Dynamic Proof Presentation[*]

Paul B. Jackson
Paul.Jackson@ed.ac.uk

20th April 2021

Abstract

For several decades there has been significant debate over the formal proof style supported by proof assistants. For example, the merits of a declarative style rather than a procedural (tactic) style have been argued. In much of the debate there has been unnecessarily rigid insistence on the languages of proof input and proof presentation being identified. When these concepts are not shackled together, many opportunities are opened up for dynamic proof presentation that take full advantage of the capabilities of computer user interfaces. With dynamic proof presentation, the proof viewer can easily focus attention on particular parts of proofs and change the level of detail presented. One viewer might be interested in just a proof outline, another might want to see how a large step of inference is composed of smaller steps. Current proof assistant user interfaces do provide some dynamic presentation capabilities, but much more could be done. Further attention to dynamic proof presentation should help make formal proofs easier to understand by a wider range of audiences, with minimal need to rewrite proof libraries that are developed with huge time investments.

1 Introduction

1.1 Proof Presentation Style

The core topic this chapter addresses is that of how formal proofs created using interactive theorem provers ought to be presented. Specifically, the concern is with the presentation of the structure of proofs

[*]This preprint is published as Chapter 4, pp 63–86, of Michaelson, G. (eds) *Mathematical Reasoning: The History and Impact of the DReaM Group.* Springer, Cham. 2021. https://doi.org/10.1007/978-3-030-77879-8_4

of individual lemmas, rather than the presentation of theories grouping lemmas and definitions, or the presentation of terms, types and formulas.

Over recent decades there has been significant discussion of this topic [10, 25, 13, 27]. It has long been recognised that tactic scripts by themselves are poor at communicating proof structure to readers. Tactic scripts describe how the prover should create or check full proofs, but typically do not show intermediate subgoal formulas and often obscure the branching structure of proofs. Often, if a reader is to understand a proof, they need access to a working version of the prover and they have to re-execute the tactic script step-by-step.

Proofs in declarative proof description languages (for example the Mizar system language and the Isar language of the Isabelle prover) are generally much more readable than tactic scripts. With such languages the proof text explicitly states many of the intermediate formulas in a proof, nested-blocks describe hierarchical proof structure, and the syntax is designed to be reminiscent of that found in mathematics papers and textbooks. These language features help a reader quickly gain a level of understanding of a proof just from study of its formal text, without running the relevant prover.

1.2 Ongoing Issues

The tactic style (sometimes called the *procedural style*) is still the norm in the user communities of a number of theorem provers (those for Coq, HOL4 and HOL Light, for example). A disincentive for tactic users to move to a more declarative style is that the intermediate formulas needed for a declarative style can be tedious to enter and can make the proofs scripts considerably more verbose. This is all the more the case when the formulas get large, as is common in formal verification applications. The lower comprehensibility of tactic-style proofs is not so much an issue when proofs are developed using tactics, as then the prover user interfaces present sufficient subgoal information to orient developers.

A key limitation of current practices of writing both declarative and tactic-style proofs is that the level of detail is fixed at proof writing time. Sometimes this level is determined by the extent of automation provided by the prover. Some arguments might need to be spelled out in more detail than a reader might want. Other times automation might enable large steps whose details are non-obvious to the reader.

In general there will be no one optimal level of detail. Different readers might have widely different degrees of familiarity with the prover and the subject formalised. And readers have different interests in proofs at different times. Sometimes readers are keen to understand the details; perhaps they wish to reproduce proofs in another prover or perhaps they are studying the proofs for a mathematics or computer science class. Other times, maybe they just want a high-level summary.

As mentioned above, replay of tactic-style proofs is usually essential for gaining a good understanding of them. Replay too is often useful even for declarative proofs, as some details of intermediate proof formulas are only viewable on replay. This need for replay is a significant barrier to any reader who does not wish to go to the trouble of installing the relevant prover and learning the basics of its usage. Further, it might take minutes or even hours to get a prover into a state where some given proof of interest can be replayed.

Additional issues concerning readability include that proof intelligibility varies significantly depending on the proof writer, and that there is a huge body of existing proofs, many written in a procedural style, to which it would be good to have better access.

1.3 The Vision

This chapter describes how to improve support for dynamically-choosing levels of proof presentation detail. The ideas discussed are relevant to proofs in both declarative and procedural styles, and they could make the differences between these styles less significant. Nearly all the ideas have already been experimented with in some way, so their further development and integration should be relatively straightforward. This improved support could significantly ease and speed the understanding of formal proofs.

Interest in using interactive theorem provers has been steadily increasing, both from those wishing to use them for formal verification and from those exploring their use for education and research in mathematics and computer science. Dynamic proof presentation technologies would be much appreciated by many of these users and could spur on further growth in theorem prover user communities.

1.4 Structure of Rest of Chapter

Section 1.5 includes a few notes on my interactions with the DReaM group over the years. Sections 2 to 6 sketch out what dynamic proof presentation can look like when both procedural and declarative proof styles are used. A number of DReaM group members have worked on relevant topics and Sect. 7 describes this work. Sections 8 and 9 are forward looking, considering desired technical requirements for supporting dynamic proof presentation and the relationship between proof viewing and proof editing. Further related work by DReaM group associates and others is considered in Sect. 10 and finally Sect. 11 summarises work strands that could be profitable in future.

1.5 My DReaM Group Connections

I came to the University of Edinburgh in 1995, having recently completed a PhD at Cornell University with Bob Constable on enhancing the Nuprl interactive theorem prover and using it for formalising some abstract algebra. Constable had previously visited Edinburgh on sabbatical, and the DReaM group's Oyster interactive theorem prover developed in the late 1980s was strongly inspired by Nuprl. Initially I had a post-doc in the LFCS (the Laboratory for Foundations of Computer Science) with Rod Burstall and then, from 1998, a lectureship.

My PhD work gave me a keen interest in the central concerns of the group about the automation of mathematical reasoning. Over the years I have enjoyed very much attending and participating in DReaM group talk meetings, and I continue to do so now. I appreciate the informality of these meetings; there can be as much or even more time spent in lively discussion as spent by the speaker talking. It is rare that they are like a regular seminar when there might be just a couple of polite questions afterwards. More generally, the group for me has been a welcoming intellectual home.

From 1998 to 2019 I was a co-investigator on grants held by the group that funded foundational and pump-priming research. For the most part, my research has followed paths closely related to but distinct from those of other group members; topics I have pursued include bounded model checking, the formal verification of software, automatic theorem proving for non-linear arithmetic, and, most recently, the verification of hybrid dynamical systems. This last topic is now also of interest to DReaM group member Jacques Fleuriot and we are currently planning a collaboration in this area.

The issue of how formal proofs should be presented has been an interest of mine ever since starting to work with Nuprl in the late 1980s. A couple of times I was involved in discussions with DReaM group members (primarily Alan Bundy and David Aspinall) about pursuing funding related to the topic of this chapter. Unfortunately, neither time did we develop ideas to the stage of completing and submitting a funding proposal. Recently I have been enthusiastic about the rise in prominence of the Lean theorem prover [7]. It has a rapidly-growing formal library and has attracted significant interest from mathematicians in using it in both their teaching and their research. This has renewed my interest in formal proof presentation and I might well use Lean for future work in this area.

2 A Running Example of a Procedural Proof

To help motivate the discussion throughout this chapter, let us use a proof of a lemma from the Nuprl system [6] that is the penultimate step in a proof of the irrationality of $\sqrt{2}$. This proof was previously presented in a comparison of 17 theorem provers [26] and it follows the shape of a proof example used by Lamport when advocating a structured proof format [15].

Figure 1 shows an automatically-generated *tactic-and-subgoal* proof-tree presentation of the proof. Nuprl is an interactive theorem prover in the LCF family: a proof of a lemma is undertaken by running tactics – procedural proof commands – on goals. Goals are sequents with numbered variable declarations and hypotheses and a single conclusion. When a tactic is run on a goal, 0 or more subgoals result. If no subgoals result, then the tactic completely proves the goal. Otherwise, the goal the tactic is run on is only proven once the subgoals are completely proven by further tactic runs. As tactics can be combined using tacticals into larger tactics, it is always possible to compose a single tactic that completely proves a top-level goal. Traditionally this is only done in Nuprl when the proof is very straightforward. Otherwise, proofs are presented in a tree form, with goal nodes and tactic nodes alternating as one moves down the tree branches. That is what one sees in Fig. 1: a tree of goals, separated by tactic calls after occurrences of the `BY` keyword.

To save space, this proof tree presentation elides the repetition of variable declarations, hypotheses and conclusions in the goal sequents. For example, Fig. 2 shows the second step of the Fig. 1 proof presentation, with and without elision of repeated sequent components. To construct a complete picture of some sequent in this kind of proof presentation, the reader needs to search up the tree for any elided components. It helps to know that proof steps in Nuprl might change existing components of the declaration and hypothesis list, or add new components, or both, but it is rare that components are deleted and a subgoal after a step has a shorter list. In the event that a subgoal's declaration and hypothesis list is shorter than that of the parent goal, there is no elision of components in the subgoal, in order to avoid ambiguity.

In the Fig. 1 proof, the first proof step shows that the proof strategy

```
*T root_2_irrat_over_int

⊢ ¬(∃m,n:ℤ. CoPrime(m,n) ∧ m * m = 2 * n * n)
|
BY (D 0 THENM ExRepD ...a)
|
1. m: ℤ
2. n: ℤ
3. CoPrime(m,n)
4. m * m = 2 * n * n
⊢ False
|
BY Assert ⌈2 | m⌉
|\
| ⊢ 2 | m
| |
| BY (BLemma `two_div_square` THENM Unfold `divides` 0
|     THENM AutoInstConcl [] ...a)
 \
  5. 2 | m
  |
  BY Assert ⌈2 | n⌉
  |\
  | ⊢ 2 | n
  | |
  | BY (BLemma `two_div_square` THENM All (Unfold `divides`)
  |     THENM ExRepD THENM Inst [⌈c * c⌉] 0
  |     THENM RWO "6" 4 ...)
   \
    6. 2 | n
    |
    BY (RWO "coprime_elim" 3 THENM FHyp 3 [5;6] ...a)
    |
    3. ∀c:ℤ. c | m ⇒ c | n ⇒ c ∼ 1
    7. 2 ∼ 1
    |
    BY (RWO "assoced_elim" 7 THENM D (-1) ...)
```

Figure 1: Tactic-and-subgoal proof tree presentation

With elision

```
...
|
1. m: ℤ
2. n: ℤ
3. CoPrime(m,n)
4. m * m = 2 * n * n
⊢ False
|
BY Assert ⌈2 | m⌉
|\
| ⊢ 2 | m
| |
| ...
 \
  5. 2 | m
  |
  ...
```

Without elision

```
...
|
1. m: ℤ
2. n: ℤ
3. CoPrime(m,n)
4. m * m = 2 * n * n
⊢ False
|
BY Assert ⌈2 | m⌉
|\
| 1. m: ℤ
| 2. n: ℤ
| 3. CoPrime(m,n)
| 4. m * m = 2 * n * n
| ⊢ 2 | m
| |
| ...
 \
  1. m: ℤ
  2. n: ℤ
  3. CoPrime(m,n)
  4. m * m = 2 * n * n
  5. 2 | m
  ⊢ False
  |
  ...
```

Figure 2: A proof step with and without elision of repeated sequent components

```
*A divides          b | a ==  ∃c:ℤ. a = b * c
*A assoced          a ~ b ==  a | b ∧ b | a
*A gcd_p            GCD(a;b;y) ==  y | a ∧ y | b ∧
                                   (∀z:ℤ. z | a ∧ z | b ⇒ z | y)
*A coprime          CoPrime(a,b) ==  GCD(a;b;1)
*T two_div_square   ∀n:ℤ. 2 | n * n ⇒ 2 | n
*T coprime_elim     ∀a,b:ℤ.  CoPrime(a,b) ⟺
                         (∀c:ℤ. c | a ⇒ c | b ⇒ c ~ 1)
*T assoced_elim     ∀a,b:ℤ.  a ~ b ⟺ a = b ∨ a = -b
```

Figure 3: Definitions and lemmas

is to assume the negation of the goal, and from this to show falsity, i.e. that we have a contradiction. The tree presentation makes clear that the proof proceeds by first establishing 2 divides m (hypothesis 5) and then that 2 divides n (hypothesis 6). The contradiction then follows because hypothesis 3 claims that m and n are co-prime, that they have no non-trivial common divisors. A divisor is trivial if it a unit as far as divisibility is concerned, i.e. if it is $+1$ or -1. We also see two side proofs in the tree presentation: the first establishing that 2 divides m and the second, knowing that 2 divides m, that also 2 divides n.

What do the tactics in Fig. 1 actually do? Nuprl's tactic language, and indeed most procedural proof languages, require study to understand. Sometimes names are suggestive (e.g. `Assert` or `Unfold`), other times they are rather abbreviated to save space and typing (e.g. `RWO` is short for *rewrite once*). Also lemma names (e.g. `coprime_elim`, `assoced_elim`) and definition names (e.g. `divides` for the infix `|` operator) provide part of the story. Once the reader sees the referenced lemmas and the definitions used, they can sometimes make a fair guess as to what is going on. For the lemmas and definitions relevant to the running example proof, see the fragment of a Nuprl library listing in Fig. 3. Here, lines starting with `*A` are for definitions (`A` is for *abstraction*, Nuprl's terminology for a definition) and with `*T` are for lemmas and theorems. Note that, when doing divisibility theory over the integers, GCDs are unique only up to associates (as specified by the `assoced` relation, with $\sim$ infix notation).

The next sections explore how dynamic proof presentation capabilities can improve proof readability and understandability.

```
* top 1 2
1. m: ℤ
2. n: ℤ
3. CoPrime(m,n)
4. m * m = 2 * n * n
5. 2 | m
⊢ False

BY Assert ⌜2 | n⌝

1* ⊢ 2 | n

2* 6. 2 | n
     ⊢ False
```

Figure 4: A proof refinement step

3 Focussing on Proof Steps in Procedural Proofs

While a tactic-and-subgoal proof tree as in Fig. 1 can provide a good overview of a proof, such trees become hard to read when spread over many pages as the vertical linearisation creates distance between a goal and its immediate subgoals. If our attention is on some subtree, it is easy to have the presentation start at the root goal of that subtree rather than the initial goal being proven. If the tree presentation is in an interactive viewer, a facility for hiding subtrees that are not of immediate interest is useful. If our attention is on a particular goal, showing just that goal and its immediate subgoals is helpful. In Nuprl, such a view is the primary way proofs are interactively presented, both when viewing proofs and editing proofs. For example, Fig. 4 shows the view Nuprl would give of the step introducing the `2 | n` hypothesis 6. As with the full tactic-and-subgoal proof trees, subgoals omit repeated sequent components to save space and enable the reader to focus on what has changed. A `*` at the start of a sequent indicates that the proof below that point is complete. If the proof is incomplete, a `#` is used instead. The `top 1 2` is the tree address of the top sequent. Users navigate up and down the proof tree just by clicking on the goal or one of the subgoals in such a view.

4 Condensing Tactic-and-Subgoal Proof Trees

The length of a proof tree presentation can be reduced by combining adjacent tactics with tacticals. For example, the two steps at the end of the presentation in Fig. 1

```
6. 2 | n
|
BY (RWO "coprime_elim" 3 THENM FHyp 3 [5;6] ...a)
|
3. ∀c:ℤ. c | m ⇒ c | n ⇒ c ~ 1
7. 2 ~ 1
|
BY (RWO "assoced_elim" 7 THENM D (-1) ...)
```

could be collapsed into the single step:

```
    6. 2 | n
|
BY (RWO "coprime_elim" 3 THENM FHyp 3 [5;6]
    THENM RWO "assoced_elim" 7 THENM D (-1) ...)
```

The reader might be wondering what the '...' and '...a' signify in the Nuprl tactics shown here and earlier. These are notational shorthand for calls of Nuprl's auto-tactic on resulting subgoals. This tactic undertakes common straightforward reasoning steps such as proving linear arithmetic facts and checking well-formedness of terms and formulas. (In Nuprl, type checking is undecidable and all type checking is undertaken using proof.) The '...' variant is for running the auto-tactic on all subgoals and the '...a' variant is for running the auto-tactic only on *auxiliary* subgoals such as well-formedness subgoals.

Collapsing tactic steps together usually decreases hints to the reader as to what is going on with each step, as intermediate goals are then no longer visible. It therefore can be helpful if the proof developer can add comments explaining steps. Indeed, a useful option is to hide the tactic text when comments are used. This can produce readable proof outlines that are accessible to those unfamiliar with the tactic language. Figure 5 shows what an outline of the whole running proof could look like if some adjacent steps are combined, comments are inserted and tactics are hidden.

Another possible viewing option could involve the replacement of tactic text with automatically-generated natural-language explanations of the tactics. One simple way to realise this would be to

```
*T root_2_irrat_over_int

⊢ ¬(∃m,n:ℤ. CoPrime(m,n) ∧ m * m = 2 * n * n)
|
BY Assume negation of goal and aim for proof by contradiction
|
1. m: ℤ
2. n: ℤ
3. CoPrime(m,n)
4. m * m = 2 * n * n
⊢ False
|
BY From hyp 4, deduce that 2 | m
|
5. 2 | m
|
BY From hyps 4 and 5, deduce that 2 | n
|
6. 2 | n
|
BY Observe that hyps 5 and 6 contradict hyp 3
```

Figure 5: Proof outline

associate every tactic with some natural language description template with slots for appropriate printing of any tactic arguments. See Fig. 6 for a mock-up of how the running proof might look with such an approach. While this might be more accessible to a reader not familiar with Nuprl, it still assumes familiarity with concepts such as forward chaining, back chaining and rewriting, and the reader needs to understand that to *decompose* a hypothesis or conclusion is to apply some relevant left or right introduction rule in a backwards fashion.

There have been more sophisticated investigations of how to produce natural language versions of whole tactic-based proofs. For example, see the work of Holland-Minkley on presenting Nuprl proofs [11]. Even if easily-understandable renditions of tactic text can be automatically generated, there still is a need for supporting display of human-written comments, as these comments might provide higher-level motivation for *why* a proof is being steered some particular way.

5 Expanding Proof Steps

Sometimes a proof reader wishes to explore a proof step in more detail. For example, they might want to split apart the tactic steps combined

```
*T root_2_irrat_over_int

⊢ ¬(∃m,n:ℤ. CoPrime(m,n) ∧ m * m = 2 * n * n)
|
BY Decompose the conclusion
|  THEN Repeatedly decompose hypotheses,
|    including existential quantifiers
|
1. m: ℤ
2. n: ℤ
3. CoPrime(m,n)
4. m * m = 2 * n * n
⊢ False
|
BY Assert ⌜2 | m⌝
|\
| ⊢ 2 | m
| |
| BY Back-chain using the lemma two_div_square
|    THEN Unfold the definition of divides (|) in the conclusion
|    THEN Instantiate the conclusion's existential quantifier
|      by matching the quantifier body against some hypothesis
 \
  5. 2 | m
  |
  BY Assert ⌜2 | n⌝
  |\
  | ⊢ 2 | n
  | |
  | BY Back-chain using the lemma two_div_square
  |    THEN Unfold the definition of divides (|)
  |      in all hypotheses and the conclusion
  |    THEN Repeatedly decompose hypotheses,
  |      including existential quantifiers
  |    THEN Instantiate the conclusion's quantifier(s)
  |      with the term(s) ⌜c * c⌝
  |    THEN Rewrite hypothesis 4 using hypothesis 6
   \
    6. 2 | n
    |
    BY Rewrite hypothesis 3 using the lemma coprime_elim
    |  THEN Forward-chain using hypothesis 3,
    |    matching with hypotheses 5 and 6
    |
    3. ∀c:ℤ. c | m ⇒ c | n ⇒ c ∼ 1
    7. 2 ∼ 1
    |
    BY Rewrite hypothesis 7 using lemma assoced_elim
       THEN Decompose the last hypothesis
       THEN Repeatedly apply straightforward reasoning techniques
```

Figure 6: Proof tree presentation with simple natural-language rendering of tactics

using the `THENM` sequencing tactical ('then on *main* subgoal') that are used to prove the `2 | n` goal. See Fig. 7 for a copy of the original tactic-and-subgoal proof tree fragment followed by a proof tree for the expanded version of this fragment. Now the reader can see, in the penultimate step, the definition of this variable `c` that is used in the term `c * c` used to instantiate the existential quantifier in the conclusion.

Further expansion could be desirable for tactics such as the auto-tactic that are defined in terms of a number of simpler tactics. Expansion of the auto-tactic at the very end of the proof could show the linear integer arithmetic tactic used to prove the main goal and the type checking tactic used to prove various well-formedness goals that are a by-product of the rewriting of hypothesis 4 with hypothesis 6.

Nuprl happens to have some support for such expansion, as it stores the proof tree fragments created by tactic runs, and a proof editor command enables the replacement of a tactic run by the resulting proof tree. Unfortunately, by default, these proof tree fragments are at the primitive rule level which is far too detailed to be of interest to almost all readers. To arrange that higher-level tactics can expand into lower level tactics, the code doing the expansion needs access to the syntax of tactic expressions and tactic definitions. With Nuprl, these details are hidden away in the ML compiler's data structures and are not accessible to the ML runtime. This is a general issue one has to face whenever tactics are expressed directly in some programming language. It is avoided when the prover adopts a custom proof command language and ASTs for commands are readily available. I did experiment with specially-defined tactics and tacticals that captured structural information about tactics and enabled incremental expansion of tactic runs into lower level tactics. However it was difficult to do this for all tactics and this facility never made it into the standard Nuprl release.

6 Dynamic Presentation of Declarative Proofs

A declarative version of our running example proof is shown in Fig. 8. This uses the Isabelle Isar declarative proof language, but, within the ⌜⌝ quotes, keeps the previously-used Nuprl notation for terms and formulas. It has been derived from a proof undertaken using the Isabelle 2020 system.

From a content point of view, this is not so different from our initial procedural proof-tree presentation in Fig. 1. The high-level flow of

Original proof:

```
1. m: ℤ
2. n: ℤ
3. CoPrime(m,n)
4. m * m = 2 * n * n
5. 2 | m
⊢ 2 | n
|
BY (BLemma ‘two_div_square‘ THENM All (Unfold ‘divides‘)
    THENM ExRepD THENM Inst [⌈c * c⌉] 0
    THENM RWO "6" 4 ...)
```

Expanded proof:

```
1. m: ℤ
2. n: ℤ
3. CoPrime(m,n)
4. m * m = 2 * n * n
5. 2 | m
⊢ 2 | n
|
BY (BLemma ‘two_div_square‘ ...a)
|
⊢ 2 | n * n
|
BY All (Unfold ‘divides‘)
|
5. ∃c:ℤ. m = 2 * c
⊢ ∃c:ℤ. n * n = 2 * c
|
BY ExRepD
|
5. c: ℤ
6. m = 2 * c
|
BY (Inst ⌈c * c⌉ 0 ...a)
|
⊢ n * n = 2 * c * c
|
BY (RWO "6" 4 ...)
```

Figure 7: Original and expanded proof of `2 | n`

```
theorem root_2_irrat_over_int:
  ⌜¬(∃m,n:ℤ. CoPrime(m,n) ∧ m * m = 2 * n * n)⌝
proof
  assume ⌜∃m,n:ℤ. CoPrime(m,n) ∧ m * m = 2 * n * n⌝
  from this obtain m n where cop: ⌜CoPrime(m,n)⌝
                         and eq: ⌜m * m = 2 * n * n⌝ by auto
  have tdm: ⌜2 | m⌝
  proof (rule two_div_square)
    from eq show ⌜2 | m * m⌝ by (unfold divides, simp)
  qed
  have tdn: ⌜2 | n⌝
  proof (rule two_div_square)
    show ⌜2 | n * n⌝
    proof (unfold divides)
      from tdm obtain c where ⌜m = 2 *c⌝ by (unfold divides, auto)
      from this eq have ⌜n * n = 2 * c * c⌝ by simp
      from this show ⌜∃k. n * n = 2 * k⌝ by simp
    qed
  qed
  have ta1: ⌜2 ∼ 1⌝
  proof -
    from cop coprime_elim have ⌜∀c. c | m ∧ c | n ⇒  c ∼ 1⌝
      by simp
    from this tdm tdn show ⌜2 ∼ 1⌝ by auto
  qed
  show ⌜False⌝
  proof -
    from ta1 assoc_elim have ⌜2 = 1 ∨ 2 = -1⌝ by simp
    from this show ⌜False⌝ by arith
  qed
qed
```

Figure 8: Declarative proof

the proof with the successively introduced hypotheses `CoPrime(m,n)`, `m * m = 2 * n * n`, `2 | m`, `2 | n` and `2` $\sim$ `1` is the same. A minor difference is in how hypotheses are referred to: here they have symbolic labels rather than numbers and the special name `this` is used to refer to an unlabelled immediately-previous hypothesis. Nested `proof-qed` blocks capture the side proofs of several of the introduced hypotheses. The `from` phrases make clear how earlier assumptions and lemmas are used in immediately-following proof steps. After the `proof` and `by` keywords are instances of *methods*, Isabelle's version of tactics. Because Isar proofs still involve invocations of procedural tactics to justify declared steps, they sometimes are referred to as being *semi-declarative*. In other more-purely-declarative systems such as Mizar, virtually all steps are either basic steps of propositional and predicate logic or involve a single implicitly-invoked procedure.

From a proof creation point-of-view, the difference between declarative proofs and proof-tree presentations of procedural proofs is usually much more radical, as all of the text in the declarative case has to be entered in the proof source file. The proof developer has not only to enter the various keywords defining the shape of the proof and suggesting what deductions depend on, but also enter all the intermediate formulas introduced in the proof. To some extent this further work by the proof developer is moderated because, knowing the result of a proof step, automation can do more to figure out how to justify a step given hints. Also, with Isabelle, the jEdit proof editor has a command that generate formula text for case splits and inductions, when the text can get rather tedious to figure out by hand.

With Isabelle, some practices act against readability. For example, proof method text automatically generated by the Sledgehammer tool [4] often contains rather more detail than many proof readers care about. (Sledgehammer is an all-purpose tool that combines a variety of automatic reasoning engines such as SMT solvers and first-order automatic theorem provers.) And there are some conventions for referring to parts of subgoals resulting from inductions and case splits that, while easing typing, avoid entry of and therefore also presentation of the full formulas involved in the subgoals.

Virtually all the ideas for dynamic proof presentation make sense in this declarative context, and some support is available. For example, with Isabelle's jEdit, the user can click at any point in a declarative proof, and a separate window shows some subgoal and context information associated with that position. And jEdit does support folding of `proof-qed` blocks, so the viewer has some control over the level of detail. It would be straightforward to allow source text to include some marks indicating blocks to be folded by default, so say just some comments on what the blocks do are visible. Expansion of proof commands would probably take some work. It may be that showing some kind of command execution traces in auxiliary windows would be easier than generating source text versions with more-detailed proof text. Indeed, Isabelle currently allows execution traces for its `simp` rewriting method to be displayed.

7 DReaM Group Contributions

Several DReaM group researchers have been concerned with the issue of how best to present partial views of proof plans so that the reader easily sees relationships between plan parts and is not overloaded with detail. The first three subsections below survey relevant work by these researchers.

A key observation in this work was the importance of being able to view hierarchies of both subgoals and proof methods. Later work covered in Sects. 7.4 and 7.5 formalised a notion of proof trees with these two hierarchies and used this formalisation to help reason about proof transformations that could make proofs easier to understand.

7.1 Barnacle and XBarnacle

Lowe, McLean and Bundy developed the Barnacle [16] and XBarnacle [17] graphical user interfaces (GUIs) to the Clam proof-planning system. These enabled a degree of interactivity when running proof plans. The GUIs displayed proof plan trees, traces of the executions of Clam proof methods at some default level of detail. Nodes in these trees were associated with method applications and were displayed as boxes labelled with method names. Edges in these trees were associated with goals: the parent edge of a method corresponded to the goal the method was applied to and the child edges of a method to any subgoals generated by the method. Goal formulas were not displayed by default, but could be viewed in pop-up windows.

If more information was desired about a method application, a pop-up window could show a proof plan tree for the method, revealing the next greater level of detail. Alternatively a method could be expanded in place into its next-level-down proof tree.

To help the user understand how planning was functioning, the user could check the status of method preconditions and method scores that the planner used to select methods.

Barnacle and XBarnacle were used and evaluated not only by researchers, but also by undergraduate students on a formal methods course. Users appreciated the graphical visualisation of proof trees and liked the ability to increase or reduce the level of detail, as the default level was often not the most useful [16].

The challenges of displaying tactic-and-subgoal proof trees have been considered in a number of provers. For example, PVS can generate two-dimensional display of proof trees where the text of PVS tactics is shown, but goal formulas are only visible in pop-up windows that appear when goal symbols are clicked on. In my experience of using both Nuprl and PVS, I have found it most useful to view the full story, seeing both goals and tactics at once. I can see what's going on more quickly, and a full view can easily be printed and studied offline. The size of goal text usually forces a one dimensional vertical layout of tree structure such as used in the proof tree presentations in this chapter. Hopefully dynamic presentation techniques can help minimise the disadvantages of a one-dimensional layout.

7.2 The Orthogonal Hierarchies of Method Trees

In 2002 Bundy authored Blue Book Note 1411 with the title *Representing Orthogonal Hierarchies in Proof Plan Presentations.* The orthogonal hierarchies in question were the hierarchy of subgoals in the method-and-subgoal proof tree, and the hierarchy of methods and their constituent methods. He considered several alternative visual presentations of these two hierarchies. He remarked how the expansion and contraction of method applications in XBarnacle prevented one seeing at a glance the relationship between a method application and its expansion. He advanced a preference for presenting higher-level methods and their constituent methods using nested boxes, and subgoals using edges between boxes with sequents labelling edges hidden by default. See Fig. 1.2 in Chap. 1 for an example of such a presentation.

Even with goals hidden, he remarked on the challenge of maintaining the readability of such presentations as the size of the tree increases. He noted obvious management techniques such as zooming in, making an internal node the root of the presentation and hiding certain sub-trees. He observed that such techniques can be applied both to the method nesting hierarchy and the subgoal hierarchy.

7.3 IsaPlanner

Dixon and Fleuriot's IsaPlanner [9] was an exploration of importing proof planning ideas into the Isabelle/Isar environment. IsaPlanner's *proof techniques*, enhanced versions of tactics, would output Isar declarative proof scripts when run on a proof goal. Particular techniques were responsible for generating script structure, and IsaPlanner provided support for unpacking a technique into lower-level constituent techniques. A graphical viewer was built for the generated proof plans along the lines Bundy had previously advocated (see Sect. 7.2), which used nested boxes to show how higher-level technique instances were composed of instances of more basic techniques.

7.4 Hiproofs and Proof Refactoring

Denney, Power and Tourlas [8] considered mathematical models of proof trees with hierarchies of both subgoals and methods as described above in Sect. 7.2. For brevity, they referred to them as *hiproofs*. Aspinall, Denny and Lüth [3] defined a simple grammar for hiproofs and a simple tactic language *Hitac* for generating hiproofs, and went on to present small-step and big-step operational semantics for Hitac. At the time Aspinall had a strong interest in *proof re-engineering*, exploring ideas analogous to software re-engineering in the world of proofs: Whiteside, Aspinall, Dixon and Grov [24] defined a simple formal declarative proof script language reminiscent of Isabelle/Isar, and gave it an operational semantics, building on the previous hiproof and Hitac work. They then consider a number of re-arrangements, *refactorings* of declarative proofs (for example, turning a backward proof into a forward proof) and argued how these refactorings are formally correct.

Refactoring of software is used to improve its maintainability and understandability. Proof refactoring is of interest in this chapter because it could make proofs easier to understand.

7.5 HipCam and Tactician

Obua, Adams and Aspinall [18] produced two systems that can automatically generate hiproof versions of HOL Light proofs and then graphically display them. The issue is that the practice in HOL Light source files is to store the proof of each lemma as a maximally-condensed single tactic. If our running example lemma were to be stored as an ML variable binding in the style used in HOL Light source files, it might look as shown in Fig. 9.

In a further paper [1] Adams explains how to use Tactician to refactor packed HOL Light proofs into sequences of individual tactic invocations on HOL Light's goal stack, with comments identifying the tree structure of of the proofs. A HOL Light proof is usually initially produced by running a sequence of separate tactics that successively refine the top goal on a stack of remaining subgoals to prove. This

```
let root_2_irrat_over_int = prove
  ⌜¬(∃m,n:ℤ. CoPrime(m,n) ∧ m * m = 2 * n * n)⌝
  ( D 0 THENM ExRepD
    THENM
    ( Assert ⌜2 | m⌝
      THENA (BLemma ‘two_div_square‘ THENM Unfold ‘divides‘ 0
             THENM AutoInstConcl []))
    THENM
    ( Assert ⌜2 | n⌝
      THENA (BLemma ‘two_div_square‘ THENM All (Unfold ‘divides‘)
             THENM ExRepD THENM Inst [⌜c * c⌝] 0
             THENM RWO "6" 4))
    THENM RWO "coprime_elim" 3 THENM FHyp 3 [5;6]
    THENM RWO "assoced_elim" 7 THENM D (-1) ...)
;;
```

Figure 9: Single tactic proof

refactoring simplifies stepped replay and viewing of HOL Light proofs, enabling novices to more easily study and learn from legacy HOL Light proofs, and also helping with proof maintenance as revisions are made to HOL Light libraries. The refactoring can also be reversed, shortcutting the tedious process of transforming a stepped proof into a packed proof.

8 Technologies for Proof Presentation

As remarked in the introduction, the hope is that improved dynamic presentations of formal proofs will help to increase the ease with which formal proofs can be understood and will broaden the audience for formal proofs. There is the potential to engage those interested in learning topics that have been formalised and attracting the attention of those who initially are just casually interested. There is the potential too to support active users of theorem provers in rapidly coming up to speed on libraries in the systems they are using and learning too from libraries in other systems.

To achieve this, proof presentations must be accessible using standard universal technologies, i.e. web browsers. Also access must be fast; delays must be at most seconds. Could this level of performance be achieved by connecting to a web server running the relevant theorem prover? Would the server need to cache pre-processed presentation information?

It is desirable that presentations of proofs from a theorem prover be long lasting, remaining accessible even after the theorem prover itself is no longer actively maintained, and perhaps after the point when running the prover on up-to-date hardware is problematic. This might steer the technology towards not relying on the theorem prover running and instead caching all relevant data. Simple hypertext presentations of Nuprl theories I developed 25 years ago are still readily browsable on the web, even though it is unlikely that the version of Nuprl I used when developing those theories still runs.

Hopefully some presentation technology could be shared across multiple theorem provers, to speed adapting it to new provers.

Presentation technologies would also need to address many of the issues not touched on here that are also highly desirable. For example, it should handle the pretty printing of formulas and terms, with control over often-hidden information such as types, implicit arguments and implicit coercions, and the provision of hyperlinks or tool-tip hover-texts that explain pointed-to proof commands, definitions and lemma names. Modern programming IDEs, e.g. VSCode, provide good examples of how such features can be engineered. For example, if a programmer using VSCode wishes to see how a function being called is defined, they can easily instruct VSCode to insert a several-line scrollable buffer immediately below the function call position that displays the function's definition. Indeed the preferred front-end for the Lean prover uses VSCode, and a VSCode front-end for Isabelle is being developed that might eventually replace the current jEdit front-end.

9 Relationship Between Viewing and Editing Proofs

Whether or not fast dynamic presentation of theorem prover libraries uses a running instance of the theorem prover, it is certainly desirable that similar functionality be available to proof developers on the proofs they are currently working on. Good dynamic proof presentation should help the developer both focus on individual proof steps and keep a good awareness of the wider proof context. It also should help them more quickly understand why a proof step might not be running or checking as they expect, and so speed the completion of proofs.

As stressed at this chapter start, good dynamic proof presentation separates the concerns of how we input proofs, the required keystrokes and mouse clicks, from the concerns of how we view and understand proofs. This could lead to simpler, easier to learn, more robust proof guidance approaches than we currently have.

10 Further Related Work

The ACL2 theorem prover [14] has a number of options for controlling the kind of information and level of detail it shows in proofs. Theorems are proved using a single sophisticated automatic strategy. As this strategy runs, it prints subgoals with their tree addresses and between these gives natural language descriptions of the reasoning techniques applied. When a proof fails, it also prints information on key steps in the failed proof that the user should first inspect in order to infer what guidance is missing. Perhaps a missing prior lemma is needed or perhaps the use of some existing previous lemma for rewriting needs to be disabled. Various options can reduce the amount of proof information printed or trace details of particular kinds of reasoning steps. Breakpoints can be set if one wants to interactively examine the prover state in particular parts of a proof attempt. To help the user appreciate how a proof is progressing, ACL2 can generate simultaneous alternate views summarising aspects of the evolving proof such as the subgoal tree structure or the applied rewrites.

A major difference between these dynamic presentation capabilities and those considered in this chapter has to do with the design purpose of the capabilities. With ACL2, the primary concern is with quickly figuring out why a proof fails and how to go about fixing it. In this chapter, a primary concern is for capabilities that help the user understand successful proofs. However it is expected that capabilities that are good for this will also help interactive proof developers to track where they are in partial proofs and to debug faulty lines of reasoning.

Another difference concerns the extent to which the prover might construct some proof data-structure which then separately can be traversed and inspected. The dynamic proof presentation discussed in this chapter assumes that such a data-structure exists. With ACL2 the capabilities seem designed to largely avoid the construction of such data-structures, perhaps because they would be prohibitively large for the formal verification applications ACL2 is typically used for. Interestingly the developers of the Imandra theorem prover [19], which has automation strongly inspired by that of ACL2 and its predecessors, are experimenting with the benefits of creating hiproof-like proof data-structures.

Siekmann et al. [21] describe a user interface for the ΩMEGA proof-planning system. ΩMEGA has a graph data structure for storing proofs that holds the multiple levels of detail of hierarchical proofs and additionally supports holding alternative proofs. Different kinds of edges in the graph record proof tree subgoal hierarchy, how method applications are related to applications of their constituent methods, and how there might be multiple proofs of a given subgoal. In one panel the interface presents a 2D layout of the interleaved subgoals and methods for a proof tree using different colours and shapes for nodes, but no visible method or goal information. Node colours and shapes distinguish whether for example a node represents a goal, a method or a primitive inference. Another panel shows a linearised natural-deduction view of the current proof and, when the proof is complete, a pop-up window can display a natural language version of the proof. Some control is provided for restricting attention to parts of a proof. The alternate views in the different panels are hyperlinked so clicking at a point in one takes the user to the corresponding point in another.

Cairns and Gow [5] explored how students on a topology course handled semi-formal hierarchical proofs in the structured proof format advocated by Lamport [15]. This format is similar to formal declarative proofs in that justifications of higher-level steps are provided in lower-level proof blocks. Of particular interest to us is that the web presentation of the hierarchical proofs allowed viewers to selectively hide or expand the more-detailed proof levels. The responses from a preliminary survey of three students was mixed. The value of being able to control the level of detail was recognised, but the unfamiliarity of the format and an awkwardness of a numerical cross-referencing scheme were obstacles to the hierarchical proofs helping improve understanding of the proofs.

Wiedijk [25] describes the notion of a *formal proof sketch* which is derived from a formal declarative proof by omitting particular details in order to produce proofs that are easier to read. He illustrates this using formal proofs from the Mizar system. These formal sketches always preserve some essential formal structure of the corresponding full formal proofs.

Kalisyk and Wiedijk [13] describe the ProofWeb system that translates arbitrary procedural proofs in Coq into the declarative Fitch-style proofs as used in the Huth and Ryan textbook on formal verification [12]. Further, it enables users to develop incomplete proofs either by directly editing the Fitch-style proofs or by running Coq tactics on statements in the declarative proofs that have not yet been justified. Related later work by Wiedijk [27] presents a light-weight front end to HOL Light that runs in the Unix vi editor and that supports the creation of declarative proofs in the style of the Mizar prover (the main inspiration for the Isabelle/Isar declarative language). The user can mix typing the declarative text in full and just typing HOL

Light tactics that run on unjustified steps and cause the system to extend the declarative proof.

Prover developers (e.g. for Isabelle, HOL4, Mizar, Coq, Lean, Metamath) do make efforts to have libraries browsable on the web, sometimes with useful hyperlinks for definitions and theorems. However, only in some cases are versions of libraries with proofs provided, and, when this happens, the proofs are usually just static proof scripts as recorded in proof script files. One exception is with the work by Tankink et al. [22, 23] on the Proviola system for Coq. This displays Coq source files in a web browser in such a way that clicking on a tactic step brings up a second pane displaying Coq's output from that step, typically a list of the subgoals generated. Another exception is with Pit-Claudel's recently-released Alectryon tool [20], again for Coq libraries. As with Proviola, Coq's output can be viewed, but here the output is interleaved with the source, and users can click to unfold the display of further information or to fold the information currently displayed. Special comments can be added to source files to control what information about subgoals and subgoals parts is folded or unfolded by default.

11 Conclusions and Future Directions

This chapter has discussed how dynamic proof presentation could help ease understanding formal proofs and broaden the audience for formal proofs.

Some ideas for directions in which future research would be worthwhile are as follows.

Source mark-up for presentation
Mark-up conventions are needed to indicate how blocks of proof are folded or unfolded by default, how comments might replace proof blocks, and how a proof has hierarchical structure that is not apparent from the proof syntax. Already provers such as Coq and Isabelle support mark-up for producing document versions of library files and the Alectryon work [20] defines further mark-up for dynamic presentation options.

Exploring further proof presentation techniques
Once proof editors and proof viewers can be engineered to support dynamic proof presentation, there are opportunities for exploring ideas for presenting proofs beyond those discussed here, perhaps bringing in too the proof refactorings and transformations mentioned in Sects. 7.4, 7.5 and 10.

Handling legacy proofs

This is vital as many formal proofs have not been developed with the reader in mind, yet there is interest in understanding these proofs. Again, some combination of proof transformation technologies such as described in Sects. 7.4, 7.5 and 10 , and dynamic presentation technologies is needed.

Proof presentation technologies

As discussed in Sect. 8, proof presentations should be viewable using web browsers and fast to access and navigate. How should this be engineered?

Prover input languages

Separating the demands of the languages for entering and displaying proofs opens up new opportunities for the input languages. In current proof languages there are compromises between the different needs of ease of input, readability and suitability for instructing the theorem prover. With separation of demands, we can imagine simplified input languages suitable for novices and more sophisticated terse input languages for experts.

Exposing proof-tree structure

Most interactive provers have tactics that transform a proof state consisting of a list or stack of unproven goals. In doing so the tree-shaped hierarchical structure of proofs is obscured. It would be good if the presentation technology can expose this tree structure so it can help with proof understanding.

Handling meta-variables

While Nuprl tactics always refine a single unproven subgoal, tactics in other provers can simultaneously modify multiple subgoals in the unproven goal list. This can make creating tactic-and-subgoal tree presentations of proofs problematic. A prime example of when this happens is when the prover supports meta-variables – implicitly existentially quantified variables – in goals. Deep down in one branch of a proof, a tactic can instantiate a meta-variable that also occurs in other proof branches. How then should tactic-and-subgoal tree presentations make such non-local modifications of a proof tree evident?

Extracting explanations from tactic runs

When a tactic encapsulates a significant amount of automation, it is desirable that the prover be able to explain tactic runs. If the tactic simply unpacks into calls of simpler tactics, then, as discussed in Sect. 5, showing a tactic-and-subgoal tree involving these simpler tactics could be appropriate. However, if it involves rewriting or involves calls of automated provers for first order logic or arithmetic, then some kind of execution trace might be relevant. But such traces can often be far too detailed. What ways are there of structuring them so detail can be incrementally revealed?

Presenting proof-terms
This chapter has not discussed the use of *proof-terms* to describe proofs. This proof style is standard with the Agda proof assistant and popular with some Lean users. While proof terms precisely express the logical structure of proofs, they do so in a way less naturally familiar to most readers, and careful use of syntactic sugar and layout is needed to produce proofs with some of the readability of declarative proofs. How could a dynamic proof presentation approach make proof terms easier to understand?

As pointed out in Sect. 7, current group members Bundy, Aspinall and Fleuriot all have a significant amount of past experience in areas closely-related to those discussed here. Further, for many years, Aspinall was the primary developer of the Proof General user interface for interactive theorem provers [2] and Fleuriot is a world-class expert in the Isabelle theorem prover and its Isar proof language. I hope this chapter will be a spur to some combination of us to now push forward on some of the topics listed here.

Acknowledgements

I would like to thank the anonymous reviewers for their helpful recommendations.

References

[1] Mark Adams. Refactoring proofs with Tactician. In Domenico Bianculli, Radu Calinescu, and Bernhard Rumpe, editors, *Software Engineering and Formal Methods - SEFM 2015 Collocated Workshops: ATSE, HOFM, MoKMaSD, and VERY*SCART, York, UK, September 7-8, 2015, Revised Selected Papers*, volume 9509 of *Lecture Notes in Computer Science*, pages 53–67. Springer, 2015. https://doi.org/10.1007/978-3-662-49224-6_6.

[2] David Aspinall. Proof General: A generic tool for proof development. In Susanne Graf and Michael I. Schwartzbach, editors, *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 2000. https://doi.org/10.1007/3-540-46419-0_3.

[3] David Aspinall, Ewen Denney, and Christoph Lüth. Tactics for hierarchical proof. *Mathematics in Computer Science*, 3(3):309–330, 2010. https://doi.org/10.1007/s11786-010-0025-6.

[4] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT solvers. *J. Autom. Reasoning*, 51(1):109–128, 2013. https://doi.org/10.1007/s10817-013-9278-5.

[5] Paul A. Cairns and Jeremy Gow. A theoretical analysis of hierarchical proofs. In Andrea Asperti, Bruno Buchberger, and James H. Davenport, editors, *Mathematical Knowledge Management, Second International Conference, MKM 2003, Bertinoro, Italy, February 16-18, 2003, Proceedings*, volume 2594 of *Lecture Notes in Computer Science*, pages 175–187. Springer, 2003. https://doi.org/10.1007/3-540-36469-2_14.

[6] Robert L. Constable, Stuart F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, Douglas J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, NJ, 1986. http://www.nuprl.org/book/.

[7] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. https://doi.org/10.1007/978-3-319-21401-6_26.

[8] Ewen Denney, John Power, and Konstantinos Tourlas. Hiproofs: A hierarchical notion of proof tree. *Electr. Notes Theor. Comput. Sci.*, 155:341–359, 2006. https://doi.org/10.1016/j.entcs.2005.11.063.

[9] Lucas Dixon and Jacques D. Fleuriot. A proof-centric approach to mathematical assistants. *J. Applied Logic*, 4(4):505–532, 2006. https://doi.org/10.1016/j.jal.2005.10.007.

[10] John Harrison. Proof style. In Eduardo Giménez and Christine Paulin-Mohring, editors, *Types for Proofs and Programs, International Workshop TYPES'96, Aussois, France, December 15-19, 1996, Selected Papers*, volume 1512 of *Lecture Notes in Computer Science*, pages 154–172. Springer, 1996. https://doi.org/10.1007/BFb0097791.

[11] Amanda M. Holland-Minkley. Planning proof content for communicating induction. In *Proceedings of the International Natural Language Generation Conference, Harriman, New York, USA, July 2002*, pages 167–172. Association for Computational Linguistics, 2002. https://www.aclweb.org/anthology/W02-2122/.

[12] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, 2 edition, 2004. https://doi.org/10.1017/CBO9780511810275.

[13] Cezary Kaliszyk and Freek Wiedijk. Merging procedural and declarative proof. In Stefano Berardi, Ferruccio Damiani, and Ugo de'Liguoro, editors, *Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers*, volume 5497 of *Lecture Notes in Computer Science*, pages 203–219. Springer, 2008. https://doi.org/10.1007/978-3-642-02444-3_13.

[14] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, June 2000.

[15] Leslie Lamport. How to write a proof. Technical Report 94, DEC Systems Research Center, February 1993. https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-94.pdf.

[16] Helen Lowe, Alan Bundy, and Duncan McLean. The use of proof planning for co-operative theorem proving. *J. Symb. Comput.*, 25(2):239–261, 1998. https://doi.org/10.1006/jsco.1997.0174.

[17] Helen Lowe and David Duncan. XBarnacle: Making theorem provers more accessible. In William McCune, editor, *Automated Deduction - CADE-14, 14th International Conference on Automated Deduction, Townsville, North Queensland, Australia, July 13-17, 1997, Proceedings*, volume 1249 of *Lecture Notes in Computer Science*, pages 404–407. Springer, 1997. https://doi.org/10.1007/3-540-63104-6_39.

[18] 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, *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*, volume 7961 of *Lecture Notes in Computer Science*, pages 184–199. Springer, 2013. https://doi.org/10.1007/978-3-642-39320-4_12.

[19] Grant O. Passmore, Simon Cruanes, Denis Ignatovich, Dave Aitken, Matt Bray, Elijah Kagan, Kostya Kanishev, Ewen Maclean, and Nicola Mometto. The Imandra automated reasoning system (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 464–471. Springer, 2020. https://doi.org/10.1007/978-3-030-51054-1_30.

[20] Clément Pit-Claudel. Untangling mechanized proofs. In Ralf Lämmel, Laurence Tratt, and Juan de Lara, editors, *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, pages 155–174. ACM, 2020. https://doi.org/10.1145/3426425.3426940.

[21] 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. LΩUI: Lovely ΩMEGA User Interface. *Formal Asp. Comput.*, 11(3):326–342, 1999. https://doi.org/10.1007/s001650050053.

[22] 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, *Intelligent Computer Mathematics, 10th International Conference, AISC 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, MKM 2010, Paris, France, July 5-10, 2010. Proceedings*, volume 6167 of *Lecture Notes in Computer Science*, pages 440–454. Springer, 2010. https://doi.org/10.1007/978-3-642-14128-7_37.

[23] Carst Tankink and James McKinna. Dynamic proof pages. In Christoph Lange and Josef Urban, editors, *Proceedings of the ITP 2011 Workshop on Mathematical Wikis, Nijmegen, The Netherlands, August 27th, 2011*, volume 767 of *CEUR Workshop Proceedings*, pages 45–48. CEUR-WS.org, 2011. http://ceur-ws.org/Vol-767/paper-08.pdf.

[24] Iain Whiteside, David Aspinall, Lucas Dixon, and Gudmund Grov. Towards formal proof script refactoring. In James H. Davenport, William M. Farmer, Josef Urban, and Florian Rabe, editors, *Intelligent Computer Mathematics - 18th Symposium, Calculemus 2011, and 10th International Conference, MKM 2011,*

Bertinoro, Italy, July 18-23, 2011. Proceedings, volume 6824 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2011. https://doi.org/10.1007/978-3-642-22673-1_18.

[25] Freek Wiedijk. Formal proof sketches. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2003. https://doi.org/10.1007/978-3-540-24849-1_24.

[26] Freek Wiedijk, editor. *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *Lecture Notes in Computer Science*. Springer, 2006. https://doi.org/10.1007/11542384.

[27] Freek Wiedijk. A synthesis of the procedural and declarative styles of interactive theorem proving. *Logical Methods in Computer Science*, 8(1), 2012. https://doi.org/10.2168/LMCS-8(1:30)2012.