates involved. For example, the *DeleteAllProcesses* operation includes two instances of *PTab* schema—a total of 28 variables—yet larger schemas can include many more. As is illustrated later, even the simple proof of theorem *tDeleteAllExtpid* has very large proof goals despite careful expansion guidance.

In the proof of *tDeleteAllExtpid*, schema and type expansion is needed right from the start. The initial goal is to show that after executing *DeleteAllProcesses*, variable $extpid'$ becomes empty:

$$DeleteAllProcesses \Rightarrow extpid' = \varnothing$$

The specification of the operation (as well as the *before* and *after* states with their properties) is hidden within the *DeleteAllProcesses* schema name. It is impossible to

[3] In this particular case the relation is not concerned with the *before* state, only with variables in the *after* state schema.

Intent: Zoom *ProofSeq*

Narrative: Expand the schemas until the desired level of discourse is reached.

In features:

- *Goal term* (*extpid′*) — The conclusion has the variable *extpid′*, hence need to expand the schemas (zoom) until we can reason about it.
- *Not* (*Assumption term* (*extpid′*)) — Variable is not visible among the assumptions.
- *Preferred level of discourse* (*extpid′*) — Prove at "*extpid′* level" (alternative feature).

Out features:

- *Assumption term* (*extpid′*) — The variable is available among the assumptions.

Children:

- *ProofSeq*: **Expand schema** ...
- *ProofSeq*: **Expand operation** ...
- *ProofSeq*: **Expand schema** ...

In goals (flattened):

$$DeleteAllProcesses \Rightarrow extpid' = \varnothing$$

Out goals (flattened): Listed in Figure 12.5.

Step K1: **Zoom**

reason about *extpid′* being empty, since nothing else can be said about the variable at this level of discourse.

To reveal the necessary concepts, the user needs to "zoom" to an appropriate level of discourse in the proof. At an abstract level, the zooming can be described as a high-level proof step **Zoom**, details of which are captured by the ProofProcess framework and are listed in step K1. The *out* goal of step K1 is listed separately in Figure 12.5 due to its size.

Marking and (not) generalising the important features

The high-level idea captured in the proof step is to expand the minimum amount of definitions (schemas) to be able to reason about the goal term *extpid′*. The proof step K1 records this using proof features: the *in* features capture that *extpid′* is a goal term but it is not visible among the assumptions, as *Goal term* (*extpid′*) and *Not* (*Assumption term* (*extpid′*)), respectively. Furthermore, it marks the important *out* proof features of a successful strategy application: i.e. that the required variable

$$
\begin{array}{l}
PTab \land \\
nextupid' \in UPID \land \\
extpid' \in UPID \pfun PID \land \\
pidext' \in PID \pfun UPID \land \\
used' \in \mathbb{F}\, PID \land \\
free' \in \mathbb{F}\, PID \land \\
tss' \in PID \pfun TSS \land \\
devmap' \in DevNo \pinj PID \land \\
state' \in PID \pfun PSTATE \land \\
ptype' \in PID \pfun PTYPE \land \\
msgq' \in PID \pfun MsgQ \land \\
devmsg' \in PID \pfun GPID \times MSG \land \\
devrpy' \in PID \pfun MSG \land \\
cdseg' \in PID \pfun SDesc \land \\
dsseg' \in PID \pfun SDesc \land \\
free' = PID \setminus used' \land \\
used' = \mathrm{dom}\, state' \land \\
\mathrm{dom}\, state' = \mathrm{dom}\, ptype' \land \\
\mathrm{dom}\, ptype' = \mathrm{dom}\, tss' \land \\
(\exists\, dprocs : \mathbb{F}\, PID;\ uprocs : \mathbb{F}\, PID \bullet \\
\quad dprocs = ptype' \sim (\!|\{dproc\}|\!) \land \\
\quad uprocs = ptype' \sim (\!|\{uproc\}|\!) \land \\
\quad dprocs = \mathrm{ran}\, devmap' \land \\
\quad \mathrm{ran}\, devmap' = \mathrm{dom}\, devmsg' \land \\
\quad \mathrm{dom}\, devmsg' = \mathrm{dom}\, devrpy' \land \\
\quad uprocs = \mathrm{dom}\, cdseg' \land \\
\quad \mathrm{dom}\, cdseg' = \mathrm{dom}\, dsseg' \land \\
\quad \mathrm{dom}\, dsseg' = \mathrm{dom}\, msgq' \land \\
\quad \mathrm{dom}\, msgq' = \mathrm{ran}\, extpid') \land \\
pidext' = extpid' \sim \land \\
(\forall\, u : UPID \mid u \geq nextupid' \bullet u \notin \mathrm{dom}\, extpid') \land \\
used' = \varnothing \\
\Rightarrow \\
extpid' = \varnothing
\end{array}
$$

Figure 12.5: Out goal of **Zoom** step (K1). Also the goal after executing the `invoke predicate` $PTab'$ proof command in the proof of *tDeleteAllExtpid*.

is available among the assumptions.

Such proof features make a somewhat blunt tool for recording the overall high-level idea and can be directly read as "reveal *any* statement about the goal variable in the assumptions". However, they do record the main idea—that the assumptions are lacking for reasoning about the goal variable—and may be enough for the majority of reuse cases for this strategy. Furthermore, because of the way schemas are constructed and packaged, revealing some fact about a variable may reveal all important facts about it (i.e. all predicates about some variables may be packaged together within the same schema, which expansion would conveniently reveal everything). When it comes to the amount and precision of the recorded proof features, in most cases it would not be necessary to be exhaustive with specifying all corner cases. Within a family of proofs, several well-placed hints may be enough to trigger the correct strategy reuse.

Step K1 also records the described idea with an alternative proof feature: *Preferred level of discourse* (*extpid′*). The "zooming" strategy can be described with the *preferred level of discourse*: i.e. by marking the level of definition at which the proof should be performed (see also examples in Chapter 11). In step K1, the user marks that the proof is to be done at the level of *extpid′* variable, i.e. that other definitions containing it should be expanded. Automating the reuse of this proof feature, however, would require extra functionality from the ProofProcess framework, because the *Preferred level of discourse()* proof feature is open-ended. Nevertheless, such a proof feature would point a human user of the strategy replay in the correct direction. Furthermore, the proof features could be combined as in step K1: the simple proof features described earlier would match automatically and the strategy would be suggested to the user during replay. He could then manually verify the proof features: seeing the "preferred level of discourse" feature would hint at the high-level idea more precisely than captured by the mechanics of "does not have a variable before—has the variable after strategy" features.

Either way, the ProofProcess framework captures proof process with hindsight: proof features mark the important parts of *this particular* proof. The challenges of generalising them for strategy reuse could be postponed: e.g. advanced algorithms of strategy extraction might fill in the gaps and generalise the strategy successfully; additional proofs with minor variations of the proof features might be captured to feed into the generalisation; the simple strategies will match too eagerly in the replay and the user will fine-tune them; etc.

The same generalisation argument applies to marking the proof features. In

the *heap* case study (Chapter 11), many proof features were marked generalised (e.g. as *shape* proof features). This yields more information for strategy extraction as the proof features directly construct the extracted strategy. However, it also steps away from recording "how the proof was achieved": i.e. the separation of proof process capture and the strategies. Marking the important proof features with the *actual* terms and variables in the goal is simpler: e.g. the system could facilitate the process by enabling the user to just click the important sub-term and include it in the proof feature. See Section 8.2.3 for more details on how the prototype ProofProcess system allows the marking of sub-terms. By quickly (and seamlessly) marking the actual important terms, the user is recording his proof process and deferring the strategy generalisation. The data can then be picked up for strategy extraction: an algorithm may decide that the actual term itself is important; or that it (or its parts) can be generalised for a more widely applicable strategy; eventually it could be combined with other examples of the same strategy or fine-tuned further manually.

Actual zoom steps

The **Zoom** proof step (K1) captures the abstract proof step that expands the necessary definitions containing the $extpid'$ variable. While it would be possible to replicate the step using a single Z/EVES proof command, the actual zooming is done in three Z/EVES proof steps.

First, the user is faced with the initial goal:

$$DeleteAllProcesses \Rightarrow extpid' = \varnothing$$

There is not much to do at this level, other than expand the *DeleteAllProcesses* schema using the `invoke` *DeleteAllProcesses* proof command. The `invoke` command in Z/EVES expands the given schema by replacing its name with inlined schema contents. In this case, the *DeleteAllProcesses* name is replaced with its definition (listed in Figure 12.2).

The user marks this proof step as **Expand schema** intent (see proof step K2). Actually, this is a generic (albeit trivial) proof strategy: if goal assumptions consist of a single schema definition, expand it. The user highlights the schema as the important feature: *Assumption schema* (*DeleteAllProcesses*). In a similar way to how invariant definitions are marked as proof features in the heap case study (see step H1 in Chapter 11), metadata about the specification can be useful here. When

Intent: Expand schema *ProofSeq* (as decoration)

Narrative: A single top-level schema in the assumptions, expand it.

In features:

- *Assumption schema* (*DeleteAllProcesses*)
- *Operation schema* (*DeleteAllProcesses*)

In goals:

$$DeleteAllProcesses \Rightarrow extpid' = \varnothing$$

Proof step: `invoke` *DeleteAllProcesses* ...
Out goals:

$$\Delta PTab \land used' = \varnothing \Rightarrow extpid' = \varnothing$$

Step K2: **Expand schema**

specified, the *DeleteAllProcesses* schema could be marked as representing an *operation*.[4] Thus by adding this proof feature (*Operation schema* (*DeleteAllProcesses*)), strategy extraction can be more eager in applying the schema expansion to other operations. This would expand the use of the extracted strategy from just single-schema assumptions to proofs involving operation schemas, e.g. if there are additional predicates within the operation, such as in precondition verification.

Predicates about the goal variable are not revealed by the expansion, however:

$$\Delta PTab \land used' = \varnothing \Rightarrow extpid' = \varnothing$$

The next step is to unpack the *before* and *after* states of the process table $\Delta PTab$. The delta notation is a shorthand for including the standard and primed versions of a Z schema. It has little value in proofs (apart from compacting the notation) and should normally be expanded. Step K3 captures this very generic strategy. Expanding $\Delta PTab$ introduces conjoined schemas $PTab \land PTab'$:

$$PTab \land PTab' \land used' = \varnothing \Rightarrow extpid' = \varnothing$$

Finally, the last "zooming" step expands $PTab'$ to reveal the predicates about the $extpid'$ variable (step K4). In this step, the proof command `invoke predicate` $PTab'$ is used to narrow the expansion to only the *after* state schema $PTab'$. The user marks this proof step as an additional instance of the **Expand schema** intent. Its proof features record that the expansion takes place because $PTab'$ contains the

[4] The fact that a schema represents an operation can also be inferred by checking for *before* and *after* states within the operation schema.

Intent: Expand operation *ProofSeq* (as decoration)

Narrative: Expand $\Delta PTab$.

In features:

- *Assumption shape* (Δ *?Sch*[a])

In goals:

$$\Delta\, PTab \land used' = \varnothing \Rightarrow extpid' = \varnothing$$

Proof step: `invoke` $\Delta\, PTab$ ...
Out goals:

$$PTab \land PTab' \land used' = \varnothing \Rightarrow extpid' = \varnothing$$

[a]The *?var* notation with the leading question mark is used to indicate placeholder variables in the ProofProcess system. It should not be confused with *var?* (trailing question mark), which denote input variables in Z notation.

Step K3: **Expand operation**

Intent: Expand schema *ProofSeq* (as decoration)

Narrative: Only expand $PTab'$, because it contains $extpid'$.

In features:

- *Contains* ($PTab'$, $extpid'$)

In goals:

$$PTab \land PTab' \land used' = \varnothing \Rightarrow extpid' = \varnothing$$

Proof step: `invoke predicate` $PTab'$ ...

Out goals: Listed in Figure 12.5.

Step K4: **Expand schema**

$extpid'$ variable within: *Contains* ($PTab'$, $extpid'$). The conjecture is only concerned with primed variables ($used'$ and $extpid'$) and does not use any *before* state information. Because of that, it is not necessary to expand the $PTab$ schema. Each $PTab$ instance contains 14 variables as well as predicates relating to them. Thus, where possible, one should avoid unnecessary expansion that would clutter the proof goal. There is enough clutter from just the necessary expansion: the goal after this proof step is already massive for such a small problem (see Figure 12.5).

The *Contains()* proof feature requires the system to "peek" inside the schema. When dealing with complex data structures, the important variables and their properties are frequently hidden under layers of abstraction or composition. The

user often has an intuition about the data structures and thus can include their contents in the high-level reasoning process, even though they are not directly visible in the goal. The intuition needs to be captured using proof features, thus first-class support for complex data structures is needed in the AI4FM system. It is not enough to select from apparent sub-terms. When capturing the proof process, the user may need to highlight the important parts contained within schemas, data types or function definitions. Similarly, when replaying strategies with such proof features, the system needs to match the proof feature with the goal by delving into the definitions and matching the contents.

Furthermore, in more complex proofs it is often preferable to avoid expanding further than some level of schemas. For example, the goal may already be very large and introducing an extra 14 variables and their predicates by expanding the *PTab* schema makes it unmanageable, particularly if only a single predicate is required from the *PTab* definition. Instead of expanding, additional lemmas could be specified that introduce the encapsulated properties as forward proof rules, in the form $SchemaRef \Rightarrow P$,[5] where P is a predicate about a variable contained within the *SchemaRef* schema (e.g. $PTab \Rightarrow nextupid \in \mathbb{Z}$). Furthermore, Z/EVES allows tagging these rules as *forward rules* (`frule`) and will apply them automatically during simplification [MS97]. This is useful for the most frequently used properties, but introducing too many lemmas can encumber the theorem prover and limit the effectiveness of its automatic tactics.

The "zooming" steps in this case study are quite simple and contribute straightforward generic strategies for reuse. The strategies are easy to remember for users of all levels of expertise, i.e. they are not "expert" strategies that "engineer"-level users need the system to learn. Nevertheless, these strategies are encountered widely in similar proofs. Thus if they are "learned" and applied automatically by the AI4FM system, it would save a lot of time doing (and redoing) similar proofs and the user could focus on the "difficult" parts of the proof.

Cleanup modelling artefacts

The invariant of the *PTab* schema (Figure 12.1) includes a large existentially quantified predicate about the *dprocs* and *uprocs* variables. These variables are derived from the *ptype* variable and represent *device* and *user* processes, respectively. To avoid including derived variables at the top level, they are existentially quantified.

[5] The forward rules can also be stated in an equivalent form $\forall SchemaRef \bullet P$.

Intent: Existential elimination *ProofSeq* (as decoration)

Narrative: Eliminate existential quantifiers from assumptions.

In features:

- *Assumption shape* ($\exists$ *?var* • *?P*)

In goals (filtered):

$$
\begin{array}{l}
?p1 \land \\
(\exists\, dprocs : \mathbb{F}\, PID;\ uprocs : \mathbb{F}\, PID \bullet \\
\quad dprocs = ptype' \sim (\!|\{dproc\}|\!) \land uprocs = ptype' \sim (\!|\{uproc\}|\!) \land \\
\quad dprocs = \operatorname{ran} devmap' \land \operatorname{ran} devmap' = \operatorname{dom} devmsg' \land \operatorname{dom} devmsg' = \operatorname{dom} devrpy' \land \\
\quad uprocs = \operatorname{dom} cdseg' \land \operatorname{dom} cdseg' = \operatorname{dom} dsseg' \land \operatorname{dom} dsseg' = \operatorname{dom} msgq' \land \\
\qquad \operatorname{dom} msgq' = \operatorname{ran} extpid') \land \\
?p2 \ \Rightarrow\ ?p3
\end{array}
$$

Proof step: `prenex` ...
Out goals (filtered):

$$
\begin{array}{l}
?p1 \land \\
dprocs \in \mathbb{F}\, PID \land uprocs \in \mathbb{F}\, PID \land \\
dprocs = ptype' \sim (\!|\{dproc\}|\!) \land uprocs = ptype' \sim (\!|\{uproc\}|\!) \land \\
dprocs = \operatorname{ran} devmap' \land \operatorname{ran} devmap' = \operatorname{dom} devmsg' \land \operatorname{dom} devmsg' = \operatorname{dom} devrpy' \land \\
uprocs = \operatorname{dom} cdseg' \land \operatorname{dom} cdseg' = \operatorname{dom} dsseg' \land \operatorname{dom} dsseg' = \operatorname{dom} msgq' \land \\
\quad \operatorname{dom} msgq' = \operatorname{ran} extpid' \land \\
?p2 \ \Rightarrow\ ?p3
\end{array}
$$

Step K5: **Existential elimination**

In a proof, the existential quantifiers in assumptions can be eliminated, giving access to these variables directly. The `prenex` proof command in Z/EVES eliminates these quantifiers—the captured details of this proof step are listed in step K5. The **Existential elimination** proof step is quite generic: the proof features only require an existential quantifier among the assumptions.

At the higher level, this proof step represents a cleanup of modelling artefacts: the existential quantifier is important when modelling, but can be eliminated during the proof. Thus the user introduces additional intent **Cleanup** by wrapping the proof step K5 (see Figure 12.4). By tagging all similar strategies as **Cleanup**, the user would capture a collection of model cleanup activities that could be performed where applicable.

12.2.2 Bridging data structures

After zooming to an appropriate level of discourse, the existing predicates about $extpid'$ can be used in the proof. As sketched at the beginning of this section,

theorem grule gEmptyRan $[X, Y]$
$\forall A : \mathbb{P}\ X;\ B : \mathbb{P}\ Y \bullet \forall P : A \leftrightarrow B \bullet \text{ran}[X, Y]\ P = \{\} \Rightarrow P = \{\}$

Figure 12.6: Theorem *gEmptyRan*.

proving the goal mainly requires following the relations between the variables (particularly between the *domains* and *ranges* of map variables). Unfortunately, with the existing set of lemmas, the automatic Z/EVES proof commands (e.g. `prove [by rewrite]` and `prove by reduce` [MS97]) are unable to complete the proof automatically. The user has to perform manual steps to advance the proof.

The strategy taken here is to nudge the Z/EVES prover towards the intended proof by manually instantiating some of the previously sketched steps, in particular, to link the $extpid'$ term in the goal and its range $\text{ran}\ extpid'$ used in the assumptions. This can be achieved using lemma[6] *gEmptyRan* (Figure 12.6), which has been created and proved beforehand. It proves that if the range of a relation is empty, so is the relation itself. In a backward-style proof, this would mean that the goal of $extpid' = \{\}$ holds if $\text{ran}\ extpid' = \{\}$ can be shown to be true.

The user employs lemma *gEmptyRan* by inserting it into the proof manually as an additional assumption and cleaning up *its* assumptions.[7] The high-level proof step that represents these activities is captured in the ProofProcess framework as step K6 with the **Bridge predicate data structures** intent. The overall high-level idea of this step is to introduce the crux of the *gEmptyRan* lemma as a new assumption. The filtered *out* goal of step K6 lists the added assumption. This is actually achieved by two high-level steps (three Z/EVES proof commands): inserting the lemma first and then dropping most of its assumptions that trivially hold. These proof steps are captured as **Insert lemma** (step K7) and **Drop lemma assumptions** (step K8), respectively.

Inserting bridging lemma

The **Insert lemma** (K7) step lists the important proof features that have triggered the user to insert lemma *gEmptyRan*. Furthermore, the *in/out* goals provide a

[6] The words *lemma* and *theorem* are used interchangeably in this thesis, just as they are in both Isabelle and Z/EVES theorem provers.

[7] Lemma *gEmptyRan* is specified as an assumption rule (`grule`) with expectation that it will be used by Z/EVES automatically where applicable to introduce additional assumptions. Unfortunately, its form does not match well with the way it is used in the kernel development and therefore it is mostly used manually by explicitly inserting into the proof.

Intent: Bridge predicate data structures *ProofSeq*

Narrative: Link the goal ($extpid' = \varnothing$) with the assumption terms ($\mathrm{ran}\ extpid'$).

Children:

- (K7) *ProofSeq*: **Insert lemma** ...
- (K8) *ProofSeq*: **Drop lemma assumptions** ...

In goals (flattened, filtered):

$$?p1 \Rightarrow ?p2$$

Out goals (flattened, filtered):[a]

$$?p1 \wedge (\mathrm{ran}\ extpid' = \{\} \Rightarrow extpid' = \{\}) \Rightarrow ?p2$$

[a] There are additional rewriting side-effects to the goal not shown here—see step K8 and the related discussion.

Step K6: **Bridge predicate data structures**

filtered view of the goal change (some of the important parts of the goal are highlighted even though they do not change). The proof features highlight that having $extpid'$ in the goal and $\mathrm{ran}\ extpid'$ among the assumptions warrants the use of the *gEmptyRan* lemma, which links them.

The important lemma shape is captured in a relaxed style: i.e. it is important that the lemma introduces some relationship between variables—it is left to the user to select the most appropriate one. An unrestricted specification of the proof feature captures the user's idea that it is important to link the variables even though the exact solution to the proof does not present itself yet. For example, if the proof features required *Goal shape* ($extpid' = ?c$) and *Assumption shape* ($\mathrm{ran}\ extpid' = ?b$), they would not match the current goal, since the fact $\mathrm{ran}\ extpid' = \{\}$, as required by the lemma, is not established in the current goal. Therefore, if the strategy was reused for other proofs, the user would receive a hint to "find some relationship between $\mathrm{ran}\ x$ and x", and selecting the best matching relationship would either be left to the user or would be based on some additional proof features.

When extracting a proof strategy from such a proof step, *termination* needs to be considered. Here the user knows that lemma introduction is needed and marks the important proof features to capture this idea. However, none of the proof features capture the negative facts: e.g. that the lemma has not been inserted

Intent: Insert lemma *ProofSeq* (as decoration)

Narrative: Use a suitable lemma *gEmptyRan* that bridges *extpid′* and ran *extpid′*.

In features:

- *Goal term* (*extpid′*)
- *Assumption term* (ran *extpid′*)
- *Used lemma* (*gEmptyRan*)
- *Lemma shape* (ran $?a = ?b \Rightarrow ?a = ?c$)

In goals (filtered):[a]

$$?p1 \land \\ \mathrm{dom}\ msgq' = \mathrm{ran}\ extpid' \land \\ ?p2 \Rightarrow extpid' = \varnothing$$

Proof step: `use` *gEmptyRan*$[\mathbb{Z}, \mathbb{Z}][A := UPID, B := PID, P := extpid']$ ...
Out goals (filtered):

$$(UPID \in \mathbb{P}\,\mathbb{Z} \land PID \in \mathbb{P}\,\mathbb{Z} \land extpid' \in UPID \leftrightarrow PID \land \\ \quad \mathrm{ran}\ extpid' = \{\} \Rightarrow extpid' = \{\}) \land \\ ?p1 \land \\ \mathrm{dom}\ msgq' = \mathrm{ran}\ extpid' \land \\ ?p2 \Rightarrow extpid' = \varnothing$$

[a] The predicates about *extpid′* and ran *extpid′* would also be filtered in the system but are shown here to highlight the important parts.

Step K7: **Insert lemma**

beforehand. Automatic application of a strategy with only this set of features will run endlessly. After inserting the lemma, the proof features in step K7 will match again—and will insert another copy of the lemma. Approaches to avoid such looping include inferring additional proof features about "negated effects of the proof step" or calculating a fixed point of strategy application (e.g. if facts no longer change, the strategy should not be applied).

The Z/EVES proof command `use` inserts the given lemma with provided instantiations at the top of assumptions. As shown in the *out* goal of step K7, no simplification has been done to discharge the lemma assumptions (particularly the type considerations). This is done in the next proof steps.

Cleaning up lemma insertion

Lemma *gEmptyRan* links ran *extpid′* being empty with the whole *extpid′* being empty. By showing the former, the overall proof will be complete, since the latter

($extpid' = \{\}$) is the overall goal. However, ran $extpid' = \{\}$ is not established here. Nevertheless, other assumptions of lemma *gEmptyRan*—the types of the variables—are either already available or can be easily shown. Discharging them simplifies the overall goal.

The cleanup is performed using two Z/EVES proof commands: `rearrange` and `rewrite`. The first one reorders the hypotheses in the goal so that "simpler" ones appear before more complicated ones. This improves the effectiveness of the rewriting/reduction commands [MS97]. In the current proof, rearranging moves the inserted lemma (a complicated hypothesis involving an implication) below the type hypotheses, thus allowing the follow-up `rewrite` command to benefit from them when discharging the lemma assumptions.

Step K8 captures the pair of `rearrange` and `rewrite` commands as a single proof step with the intent **Drop lemma assumptions**. The proof features highlight that an implication (inserted lemma) appears among the assumptions, which would be cleaned up with the proof step. Furthermore, to make sure that an appropriate effect is reached, the *out* features record the result of the proof step: some of the conjuncts in the implication have been proved and hence dropped from the goal. The proof features describe the current proof (i.e. four lemma assumptions before the step—only a single one remaining afterwards). When a strategy is extracted from this captured proof step, a "perfect" heuristic would figure out that the strategy basically requires the number of assumptions to decrease to qualify as a successful application of the **Drop lemma assumptions** strategy. Such heuristics could be encoded in the strategy extraction algorithm, however the user may choose not to do any manual generalisations during the proof process capture.

Tactic side-effects and lemma usage

The `rewrite` proof command applies all possible rewrite rules to the goal. Using such a command may change other parts of the goal, as happens in step K8. The filtered goals show that in addition to dropping the lemma assumptions, the prover replaces dom $state'$ and dom $ptype'$ with empty sets, because $used' = \{\}$ and these sets are transitively equal to the $used'$ set. To avoid such side effects, the user could be more selective with the proof commands, only applying them to a narrow sub-term, e.g. using the following command:

`with predicate (`$extpid' \in UPID \fun PID \land UPID \in \power \zed \land$
$PID \in \power \zed \land \ran extpid' = \{\} \implies extpid' = \{\}$`) rewrite;`

Intent: Drop lemma assumptions *ProofSeq*

Narrative: Prove that lemma assumptions hold, dropping them.

In features:

- *Assumption shape* ($?a1 \land ?a2 \land ?a3 \land ?a4 \Rightarrow ?g1$)

Out features:

- *Assumption shape* ($?a4 \Rightarrow ?g1$)

Children:

- *ProofEntry*: `rearrange ...`
- *ProofEntry*: `rewrite ...`

In goals (filtered):

$$
\begin{array}{l}
(UPID \in \mathbb{P}\,\mathbb{Z} \land PID \in \mathbb{P}\,\mathbb{Z} \land extpid' \in UPID \leftrightarrow PID \land \\
\quad \operatorname{ran} extpid' = \{\} \Rightarrow extpid' = \{\}) \land \\
?p1 \land \\
used' = \operatorname{dom} state' \land \\
?p2 \land \\
\operatorname{dom} state' = \operatorname{dom} ptype' \land \\
\operatorname{dom} ptype' = \operatorname{dom} tss' \land \\
?p3 \land \\
used' = \varnothing \land \\
\Rightarrow \\
?p4
\end{array}
$$

Used lemmas: *weakening* and others (all listed in Figure 12.7).

Out goals (filtered):

$$
\begin{array}{l}
?p1 \land \\
used' = \operatorname{dom} state' \land \\
used' = \{\} \land \\
?p2 \land \\
\{\} = \operatorname{dom} ptype' \land \\
\{\} = \operatorname{dom} tss' \land \\
?p3 \land \\
(\operatorname{ran} extpid' = \{\} \Rightarrow extpid' = \{\}) \\
\Rightarrow \\
?p4
\end{array}
$$

Step K8: **Drop lemma assumptions**

- Rewrite rules: *applicationInDeclaredRangeFun*, *weakening*, *rel_sub*, *power_sub*, *notInRule*, *diffEmptyRight*, *nullFinite*, *emptyDefinition*
- Forward rules: *KnownMember$declarationPart*, *fNextupidUPIDType*, *knownMember*, *PTab$declarationPart*, *fNextupidType*, *fPTabDprocsPowerUsed*, *fPTabUprocsPowerUsed*, *fPTabDomPidext*, *fDevmsgMaxType*, *fCdsegMaxType*, *fDssegMaxType*, [internal items]
- Assumptions: *relDefinition*, *notin$declaration*, *uproc$declaration*, *ran$declaration*, *dproc$declaration*, *rimg$declaration*, *gSDescType*, *gGPIDMaxType*, *MsgQ$declaration*, *pinj_type*, *gDevNoType*, *TSS$declaration*, *select_2_1*, *select_2_2*, *setminus$declaration*, *finset_type*, *dom$declaration*, *inverse$declaration*, *pfun_type*, *gPIDMaxType*, *gPIDFinType*, *gPIDIsFinset*, *gPIDNotEmpty*, *gUPIDMaxType*, [internal items]

Figure 12.7: Lemmas used by `rewrite` proof command in **Drop lemma assumptions** (step K8).

In this particular case, the side-effects of the `rewrite` command are not too disruptive: i.e. they do not make a substantial change to the goal and do not affect the general high-level proof idea. Therefore the user chooses to apply the command broadly. Furthermore, in similar situations the user can often miss some of the additional changes to the goal when using automatic proof commands. The *filtered* goal view in the ProofProcess system helps with the issue by showing just the changes within a proof step (see Section 8.2.3). However, the eventual decision of how to advance the proof is with the user and in this case a simple `rewrite` command is used. The ProofProcess framework aims to accommodate all proof processes, thus the captured proof information is faithful to how the proof is done, even though extracting strategies from it may be difficult.

In a more general case, the side-effects may be more disruptive. Furthermore, a single powerful proof command may advance proof in multiple ways, realising several high-level ideas. For example, if sufficient lemmas are available, multiple parts of the goal can be transformed when respective lemmas are applied by the same `rewrite` command. In such a scenario, the user may want to record the different proof ideas separately. This can be achieved by having multiple nested *ProofSeq* elements in the ProofProcess framework. Each higher-level element wrapping the same proof command would have its own proof intent and associated proof features describing the particular high-level proof idea.

Step K8 also lists the captured lemma usage information. In particular the *weakening* lemma is highlighted. The full list of lemmas used by the `rewrite` proof command is much larger and due to space issues is listed in Figure 12.7. The Z/EVES theorem prover traces lemma usage in proof tactics, which perform

simplification and rewriting. This information is displayed along with the goal changes. The Z/EVES ProofProcess integration captures this information as part of the *ZEvesTrace* data structure, which records the prover command data. The captured data lists the *rewrite* rules (used by rewrite steps) as well as *forward* and *assumption* rules (used by both rewrite and simplification steps). Figure 12.7 shows that the number of lemmas can be large even for simple models and proofs. By filtering the important ones (e.g. excluding the built-in Z/EVES toolkit lemmas, the *datatype$declaration* lemmas that are generated automatically from datatypes, etc.), the user would have a better overview of what is done by the proof command—and particularly *how* the goal is changed.

In the case of the **Drop lemma assumptions** (K8) step, the *weakening* lemma is highlighted among the used lemmas. The *weakening* lemma is used to prove the type assumption $extpid' \in UPID \leftrightarrow PID$ from an existing hypothesis $extpid' \in UPID \pfun PID$: i.e. if a variable is a partial function, it is also a relation. However, the used lemma is not marked explicitly as an important proof feature. This is because the *weakening* lemma is a built-in Z/EVES toolkit lemma: i.e. it would also be available for other similar proofs automatically. Because of this ubiquity, its availability and shape are not as important and therefore are not recorded as proof features. In other cases, the user would select from the captured list of used lemmas and record the important lemmas as proof features.

12.2.3 Simulating backward proof step

The **Bridge predicate data structures** step introduces a lemma that proves the overall goal: $\mathrm{ran}\ extpid' = \{\} \Rightarrow extpid' = \{\}$. Thus if the fact $\mathrm{ran}\ extpid' = \{\}$ can be established, the overall goal follows. Unfortunately, performing such backward reasoning in Z/EVES proofs and establishing $\mathrm{ran}\ extpid' = \{\}$ as a new subgoal is not straightforward. Instead, a workaround resembling a *proof by contradiction* is employed to be able to focus on a proof involving $\mathrm{ran}\ extpid'$.

Splitting the goal into contradicting cases

Z/EVES provides a `split` command, which introduces an **if** P **then** G **else** G predicate for the given predicate P on which the goal G is split [MS97]. This way any fact can be inserted among the assumptions with two cases to prove: either when the fact is *true* or *false*. In general, it can be used to introduce a cut rule: the proof can rely on the cut predicate in the "true" case. In the "false" case, the negated

Intent: Split on lemma assumption *ProofSeq* (as decoration)

Narrative: Make a case split on the lemma assumption that proves the necessary goal.

In features:

- *Assumption shape* (ran $extpid' = \{\} \Rightarrow ?g$)
- *Goal shape* ($?g$)

In goals (filtered):

$?p1 \Rightarrow ?p2$

Proof step: `split` ran $extpid' = \{\}$ ...
Out goals (filtered):

if (ran $extpid' = \{\}$)
then $?p1 \Rightarrow ?p2$
else $?p1 \Rightarrow ?p2$

Step K9: **Split on lemma assumption**

predicate $\neg P$ appears among the assumptions. However, as all the assumptions are conjoined, showing that the negated predicate is false (or that it can falsify another assumption predicate) makes the conjoined assumptions predicate false and therefore the overall goal true (*false* implies anything). Showing that the negated predicate is false ($\neg \neg P$) actually results in showing that P is true.

In this proof, the split is done on ran $extpid' = \{\}$. The high-level idea **Split on lemma assumption** is captured in step K9. The triggers for this strategy are recorded as proof features: there is a lemma among the assumptions that proves the overall goal $?g$. The assumption of this lemma should then be used as the argument for the `split` proof command. The user marks the matching goal with a placeholder variable $?g$, simplifying strategy extraction and making it easier to comprehend that the assumption lemma matches the goal.

After splitting on the ran $extpid' = \{\}$, there are two cases in the goal: when ran $extpid' = \{\}$ (**then**/"true" case) and when ran $extpid' \neq \{\}$ (**else**/"false" case). A basic simplification command would prove and discard the "true" case and simplify the "false" case automatically, making it possible to jump directly to step K16 in the proof. However, the user elects to tackle each case individually.

To split the **if – then –else** argument of the goal into individual sub-goals, the `cases` proof command can be used. It performs the splitting and narrows down the goal to the first sub-goal. Unlike Isabelle, the Z/EVES prover focuses only on a single goal. Thus when the `cases` command is used, instead of displaying two

Intent: Select case *ProofSeq* (as decoration)

Narrative: Select the first case in a case split.

In features:

- *Top shape* (**if** ?$p1$ **then** ?$p2$ **else** ?$p3$)

In goals (filtered):

$$\begin{array}{l}\textbf{if}\,(\mathrm{ran}\,extpid' = \{\}) \\ \textbf{then}\ ?p1 \ \Rightarrow\ ?p2 \\ \textbf{else}\ ?p1 \ \Rightarrow\ ?p2\end{array}$$

Proof step: `cases ...`

- Proof case: #2.

Out goals (filtered):

$$\mathrm{ran}\,extpid' = \{\} \land ?p1 \ \Rightarrow\ ?p2$$

Step K10: **Select case**

sub-goals, it shows just one but indicates that this is a sub-case of the overall proof. The ProofProcess system tracks individual proof branches within the ProofProcess tree structure (Section 4.3). In Z/EVES ProofProcess, the branching information is provided by the prover, where each branch is marked with a case number. The system collects sequential proof steps with the same case number into one branch and joins all same-level branches within a single *ProofParallel* element. Furthermore, the case number is captured as part of the *ZEvesTrace* data structure for future reference. It is used to identify proof branches, whereas Isabelle ProofProcess has to do goal change analysis. However, the branching proof commands have to be captured within the proof process (they are filtered in Isabelle ProofProcess capture) as the commands perform both goal transformation and sub-goal selection. This can introduce awkward proof steps, particularly for the `next` proof commands (see steps K13 and K19).

Trivially proving the position case

The case splitting in the current proof is captured as **Select case** step K10. The proof features capture the fact that this generic proof step can be used when the goal is a top-level **if – then –else** predicate. A general strategy can be extracted directly from the captured proof step, suggesting case analysis in all such situations. The `cases` command is also applicable in other situations: e.g. when the goal is a conjunction, disjunction, etc. [MS97]. These other cases of the strategy will be

Intent: Use assumption *ProofSeq* (as decoration)

Narrative: Use the assumption lemma to prove the overall goal.

In features:

- *Assumption shape* ($?p1 \Rightarrow ?p2$)
- *Goal shape* ($?p2$)
- *Assumption shape* ($?p1$)

Out features:

- *Number of goals* (0)

In goals (filtered):

$$\begin{array}{l} \mathrm{ran}\, extpid' = \{\} \land ?p1 \land (\mathrm{ran}\, extpid' = \{\} \Rightarrow extpid' = \{\}) \\ \Rightarrow \\ extpid' = \{\} \end{array}$$

Proof step: `simplify` ...

- Proof case: #2.

Out goals (filtered): ✓ *(none)*

Step K11: **Use assumption**

captured when used in other proofs and tagged with the same **Select case** intent.

After the case split, the predicate $\mathrm{ran}\ extpid' = \{\}$ is added to the list of assumptions. Together with the previously added lemma, the overall goal is discharged, completing this proof branch. The details are captured as the **Use assumption** (K11) proof step. With the necessary assumptions in place, a simple `simplify` command suffices to complete the proof branch: Z/EVES produces the final predicate **true** to indicate a complete proof. When capturing the proof using Z/EVES ProofProcess, the predicate is dropped—a complete proof in the ProofProcess framework is represented with the empty list of *out* goals.

The user also marks this proof branch as a higher-level proof step to provide descriptive proof process information. Step K12 captures the abstract proof step **Use lemma conclusion**, containing both proof steps in this proof branch.

Selecting the negation case

With the "position" (where the assumption is positive) branch complete, the user focuses on the "negation" case. To switch to the other proof branch, Z/EVES provides the `next` proof command. At this point, the high-level idea is the same as with the earlier `cases` application: selecting one of the proof cases. Therefore the

Intent: **Use lemma conclusion** *ProofSeq*

Narrative: Prove the overall goal with the necessary assumption established.

Children:

- (K10) *ProofSeq*: **Select case** ...
- (K11) *ProofSeq*: **Use assumption** ...

Step K12: **Use lemma conclusion**

Intent: **Select case** *ProofSeq* (as decoration)

Narrative: Select the next case.

In features:

- *In case split* () — Verify that the proof is currently in a case split (e.g. *Case* # 2).
- *Number of goals* (0)

In goals (filtered): *(none)*

Proof step: `next` ...

- Proof case: # 1.

Out goals (filtered):

$$\neg\ \mathrm{ran}\ extpid' = \{\} \wedge ?p1 \ \Rightarrow\ ?p2$$

Step K13: **Select case**

proof command is tagged with the same **Select case** intent (step K13).

The captured proof step records why the `next` command is used: the proof is in a case split (the prover reports a case number). The `next` command allows a change to another proof branch, even if the previous branch is not yet finished. However, in this case the previous branch is finished and the user marks that as an important proof feature: *Number of goals* (0). This restricts the strategy reuse: e.g. the **Select case** proof strategy should be suggested in order to switch to a new branch when the previous branch is finished, otherwise it would match whenever a proof within a case split is being done.

The `next` proof command (as well as its companion `cases` command), however, introduces some issues to the captured proof process structure. Because some of the goal transformations that are done by the commands are hidden, it is difficult to correctly represent the proof process in a branching structure. For example, when capturing an Isabelle proof process, a proof command that introduces multiple sub-goals can be tracked as an entry point to the branch split. Then by analysing

the proof goal changes, the system assigns proof commands to particular goals of the explicitly listed sub-goals (see Section 6.2). By separating the "splitting" and "sub-goal" commands, the system can produce a correct structure to represent the branching: e.g. the splitting command is followed by a *ProofParallel* that contains branches of sub-goal commands.

When capturing Z/EVES proof commands, the branch identifiers are reported by the prover and the Z/EVES ProofProcess system utilises them to partition proof commands into branches. However, it is impossible to separate the functionality of these commands into "splitting" and "selecting/working on sub-goal". The `cases` command transforms the goal into multiple sub-goals ("true" and "false" cases in the current proof), then selects one of the sub-goals to work on—a single command does both the splitting and the selection. After applying the `cases` command, the system can only capture a single sub-goal, because the other sub-goals are not visible until a `next` command is issued. In turn, the `next` command is used on a previous branch goal and it changes the goal to the next one in the split that has been done by the `cases` command. Because of this, the captured proof step records an incorrect transformation, as if the proof step introduces a proof goal out of nowhere (e.g. as in step K13), or, if the user switches the proof branch before completing the previous one, an invalid goal transformation.

A more accurate record would have the "before-split" goal (e.g. the *in* goal of step K10) as the *in* goal. However, it is not correct either, because the `next` proof command does not do the splitting actually—it is part of the `cases` command. Furthermore, in this approach the captured proof features, which record why the proof step was taken, would not match the *in* goal in the proof step. Alternatively, the `cases` and `next` commands could be treated as "meta" commands that are not important to the high-level proof structure (e.g. goal rearranging commands in Isabelle ProofProcess are discarded as they do not change the high-level proof). However, this is also not ideal, because the `cases` command actually does split the goal into multiple sub-goals. Finding a more nuanced approach to handle and capture branching in Z/EVES proof, one that would take into account these considerations, is among the future work of this research.

Cleaning up the negation case

The `next` command selects the proof case with a negated lemma assumption: $\neg\ \mathrm{ran}\, extpid' = \{\}$. The "position" case allows for proof of the overall goal by

Intent: Cleanup *ProofSeq* (as decoration)
Narrative: Drop the implications depending on negated assumptions.
In features:

- *Assumption shape* ($\neg$?*p*1)
- *Assumption shape* (?*p*1 $\Rightarrow$?*p*2)

In goals (filtered):

$$\neg\ \mathrm{ran}\ extpid' = \{\} \wedge ?p1 \wedge (\mathrm{ran}\ extpid' = \{\} \Rightarrow extpid' = \{\}) \Rightarrow ?p2$$

Proof step: `simplify` ... (case # 1)
Out goals (filtered):

$$\neg\ \mathrm{ran}\ extpid' = \{\} \wedge ?p1 \wedge \Rightarrow ?p2$$

Step K14: **Cleanup**

assuming the lemma hypothesis (step K11). The "negation" case focuses on proving that the said hypothesis actually holds. As sketched earlier, the negated hypothesis is proved to be false, thus falsifying the whole assumptions conjunct.

Furthermore, the negated assumption allows for cleaning up the remains of lemma *gEmptyRan*. Because the assumption is **false**, the implication $\mathrm{ran}\ extpid' = \{\} \Rightarrow extpid' = \{\}$ can be dropped using a simple `simplify` command. The details of this proof step are captured in step K14. This proof step is always applicable when performing such a workaround, simulating a backward proof step. The strategy splits on lemma assumptions, thus the "negation" case always has the assumptions negated and permits the cleanup of the lemma itself. The captured proof step is tagged with a generic **Cleanup** intent, adding to the collection of generally applicable cleanup steps. The proof features in step K14 capture the generic scenario that the strategy is applicable for implications with negated assumptions.

After cleanup, the user is finished with simulating the backward proof step: the goal is to show that $\mathrm{ran}\ extpid' = \{\}$, even though the goal itself is not structured as such. To record the high-level idea (that this case split and proof branching is done as a backward proof step), the user introduces a higher-level proof step tagged with a **Do backward proof** intent (step K15). This high-level proof step wraps the case splitting and handling, providing a high-level description. Such abstract proof steps describe the overall idea of how the proof is achieved. For example, the current proof of lemma *tDeleteAllExtpid* can be described as follows:

1. **Zoom**

Intent: **Do backward proof** *ProofSeq*

Narrative: Show that ran $extpid'$ is empty by contradiction (simulating a backward proof step by negating the lemma assumption).

Children:

- (K9) *ProofSeq*: **Split on lemma assumption** ...
- *ProofParallel*: **Contradiction cases** ...
  - (K12) *ProofSeq*: **Use lemma conclusion** ...
  - *ProofSeq*: **Prove negated assumption** ...

In goals (flattened):[a]

$$?assms \land (\text{ran}\, extpid' = \{\} \Rightarrow extpid' = \{\}) \Rightarrow extpid' = \{\}$$

Out goals (flattened): ✓ *(none)*

[a] *?assms* variable denotes the same assumptions as in the *out* goal of step K6.

Step K15: **Do backward proof**

2. **Cleanup**
3. **Bridge predicate data structures**
4. **Do backward proof**
5. *(Nested, see step K18)* ***Prove at element level***

See Figure 12.4 for a structural overview. The last high-level step (**Prove at element level**) is actually nested within **Do backward proof**. This happens because the proof continues within a sub-branch and its high-level proof steps are contained within the parent *ProofParallel* element, resulting in this nested structure (a similar situation is encountered in the heap proof, see Section 11.2.3). The ProofProcess framework makes it possible to circumvent a nested structuring by exporting a branch result outside the parent *ProofParallel* element. See full discussion on wrapping and exporting proof branches in Section 4.3.7.

In fact, such a structure, where unfinished branch goals are exported to the "main" proof, can be done directly in Z/EVES proofs. Using the `next` proof command, the user may switch to a next proof branch without having finished it. If there are no more branches to switch to, the `next` command completes the branching and collects the unfinished branches back into a single goal. In the current proof, there is a single unfinished branch remaining (proving ran $extpid' = \{\}$). The user can close the branch by executing the `next` command: its goal becomes

a top-level one. Z/EVES ProofProcess system will notice that the case number has changed and will register the subsequent proof steps at the top level, not as part of a branch. The resulting ProofProcess tree structure will be the same as if the user exports the branch at the capture level.

The ability to abandon unfinished proof branches in Z/EVES results in proof "merge" points when capturing the proof process. Each proof branch would get a *ProofId* element referencing the same proof step, which closes the branching and continues the overall proof as a single goal. It is another case where a merging tree structure is needed (see further discussion and other examples in Section 4.3.7).

12.2.4 Completing the proof at the element level

At this point of the proof, the aim is no longer proving the initial $extpid' = \{\}$, but rather proving the fact $\mathrm{ran}\ extpid' = \{\}$ introduced by using lemma *gEmptyRan*. The goal itself, however, is structured with this fact negated (line 2):[8]

$$
\begin{array}{l}
?type\_assms \land ?other\_assms \land \\
\neg\ \mathrm{ran}\ extpid' = \{\} \land \\
used' = \mathrm{dom}\ state' \land \\
used' = \{\} \land \\
uprocs = ptype' \sim (\!|\{uproc\}|\!) \land \\
uprocs = \mathrm{dom}\ cdseg' \land \\
\{\} = \mathrm{dom}\ ptype' \land \\
\{\} = \mathrm{dom}\ tss' \land \\
\mathrm{dom}\ cdseg' = \mathrm{dom}\ dsseg' \land \\
\mathrm{dom}\ dsseg' = \mathrm{dom}\ msgq' \land \\
\mathrm{dom}\ msgq' = \mathrm{ran}\ extpid' \land \\
\Rightarrow \\
extpid' = \{\}
\end{array}
$$

The overall goal is proved if $\mathrm{ran}\ extpid' = \{\}$ or if this fact falsifies another assumption in the conjoined list. In that case, the whole assumptions predicate is **false** and thus the overall implication is proved (**false** implies anything).

The inspection of the goal shows that the equalities can be followed easily to get to $ptype' \sim (\!|\{uproc\}|\!) = \mathrm{ran}\ extpid'$. The assumption $\{\} = \mathrm{dom}\ ptype'$ comes from the other side to show the emptiness. Bridging the final step from the $\mathrm{dom}\ ptype'$ being empty to the relational image of inverse $ptype'$ map being empty, however, requires manual proof steps. Automatic Z/EVES proof commands lack lemmas to work at this level of reasoning to establish this proof argument.

[8] Only important parts of the goal are highlighted, type information and predicates not used in the proof are omitted from this goal presentation.

Intent: Zoom *ProofSeq* (as decoration)

Narrative: Zoom to set-element level from set level.

In features:

- *Used lemma* (*extensionality*)
- *Assumption term* ($\mathrm{ran}\ extpid' = \{\}$)
- *Preferred level of discourse* ($?elem \in \mathrm{ran}\ extpid'$)

In goals (filtered):

$$\neg\ \mathrm{ran}\ extpid' = \{\} \land$$
$$?p1 \Rightarrow ?p2$$

Proof step: `apply` *extensionality* `to predicate` $\mathrm{ran}\ extpid' = \{\}$ ... (case # 1)
Out goals (filtered):

$$\neg\ ((\forall x : \mathrm{ran}\ extpid' \bullet x \in \{\}) \land (\forall y : \{\} \bullet y \in \mathrm{ran}\ extpid')) \land$$
$$?p1 \Rightarrow ?p2$$

Step K16: **Zoom**

Zooming to set-element level

Automation can be improved by adding new general lemmas to manipulate predicates at the level of sets and maps. However, this requires skills in conjecturing good lemmas in a correct shape for the Z/EVES theorem prover. Section 12.3 discusses how good lemmas can simplify the proof in this case study.

Alternatively, the user can seek assistance at a lower level of discourse in proof. Z/EVES has a rich set of lemmas about set-membership in its base library. The user can "zoom" into the proof to go from reasoning about sets to reasoning about set elements. Step K16 captures this as a **Zoom** high-level intent. Just like with previous instances of the **Zoom** intent, "zooming" early can introduce too many low-level facts into the goal and complicate the proof. Thus the user zooms to the set-element level only after advancing the proof enough at the higher (set) level.

To change the level of discourse, the user applies the *extensionality* lemma to the predicate $\mathrm{ran}\ extpid' = \{\}$. This lemma replaces set equality with bidirectional set membership: every element of one set must belong to the other and vice-versa. The *out* goal of step K16 records the application results. This complex-looking negated predicate becomes a simple $x \in \mathrm{ran}\ extpid'$ after quantifier elimination and rewriting, and achieves the intended level of discourse.

Intent: Prove automatically *ProofSeq* (as decoration)
Narrative: Complete the proof blindly using automatic proof command.
In features:

- *Used lemma* (*inImageInv*)
- *Assumption term* (*ptype′* ~ (|{*uproc*}|))

Out features:

- *Number of goals* (0)

In goals (filtered): ...
Proof step: `prove` ... (case # 1)
Used lemmas: *inNull*, *inImageInv*, *weakening*, *pfun_sub* and others.
Out goals (filtered): ✓ *(none)*

Step K17: **Prove automatically**

Blind proof

At the lower level of discourse, the user blindly tries one of the automatic proof commands in Z/EVES: `prove [by rewrite]`. This command performs quantifier elimination, hypotheses rearrangement, equality substitution and goal rewriting in a loop a number of times, until the goal no longer changes [MS97]. At the set-element level, the automatic proof command succeeds and completes this proof branch. The proof step is captured by the ProofProcess framework as **Prove automatically** and is listed in step K17.

Blindly running automated proof commands and similar tools (e.g. the Sledgehammer automatic theorem provers in Isabelle) is a tactic taken frequently by the users. This is particularly the case in industrial settings, where *how* the proof is achieved is not as as important as getting it done. A blind `prove by reduce` command early in the proof is usually attempted in Z/EVES proofs. If it succeeds, the user is often not even bothered to get familiar with the goal. The approach is understandable and modern proof assistants should run such automatic proof commands in the background,[9] "just in case" they succeed. If so, then the proof is found and the high-level idea may not be as important. The approach, however, is not without problems: the success of "blind" application of automated proof commands may mask an error in the theorem. If the theorem is specified incorrectly, its proof may be trivial, but the user will have mistaken assurance that some property about the specification holds.

[9] Isabelle/jEdit already employs automatic execution of a number of proof methods [Wen14].

theorem rule inImageInv $[X, Y]$
$\forall f : Y \pfun X; S : \mathbb{P}\, X \bullet x \in f^{\sim}(\!|\, S \,|\!) \Leftrightarrow x \in \mathrm{dom}\, f \land f(x) \in S$

Figure 12.8: Lemma *inImageInv* (Z/EVES toolkit).

Nevertheless, **Prove automatically ("blindly")** is a valid proof step. It does capture the high-level idea that the user is attempting: that he does not actually have a strategy and at this point attempts the proof blindly. The "blind" strategy is also reusable: in similar proofs, a blind application of the automatic `prove` command may also complete the proof.

Furthermore, after doing the proof step, the user may be able to mark the important parts of the proof that allowed the strategy to succeed. For example, Z/EVES ProofProcess tracks all lemmas used by an automatic proof step. By inspecting them, the user may identify the important ones and recognise the direction that the prover has taken. In step K17, the user highlights lemma *inImageInv* as an important one. This lemma (listed in Figure 12.8) links the set membership in a relational image of inverse function with set membership in the domain of said function. As discussed earlier, this link is essential to complete the proof. "Zooming" to the set-element level allowed Z/EVES to use this lemma.

By recognising the key *inImageInv* lemma among the used lemmas, the user can infer why the proof succeeds. Then he can mark the important parts of the proof retroactively, as if this was indeed the intended proof direction. The proof features in step K17 record that having the relational image of an inverse function is key to using this lemma. Furthermore, the "blind" strategy is restricted to be used only if it completes the proof by marking *Number of goals* (0) as an *out* proof feature. With hindsight, the user constructs a proof step that identifies the features of why the proof succeeds. The extracted strategy presents multiple scenarios for reuse: e.g. if a similar data structure is encountered ($f^{\sim}(\!|\, S \,|\!)$), the user is encouraged to benefit from the *inImageInv* lemma; otherwise, if the user follows the higher-level strategy in a similar proof but does not have such a data structure, the captured proof will hint at blindly trying to complete the proof.

With the proof complete, further higher-level proof steps can be identified and tagged in the captured ProofProcess tree structure (see Figure 12.4 for overview). For example, the last two proof steps are tagged with **Prove at element level** intent (step K18). This highlights a higher-level strategy: try zooming to a set-element level and then blindly prove using the lower-level lemmas in the Z/EVES toolkit.

Intent: **Prove at element level** *ProofSeq*
Narrative: Zoom to the set-element level of discourse and prove blindly.
Children:

- (K16) *ProofSeq*: **Zoom** ...
- (K17) *ProofSeq*: **Prove automatically** ...

Step K18: **Prove at element level**

Intent: **Finish split** *ProofSeq* (as decoration)
Narrative: Finish case split by closing the last proof branch.
In features:

- *Case number* (#1)

In goals: ✓ *none*
Proof step: `next` ...
Out goals: ✓ *(none)*

Step K19: **Finish split**

Completing proof branches

The last proof step completes the "negation" proof branch. However, to complete the overall proof, one more instance of the `next` proof command is needed. It collects the (possibly unfinished) goals of each proof branch back into the top-level goal. At this point in the *tDeleteAllExtpid* proof, all proof branches are complete, thus the overall proof is also complete.

Step K19 records the minimal proof process information about this proof step. To finish the split, the case number must be #1. In Z/EVES, the higher case split numbers are addressed first, thus the final proof branch always has case number #1. There are no more goal transformations to record in step K19. Because of this, recording the proof step itself is somewhat redundant. This is another issue with the current handling of `cases`/`next` proof commands in Z/EVES ProofProcess (see also steps K10 and K13). One approach is to drop the capture of these proof steps altogether, but encode the proof commands in the replay tools.

12.3 Automating proof with lemmas

The proof of lemma *tDeleteAllExtpid* is quite convoluted for a simple problem. The sketched manual proof in Section 12.2 shows that the goal follows by simply tracing the relationships between process table variables. However, due to the lack of appropriate lemmas, the prover cannot discharge the conjecture automatically.

The captured proof process and marked important proof meta-information produce a clearer big-picture view of the proof. When tagging the proof intent, the user is forced to express the high-level idea of how the proof is advanced. Furthermore, marking the important proof features isolates the crucial parts of the goal from the clutter and allows clarification of the proof process. In the proof of lemma *tDeleteAllExtpid*, two points stand out in particular:

- The **Bridge predicate structures** (K6) proof step requires three proof commands to insert a lemma and clean up its assumptions. Better automation of the lemma use could simplify its manual insertion (including the need to fully qualify the `use` proof command in step K7), or avoid it altogether.

  Furthermore, the shape of lemma *gEmptyRan* used to bridge the data structures (i.e. the implication) requires a backward proof step later (see Section 12.2.3). Simulating the backward proof step requires additional proof commands and takes the focus away from the proof, requiring construction of the workaround in Z/EVES.

- The **Prove at element level** (K18) proof step goes to a lower level of discourse in the proof, where it can utilise additional lemmas about set-membership in the Z/EVES toolkit. While being a valid strategy, it highlights that too few appropriate lemmas are available at the higher level of discourse (sets and maps). If said lemmas were available, the zooming step could be avoided and the prover could solve the proof automatically at the level of sets.

As explained earlier, the original proof of lemma *tDeleteAllExtpid* was done by the user without extensive experience in theorem proving and, particularly, without intimate knowledge of the Z/EVES theorem prover. Because of that, lemma *gEmptyRan* was formulated in a way that inhibited its automatic use. The manual use of this lemma introduces significant inefficiency, thus the lemma needs to be reformulated. Figure 12.9 lists the new generic *lEmptyRan* lemma, specified as a *rewrite rule*. Instead of an implication, the lemma states that the *range* of a relation

theorem rule lEmptyRan $[X, Y]$
$\forall P : X \leftrightarrow Y \bullet \mathrm{ran}[X, Y]\ P = \{\} \Leftrightarrow P = \{\}$

theorem rule lEmptyDom $[X, Y]$
$\forall P : X \leftrightarrow Y \bullet \mathrm{dom}[X, Y]\ P = \{\} \Leftrightarrow P = \{\}$

theorem rule lEmptyFlip
$\{\} = S \Leftrightarrow S = \{\}$

Figure 12.9: General lemmas about empty sets and maps.

```
proof tDeleteAllExtpid
  invoke DeleteAllProcesses;
  invoke ΔPTab;
  invoke predicate PTab';
  prove;
```

Figure 12.10: New proof of theorem *tDeleteAllExtpid*.

being empty is the same as if the relation itself (a map in the current proof) is empty. Furthermore, a counterpart lemma about the *domains* of relations—*lEmptyDom*—is also specified. As *rewrite rules*, these lemmas are used automatically by the prover during the rewrite steps—the shape of the lemma is chosen to match the Z/EVES requirements. Furthermore, an additional lemma *lEmptyFlip* is needed: it transforms empty-set equalities into a canonical form used by most lemmas (e.g. $\{\} = \mathrm{dom}\ ptype'$ in step K8 would be replaced with $\mathrm{dom}\ ptype' = \{\}$). The latter shape is more suitable for automated use in Z/EVES.

Lemmas *lEmptyRan* and *lEmptyDom* address the issues mentioned in both the points above: they automate the use of facts within the previous lemma *gEmptyRan*, as well as introduce additional lemmas to reason about maps at the level of sets. With the generic lemmas in Figure 12.9 added to the specification, the overall proof of lemma *tDeleteAllExtpid* can be found automatically: the new proof steps are listed in Figure 12.10. With the new lemmas available, the proof can actually be shortened to just a single line: `prove by reduce`. This is the most powerful proof tactic in Z/EVES: it expands all datatypes and performs the rewriting steps in a loop. However, full expansion clutters the goal and finding the proof takes much longer. The proof listed in Figure 12.10 is a good compromise between the manual effort needed and the speed of the prover: the zooming part is quite straightforward and the proof is fully automatic thereafter.

The addition of generic lemmas simplifies the proof to the extent that it may no longer be interesting to capture it using the ProofProcess framework. The automatic `prove` command performs a large number of important rewrites and identifying them all as higher-level proof steps is an unnecessary overhead. The redundancy of the ProofProcess framework and the AI4FM tools in these situations should not be seen as a negative: after all, the AI4FM project aims to help the user in the places where the automation fails, i.e. it tries to supplement the automatic tools, not replace them. Nevertheless, the user can still mark the lemmas listed in Figure 12.9 as important, together with some other important lemmas traced from the rewriting process. This way the user can indicate why the automation is successful—in similar proofs the user could check whether their analogues are available for the new problem.

Furthermore, the capture of high-level ideas using the ProofProcess framework makes the user think about the proof and identify that such lemmas are needed and useful. Thus the recording of a higher-level description can be useful for proof strategies, but it can also invoke a more thoughtful approach to theorem proving, instead of blindly attempting all available proof commands until successful.

A good selection of generic lemmas simplifies the proof of *tDeleteAllExtpid*. Nevertheless, even clumsy proofs without good lemmas, such as captured in this case study, can yield reusable strategies. The next section discusses reuse of this awkward proof for similar lemmas.

12.4 Reuse of awkward strategy

The high-level proof process of lemma *tDeleteAllExtpid* captured in Section 12.2 can give rise to reusable strategies despite being quite convoluted. The original formal development of the separation kernel [Vel09], which includes this proof, contains a number of instances where parts of these high-level ideas are reused in similar proofs. Proof simplification as described in Section 12.3 is a recent development, thus the other proofs in the original development have not benefited from the newly added generic lemmas.

The proof of *tDeleteAllExtpid* only verifies a single property: that the $extpid'$ variable is empty after executing the *DeleteAllProcesses* operation. Proving the same property for all other variables yields very similar proofs. For example, proving that the $dsseg'$ variable is also empty after the execution of *DeleteAllProcesses*

```
theorem tDeleteAllDsseg
  DeleteAllProcesses ⇒ dsseg′ = ∅

proof tDeleteAllDsseg
  invoke DeleteAllProcesses;
  invoke ΔPTab;
  invoke predicate PTab′;
  prenex;
  use gEmptyDom[ℤ, ℤ × ℤ][A := PID, B := SDesc, P := dsseg′];
  rearrange;
  rewrite;
  split dom dsseg′ = {};
  cases;
  simplify;
  next;
  simplify;
  apply extensionality to predicate dom dsseg′ = {};
  prove;
  next;
```

Figure 12.11: Theorem *tDeleteAllDsseg* and its proof in Z/EVES.

succumbs to the same proof commands: the only difference is the variable names (*dsseg′* instead of *extpid′*), associated types and focusing on the domain of *dsseg′* (see Figure 12.11). Because dom *dsseg′* variable is used within the *PTab* invariants (cf. ran *extpid′*, see Figure 12.1), an analogous lemma *gEmptyDom* is used instead of *gEmptyRan*. The lemma states that an empty domain of a map (relationship) implies that the map itself is empty. It is inserted in the proof manually and used by simulating a backward proof step, just as in the proof of lemma *tDeleteAllExtpid*.

This is a trivial example of the strategy reuse: the same proof process applies to a similar proof almost intact. Nevertheless, manual intervention is needed to chase the correct types, appropriate lemma and specific sub-terms to split or apply lemmas on. However, if the strategy gets automated and such a proof is discharged automatically without manual intervention, it will be beneficial for industrial-size formal developments.

Furthermore, *partial* strategies extracted from the captured proof can be reused in a number of cases. In the original formal development of a separation kernel [Vel09], a brief search reveals that the following strategies were used within the proofs about the process table:

- **Prove at element level** (step K18, i.e. "zooming" to the set-element level using the *extensionality* lemma and then proving automatically), is used four

times. Particularly, this step is used when the proof involves variables with a relational image of an inverse function: better lemmas are available at the set-element level in the Z/EVES toolkit.

Two of these lemmas (*gPTabEmptyDprocs* and *gPTabEmptyUprocs*, see [Vel09] for full details) are noteworthy: they are used to simplify later proofs about individual *PTab* variables by introducing the fact that:

$$ptype^{\sim}(\!|\{uproc\}|\!) = \{\} \text{ (same for } dproc\text{)}.$$

Basically, these lemmas split the proofs in a similar manner to that described earlier. Then, instead of doing the **Prove at element level** step everywhere, these lemmas are used.

- **Bridge predicate data structures** (step K6) is used 10 times, twice using the *gEmptyRan* lemma, the rest using the *gEmptyDom* lemma.
- The **Zoom** steps (e.g. step K1) at the beginning of the proof are employed widely: in such industrial-style proofs expanding the necessary schemas and definitions is an almost universal first step.

The reuse of the full proof process or just parts of the overall strategy suggests that proof families exist even in small formal developments like the separation kernel in [Vel09]. Formal verification can be done successfully by replaying the same high-level ideas and without spending additional effort to generalise proofs, extract lemmas, etc. Even the "awkward" proof of lemma *tDeleteAllExtpid* yields viable reusable information—if that gets the similar proofs done, there is nothing wrong with it. In industrial settings, results matter more rather than "tidy" lemmas, data structures or perfect strategies.

CHAPTER 13

Conclusions and further work

This thesis investigates capturing reusable interactive proof ideas, via abstraction and holistic inspection of the overall proof process. The proposed abstractions and other data points provide rich descriptions of interactive proofs. Such descriptions can give rise to proof strategies that help tackle similar proofs; or just provide a comprehensive view of how proofs have been discovered—capturing the expert's insight rather than low-level instructions to the theorem proving system.

This chapter recaps the key points of this thesis, from the proposed proof process representation to the system implementation and proof process capture evaluation using case studies. Also, this chapter draws directions for further research. These include avenues towards extracting and replaying proof strategies (including testing the existing approaches) as well as new applications for the captured proof process data.

13.1 Thesis summary

A review of related work on extracting proof strategies highlights the limitations of the use of low-level proof scripts as the source for strategies. Some success has been found in proof reuse using analogy, particularly in the area of *proof planning*, where the use of higher-level abstractions (such as "specifications" of proof tactics) enable

adapting one proof plan to similar proofs. Capturing important abstractions of the interactive *proof process* (and in this way representing the theorem proving expert's insight) is the aim of this research. Furthermore, the focus is on the industrial formal verification proofs, where automatic generation of proof obligations and reuse of data structures give rise to "families" of similar proofs.

This thesis proposes a ProofProcess model to represent the important information about an interactive proof development. The important proof steps are described using a *proof intent* "tag" with various *proof features* marking the terms within the goal, used lemmas and other proof meta-information relevant to the particular proof step. This information provides an abstract description of a proof step and can be captured at any granularity. Furthermore, the proposed model provides a representation for *proof structure*, can capture multiple *proof attempts* and establishes links between the abstract proof step descriptions and the actual prover commands "implementing" them.

The generic core model allows extensions for prover specific data, enabling integration with different theorem proving systems. Furthermore, supplementary *proof history* data is easy to capture during interactive proof and unlocks new opportunities: from enhancing the proof process capture and analysis to calculating proof metrics, assisting with proof tutoring, etc.

Automating the capture of proof process meta-information is a difficult challenge. The thesis presents approaches to infer the proof structure from low-level proof commands as well as to recognise different proof attempts. In the meantime, other details such as the proof intent and proof features need to be recorded manually (the thesis proposes approaches to infer this data automatically).

The presented ideas are implemented within the ProofProcess framework—a proof process capture system. The thesis presents a description of the system as well as the implementation details. Integrations with two theorem provers (Isabelle and Z/EVES) are available, extending and reusing the generic core framework. The system "wire-taps" the prover communication. It captures and analyses the low-level data as well as enables the user to mark the important facets of the proof, thus constructing the proposed high-level representation of the interactive proof and the expert's insight.

The proof process capture approach presented in this thesis is illustrated and evaluated using two case studies, each done using a different theorem prover. They show how the proof can be represented at an abstract level, carry both proof-specific and domain-specific descriptions, how proof features are used to capture

key ideas about particular proof directions, how different prover commands comprise alternative realisations of the same high-level proof intent, etc.

Furthermore, the evaluation involves reusing the captured high-level proof process information for similar proofs. *Analogy* is employed to construct the similar proofs, as strategy extraction is not yet available from the AI4FM project. However, the case studies propose what the extracted strategies would be like, and the general ideas about how the captured proof process data facilitates strategy extraction are also presented in the thesis.

13.2 Conclusions and discussion

The research presented in this thesis is a combination of tangible results (e.g. the abstract model of high-level interactive proof process, the built prototype systems and the case studies demonstrating the validity of the proposed approach) as well as ideas on how to produce such results in the areas where the current execution is lacking. Together they describe a system that can evaluate the thesis hypotheses: that enough information can be extracted about interactive proof process in an automated manner to facilitate creation of reusable, high-level proof strategies. The implementation of the prototype shows that such a proof capture system is viable. The case studies confirm the main hypothesis and show that proofs can be reused even without strategy extraction. Improving and polishing the prototype implementation, also implementing the full system and evaluating it with large-scale industrial case studies in order to prove the hypotheses empirically and in an assured manner is left for future work.

A significant question underpinning the H_1 hypothesis is "how does one do proof?", more specifically: "how to describe interactive proof?". This thesis explores the question extensively, with results including an abstract model of proof processes; categories and examples of proof intents and proof features that can be used to describe how proofs are developed; an implemented prototype system enabling proof process capture and description; case studies illustrating the discussion of necessary and sufficient proof description for reuse; etc. The results show that the approach works and can capture high-level descriptions of interactive proof processes. The prototype framework supports multiple theorem provers. Though the currently supported Isabelle and Z/EVES provers have similarities, the framework is designed to accommodate other theorem provers and proof styles.

The ultimate goal of learning proofs strategies from an expert is approached from a pragmatic standpoint. The aim is to succeed in more straightforward cases, providing proof assistance where possible, without aiming for *perfect* proof strategies or their perfect specifications. The proposed model leans toward interactive proofs constructed using mechanised proof assistants, rather than aiming to solve a more general question of "how is proof constructed in general?". Utilising proof features outside the scope of the current goal is novel, as is allowing free-style description rather than restricting to a small set of predefined building blocks to describe proofs. Of course, the description flexibility may create obstacles in strategy replay, but is needed to express the expert's proof insight. Furthermore, the balance between precise and exhaustive description of proof versus quick hints and under-specification tilts towards the latter: the use cases show that strategies can be identified and reused from the limited description. Relying on the actual theorem prover to verify the proof gives the flexibility to err in corner cases.

The types of strategies that the proposed approach would facilitate can range from generic, widely reusable strategies to problem-specific ones, reusable within a very narrow family of proofs. Discovering new generic strategies would be very valuable, akin to developing (or rather, cataloguing) design patterns in software engineering. However, the case studies in this thesis illustrate that the majority of reusable proof strategies are problem-specific. learning and replaying them, however, would still improve the overall automation of using formal methods.

A significant contribution of this thesis is the development of a platform to capture data about interactive proof. Capturing and having access to data about interactive proof can yield interesting case studies and insights (e.g. see the discussion in Section 13.4), enabling various further research directions. With proof process capture, one can start running large-scale studies on interactive proof processes in industry; or investigate how to infer or learn advanced proof features automatically. Thus the research presented in this thesis is a valuable contribution enabling such directions.

The results presented in this thesis are limited when it comes to user-oriented surveys and studies. The examples of proof descriptions in this thesis come from the thesis author and colleagues in the AI4FM project. Enabled by data capture, one can start running studies of larger scale: how do people describe proofs? How does the process of developing proofs change if one is asked to describe and qualify the choices made? The kernel case study shows that having to explain how the proof is advanced makes the user identify better proof approaches (Section 12.3).

Part of this thesis focuses on building tools to improve theorem proving. The state of prover IDEs and the usefulness and adequacy of available tools warrants a wider discussion. Reports from industry highlight the preference of automated tools (automated provers, model checkers, etc) to interactive proof. This statement can be a signal of inadequate tool support in the latter area. Theorem provers have been available for many years, however their functionality and convenience are lacking when compared, for example, to modern IDEs for software development. There are a number of areas that can be improved without large effort. For example, identifying one's place in the proof is important: is would be useful to have a high-level overview of the proof, a description of what is happening. This thesis and the prototype implementation takes steps towards improving the state of the art. For example, goal filtering to identify "affected" parts (Section 8.2.3) has not been seen in proof assistants, yet is not difficult to implement and provides a convenient way of viewing goals, particularly in industrial-scale developments. Other avenues towards a better theorem proving interface are also explored: the capture of high-level proof description can improve the usability (see discussions in Section 13.4.2).

13.3 Future work

The scope of the overall goal of this thesis—to (automatically) capture the expert's high-level proof process—is very large. This thesis builds the foundations that can be taken further in various directions. A number of avenues and approaches to improve the current state of work are proposed within the thesis, such as ways to infer proof process data (Chapter 6), better user interface solutions or prover integration (e.g. in Chapter 8, Sections 9.1.2 and 10.2), etc. This section reiterates the key future improvements to proof process capture as well as outlines additional directions for further research, building up the ProofProcess platform towards the AI4FM goal of reusing proof strategies.

13.3.1 Capturing proof process

This thesis presents an architecture, a prototype implementation and a brief evaluation of a proof capture system. Further improvements to the current state of research and the implemented system are discussed in detail within the thesis. The following paragraphs provide a summary of this future work, categorising it into things that can be addressed immediately and topics that would take longer

to investigate and build. This information can be used to direct research and development resources towards improving the proof process capture system.

The immediate areas where the ProofProcess system and approach can be improved are the following:

- Polish and improve the user interface. The current solutions focus on basic data manipulation, but they need to be more user-friendly to entice new users into providing the data.
- Better support for deriving new proof attempts. Currently high-level proof process data is not moved onto a new attempt when it branches off the old one (see Section 8.6.2).
- Improve prover integration. Upgrade proof capture and analysis functionality when necessary APIs become available within the Isabelle theorem prover (see Section 9.1.2). The Z/EVES integration can also be improved, as discussed in Section 10.2.
- Support tracing of lemma usage in Isabelle either by parsing the proof terms or using the existing tracing mechanisms (see Section 6.5.4). More research is needed on the best way to mark the important features of the used lemmas.
- Support straightforward proof features. Section 6.5 outlines some proof feature types that are not difficult to infer automatically. Features such as *Top symbol* are straightforward to implement to be suggested to the expert. Proof context features such as *domain* or *provenance* can be easily derived.
- Support inferring proof intent in straightforward cases. For example, finding the same proof command or the same lemma being used in a previous proof can suggest the same proof intent for reuse (Section 6.4).
- Evaluate the system using large-scale case studies involving real industrial formal developments and proofs, different users of varying expertise, etc.

In addition to these medium term improvements, longer term research goals can be outlined:

- Implement a library of *known* proof features (see Section 6.5.1). The order of implementation could be informed by the proof process data capture: e.g. by first support the most popular features. The *shape* proof features are often

used and should be prioritised. Their implementation could reuse similar capabilities (e.g. unification) within theorem provers.

- Support matching captured proof process data with previous proofs. This requires generalisation of proof features, proof structure, etc.
- Support re-running the proof history. The current system captures all proof history including proof script versions. Section 13.4.2 suggests using this information for "proof movies", but it would be beneficial to at least re-run the proof script to perform new analysis.
- Integrate with other theorem provers. For example, supporting proof process capture for Rodin toolset would give access to a library of large-scale industrial proofs to evaluate the system.
- Evaluate the benefits of proof *attempts* as a source of proof strategies. The design supports capture of multiple proof attempts with the aim that even failed attempts can yield reusable strategies. This claim needs to be evaluated in larger case studies. Furthermore, the use of failed attempts as *negative* information can also be explored.
- Develop truly modern, immersive user interfaces for the ProofProcess system and theorem provers. Marking important proof features should be a streamlined process to avoid distracting the expert from the task of actually doing the proof.
- Extract and replay proof strategies. The issue of how to generalise the captured proof process data into reusable proof strategies is still open within AI4FM and needs to be explored. Replaying strategies also has a number of open questions, as discussed in the next section.

A number of these future research and development goals are independent and can be done in a modular manner. For example, adding support for new proof features can be done one-by-one, thus each implementation can be of a small scale and distributed among different developers.

13.3.2 Testing proof strategies

Proof strategies are closely aligned with proof process capture. The captured proof process information is the data source to extract strategies but also affects their

development: e.g. the strategies must accommodate the different proof vocabularies and proof styles. In the other direction, development of proof strategies informs the research on proof process capture: enough information needs to be captured to facilitate strategy extraction.

The case studies in Chapters 9–10 simulate proof strategies using *ad hoc* analogy to reuse the captured proof process data. Chapter 7 describes the research within the AI4FM project on providing more detailed models and representations of proof strategies. In particular, two approaches are presented: an abstract top-down model of proof strategies and an implementation using *proof-strategy graphs*.

Some of the differences between the approaches can be attributed to coming from different sides of the same problem: top-down versus bottom-up development. The abstract model of proof strategies in Section 7.2 establishes a high-level view with different types of meta-information used in describing and matching proof strategies. Furthermore, it focuses on finding the most applicable strategy step every time, enabling seamlessly switching between strategies coming from different proofs and thus constructing an optimal proof from the start. Proof-strategy graphs (Section 7.3) encode multi-step strategies—the approach has been developed by generalising actual proofs. Strategy replay follows the selected strategy until the end or until it gets stuck, with subgoals continuing along the edges of a proof-strategy graph. In this case, the proof search is more constrained and follows some earlier proof (from which the strategy got extracted) to its entirety.

Further experiments and an implementation workbench with real-world formal verification examples are needed to resolve the balance between these approaches. In fact, Chapter 7 discusses that there may be some optimal middle ground for strategy replay, where the new most applicable strategies are queried when an existing strategy gets stuck; alternatively, the user could choose how often to search for new strategies.

13.3.3 Lemma discovery

During proof process capture, important lemmas can be marked using *used lemma* proof features (Section 4.2.1). They record the key lemmas needed by the proof step; further details about the lemma shape and its other properties could also be marked. When the proof is reused, a *similar* lemma may be needed.

This thesis suggests that the user would be capable of recognising what similar lemma is needed in most cases, particularly if important features of the original

lemma had been identified. However, automating this process and discovering the lemmas without user interaction would be beneficial.

Lemma discovery (or *lemma synthesis*, *theory exploration*) is an active research area in supporting both the interactive and automated theorem proving. IsaScheme [MRMDB10] uses 'schemes' (terms in higher-order logic) as templates to generate new definitions and conjectures. IsaCoSy [JDB11] only generates irreducible terms, implemented as constraints to the lemma synthesis process. In both systems the generated candidate lemmas are filtered through a counter-example checker and an automated theorem prover to only generate valid ones. Heras et al. [HKJM13] use lemma discovery with machine learning to generate similar lemmas in ACL2 proofs. Machine learning is employed to identify proofs similar to the current one, then *lemma analogy* is used to adapt the lemmas in the similar proofs to the current one.

These approaches could successfully complement the proof strategy replay process in AI4FM. The *used lemma* proof features capture the important lemmas—in similar proofs, lemma analogy could be used to automatically generate appropriate sibling lemmas. When the used lemma shape is recorded, it could be used as a template ('scheme') within the IsaScheme approach, and so on. Further work is needed to investigate the relation between the quality and amount of proof information being captured and the success in generating necessary lemmas; as well as to build the integration with existing lemma discovery approaches.

13.3.4 Querying proof processes

The architecture of a proof process capture system proposed in this thesis separates the capture and storage of the data from its uses (Section 3.2.3). This improves the modularity of the main components and encourages further extensions and uses of proof process data that are independent of strategy extraction.

To facilitate the access to the captured data, a query interface is needed. It could be used to examine the captured data that matches some set of parameters, with applications in viewing the captured data, running analysis techniques or inspecting interesting facets of the proof processes. Even when capturing new proof process, querying is needed to link with the existing data: e.g. a newly captured *Attempt* needs querying for a correct *Proof* object to attach it to. More detailed queries are needed to check whether a new *Attempt matches* with some existing one, in case it is a replay or an extension of some previous proof (see also

Section 6.3). Currently the prototype ProofProcess system accesses the stored data by exhaustively traversing the whole data structure, starting from a top-level *proof store*. A proof process query API would abstract the concept of a proof store and provide a more convenient access to the captured data.

Developing a proof process query language and implementing a programming interface is among the future work. Aspinall et al. [ADL12, ADL13] propose a proof query language PrQL that supports *hiproofs*. As the ProofProcess data structures are similar to hiproofs, this work could be extended with support for proof features, multiple attempts, etc.

13.3.5 Capturing declarative proof

Declarative proof style allows explicitly stating the facts to be proven and using them to construct the proof argument. This style of proof is well suited for developing mathematical proofs and provides a human-readable representation similar to that found in textbooks. However, for industrial-style formal verification proofs, procedural proof style is preferred, as observed in Section 2.2.1.

Working towards a complete system for capturing *any* interactive proof style, it is important to support the capture of declarative proofs. The research presented in this thesis focuses on capturing procedural proofs. However, rudimentary support for declarative proofs is also proposed and implemented for the Isabelle/Isar proof language [WW07]. This section outlines the current state of this support and identifies the main issues for further work.

Isabelle/Isar allows switching between procedural and declarative proof styles within the same proof. In the declarative style, stated facts can be named and used as assumptions later in the proof. This establishes a relationship, where a proof step "depends" on previous proof steps that conjecture its assumptions. A single proof step can have a number of such assumptions, as illustrated in Section 9.2.3. The ProofProcess system captures these relationships and represents them within the graph structure for proofs (Section 8.6): e.g. each dependency is represented as an *edge* between the assumption and its use. The various assumption edges, unfortunately, make the overall graph structure of such proofs very complex.

The current approach tries to make both proof styles coexist within the same data structure. Section 9.2.3 discusses how the goals from the two styles are normalised. Furthermore, the "consumption" and "production" of assumption facts are captured as part of the *in/out goals*: e.g. if a proof step uses (depends on) some

assumption, it becomes one of the *in* goals. In a similar manner, if a proof step declares an assumption fact, it is recorded as one of the *out* goals. In each case, the assumptions are wrapped into *AssumptionTerm*s to differentiate them from the actual goals, which are wrapped into *JudgementTerm*s. Thus each declarative proof step has outgoing edges to where the declared assumption is used as well as to a (possibly) procedural proof that establishes the validity of the stated assumption. This proof structure is inferred automatically using the generic goal change analysis algorithm presented in Section 6.2.

The main issue with the current approach is that switching between the procedural and declarative style loses the linear account of how the goals change. The declarative proof graph with assumption edges has a large number of "merge points", which make the representation very complex when converted to a ProofProcess tree structure. Furthermore, describing high-level proof steps (i.e. "grouping" the proof commands) over a non-linear structure is also not intuitive. Some approaches exist for converting between proof styles (e.g. in [KW09, Whi13]), but conversion would lose the representation in which the user constructs the proof. It may be difficult to record the proof insight within an unfamiliar structure.

The current support for capturing declarative proofs is limited and further work is needed to investigate the best way to represent both declarative and procedural proofs. Furthermore, ideas on how proof strategies can be extracted from declarative proof need to be considered.

13.4 Other uses of proof process data

Capturing rich meta-information about proofs and the proof process can be beneficial for various applications in addition to strategy extraction. These opportunities show the usefulness of the proposed proof process capture system but further research is needed to explore them. This section discusses some of these research avenues that could use the captured proof process data.

13.4.1 Data for machine learning

Machine learning can be used to improve the premise selection of automated theorem proving, to cluster similar proofs or to identify proof patterns (see Section 2.3.2). The current approaches use quite low-level features of the proofs as the data source: the top symbol, the main operators, the number of subgoals, etc.

A dataset of captured proof process data as proposed in this thesis can provide richer data points for data mining. For example, the important operators or subterms could be used instead of taking the top symbol or all function names; proof intents give alternative names to proof steps; relationships between proof steps (both the branching structure and the levels of granularity) could be exploited; etc. Furthermore, the capture of all proof attempts enlarges the dataset needed for data mining: the failed attempts can carry proof information of general interest.

13.4.2 Proof explanation, teaching and training

High-level descriptions of proof processes can improve proof comprehension in various human-related use cases: from understanding the proof for proof maintenance to learning how to do interactive proof.

Understanding proofs

Getting oneself familiar with an existing proof is not a rare activity in interactive proof development. It is necessary to understand how the proof was achieved when trying to get to the bottom of someone else's proof, when training a new person to work on formal development, and even when coming back to one's own proofs after some period of time (e.g. for proof maintenance). In all these cases, the lost proof insight inflicts additional effort to understand the proof.

Communicating proofs is important in formal mathematics, where proofs are more complex and difficult in comparison to formal verification proofs. One solution is using the declarative proof style to construct the proof by explicitly stating the intermediate facts. Alternatively, a "wiki for formal mathematics" [TGMW10, TGM10, TKUG13] combines the recording of proof commands and their results with additional narrative to explain procedural proofs. In fact, the latter approach has a number of similarities to proof process capture proposed in this thesis. The proofs are captured by "wire-tapping" the prover communication; complete proof steps with their results are stored for independent inspection; etc.

The work presented in this thesis has similar capabilities in capturing and describing proofs. Proposed proof abstractions would further enhance proof inspection experience: users could view the proof at varying levels of granularity, with important proof features highlighted, etc.

Proof maintenance

The proposed proof abstractions can be beneficial for proof maintenance. When specification definitions, libraries or proof tactics change, the subsequent proofs are invalidated and may no longer work. A significant obstacle to fixing broken proofs is the loss of information about what made the proof work previously. Declarative proof style helps with this issue: explicitly stated intermediate goals act as checkpoints of what is expected in the proof. Captured proof descriptions would provide a similar benefit: information about the original intermediate goals and other proof properties would be available. Furthermore, *proof features* can highlight important parts of the proof, pinpointing the differences from the original proof and hinting at specific fixes.

This use case could be automated further by using proof strategies. For example, a proof strategy extracted by an AI4FM system from the original proof could be applicable to the changed goal. Then it could automatically produce a proof *similar* to the original.

"Watching" the expert

Assistance and training from existing theorem proving experts is very important when a new person starts doing formal verification work [AJK+12]. Similarly, teaching theorem proving to students can benefit from "learning by example", i.e. if the students can watch an expert do interactive proof.

The ProofProcess system presented in this thesis captures enough data (e.g. see Chapter 5) to enable viewing of "proof movies" about formal development and interactive proof process. The proposed approach is to animate an expert doing a previously captured proof so that students and trainees can follow his example and get familiar with the "big picture" of how proof is done.[1] This is particularly useful when the user lacks familiarity with the theorem proving system, used libraries or theorem proving in general.

Following the full development including backtracking, intermediate structuring and editing of the formal specification, as well as other important activities, can lead to easier understanding of what is going on and how the formal verification

[1] A similar functionality of recording "proof movies" is available from the Proviola tool [TGMW10], which aims to improve proof reviews by recording proof states to complement the proof script and allowing additional informal narrative to describe the proof.

is achieved. Furthermore, the proof intent and proof features information could help with descriptions of the high-level insight.

13.4.3 Towards proof process metrics

Industrial use of formal methods and formal verification is often argued to be more cost-effective than testing and certification of a comparable assurance [WLBF09]. However, estimating cost-effectiveness in advance requires development of proof process metrics. Andronick and Staples et al. [AJK+12, SJA+14] conclude that the *lines of proof*[2] metric is a poor size measure for proof productivity and cost.

The proof process data capture proposed in this thesis can provide new data points to evaluate and develop metrics about the proof process. For example, proof structure, granularity and proof features could be used for various *complexity* metrics; capture of different proof attempts and proof re-runs can provide data on proof rework or the difficulty of finding the correct proof; timestamped proof activities can yield data about the durations of proofs and the efficiency of interactive proof; etc. Further work is needed on both capturing significant examples and developing proof metrics, but the rich proof process meta-data can be beneficial in this research area.

[2] Akin to the *lines of code* metric in software engineering.

APPENDIX A

ProofProcess model

A.1 Core model

A.1.1 Top

$$
\begin{array}{lll}
\mathit{ProofStore} :: & \mathit{proofs} & : \mathit{Proof}\text{-}\mathbf{set} \\
& \mathit{intents} & : \mathit{IntentId} \xrightarrow{m} \mathit{Intent} \\
& \mathit{features} & : \mathit{FeatureId} \xrightarrow{m} \mathit{FeatureDef}
\end{array}
$$

A.1.2 Attempts

$$
\begin{array}{lll}
\mathit{Proof} :: & \mathit{goals} & : \mathit{Term}^{+} \\
& \mathit{label} & : [\mathit{Name}] \\
& \mathit{attempts} & : \mathit{AttemptId} \xrightarrow{m} \mathit{Attempt}
\end{array}
$$

where

$$
\mathit{inv\text{-}Proof}(\mathit{mk\text{-}Proof}(\mathit{goals}, \mathit{label}, \mathit{attempts})) \triangleq
\forall a \in \mathbf{rng}\, \mathit{attempts} \cdot \mathit{inGoals}(a.\mathit{proof}) =_m \mathit{goals}
$$

$$
\mathit{Term} = \mathbf{token}
$$

$$
\begin{array}{lll}
\mathit{Attempt} :: & \mathit{proof} & : \mathit{ProofTree} \\
& \mathit{derivedFrom} & : \mathit{AttemptId}
\end{array}
$$

A.1.3 Proof tree

$$
\mathit{ProofTree} = \mathit{ProofEntry} \mid \mathit{ProofSeq} \mid \mathit{ProofParallel} \mid \mathit{ProofId}
$$

$$\begin{array}{l} \mathit{ProofSeq} :: \mathit{info} : \mathit{ProofInfo} \\ \qquad\qquad \mathit{steps} : \mathit{ProofTree}^+ \end{array}$$

where

$$\begin{array}{l} \mathit{inv\text{-}ProofSeq}(\mathit{mk\text{-}ProofSeq}(\mathit{info}, \mathit{steps})) \triangleq \\ \quad (\forall i \in \mathbf{inds}\ \mathit{steps} \cdot i > 0 \Rightarrow \\ \qquad \mathit{inGoals}(\mathit{steps}(i)) =_m \mathit{outGoals}(\mathit{steps}(i\text{-}1))^1) \wedge \\ \quad (\exists s \in \mathbf{elems}\ \mathit{steps} \cdot \neg\ \mathit{is\text{-}ProofId}(s)) \end{array}$$

$$\begin{array}{l} \mathit{ProofParallel} :: \mathit{info} : \mathit{ProofInfo} \\ \qquad\qquad \mathit{branches} : \mathit{ProofTree}\text{-}\mathbf{set} \end{array}$$

where

$$\mathit{inv\text{-}ProofParallel}(\mathit{mk\text{-}ProofParallel}(\mathit{info}, \mathit{branches})) \triangleq \mathit{branches} \neq \{\}$$

$$\mathit{ProofId} :: \mathit{goals} : \mathit{Term}^+$$

$$\begin{array}{l} \mathit{ProofEntry} :: \mathit{info} : \mathit{ProofInfo} \\ \qquad\qquad \mathit{step} : \mathit{ProofStep} \end{array}$$

$$\begin{array}{l} \mathit{ProofStep} :: \mathit{inGoals} : \mathit{Term}^+ \\ \qquad\qquad \mathit{outGoals} : \mathit{Term}^* \\ \qquad\qquad \mathit{justification} : \mathit{Justification} \end{array}$$

$$\mathit{Justification} = \textsc{Trusted} \mid \textsc{Gap} \mid \mathit{ProofTrace}$$

$$\mathit{ProofTrace} = \mathit{NaturalDeduction} \mid \mathit{IsabelleTrace} \mid \mathit{ZEvesTrace} \mid \ldots$$

Proof tree functions

$$\mathit{inGoals} : \mathit{ProofTree} \to \mathit{Term}^+$$

$$\mathit{inGoals}(\mathit{ptree}) \triangleq \text{given by cases below}$$

$$\mathit{inGoals}(\mathit{mk\text{-}ProofEntry}(\mathit{info}, \mathit{mk\text{-}ProofStep}(\mathit{in}, \mathit{out}, \mathit{justif}))) \triangleq \mathit{in}$$

$$\mathit{inGoals}(\mathit{mk\text{-}ProofSeq}(\mathit{info}, \mathit{steps})) \triangleq \mathit{inGoals}(\mathbf{hd}\ \mathit{steps})$$

$$\begin{array}{l} \mathit{inGoals}(\mathit{mk\text{-}ProofParallel}(\mathit{info}, \mathit{branches})) \triangleq \\ \quad \mathbf{let}\ \mathit{brGoals} = [\mathit{inGoals}(b) \mid b \in \mathit{branches}]\ \mathbf{in}\ \mathbf{dconc}\ \mathit{brGoals} \end{array}$$

[1] Here $=_m$ is a multiset equality, as the order must be ignored.

$inGoals(mk\text{-}ProofId(goals))) \triangleq goals$

$outGoals : ProofTree \to Term^*$

$outGoals(ptree) \triangleq$ given by cases below

$outGoals(mk\text{-}ProofEntry(info, mk\text{-}ProofStep(in, out, justif))) \triangleq out$

$outGoals(mk\text{-}ProofSeq(info, steps)) \triangleq$
let $last = steps(\textbf{len}\ steps)$ **in** $outGoals(last)$

$outGoals(mk\text{-}ProofParallel(info, branches)) \triangleq$
let $brGoals = [outGoals(b) \mid b \in branches]$ **in dconc** $brGoals$

$outGoals(mk\text{-}ProofId(goals)) \triangleq goals$

$isDischarged : ProofTree \to \mathbb{B}$

$isDischarged(ptree) \triangleq \textbf{len}\ outGoals(ptree) = 0$

A.1.4 Proof info

$$\begin{array}{lll} ProofInfo :: & why & : [IntentId] \\ & inFeatures & : Feature\text{-}\textbf{set} \\ & outFeatures & : Feature\text{-}\textbf{set} \\ & narrative & : Text \\ & score & : Score \end{array}$$

$Intent = \textbf{token}$

$$\begin{array}{lll} Feature :: & name & : FeatureId \\ & params & : FeatureParam^+ \\ & type & : \textsc{User} \mid \textsc{Inferred} \end{array}$$

$FeatureParam = Term \mid \ldots$

$FeatureDef = KnownFtr \mid CustomFtr \mid \ldots$

$KnownFtr = TermFtr \mid ShapeFtr \mid UsedLemmaFtr \mid ContextFtr \mid \ldots$

$TermFtr = ExiSymFtr \mid TopSymFtr \mid ExiTermFtr \mid TypeFtr \mid \ldots$

$$ShapeFtr = MainTermShapeFtr \mid SubTermShapeFtr \mid \ldots$$

$$UsedLemmaFtr = UsedLemmaNameFtr \mid UsedLemmaShapeFtr \mid \ldots$$

$$ContextFtr = OriginFtr \mid ProvenanceFtr \mid DomainFtr \mid \ldots$$

$$\begin{aligned} CustomFtr :: \; & name \; : \; Text \\ & template \; : \; ParamTemplate \end{aligned}$$

$$ParamTemplate = \textbf{token}$$

$$Score = \textbf{token}$$

A.2 Proof history

A.2.1 Proof log

$$ProofLog :: activities : (Activity \times Timestamp)^*$$

$$Timestamp = \mathbb{N}$$

$$Activity = ProofActivity \mid DefActivity \mid \ldots$$

$$ProofActivity :: proofStep : ProofEntry$$

$$DefActivity = \textbf{token}$$

$$\begin{aligned} IsabelleTrace :: \; & \ldots \\ & source \; : \; [TextLoc] \end{aligned}$$

$$\begin{aligned} TextLoc :: \; & file \; : \; File \\ & offset \; : \; \mathbb{N} \\ & length \; : \; \mathbb{N} \end{aligned}$$

$$\begin{aligned} ZEvesTrace :: \; & \ldots \\ & source \; : \; [TextLoc] \end{aligned}$$

A.2.2 File history

$$FileHistoryProject :: files : FileId \xrightarrow{m} FileEntry$$

$$FileEntry :: versions : FileVersion^*$$

where

$$inv\text{-}FileEntry(mk\text{-}FileEntry(versions)) \triangleq$$
$$\forall i, j \in \textbf{inds}\ versions \cdot$$
$$i < j \Rightarrow versions(i).timestamp \leq versions(j).timestamp$$

$$\begin{aligned} FileVersion :: \ & contents : File \\ & timestamp : Timestamp \\ & syncPoint : \mathbb{N} \\ & checksum : Checksum \\ & syncChecksum : Checksum \end{aligned}$$

References

[ABD+06] Serge Autexier, Christoph Benzmüller, Dominik Dietrich, Andreas Meier, and Claus-Peter Wirth. A generic modular data structure for proof attempts alternating on ideas and granularity. In Michael Kohlhase, editor, *Mathematical Knowledge Management*, volume 3863 of *LNCS*, pages 126–142. Springer, 2006. doi:10.1007/11618027_9.

[ABH+10] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010. doi:10.1007/s10009-010-0145-y.

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[Abr07] Jean-Raymond Abrial. Formal methods: Theory becoming practice. *J. UCS*, 13(5):619–628, 2007. doi:10.3217/jucs-013-05-0619.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[AD10] Serge Autexier and Dominik Dietrich. A tactic language for declarative proofs. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP*, volume 6172 of *LNCS*, pages 99–114, Edinburgh, Scotland, July 2010. Springer. doi:10.1007/978-3-642-14052-5_9.

[ADL10] David Aspinall, Ewen Denney, and Christoph Lüth. Tactics for hierarchical proof. *Mathematics in Computer Science*, 3(3):309–330, 2010. doi:10.1007/s11786-010-0025-6.

[ADL12] David Aspinall, Ewen Denney, and Christoph Lüth. Querying proofs. In Nikolaj Bjørner and Andrei Voronkov, editors, *LPAR-18*,

volume 7180 of *LNCS*, pages 92–106. Springer, 2012. doi:10.1007/978-3-642-28717-6_10.

[ADL13] David Aspinall, Ewen Denney, and Christoph Lüth. A semantic basis for proof queries and transformations. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *LPAR-19*, volume 8312 of *LNCS*, pages 53–70. Springer, 2013. doi:10.1007/978-3-642-45221-5_4.

[AHL+08] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert Schirmer, and Artem Starostin. The verisoft approach to systems verification. In Natarajan Shankar and Jim Woodcock, editors, *VSTTE 2008*, volume 5295 of *LNCS*, pages 209–224. Springer, 2008. doi:10.1007/978-3-540-87873-5_18.

[AI4] AI4FM. AI4FM project. http://www.ai4fm.org.

[AJK+12] June Andronick, D. Ross Jeffery, Gerwin Klein, Rafal Kolanski, Mark Staples, He Zhang, and Liming Zhu. Large-scale formal verification in practice: A process perspective. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *ICSE 2012*, pages 1002–1011. IEEE, 2012. doi:10.1109/ICSE.2012.6227120.

[BBHI05] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, June 2005. doi:10.2277/052183449X.

[BBM98] Patrick Behm, Lilian Burdy, and Jean-Marc Meynadier. Well defined B. In Didier Bert, editor, *B'98*, volume 1393 of *LNCS*, pages 29–45. Springer, 1998. doi:10.1007/BFb0053354.

[BCF+97] Christoph Benzmüller, Lassaad Cheikhrouhou, Detlef Fehrer, Armin Fiedler, Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, Wolf Schaarschmidt, Jörg H. Siekmann, and Volker Sorge. OMEGA: Towards a mathematical assistant. In William McCune, editor, *CADE*, volume 1249 of *LNCS*, pages 252–255. Springer, 1997. doi:10.1007/3-540-63104-6_23.

[BCJ84] Howard Barringer, J. H. Cheng, and Cliff B. Jones. A logic covering undefinedness in program proofs. *Acta Inf.*, 21:251–269, 1984. doi:10.1007/BF00264250.

[BDKK12] Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. Challenges and experiences in managing large-scale proofs. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *CICM 2012*, volume 7362 of *LNCS*, pages 32–48. Springer, 2012. doi:10.1007/978-3-642-31374-5_3.

[BFW09] Andrew Butterfield, Leo Freitas, and Jim Woodcock. Mechanising a formal model of flash memory. *Sci. Comput. Program.*, 74(4):219–237, 2009. doi:10.1016/j.scico.2008.09.014.

[BGJ09] Alan Bundy, Gudmund Grov, and Cliff B. Jones. Learning from experts to aid the automation of proof search. In Liam O'Reilly and Markus Roggenbach, editors, *AVoCS'09*, Technical Report of Computer Science CSR-2-2009, pages 229–232. Swansea University, Wales, UK, 2009.

[BGJ10] Alan Bundy, Gudmund Grov, and Cliff B. Jones. An outline of a proposed system that learns from experts how to discharge proof obligations automatically. In Jean-Raymond Abrial, Michael Butler, Rajeev Joshi, Elena Troubitsyna, and Jim C. P. Woodcock, editors, *Refinement Based Methods for the Construction of Dependable Systems*, number 09381 in Dagstuhl Seminar Proceedings, pages 38–42, Dagstuhl, Germany, 2010. Schloss Dagstuhl. URL: http://drops.dagstuhl.de/opus/volltexte/2010/2374.

[BHW06] Juan Bicarregui, C. A. R. Hoare, and J. C. P. Woodcock. The verified software repository: a step towards the verifying compiler. *Formal Asp. Comput.*, 18(2):143–151, 2006. doi:10.1007/s00165-005-0079-4.

[BN00] Stefan Berghofer and Tobias Nipkow. Proof terms for simply typed higher order logic. In Mark Aagaard and John Harrison, editors, *TPHOLs 2000*, volume 1869 of *LNCS*, pages 38–52. Springer, 2000. doi:10.1007/3-540-44659-1_3.

[BRS+00] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. Formal system development with KIV. In T. S. E. Maibaum, editor, *FASE 2000*, volume 1783 of *LNCS*, pages 363–366. Springer, 2000. doi:10.1007/3-540-46428-X_25.

[Bun88] Alan Bundy. The use of explicit plans to guide inductive proofs. In Ewing Lusk and Ross Overbeek, editors, *CADE*, volume 310 of *LNCS*, pages 111–120. Springer, 1988. doi:10.1007/BFb0012826.

[Bun91] Alan Bundy. A science of reasoning. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 178–198, 1991.

[BvHHS90] Alan Bundy, Frank van Harmelen, Christian Horn, and Alan Smaill. The oyster-clam system. In Mark E. Stickel, editor, *CADE 1990*, volume 449 of *LNCS*, pages 647–648. Springer, 1990. doi:10.1007/3-540-52885-7_123.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus - A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998. doi:10.1007/978-1-4612-1674-2.

[CB08] David Cooper and Janet Barnes. Tokeneer ID station EAL5 demonstrator: Summary report. Technical Report S.P1229.81.1 Issue: 1.1, Altran-Praxis, August 2008.

[CCL+14] Joey W. Coleman, Luis Diogo Couto, Kenneth Lausdahl, Claus Ballegaard Nielsen, Anders Kaels Malmos, Peter Gorm Larsen, Richard Payne, Simon Foster, Alvaro Miyazawa, Uwe Schulze, Adalberto Cajueiro, and Andre Didier. Fourth release of the COMPASS tool — Symphony IDE user manual. Deliverable number D31.4a, version 1.1, COMPASS project, September 2014. URL: http://www.compass-research.eu/Project/Deliverables/D31.4a.pdf.

[CFSV04] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004. doi:10.1109/TPAMI.2004.75.

[Cra07] Iain D. Craig. *Formal Refinement for Operating System Kernels*. Springer, 2007. doi:10.1007/978-1-84628-967-5.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996. doi:10.1145/242223.242257.

[Del02] David Delahaye. A proof dedicated meta-language. *Electr. Notes Theor. Comput. Sci.*, 70(2):96–109, 2002. doi:10.1016/S1571-0661(04)80508-5.

[DEPa] DEPLOY. DEPLOY project. http://www.deploy-project.eu.

[DEPb] DEPLOY. Event-B examples. http://deploy-eprints.ecs.soton.ac.uk/view/subjects/examples.html.

[DF03] Lucas Dixon and Jacques Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In Franz Baader, editor, *CADE*, volume 2741 of *LNCS*, pages 279–283. Springer, 2003. doi:10.1007/978-3-540-45085-6_22.

[DG96] L. Peter Deutsch and Jean-Loup Gailly. ZLIB compressed data format specification version 3.3. RFC 1950, Network Working Group, 1996.

[Die11] Dominik Dietrich. *Assertion Level Proof Planning with Compiled Strategies*. PhD thesis, Saarland University, Saarbrücken, Germany, 2011.

[DPT06] Ewen Denney, John Power, and Konstantinos Tourlas. Hiproofs: A hierarchical notion of proof tree. *Electronic Notes in Theoretical Computer Science*, 155:341–359, 2006. doi:10.1016/j.entcs.2005.11.063.

[Dun07] Hazel Duncan. *The Use of Data-Mining for the Automatic Formation of Tactics*. PhD thesis, School of Informatics, University of Edinburgh, 2007.

[Ecla] Eclipse.org. CDO model repository. http://eclipse.org/cdo.

[Eclb] Eclipse.org. Eclipse project. http://eclipse.org/eclipse.

[Emp] Peter Empen. Graph for scala. http://www.scala-graph.org.

[FH94] Amy P. Felty and Douglas J. Howe. Generalization and reuse of tactic proofs. In Frank Pfenning, editor, *LPAR*, volume 822 of *LNCS*, pages 1–15. Springer, 1994. doi:10.1007/3-540-58216-9_25.

[FJ10] Leo Freitas and Cliff B. Jones. Learning from an expert's proof: AI4FM. In Tom Ball, Lenore Zuck, and Natarajan Shankar, editors, *UV10 (Usable Verification)*, November 2010.

[FJV14] Leo Freitas, Cliff B. Jones, and Andrius Velykis. Can a system learn from interactive proofs? In Andrei Voronkov and Margarita Korovina, editors, *HOWARD-60. A Festschrift on the Occasion of Howard Barringer's 60th Birthday*, pages 124–139. EasyChair, 2014.

[FJVW13] Leo Freitas, Cliff B. Jones, Andrius Velykis, and Iain Whiteside. How to say why (in AI4FM). Technical Report CS-TR-1398, School of Computing Science, Newcastle University, October 2013.

[FJVW14] Leo Freitas, Cliff B. Jones, Andrius Velykis, and Iain Whiteside. A model for capturing and replaying proof strategies. In Dimitra Giannakopoulou and Daniel Kroening, editors, *VSTTE 2014*, volume 8471 of *LNCS*, pages 183–199. Springer, 2014. doi:10.1007/978-3-319-12154-3_12.

[FW08] Leo Freitas and Jim Woodcock. Mechanising Mondex with Z/Eves. *Formal Asp. Comput.*, 20(1):117–139, 2008. doi:10.1007/s00165-007-0059-y.

[FW14] Leo Freitas and Iain Whiteside. Proof patterns for formal methods. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014*, volume 8442 of *LNCS*, pages 279–295. Springer, 2014. doi:10.1007/978-3-319-06410-9_20.

[FWF09] Leo Freitas, Jim Woodcock, and Zheng Fu. POSIX file store in Z/Eves: An experiment in the verified software repository. *Sci. Comput. Program.*, 74(4):238–257, 2009. doi:10.1016/j.scico.2008.08.001.

[Geu09] Herman Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009. doi:10.1007/s12046-009-0001-5.

[Git] Git. Git distributed version control system. http://git-scm.com.

[GKL13] Gudmund Grov, Aleks Kissinger, and Yuhui Lin. A graphical language for proof strategies. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *LPAR-19*, volume 8312 of *LNCS*, pages 324–339. Springer, 2013. doi:10.1007/978-3-642-45221-5_23.

[GKL14] Gudmund Grov, Aleks Kissinger, and Yuhui Lin. Tinker, tailor, solver, proof. In Christoph Benzmüller and Bruno Woltzenlogel Paleo, editors, *UITP 2014*, volume 167 of *EPTCS*, pages 23–34, 2014. doi:10.4204/EPTCS.167.5.

[GM10] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *J. Formalized Reasoning*, 3(2):95–152, 2010. doi:10.6092/issn.1972-5787/1979.

[GM13] Gudmund Grov and Ewen Maclean. Towards automated proof strategy generalisation. Pre-print arXiv:1303.2975, 2013. URL: http://arxiv.org/abs/1303.2975.

[GMM+78] Michael J. C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. A metalanguage for interactive proof in LCF. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *POPL'78*, pages 119–130. ACM Press, 1978. doi:10.1145/512760.512773.

[H2] H2. H2 database engine. http://www.h2database.com.

[Har96] John Harrison. Proof style. In Eduardo Giménez and Christine Paulin-Mohring, editors, *TYPES'96*, volume 1512 of *LNCS*, pages 154–172. Springer, 1996. doi:10.1007/BFb0097791.

[Har06] John Harrison. Floating-point verification using theorem proving. In Marco Bernardo and Alessandro Cimatti, editors, *SFM 2006*, volume 3965 of *LNCS*, pages 211–242. Springer, 2006. doi:10.1007/11757283_8.

[Har09] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[HC02] Anthony Hall and Roderick Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, 19(1):18–25, 2002. doi:10.1109/52.976937.

[Hen06] Alex Heneveld. *Using Features for Automated Problem Solving*. PhD thesis, School of Informatics, University of Edinburgh, February 2006.

[HK91] Ian Houston and Steve King. CICS project report: Experiences and results from the use of Z in IBM. In Søren Prehn and W. J. Toetenel, editors, *VDM '91*, volume 551 of *LNCS*, pages 588–596. Springer, 1991. doi:10.1007/3-540-54834-3_34.

[HK14] Jónathan Heras and Ekaterina Komendantskaya. Recycling proof patterns in Coq: Case studies. *Mathematics in Computer Science*, 8(1):99–116, 2014. doi:10.1007/s11786-014-0173-1.

[HKJM13] Jónathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. Proof-pattern recognition and lemma discovery in ACL2. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *LPAR-19*, volume 8312 of *LNCS*, pages 389–406. Springer, 2013. doi:10.1007/978-3-642-45221-5_27.

[HM05] Tony Hoare and Jayadev Misra. Verified software: Theories, tools, experiments—Vision of a Grand Challenge project. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE 2005*, volume 4171 of *LNCS*, pages 1–18. Springer, 2005. doi:10.1007/978-3-540-69149-5_1.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972. doi:10.1007/BF00289507.

[Hoa03] C. A. R. Hoare. The verifying compiler: A grand challange for computing research. In László Böszörményi and Peter Schojer, editors, *Modular Programming Languages, JMLC 2003*, volume 2789 of *LNCS*, pages 25–35. Springer, 2003. doi:10.1007/978-3-540-45213-3_4.

[IB96] Andrew Ireland and Alan Bundy. Productive use of failure in inductive proof. *J. Autom. Reasoning*, 16(1-2):79–111, 1996. doi:10.1007/BF00244460.

[IET06] IETF. The Base16, Base32, and Base64 data encodings. RFC 4648, Network Working Group, 2006.

[IGLB13] Andrew Ireland, Gudmund Grov, Maria Teresa Llano, and Michael J. Butler. Reasoned modelling critics: Turning failed proofs into modelling guidance. *Sci. Comput. Program.*, 78(3):293–309, 2013. doi:10.1016/j.scico.2011.03.006.

[ISO02] ISO/IEC. Information technology – Z formal specification notation – Syntax, type system and semantics. International Standard ISO/IEC 13568:2002, International Organization for Standardization, 2002.

[ISO08] ISO/IEC. Information technology – Open Systems Interconnection – Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components. International Standard ISO/IEC 9834-8:2008, International Organization for Standardization, 2008.

[JDB11] Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture synthesis for inductive theories. *J. Autom. Reasoning*, 47(3):251–289, 2011. doi:10.1007/s10817-010-9193-y.

[JFV13] Cliff B. Jones, Leo Freitas, and Andrius Velykis. Ours is to reason why. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *LNCS*, pages 227–243, Shanghai, China, September 2013. Springer. doi:10.1007/978-3-642-39698-4_14.

[JGB10] Cliff B. Jones, Gudmund Grov, and Alan Bundy. Ideas for a high-level proof strategy language. In Bruno Dutertre and Hassen Saïdi, editors, *AFM'10 (Automated Formal Methods)*, July 2010.

[JJLM91] Cliff B. Jones, Kevin D. Jones, Peter A. Lindsay, and Richard C. Moore. *mural: A Formal Development Support System*. Springer, 1991. doi:10.1007/978-1-4471-3180-9.

[JKPB03] Mateja Jamnik, Manfred Kerber, Martin Pollet, and Christoph Benzmüller. Automatic learning of proof methods in proof plan-

ning. *Logic Journal of the IGPL*, 11(6):647–673, 2003. doi:10.1093/jigpal/11.6.647.

[JL04] Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *TPHOLs*, volume 3223 of *LNCS*, pages 152–167. Springer, 2004. doi:10.1007/978-3-540-30142-4_12.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[JOW06] Cliff B. Jones, Peter W. O'Hearn, and Jim Woodcock. Verified software: A grand challenge. *IEEE Computer*, 39(4):93–95, 2006. doi:10.1109/MC.2006.145.

[JS90] Cliff B. Jones and Roger C. F. Shaw, editors. *Case Studies in Systematic Software Development*. Prentice Hall International, 1990. URL: http://homepages.cs.ncl.ac.uk/cliff.jones/publications/cases90.pdf.

[JW08] Cliff B. Jones and Jim Woodcock, editors. *Formal Asp. Comput.—Special issue on the Mondex verification*, volume 20(1). Springer, 2008.

[KBKU13] Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. MaSh: Machine learning for Sledgehammer. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *ITP 2013*, volume 7998 of *LNCS*, pages 35–50. Springer, 2013. doi:10.1007/978-3-642-39634-2_6.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *SOSP 2009*, pages 207–220. ACM, 2009. doi:10.1145/1629575.1629596.

[KHG13] Ekaterina Komendantskaya, Jónathan Heras, and Gudmund Grov. Machine learning in Proof General: Interfacing interfaces. In Cezary Kaliszyk and Christoph Lüth, editors, *UITP 2012*, volume 118 of *EPTCS*, pages 15–41, 2013. doi:10.4204/EPTCS.118.2.

[KNP] Gerwin Klein, Tobias Nipkow, and Lawrence C. Paulson, editors. *Archive of Formal Proofs*. `http://afp.sf.net`, Formal proof developments.

[KSGK07] Ulf Krumnack, Angela Schwering, Helmar Gust, and Kai-Uwe Kühnberger. Restricted higher-order anti-unification for analogy making. In Mehmet A. Orgun and John Thornton, editors, *Australian Conference on Artificial Intelligence*, volume 4830 of *LNCS*, pages 273–282. Springer, 2007. `doi:10.1007/978-3-540-76928-6_29`.

[KW09] Cezary Kaliszyk and Freek Wiedijk. Merging procedural and declarative proof. In Stefano Berardi, Ferruccio Damiani, and Ugo de'Liguoro, editors, *TYPES 2008*, volume 5497 of *LNCS*, pages 203–219. Springer, 2009. `doi:10.1007/978-3-642-02444-3_13`.

[LAB+06] Gary T. Leavens, Jean-Raymond Abrial, Don S. Batory, Michael J. Butler, Alessandro Coglio, Kathi Fisler, Eric C. R. Hehner, Cliff B. Jones, Dale Miller, Simon L. Peyton Jones, Murali Sitaraman, Douglas R. Smith, and Aaron Stump. Roadmap for enhanced languages and methods to aid verification. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, *GPCE 2006*, pages 221–236. ACM, 2006. `doi:10.1145/1173706.1173740`.

[Lla12] Maria Teresa Llano. *Invariant Discovery and Refinement Plans for Formal Modelling in Event-B*. PhD thesis, School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, December 2012.

[Meh07] Farhad Mehta. Supporting proof in a reactive development environment. In *SEFM*, pages 103–112, London, England, UK, 2007. IEEE Computer Society. `doi:10.1109/SEFM.2007.40`.

[Meh08] Farhad Mehta. *Proofs for the Working Engineer*. PhD thesis, ETH Zurich, 2008. Diss. ETH No. 17671.

[MLK98] J. Strother Moore, Thomas W. Lynch, and Matt Kaufmann. A mechanically checked proof of the AMD5$_{\mathrm{K}}$86™ floating point division program. *IEEE Trans. Computers*, 47(9):913–926, 1998. `doi:10.1109/12.713311`.

[Mor94] Carroll C. Morgan. *Programming from specifications*. Prentice Hall International series in computer science. Prentice Hall, 2nd edition, 1994.

[MRMDB10] Omar Montano-Rivas, Roy L. McCasland, Lucas Dixon, and Alan Bundy. Scheme-based synthesis of inductive theories. In Grigori Sidorov, Arturo Hernández Aguirre, and Carlos A. Reyes García, editors, *MICAI (1)*, volume 6437 of *LNCS*, pages 348–361. Springer, 2010. doi:10.1007/978-3-642-16761-4_31.

[MS97] Irwin Meisels and Mark Saaltink. The Z/EVES reference manual (for version 1.5). Reference manual TR-97-5493-03d, ORA Canada, 1997.

[MS99] Erica Melis and Jörg Siekmann. Knowledge-based proof planning. *Artificial Intelligence*, 115(1):65–105, 1999. doi:10.1016/S0004-3702(99)00076-4.

[MU05] Petra Malik and Mark Utting. CZT: A framework for Z tools. In Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider, editors, *ZB*, volume 3455 of *LNCS*, pages 65–84. Springer, 2005. doi:10.1007/11415787_5.

[MW99] Erica Melis and Jon Whittle. Analogy in inductive theorem proving. *Journal of Automated Reasoning*, 22:117–147, 1999. doi:10.1023/A:1005936130801.

[MW10] David C. J. Matthews and Makarius Wenzel. Efficient parallel programming in Poly/ML and Isabelle/ML. In Leaf Petersen and Enrico Pontelli, editors, *DAMP 2010*, pages 53–62. ACM, 2010. doi:10.1145/1708046.1708058.

[MWM14] Daniel Matichuk, Makarius Wenzel, and Toby C. Murray. An isabelle proof method language. In Gerwin Klein and Ruben Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 390–405. Springer, 2014. doi:10.1007/978-3-319-08970-6_25.

[NIS12] NIST. Secure hash standard. Federal Information Processing Standard Publication 180-4, U.S. Department of Commerce, National Institute of Standards and Technology (NIST), 2012.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, May 2002. doi:10.1007/3-540-45949-9.

[OAA13] Steven Obua, Mark Adams, and David Aspinall. Capturing hiproofs in HOL Light. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka, and Wolfgang Windsteiger, editors, *MKM/Calculemus/DML*, volume 7961 of *LNCS*, pages 184–199. Springer, 2013. doi:10.1007/978-3-642-39320-4_12.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: a prototype verification system. In Deepak Kapur, editor, *CADE-11*, volume 607 of *LNCS*, pages 748–752. Springer, 1992. doi:10.1007/3-540-55602-8_217.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 1st edition, 2008.

[OZGS99] John O'Leary, Xudong Zhao, Rob Gerth, and Carl-Johan H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, (Q1):10, 1999.

[Plo69] Gordon D. Plotkin. A note on inductive generalization. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 5*, pages 153–163. Edinburgh University Press, 1969.

[RS93] Wolfgang Reif and Kurt Stenzel. Reuse of proofs in software verification. In Rudrapatna Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *LNCS*, pages 284–293. Springer, 1993. doi:10.1007/3-540-57529-4_61.

[RSS96] Harald Rueß, Natarajan Shankar, and Mandayam K. Srivas. Modular verification of SRT division. In Rajeev Alur and Thomas A. Henzinger, editors, *CAV '96*, volume 1102 of *LNCS*, pages 123–134. Springer, 1996. doi:10.1007/3-540-61474-5_63.

[RT13] Alexander Romanovsky and Martyn Thomas, editors. *Industrial Deployment of System Engineering Methods*. Springer, 2013. doi:10.1007/978-3-642-33170-1.

[RV01] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[Saa97] Mark Saaltink. The Z/EVES system. In Jonathan Bowen, Michael Hinchey, and David Till, editors, *ZUM '97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 72–85. Springer, 1997. doi:10.1007/BFb0027284.

[SB94] Harsh P. Sharangpani and M. L. Barton. Statistical analysis of floating point flaw in the Pentium™ processor (1994). White paper, Intel Corporation, 1994.

[SBB+02] Jörg Siekmann, Christoph Benzmüller, Vladimir Brezhnev, Lassaad Cheikhrouhou, Armin Fiedler, Andreas Franke, Helmut Horacek, Michael Kohlhase, Andreas Meier, Erica Melis, Markus Moschner, Immanuel Normann, Martin Pollet, Volker Sorge, Carsten Ullrich, Claus-Peter Wirth, and Jürgen Zimmer. Proof development with OMEGA. In Andrei Voronkov, editor, *CADE*, number 2392 in LNCS, pages 144–149, Copenhagen, Denmark, 2002. Springer. doi:10.1007/3-540-45620-1_12.

[SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, second edition, December 2008.

[Sch12] Matthias Schmalz. *Formalizing the Logic of Event-B*. PhD thesis, ETH Zurich, Zurich, Switzerland, 2012. doi:10.3929/ethz-a-007577749.

[SCW00] Susan Stepney, David Cooper, and Jim Woodcock. An electronic purse: Specification, refinement, and proof. Technical Monograph PRG-126, Oxford University Computing Laboratory, July 2000.

[SHB+99] Jörg H. Siekmann, Stephan M. Hess, Christoph Benzmüller, Lassaad Cheikhrouhou, Armin Fiedler, Helmut Horacek, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, Martin Pollet, and Volker Sorge. LOUI: Lovely OMEGA User Interface. *Formal Aspects of Computing*, 11(3):326–342, 1999. doi:10.1007/s001650050053.

[SJA+14] Mark Staples, D. Ross Jeffery, June Andronick, Toby C. Murray, Gerwin Klein, and Rafal Kolanski. Productivity for proof engineering. In Maurizio Morisio, Tore Dybå, and Marco Torchiano, editors, *ESEM '14*, page 15. ACM, 2014. doi:10.1145/2652524.2652551.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice Hall, 2nd edition, 1992.

[SW08] Natarajan Shankar and Jim Woodcock, editors. *Verified Software: Theories, Tools, Experiments, VSTTE 2008*, volume 5295 of *LNCS*. Springer, 2008.

[Tew11] Hendrik Tews. Automatic library compilation and proof tree visualization for Coq Proof General. Presentation at the 3rd Coq Workshop, Nijmegen, 2011. URL: http://askra.de/software/prooftree/.

[TGM10] Carst Tankink, Herman Geuvers, and James McKinna. Narrating formal proof (work in progress). In Claudio Sacerdoti Coen and David Aspinall, editors, *User Interfaces for Theorem Provers (UITP 2010), FLOC 2010 Satellite Workshop*, Edinburgh, Scotland, July 2010.

[TGMW10] Carst Tankink, Herman Geuvers, James McKinna, and Freek Wiedijk. Proviola: A tool for proof re-animation. In Serge Autexier, Jacques Calmet, David Delahaye, Patrick D. F. Ion, Laurence Rideau, Renaud Rioboo, and Alan P. Sexton, editors, *AISC/MKM/Calculemus*, volume 6167 of *LNCS*, pages 440–454. Springer, 2010. doi:10.1007/978-3-642-14128-7_37.

[TKUG13] Carst Tankink, Cezary Kaliszyk, Josef Urban, and Herman Geuvers. Formal mathematics on display: A wiki for Flyspeck. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka, and Wolfgang Windsteiger, editors, *MKM/Calculemus/DML*, volume 7961 of *LNCS*, pages 152–167. Springer, 2013. doi:10.1007/978-3-642-39320-4_10.

[Tru14] Trustworthy Systems Team. seL4 v1.03, release 2014-08-10, 2014. https://github.com/seL4/seL4. doi:10.5281/zenodo.11247.

[USPV08] Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jirí Vyskocil. MaLARea SG1 - machine learner for automated reasoning with semantic guidance. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR 2008*, volume 5195 of *LNCS*, pages 441–456. Springer, 2008. doi:10.1007/978-3-540-71070-7_37.

[UTS+03] Mark Utting, Ian Toyn, Jing Sun, Andrew Martin, Jin Song Dong, Nicholas Daley, and David W. Currie. ZML: XML support for standard Z. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina A. Waldén, editors, *ZB 2003*, volume 2651 of *LNCS*, pages 437–456. Springer, 2003. doi:10.1007/3-540-44880-2_26.

[Vad95] Sunil Vadera. Proof by analogy in mural. *Formal Asp. Comput.*, 7(2):183–206, 1995. doi:10.1007/BF01211605.

[Vel] Andrius Velykis. Isabelle/Eclipse prover IDE. http://andriusvelykis.github.io/isabelle-eclipse.

[Vel09] Andrius Velykis. Formal modelling of separation kernels. Master's thesis, Department of Computer Science, University of York, UK, September 2009.

[Vel12a] Andrius Velykis. Capturing and inferring the proof process (Part 2: Architecture). In Alan Bundy, Dieter Hutter, Cliff B. Jones, and J Strother Moore, editors, *AI meets Formal Software Development*, number 12271 in Dagstuhl Seminar Proceedings, pages 27–27, Dagstuhl, Germany, 2012. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. doi:10.4230/DagRep.2.7.1.

[Vel12b] Andrius Velykis. Inferring the proof process. In Christine Choppy, David Delayahe, and Kaïs Klaï, editors, *FM2012 Doctoral Symposium*, Paris, France, August 2012.

[VF10] Andrius Velykis and Leo Freitas. Formal modelling of separation kernel components. In Ana Cavalcanti, David Déharbe, Marie-Claude Gaudel, and Jim Woodcock, editors, *ICTAC*, volume 6255 of *LNCS*, pages 230–244. Springer, 2010. doi:10.1007/978-3-642-14808-8_16.

[WAL05] Daniel Winterstein, David Aspinall, and Christoph Lüth. Proof General / Eclipse: A generic interface for interactive proof. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05*, pages 1587–1588. Professional Book Center, 2005.

[WB] Makarius Wenzel and Stefan Berghofer. The Isabelle system manual. `http://isabelle.in.tum.de/doc/system.pdf`.

[WD96] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International, March 1996.

[Wen02] Markus M. Wenzel. *Isabelle/Isar - a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München, 2002.

[Wen12] Makarius Wenzel. Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. *Electr. Notes Theor. Comput. Sci.*, 285:101–114, 2012. `doi:10.1016/j.entcs.2012.06.009`.

[Wen14] Makarius Wenzel. Asynchronous user interaction and tool integration in Isabelle/PIDE. In Gerwin Klein and Ruben Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 515–530. Springer, 2014. `doi:10.1007/978-3-319-08970-6_33`.

[Whi13] Iain Whiteside. *Refactoring Proofs*. PhD thesis, School of Informatics, University of Edinburgh, 2013.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4), 2009. `doi:10.1145/1592434.1592436`.

[WW07] Makarius Wenzel and Burkhart Wolff. Building formal method tools in the Isabelle/Isar framework. In Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732 of *LNCS*, pages 352–367. Springer, 2007. `doi:10.1007/978-3-540-74591-4_26`.

[ZDK+13] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: a monad for typed tactic programming in Coq. In Greg Morrisett and Tarmo Uustalu, editors, *ICFP'13*, pages 87–100. ACM, 2013. `doi:10.1145/2500365.2500579`.

Index