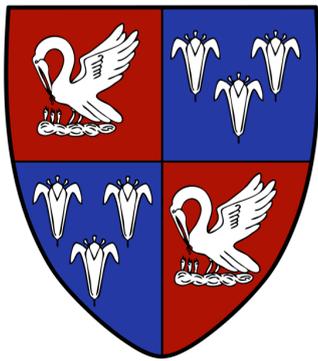


A Tool for Producing Verified, Explainable Proofs.

Edward William Ayers
Corpus Christi College
University of Cambridge
Submission Date: 2021-09-06

This thesis is submitted for the degree of Doctor of Philosophy.

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the preface and specified in the text. It is not substantially the same as any work that has already been submitted before for any degree or other qualification except as declared in the preface and specified in the text. It does not exceed the prescribed word limit for the Mathematics Degree Committee.



Abstract

Mathematicians are reluctant to use interactive theorem provers. In this thesis I argue that this is because proof assistants don't emphasise explanations of proofs; and that in order to produce good explanations, the system must create proofs in a manner that mimics how humans would create proofs. My research goals are to determine what constitutes a human-like proof and to represent human-like reasoning within an interactive theorem prover to create formalised, understandable proofs. Another goal is to produce a framework to visualise the goal states of this system.

To demonstrate this, I present *HumanProof*: a piece of software built for the Lean 3 theorem prover. It is used for interactively creating proofs that resemble how human mathematicians reason. The system provides a visual, hierarchical representation of the goal and a system for suggesting available inference rules. The system produces output in the form of both natural language and formal proof terms which are checked by Lean's kernel. This is made possible with the use of a structured goal state system which interfaces with Lean's tactic system which is detailed in Chapter 3.

In Chapter 4, I present the *subtasks* automation planning subsystem, which is used to produce equality proofs in a human-like fashion. The basic strategy of the subtasks system is break a given equality problem in to a hierarchy of tasks and then maintain a stack of these tasks in order to determine the order in which to apply equational rewriting moves. This process produces equality chains for simple problems without having to resort to brute force or specialised procedures such as normalisation. This makes proofs more human-like by breaking the problem into a hierarchical set of tasks in the same way that a human would.

To produce the interface for this software, I also created the ProofWidgets system for Lean 3. This system is detailed in Chapter 5. The ProofWidgets system uses Lean's metaprogramming framework to allow users to write their own interactive, web-based user interfaces to display within the VSCode editor and in an online web-editor. The entire tactic state is available to the rendering engine, and hence expression structure and types of subexpressions can be explored interactively. The ProofWidgets system also allows the user interface to interactively edit the proof document, enabling a truly interactive modality for creating proofs; human-like or not.

In Chapter 6, the system is evaluated by asking real mathematicians about the output of the system, and what it means for a proof to be understandable to them. The user group study asks participants to rank and comment on proofs created by HumanProof alongside natural language and pure Lean proofs. The study finds that participants generally prefer the HumanProof format over the Lean format. The verbal responses collected during the study indicate that providing intuition and signposting are the most important properties of a proof that aid understanding.

Contents

- **1. Introduction**
 - 1.1 Mathematicians and proof assistants
 - 1.2 Research questions
 - 1.3 Contributions
 - 1.4 Structure of this document
 - 1.5 Previously published work and collaboration
 - 1.6 Acknowledgements
- **2. Background**
 - 2.1 The architecture of proof assistants
 - 2.2 Preliminaries
 - 2.3 Inductive gadgets
 - 2.4 Metavariables
 - 2.5 Understandability and confidence
 - 2.6 Human-like reasoning
 - 2.7 Natural language for formal mathematics
 - 2.8 Chapter summary
- **3. A development calculus**
 - 3.1 Motivation
 - 3.2 Overview of the software
 - 3.3 The Box datastructure
 - 3.4 Creating valid proof terms from a Box
 - 3.5 Human-like-tactics for Box.
 - 3.6 Natural language generation of proofs
 - 3.7 Conclusion
- **4. Subtasks**
 - 4.1 Equational reasoning
 - 4.2 Example
 - 4.3 Design of the algorithm
 - 4.4 Qualitative comparison with related work
 - 4.5 Evaluation
 - 4.6 Conclusions and Further Work
- **5. A graphical user interface framework for formal verification**
 - 5.1 Survey of user interfaces for provers
 - 5.2 Background on web-apps and functional GUIs
 - 5.3 Research goals
 - 5.4 System description
 - 5.5 Interactive expressions
 - 5.6 Related work
 - 5.7 Implementation of Widgets in Lean
 - 5.8 Visualising Boxes
 - 5.9 Future work
- **6. Evaluation**
 - 6.1 Objectives
 - 6.2 Methodology
 - 6.3 Agenda of experiment
 - 6.4 Results

- 6.5 Quantitative analysis of ratings
- 6.6 Qualitative analysis of verbal responses
- 6.7 Threats to validity and limitations
- 6.8 Conclusions and future work
- 7. Conclusion
 - 7.1 Revisiting the research questions
 - 7.2 Future work and closing remarks
- A. Zippers and tactics for boxes
 - A.1 Typing of expressions containing metavariables
 - A.2 Zippers on Boxes
 - A.3 Running tactics in Boxes
- B. ProofWidgets tutorial
- C. The rendering algorithm of ProofWidgets
 - C.1 Motivating ProofWidgets with a todo list app
 - C.2 Abstracting the Html datatype
 - C.3 Holding a general representation of the UI state.
 - C.4 Reconciliation
 - C.5 Implementation
- D. Material for evaluation study
 - D.1 Advertising email
 - D.2 Training Document
 - D.3 Proof scripts
 - D.4 Consent form

Chapter 1

Introduction

My first contact with the ideas of formalised mathematics came from reading the anonymously authored *QED Manifesto* [Ano94]¹ which envisions a 'QED system' in which all mathematical knowledge is stored in a single, computer-verified repository. This idea dizzied me: perhaps review of mathematics will amount to remarking on style and interest, with checking of proofs performed automatically from a machine readable document.

The general term that I will use for software that works towards this vision is **proof assistant** or **Interactive Theorem Prover** ITP. A proof assistant at its most general is a piece of software that allows users to create and verify mathematical proofs. In Section 2.1 I will provide more detail how proof assistants are generally constructed.

In 2007, Freek Wiedijk [Wie07] pronounced the QED project to have "not been a success (yet)", citing not enough people working on formalised mathematics and the severe differences between formalised and 'real' mathematics, both at a syntactic level (formalised mathematics resembles source code) and at a foundational level (formalised mathematics is usually constructive and procedural as opposed to classical and declarative). Similarly, Alan Bundy [Bun11] notes that although mathematicians have readily adopted computational tools such as \TeX [Knu86] and computer algebra systems², computer aided proving has had very little impact on the workflow of a working mathematician. Bundy cites several reasons for this which will be discussed in Section 1.1.

Now, a decade later, the tide may be turning. In 2021, proof assistants are pretty good. There are several well-supported large-scale systems such as *Isabelle* [Pau89], *Coq* [Coq], *Lean* [MKA+15], *HOL Light* [Har09], *Agda* [Noro8], *Mizar* [GKN15], *PVS* [SORS01] and many more. These systems are used to define and prove mathematical facts in a variety of logics (e.g. FOL, HOL, CIC, univalent foundations). These systems are bridged to powerful, automated reasoning systems (e.g. *Vampire* [RVo2], *Z3* [MBo8], *E* [SCV19] and *Leo-III* [SB18a]). Within these systems, many theorems big and small (4-colour theorem [Gon08], Feit-Thompson theorem [GAA+13], Kepler conjecture [HAB+17]) have been proved in a variety of fields, accompanied by large mathematical libraries (*Isabelle's Archive of Formal Proofs*, *Lean's mathlib*, *Coq's Mathematical Components*, *Mizar's Formalized Mathematics*) whose intersection with undergraduate and research level mathematics is steadily growing³.

However, in spite of these advances, we are still yet to see widespread adoption of ITP by mathematicians outside of some (growing) cliques of enthusiasts. In this thesis I wish to address this problem through engaging with how mathematicians use and understand proofs to create new ways of interacting with formalised proof. Let's first expand on the problem a little more and then use this to frame the research questions that I will tackle for the remainder of the thesis.

1.1. Mathematicians and proof assistants

Here I offer 3 possible explanations for why mathematicians have not adopted proof assistants. Many have commented on these before: Bundy [Bun11] summarises the main challenges well.

1. Differing attitudes towards correctness and errors. Mathematicians don't worry about mistakes in the same way as proof assistants do⁴. Mathematicians care deeply about correctness, but historically the dynamics determining whether a result is considered to be true are also driven by sociological mechanisms such as peer-review; informal correspondences; 'folk' lemmas and principles; reputation of authors; and so on [MUP79].

[Ano94] **Anonymous** *The QED manifesto* (1994) Automated Deduction--CADE

¹ In this thesis, shortened citation references will appear in the sidebar, a full bibliography with all reference details is provided at the end of the document. Some sidebar citations will be omitted if there is not enough space.

[Wie07] **Wiedijk, Freek** *The QED manifesto revisited* (2007) *Studies in Logic, Grammar and Rhetoric*

[Bun11] **Bundy, Alan** *Automated theorem provers: a practical tool for the working mathematician?* (2011) *Annals of Mathematics and Artificial Intelligence*

[Knu86] **Knuth, Donald E.** *The TeXbook* (1986) **publisher** Addison-Wesley

² A computer algebra system (CAS) is a tool for symbolic manipulation of formulae and expressions, without necessarily having a formalised proof that the manipulation is sound. Examples of CASes include Maple and Mathematica.

³ See, for example, the rate of growth of the Lean 3 mathematical library https://leanprover-community.github.io/mathlib_stats.html.

⁴ I will present some evidence for this in Section 2.5.

A proxy for trustworthiness of a result is the number of other mathematicians that have scrutinized the work. That is, if the proof is found on an undergraduate curriculum, you can predict with a high degree of confidence that any errors in the proof will be brought to the lecturer's attention. In contrast, a standalone paper that has not yet been used for any subsequent work by others is typically treated with some degree of caution.

2. High cost. Becoming proficient in an ITP system such as Isabelle or Coq can require a lot of time. And then formalising an area of maths can take around ten times the amount of time required to write a corresponding paper or textbook on the topic. This time quickly balloons if it is also necessary to write any underlying assumed knowledge of the topic (e.g., measure theory first requires real analysis). This 'loss factor' of the space cost of developing a formalised proof over that of a natural language proof was first noted by de Bruijn in relation to his AUTOMATH prover [DeB80]. De Bruijn estimates a factor of 20 for AUTOMATH, and Wiedijk later estimates this factor to be closer to three or four in Mizar [Wie00]. There are costs other than space too, the main one of concern here being the time to learn to use the tools and the amount of work required per proof.

3. Low reward. What does a mathematician have to gain from formalising their research? In many cases, there is little to gain other than confirming something the researcher knew to be true anyway. The process of formalisation may bring to light 'bugs' in the work: perhaps there is a trivial case that wasn't accounted for or an assumption needs to be strengthened. Sometimes the reward is high enough that there is a clear case for formalisation, particularly when the proof involves some computer-generated component. This is exemplified by Hales' proof [Hal05] and later formalised proof [HAB+17] of the Kepler conjecture. The original proof involved lengthy computer generated steps that were difficult for humans to check, and so Hales led the *Flyspeck* project to formalise it, taking 21 collaborators around a decade to complete. Another celebrated example is Gonthier's formalisation of the computer-generated proof of the four-colour theorem [Gon08]. Formalisation is also used regularly in formalising expensive hardware and safety-critical computer software (e.g., [KEH+09, Pau98]).

The economics of the matter are such that the gains of using ITP are too low compared to the benefits for the majority of cases. Indeed, since mathematicians have a different attitude to correctness, there are sometimes *no* benefits to formalisation. As ITP developers, we can improve the situation by either decreasing the learning cost or increasing the utility.

How can we make ITP easier to learn? One way is to teach it in undergraduate mathematics curricula (not just computer science). An example of such a course is Massot's *Introduction aux mathématiques formalisées* taught at the Université Paris Sud. Another way is to improve the usability of the user interface for the proof assistant; I will consider this point in more detail in Chapter 5.

How can we increase the utility that mathematicians gain from using a proof assistant? In this thesis I will argue that one way to help with these three issues is to put more emphasis on interactive theorem provers providing *explanations* rather than a mere guarantee of correctness. We can see that explanations are important because mathematicians care about new proofs of old results that treat the problem in a new way. *Proofs from the Book* [AZHE10] catalogues some particularly lovely examples of this.

Can computers also provide informal proofs with more emphasis on explanations? Gowers [Gow10 §2] presents an imagined interaction between a mathematician and a proof assistant of the future.

Mathematician. Is the following true? Let $\delta > 0$. Then for N sufficiently large, every set $A \subseteq \{1, 2, \dots, N\}$ of size at least δN contains a subset of the form $\{a, a + d, a + 2d\}$?

Computer. Yes. If A is non-empty, choose $a \in A$ and set $d = 0$.

M. All right all right, but what if d is not allowed to be zero?

[MUP79] **de Millo, Richard A; Upton, Richard J; Perlis, Alan J** *Social processes and proofs of theorems and programs* (1979) Communications of the ACM

[DeB80] **De Bruijn, Nicolaas Govert** *A survey of the project AUTOMATH* (1980) To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism

[Wie00] **Wiedijk, Freek** *The de Bruijn Factor* (2000) <http://www.cs.ru.nl/F.Wiedijk/factor/factor.pdf>

[Hal05] **Hales, Thomas C** *A proof of the Kepler conjecture* (2005) Annals of mathematics

[HAB+17] **Hales, Thomas C; Adams, Mark; Bauer, Gertrud; et al.** *A formal proof of the Kepler conjecture* (2017) Forum of Mathematics, Pi

[Gon08] **Gonthier, Georges** *Formal proof--the four-color theorem* (2008)

[KEH+09] **Klein, Gerwin; Elphinstone, Kevin; Heiser, Gernot; et al.** *seL4: Formal verification of an OS kernel* (2009) Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles

[Pau98] **Paulson, Lawrence C** *The inductive approach to verifying cryptographic protocols* (1998) Journal of Computer Security

[AZHE10] **Aigner, Martin; Ziegler, Günter M; Hofmann, Karl H; et al.** *Proofs from the Book* (2010) **publisher** Springer

[Gow10] **Gowers, W. T.** *Rough structure and classification* (2010) Visions in Mathematics

C. Have you tried induction on N , with some $\delta = \delta(N)$ tending to zero?

M. That idea is no help at all. Give me some examples please.

C. The obvious greedy algorithm gives the set

$\{1, 2, 4, 5, 10, 11, 13, 14, 28, 29, 31, \dots\}$

An interesting feature of this conversation is that the status of the formal correctness of any of the statements conveyed by the computer is not mentioned. Similar notions are brought to light in the work of Corneli *et al.* [CMM+17] in their modelling of informal mathematical dialogues and exposition.

Why not have both explanatory and verified proofs? I suspect that if an ITP system is to be broadly adopted by mathematicians, it must concisely express theorems and their proofs in a way similar to that which a mathematician would communicate with fellow mathematicians. This not only requires constructing human-readable explanations, but also a reimagining of how the user can interact with the prover.

In this thesis, I will focus on problems that are considered 'routine' for a mathematician. That is, problems that a mathematician would typically do 'on autopilot' or by 'following their nose' ⁵. I choose to focus on this class of problem because I believe it is an area where ITP could produce proofs that explain why they are true rather than merely provide a certificate of correctness. The typical workflow when faced with a problem like this is to either meticulously provide a low-level proof or apply automation such as Isabelle's `auto`, or an automation orchestration tool such as Isabelle's Sledgehammer [BN10]. In the case of using an automation tactic⁶ like `auto` the tactic will either fail or succeed, leaving the user with little feedback on why the result is true. There are some tools for producing intelligible proofs from formalised ones, for example, the creation of Isar [Wen99] proofs from Sledgehammer by Blanchette *et al.* [BBF+16]. However, gaining an intuition for a proof will be easier if the proof is generated in a way that reflects how a human would solve the problem, and so translating a machine proof to a proof which a human will extract meaning from is an uphill battle.

1.1.1. Types of understandability

The primary motivation of the work in this thesis is to help make ITP systems more appealing to mathematicians. The approach I chosen to take towards this is to research ways of making ITP systems more *understandable*. There are many components of ITP that I consider with respect to understandability:

- **interaction:** is the way in which the user interacts and creates a proof easy to understand?
- **system-output:** is the final proof rendered to the user easy to understand?
- **underlying representations:** is the way in which the proof is stored similar to the user's understanding of the proof?
- **automation:** if a proof is generated automatically, is it possible for a user to follow it?

The different parts of my thesis will address different sets of these ways in which a proof assistant can be understandable. With respect to the automation and underlying-representation aspects of understandability, we will see in Section 2.6 that there is some debate over whether prover automation needs to be easy to follow for a human or not (machine-like vs. human-like). In this thesis I take a pragmatic stance that the understandability of automation and underlying-representation need not be human-like provided that the resulting interaction and output is understandable. However, as I investigate in Chapter 4, there may be ways of creating automation that are more conducive to creating understandable output and interaction.

[CMM+17] **Corneli, Joseph; Martin, Ursula; Murray-Rust, Dave; et al.** *Modelling the way mathematics is actually done* (2017) Proceedings of the 5th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design

⁵ For example, showing that $(a + b)^2 = a^2 + 2ab + b^2$ from the ring axioms.

[BN10] **Böhme, Sascha; Nipkow, Tobias** *Sledgehammer: judgement day* (2010) International Joint Conference on Automated Reasoning

⁶ Broadly, a tactic is a program for creating proofs. I will drill down on how this works in Chapter 2.

[Wen99] **Wenzel, Markus** *Isar - A Generic Interpretative Approach to Readable Formal Proof Documents* (1999) Theorem Proving in Higher Order Logics

[BBF+16] **Blanchette, Jasmin Christian; Böhme, Sascha; Fleury, Mathias; et al.** *Semi-intelligible Isar proofs from machine-generated proofs* (2016) Journal of Automated Reasoning

1.2. Research questions

In the context of these facets of an understandable ITP system, there arise some key research questions that I seek to study.

Question 1. What constitutes a human-like, understandable proof?

Objectives:

- Identify what 'human-like' and 'understandable' mean to different people.
- Distinguish between human-like and machine-like proofs in the context of ITP.
- Merge these strands to determine a working definition of human-like proof.

Question 2. How can human-like reasoning be represented within an interactive theorem prover to produce *formalised*, understandable proofs?

Objectives:

- Form a calculus of representing goal states and inference steps that act at the abstraction layer that a human uses when solving proofs.
- Create a system for producing natural language proofs from this calculus.
- Evaluate the resulting system by performing a study on real mathematicians.

Question 3. How can this mode of human-like reasoning be presented to the user in an interactive, multimodal way?

Objectives:

- Investigate new ways of interacting with proof objects.
- Make it easier to create novel *graphical user interfaces* (GUIs) for interactive theorem provers.
- Produce an interactive interface for a human-like reasoning system.

1.3. Contributions

This thesis presents a number of contributions towards the above research questions:

1. An abstract calculus for developing human-like proofs (Chapter 3).
2. An interface between this abstraction layer and a metavariable-driven tactic state, as is used in theorem provers such as Coq and Lean, producing formally verified proofs (Chapter 3 and Appendix A).
3. A procedure for generating natural `language` proofs from this calculus (Section 3.6).
4. The 'subtasks' algorithm, a system for automating the creation of chains of equalities and inequalities. This work has been published in [AGJ19] (Chapter 4).
5. A graphical user interface framework for interactive theorem provers (Chapter 5). This has been published in [AJG21].
6. An implementation of all of the above contributions in the Lean 3 theorem prover.
7. A study assessing the impact of natural language proofs with practising mathematicians (Chapter 6).

The implementations for these contributions can be found at the following links:

- <https://github.com/edayers/lean-humanproof-thesis> for the primary implementation of HumanProof.
- <https://github.com/edayers/lean-subtask> for a supplementary implementation of the subtasks algorithm presented in Chapter 4 and originally presented at [AGJ19]
- An implementation of the ProofWidgets code has been incorporated in to the [leanprover-community](#) fork of the Lean theorem prover. The relevant pull requests are:
 - <https://github.com/leanprover-community/lean/pull/258>

[AGJ19] **Ayers, E. W.; Gowers, W. T.; Jamnik, Mateja** *A human-oriented term rewriting system* (2019) KI 2019: Advances in Artificial Intelligence - 42nd German Conference on AI

[AJG21] **Ayers, E. W.; Jamnik, Mateja; Gowers, W. T.** *A graphical user interface framework for formal verification* (2021) Interactive Theorem Proving

1.4. Structure of this document

In Chapter 2, I will provide an overview of the background material needed for the remaining chapters. Next, in Chapter 3, I introduce the HumanProof software for producing human-like proofs within the Lean proof assistant. I provide motivation of the design in Section 3.1, an overview of the system in Section 3.2 and then dive in to the details of how the system is designed, including the natural-language generation engine in Section 3.6. Chapter 4 discusses a system for producing equational reasoning proofs called the subtask algorithm. Chapter 5 details the ProofWidgets system, which is used to produce the user interface of HumanProof. Chapter 6 provides the design and results of a user study that I conducted on mathematicians to determine whether HumanProof really does provide understandable proofs. Finally, Chapter 7 wraps things up with some reflection on my progress and a look ahead to future work.

There are also four appendices:

- Appendix A presents some additional technical detail on interfacing HumanProof with Lean.
- Appendix B is a tutorial for using ProofWidgets.
- Appendix C is some additional detail on the algorithms used by ProofWidgets.
- Appendix D provides supplementary material for Chapter 6.

1.5. Previously published work and collaboration

The work in Chapter 3 is my own, although the box calculus presented is inspired through many sessions of discussion with W.T. Gowers and the design of Gowers' previous collaboration with Ganesalingam [GG17]. More on this will be given when it is surveyed in Section 2.6 and Section 3.3.5.

The work in Chapter 4 is previously published at KI 2019 [AGJ19].

The work presented in Chapter 5 is pending publication in ITP 2021 [AJG21] and is also [merged in to the Lean 3 community repository](#). The design is strongly influenced by Elm and React; however, there are a number of novel architectural contributions necessitated by the unique challenges of implementing a portable framework within a proof assistant.

The user study presented in Chapter 6 is all my own work with a lot of advisory help from Mateja Jamnik, Gem Stapleton and Aaron Stockdill on designing the study.

1.6. Acknowledgements

I thank my supervisors W. T. Gowers and Mateja Jamnik for their ideas, encouragement and support and for letting me work on such a wacky topic. I thank Gabriel Ebner and Brian Gin-ge Chen for reading through my ProofWidgets PRs. I thank Patrick Massot, Kevin Buzzard and the rest of the Lean Prover community for complaining about my PRs after the fact. I thank Jeremy Avigad for taking the time to introduce me to Lean at the *Big Proof* conference back in 2017. I thank Bohua Zhan, Chris Sangwin, and Makarius Wenzel and many more for the enlightening conversations on automation for mathematicians at *Big Proof* and beyond. I thank Peter Koepke for being so generous in inviting me to Bonn to investigate Naproche/SAD with Steffan Frerix and Andrei Paskevich. I thank Larry Paulson and the ALEXANDRIA team for letting me crash their weekly meetings. I thank my parents for letting me write up in the house during lockdown.

[GG17] **Ganesalingam, Mohan; Gowers, W. T.** *A fully automatic theorem prover with human-style output* (2017) Journal of Automated Reasoning

[AGJ19] **Ayers, E. W.; Gowers, W. T.; Jamnik, Mateja** *A human-oriented term rewriting system* (2019) KI 2019: Advances in Artificial Intelligence - 42nd German Conference on AI

[AJG21] **Ayers, E. W.; Jamnik, Mateja; Gowers, W. T.** *A graphical user interface framework for formal verification* (2021) Interactive Theorem Proving

I thank my friends and colleagues in the [CMS](#). Andrew, Eric, Sammy P, Sven, Ferdia, Mithuna, Kasia, Sam O-T, Bhavik, Wojciech, and many more. In parallel, the [Computer Laboratory](#): Chaitanya, Botty, Duo, Daniel, Aaron, Angeliki, Yiannos, Wenda, Zoreh.

This research was supported by EPSRC and the [Cantab Capital Institute for the Mathematics of Information](#).

1.6.1. Typesetting acknowledgements

I decided to typeset this thesis as HTML-first, print second. The digital copy may be found at <https://edayers.com/thesis>. The printed version of this thesis was generated by printing out the website version and concatenating.

I was able to create the thesis in this way thanks to many open-source projects. I will acknowledge the main ones here. [React](#), [Gatsby](#), [Tachyons](#), [PrismJS](#). Thanks to [Titus Woormer](#) for [remarkJS](#) and also adding my feature request in less than 24 hours! The code font is [PragmataPro](#) created by Fabrizio Schiavi. The style of the site is a modified version of the [Edward Tufte Handout style](#). The syntax colouring style is based on the [VS theme](#) by [Andrew Lock](#). I also use some of the [vscode-icons](#) icons.

Chapter 2

Background

In this chapter I will provide a variety of background material that will be used in later chapters. Later chapters will include links to the relevant sections of this chapter. I cover a variety of topics:

- Section 2.1 gives an overview of how proof assistants are designed. This provides some context to place this thesis within the field of ITP.
- Section 2.2 contains some preliminary definitions and notation for types, terms, datatypes and functors that will be used throughout the document.
- Section 2.3 contains some additional features of inductive datatypes that I will make use of in various places throughout the text.
- Section 2.4 discusses the way in which metavariables and tactics work within the Lean theorem prover, the system in which the software I write is implemented.
- Section 2.5 asks what it means for a person to understand or be confident in a proof. This is used to motivate the work in Chapter 3 and Chapter 4. It is also used to frame the user study I present in Chapter 6.
- Section 2.6 explores what the automated reasoning literature has to say on how to define and make use of 'human-like reasoning'. This includes a survey of proof planning (Section 2.6.2).
- Section 2.7 surveys the topic of natural language generation of mathematical texts, used in Section 3.6.

2.1. The architecture of proof assistants

In this section I am going to provide an overview of the designs of proof assistants for non-specialist. The viewpoint I present here is largely influenced by the viewpoint that Andrej Bauer expresses in a MathOverflow answer [Bau20].

The essential purpose of a proof assistant is to represent mathematical theorems, definitions and proofs in a language that can be robustly checked by a computer. This language is called the **foundation language** equipped with a set of **derivation rules**. The language defines the set of objects that formally represent mathematical statements and proofs, and the inference rules and axioms provide the valid ways in which these objects can be manipulated⁷. Some examples of foundations are [first-order logic \(FOL\)](#), [higher-order logic \(HOL\)](#), and various forms of dependent type theory (DTT) [Mar84, CH88, PP89, Pro13].

A component of the software called the **kernel** checks proofs in the foundation. There are numerous foundations and kernel designs. Finding new foundations for mathematics is an open research area but FOL, HOL and DTT mentioned above are the most well-established for performing mathematics. I will categorise kernels as being either 'checkers' or 'builders'.

A 'checker' kernel takes as input a proof expression and outputs a yes/no answer to whether the term is a valid proof. An example of this is the Lean 3 kernel [MKA+15].

A 'builder' kernel provides a fixed set of partial functions that can be used to build proofs. Anything that this set of functions accepts is considered as valid. This is called an LCF architecture, originated by Milner [Mil72, Goroo]. The most widely used 'builder' is the Isabelle kernel by Paulson [Pau89].

[Bau20] **Bauer, Andrej** *What makes dependent type theory more suitable than set theory for proof assistants?* (2020) <https://mathoverflow.net/q/376973>

⁷At this point, we may raise a number of philosophical objections such as whether the structures and derivations 'really' represent mathematical reasoning. The reader may enjoy the account given in the first chapter of *Logic for Mathematicians* by J. Barkley Rosser [Ros53].

[MKA+15] **de Moura, Leonardo; Kong, Soonho; Avigad, Jeremy; et al.** *The Lean theorem prover (system description)* (2015) International Conference on Automated Deduction

Most kernels stick to a single foundation or family of foundations. The exception is Isabelle, which instead provides a 'meta-foundation' for defining foundations, however the majority of work in Isabelle uses the HOL foundation.

2.1.1. The need for a vernacular

One typically wants the kernel to be as simple as possible, because any bugs in the kernel may result in 'proving' a false statement ⁸. For the same reason, the foundation language should also be as simple as possible. However, there is a trade-off between kernel simplicity and the usability and readability of the foundation language; a simplified foundation language will lack many convenient language features such as implicit arguments and pattern matching, and as a result will be more verbose. If the machine-verified definitions and lemmas are tedious to read and write, then the prover will not be adopted by users.

Proof assistant designers need to bridge this gap between a human-readable, human-understandable proof and a machine-readable, machine-checkable proof. A common approach is to use a second language called the **vernacular** (shown on Figure 2.5). The vernacular is designed as a human-and-machine-readable compromise that is converted to the foundation language through a process called **elaboration** (e.g., [MAKR15]). The vernacular typically includes a variety of essential features such as implicit arguments and some form of type inference, as well as high-level programming features such as pattern matching. Optionally, there may be a compiler (see Figure 2.5) for the vernacular to also produce runnable code, for example Lean 3 can compile vernacular to bytecode [EUR+17].

I discuss some work on provers with the vernacular being a restricted form of natural language as one might find in a textbook in Section 2.7.2.

2.1.2. Programs for proving

Using a kernel for checking proofs and a vernacular structure for expressing theorems, we now need to be able to construct proofs of these theorems.

An **Automated Theorem Prover** (ATP) is a piece of software that produces proofs for a formal theorem statement automatically with a minimal amount of user input as to how to solve the proof, examples include [Z3](#), [E](#) and [Vampire](#).

Interactive Theorem Proving (ITP) is the process of creating proofs incrementally through user interaction with a prover. I will provide a review of user interfaces for ITP in Section 5.1. Most proof assistants incorporate various automated and interactive theorem proving components. Examples of ITPs include [Isabelle](#) [Pau89], [Coq](#) [Coq], [Lean](#) [MKA+15], [HOL Light](#) [Har09], [Agda](#) [Noro8], [Mizar](#) [GKN15], [PVS](#) [SORSo1].

A common modality for allowing the user to interactively construct proofs is with the **proof script** (Figure 2.1), this is a sequence of textual commands, written by the user to invoke certain proving programs called **tactics** that manipulate a state representing a partially constructed proof. An example of a tactic is the `assume` tactic in Figure 2.1, which converts a goal-state of the form $\vdash X \rightarrow Y$ to $X \vdash Y$. Some of these tactics may invoke various ATPs to assist in constructing proofs. Proof scripts may be purely linear as in Figure 2.1 or have a hierarchical structure such as in [Isar](#) [Wen99] or [HiProof](#) [ADL10].

An alternative to a proof script is for the prover to generate an auxiliary proof object file that holds a representation of the proof that is not intended to be human readable. This is the approach taken by [PVS](#) [SORSo1] although I will not investigate this approach further in this thesis because most of the ITP systems use the proof-script approach.

In the process of proving a statement, a prover must keep track of partially built proofs. I will refer to these representations of partially built proofs as **development calculi**. I will return to development calculi in Section 2.4.

[Mil72] **Milner, Robin** *Logic for computable functions description of a machine implementation* (1972) Technical Report

[Goro0] **Gordon, Mike** *From LCF to HOL: a short history* (2000) Proof, language, and interaction

[Pau89] **Paulson, Lawrence C** *The foundation of a generic theorem prover* (1989) Journal of Automated Reasoning

⁸ An alternative approach is to 'bootstrap' increasingly complex kernels from simpler ones. An example of this is the [Milawa theorem prover](#) for ACL2 [Dav09].

[MAKR15] **de Moura, Leonardo; Avigad, Jeremy; Kong, Soonho; et al.** *Elaboration in Dependent Type Theory* (2015) CoRR

[EUR+17] **Ebner, Gabriel; Ullrich, Sebastian; Roesch, Jared; et al.** *A metaprogramming framework for formal verification* (2017) Proceedings of the ACM on Programming Languages

Figure 2.1. An example proof script from the Lean 3 theorem prover. The script proper are the lines between the `begin` and `end` keywords. Each line in the proof script corresponds to a tactic.

```
lemma p : P ∧ Q → P ∨ Q :=
begin
  assume (h : P ∧ Q),
  cases h,
  apply or.inl,
  assumption
end
```

[Wen99] **Wenzel, Markus** *Isar - A Generic Interpretative Approach to Readable Formal Proof Documents* (1999) Theorem Proving in Higher Order Logics

2.1.3. Foundation

A **foundation** for a prover is built from the following pieces:

1. A **language**: defining inductive trees of data that we wish to talk about and also syntax for these trees.
2. The **judgements**: meta-level predicates over the above trees.
3. The **inference rules**: a generative set of rules for deriving judgements from other judgements.

To illustrate, the language of simply typed lambda calculus would be expressed as in (2.2).

```

x, y, z ::= X           -- variable
α, β   ::= A | α → β   -- type
s, t   ::= s t | λ (x : α), s | X -- term
Γ      ::= ϕ | Γ, (x : α) -- context

```

In (2.2), the purple greek and italicised letters (x , y , α , ...) are called **nonterminals**. They say: "You can replace me with any of the $\boxed{\quad}$ -separated items on the right-hand-side of my $::=$ ". So, for example, " α " can be replaced with either a member of \boxed{A} or " $\alpha \rightarrow \beta$ ". The green words in the final column give a natural language noun to describe the 'kind' of the syntax.

In general terms, **contexts** Γ perform the role of tracking which variables are currently in scope. To see why contexts are needed, consider the expression $x + y$; its resulting type depends on the types of the variables x and y . If x and y are both natural numbers, $x + y$ will be a natural number, but if x and y have different types (e.g. vectors, strings, complex numbers) then $x + y$ will have a different type too. The correct interpretation of $x + y$ depends on the *context* of the expression.

Next, define the judgements for our system in (2.3). Judgements are statements about the language.

```

Γ ⊢ ok      Γ ⊢ t : α

```

Then define the **natural deduction** rules (2.4) for inductively deriving these judgements.

```

_____ ϕ-ok _____      Γ ok      (x : α) ∈ Γ
ϕ ok                        (x : α) ∉ Γ      _____ var-typing _____
                                _____ append-ok _____      Γ ⊢ x : α
                                [..Γ, (x : α)] ok

Γ ⊢ s : α → β      x ∉ Γ
Γ ⊢ t : α           Γ, (x : α) ⊢ t : β
_____ app-typing _____      _____ λ-typing _____
Γ ⊢ s t : β                    Γ ⊢ (λ (x : α), t) : α → β

```

And from this point, it is possible to start exploring the theoretical properties of the system. For example: is $\Gamma \vdash s : \alpha$ decidable?

Foundations such as the example above are usually written down in papers as a BNF grammar and a spread of gammas, turnstiles and lines as illustrated in (2.2), (2.3) and (2.4). LISP pioneer Steele calls it *Computer Science Metanotation* [Ste17].

In *implementations* of proof assistants, the foundation typically doesn't separate quite as cleanly in to the above pieces. The language is implemented with a number of optimisations such as de Bruijn indexing [deB72] for the sake of efficiency. Judgements and rules are implicitly encoded in algorithms such as type checking, or appear in forms different from that in the corresponding paper. This is primarily for efficiency and extensibility.

[ADL10] **Aspinall, David; Denney, Ewen; Lüth, Christoph** *Tactics for hierarchical proof* (2010) Mathematics in Computer Science

[SORS01] **Shankar, Natarajan; Owre, Sam; Rushby, John M; et al.** *PVS prover guide* (2001) Computer Science Laboratory, SRI International, Menlo Park, CA

(2.2). Example of a BNF grammar specification. \boxed{A} and \boxed{X} are some sets of variables (usually strings of unicode letters).

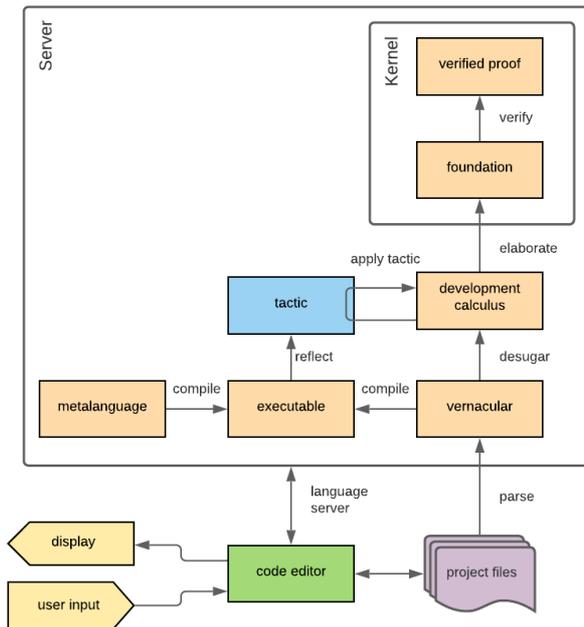
(2.3). Judgements for an example lambda calculus foundation. Γ , t and α may be replaced with expressions drawn from the grammar in (2.2)

(2.4). Judgement derivation rules for the example lambda calculus (2.2). Each rule gives a recipe for creating new judgements: given the judgements above the horizontal line, we can derive the judgement below the line (substituting the non-terminals for the appropriate ground terms). In this way one can inductively produce judgements.

[Ste17] **Steele Jr., Guy L.** *It's Time for a New Old Language* (2017) <http://2017.clojure-conj.org/guy-steele/>

In this thesis the formalisation language that I focus on is the calculus of inductive constructions (CIC) ⁹This is the the type theory used by Lean 3 as implemented by de Moura *et al* and formally documented by Carneiro [Car19]. A good introduction to mathematical formalisation with dependent type theory is the first chapter of the *HoTT Book* [Pro13 ch. 1]. Other foundations are also available: Isabelle's foundation is two-tiered [Pau89]: there is a meta-level foundation upon which many foundations can be implemented. A lot of the work in this thesis is independent of foundation and so I will try to indicate how the contributions can be augmented to work in other foundations.

A typical architecture of a modern, full-fledged checker-style proof assistant is given in Figure 2.5.



[deB72] **de Bruijn, Nicolaas Govert** *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem* (1972) *Indagationes Mathematicae* (Proceedings)

⁹ Calculus of Inductive Constructions. Inductive datastructures (Section 2.2.3) for the Calculus of Constructions [CH88] were first introduced by Pfenning *et al* [PP89].

[Car19] **Carneiro, Mario** *Lean's Type Theory* (2019) Masters' thesis (Carnegie Mellon University)

[Pro13] **The Univalent Foundations Program** *Homotopy Type Theory: Univalent Foundations of Mathematics* (2013) **publisher** Institute for Advanced Study

[Pau89] **Paulson, Lawrence C** *The foundation of a generic theorem prover* (1989) *Journal of Automated Reasoning*

Figure 2.5. Schematic overview of a typical modern kernel-based proof assistant.

2.2. Preliminaries

This section contains a set of quick preliminary definitions for the concepts and notation that I will be using later. In this thesis I will be using a pseudo-language which should be familiar to functional programming enthusiasts. This pseudo-language is purely presentational and is used to represent algorithms and datastructures for working with theorem provers.

2.2.1. Some notation for talking about type theory and algorithms

The world is built of *types* and *terms*. New variables are introduced as " $x : A$ "; x is the variable and it has the type A . Lots of variables with the same type can be introduced as $x y z : A$. Types $A B C : \text{Type}$ start with an uppercase letter and are coloured turquoise. Type is a special 'type of types'. Meanwhile terms start with a lowercase letter and term variables are purple and italicised. $A \rightarrow B$ is the function type. \rightarrow is *right associative* which means that $f : A \rightarrow B \rightarrow C$ should be read as $f : A \rightarrow (B \rightarrow C)$. This is called a *curried function*, we may consider A and B to be the input arguments of f and C to be its return type. Given $a : A$ we may *apply* f to a by writing $f a : B \rightarrow C$. Functions are introduced using maps-to notation $(a : A) \mapsto (b : B) \mapsto f a b$. Write the identity function $x \mapsto x$ as $1 : X \rightarrow X$. Given $f : A \rightarrow B$, $g : B \rightarrow C$, write function composition as $g \circ f : A \rightarrow C$. Function application is left associative, so $f a b$ should be read as $(f(a))(b)$. The input types of functions may optionally be given argument names, such as: $(a : A) \rightarrow (b : B) \rightarrow C$. We also allow 'dependent types' where the return value C is

allowed to depend on these arguments: $(a : A) \rightarrow \mathcal{C} a$ where $\mathcal{C} : A \rightarrow \text{Type}$ is a *type-valued function*.

- `Empty` is the empty type.
- `Unit` is the type containing a single element `()`.
- `Bool` is the boolean type ranging over values `true` and `false`.
- `Option X` is the type taking values `some x` for $x : X$ or `none`. `some` will usually be suppressed. That is, $x : X$ will be implicitly cast to `some x : Option X` in the name of brevity.
- `List X` is the type of finite lists of X . Given $x y : X$ and $l_1 l_2 : \text{List } X$, we can write $x :: l_1$ for list cons and $l_1 ++ l_2$ for concatenating (i.e. appending) two lists. For list construction and pattern matching, list *spreads* will be used. For example `[..l1, x, y, ..l2]` denotes the list formed by concatenating l_1 , `[x, y]` and l_2 . Python-style list comprehensions are also used: `[i2 for i in 1..20]` is a list of the first 20 square numbers.
- `N` is the type of natural numbers. Individual numbers can be used as types: $x : 3$ means that x is a natural number taking any value $x < 3$, i.e. $x \in \{0, 1, 2\}$.
- $A \times B$ is the type of **tuples** over A and B . Elements are written as `(a, b) : A × B`. As usual we have projections $\pi_1 (a, b) := a$ and $\pi_2 (a, b) := b$. Members of tuples may be given names as $(a : A) \times (b : B)$. In this case, supposing $p : (a : A) \times (b : B)$, we can write `p.a` and `p.b` instead of $\pi_1 p$ and $\pi_2 p$. Similarly to above, we can have a dependent tuple or 'sigma type' $(a : A) \times (b : B(a))$.
- $A + B$ is the discriminated union of A and B with constructors `inl : A → A + B` and `inr : B → A + B`.

2.2.2. Functors and monads

I will assume that the readers are already familiar with the motivation behind functors and monads in category theory and as used in e.g. Haskell but I will summarise them here for completeness. I refer the unfamiliar reader to the [Haskell Typeclassopedia](https://wiki.haskell.org/Typeclassopedia)¹⁰.

Definition 2.6 (functor): A **functor** is a type-valued function $F : \text{Type} \rightarrow \text{Type}$ equipped with a function mapper $F (f : A \rightarrow B) : F A \rightarrow F B$ ¹¹. I always assume that the functor is **lawful**, which here means it obeys the functor laws (2.7).

¹⁰ <https://wiki.haskell.org/Typeclassopedia>

¹¹ Here, the word 'functor' is used to mean the special case of category-theoretical functors with the domain and codomain category being the category of `Type`.

(2.7). Laws for functors.

$$F (f \circ g) = (F f) \circ (F g) \quad F (x \mapsto x) y = y$$

Definition 2.8 (natural function): A **natural function** $a : F \Rightarrow G$ between functors $F G : \text{Type} \rightarrow \text{Type}$ is a family of functions $a[A] : F A \rightarrow G A$ indexed by $A : \text{Type}$ such that $a[B] \circ F f = G f \circ a[A]$ for all $f : A \rightarrow B$. Often the type argument to a will be suppressed. It is quick to verify that the functors and natural functions over them form a category.

Definition 2.9 (monad): A **monad**¹² $M : \text{Type} \rightarrow \text{Type}$ is a functor equipped with two natural functions `pure : 1 ⇒ M` and `join : M M ⇒ M` obeying the monad laws (2.10). Write $m \gg= f := \text{join } (M f m)$ for $m : M A$ and $f : A \rightarrow M B$. `do` notation is used in places¹³.

¹² For learning about programming with monads, see https://wiki.haskell.org/All_About_Monads

¹³ <https://wiki.haskell.org/Keywords#do>

(2.10). Laws for monads.

$$\begin{aligned} \text{join}[X] \circ (M \text{ join}[X]) &= \text{join}[X] \circ (\text{join}[M X]) & \text{join}[X] \circ (M \text{ pure}[X]) &= \text{pure } X \\ \text{join}[X] \circ (\text{pure}[M X]) &= \text{pure } X \end{aligned}$$

Definition 2.11 (applicative): An **applicative functor** [MPo8 §2] $M : \text{Type} \rightarrow \text{Type}$ is equipped with `pure : A → M A` and `seq : M (A → B) → M A → M B`. Write $f <*> a := \text{seq } f x$ ¹⁴ and $a *> b := \text{seq } (_ \mapsto a) b$. Applicative functors obey the laws given in (2.12).

[MPo8] **McBride, Conor; Paterson, Ross** *Applicative programming with effects* (2008) J. Funct. Program.

(pure 1) <*> u = u (pure (◦)) <*> u <*> v <*> w = u <*> (v <*> w)

(pure f) <*> (pure x) = pure (f x) u <*> pure x = pure (f ↦ f x) <*> u

2.2.3. Inductive datatypes

New inductive datatypes are defined with a [GADT-like syntax](#) (2.13).

```
List (X : Type) ::=
| nil
| cons (x : X) (l : List X)
```

In cases where it is obvious which constructor is being used, the tag names are suppressed. Function definitions with pattern matching use the syntax given in (2.14).

```
f : Bool + (X × Y) → ℕ
| true      ↦ 3
| false     ↦ 0
| (x, y)    ↦ 2
```

One can express inductive datatypes D as fixpoints of functors $D = \text{Fix } P$ where $\text{Fix } P := P (\text{Fix } P)$. Depending on the underlying category, $\text{Fix } P$ may not exist for all P ¹⁵.

Definition 2.15 (base functor): When a $D : \text{Type}$ is written as $\text{Fix } P$ for some P (and there is no Q such that $P = Q \circ Q \circ \dots \circ Q$), P is called the **base functor** for D . This conceptualisation is useful because we can use the base functor to make related types without needing to explicitly write down the constructors for the modified versions. For example we can make the list lazy with $\text{Lazy } P \ X := \text{Fix } ((X \mapsto \text{Unit} \rightarrow X) \circ P)$.

2.3. Inductive gadgets

For the rest of this thesis, I will make use of a few motifs for discussing inductive datastructures, particularly in Section 2.4, Chapter 3, Appendix A and Appendix C. In this section I will lay some background material for working with inductive datatypes.

2.3.1. Traversable functors

Given a monad M , a common task is performing a monad-map with $f : A \rightarrow M B$ over a list of objects $l : \text{List } X$. This is done with the help of a function called `mmap` (2.16).

```
mmap (f : A → M B)
: List A → M (List B)
| []      ↦ pure []
| (h::l) ↦ pure cons <*> f h <*> mmap f l
```

But we can generalise `List` to some functor $T : \text{Type} \rightarrow \text{Type}$; when can we equip an analogous `mmap` to T ? For example, in the case of binary trees (2.17).

```
Tree A ::=
| leaf : Tree A
| branch : Tree A → A → Tree A → Tree A

mmap (f : A → M B)
: Tree A → M (Tree B)
| leaf      ↦ pure leaf
| (branch l a r) ↦
  pure branch <*> mmap f l <*> f a <*> mmap f r
```

Definition 2.18 (traversable): A functor $T : \text{Type} \rightarrow \text{Type}$ is **traversable** when for all applicative functors (Definition 2.11) $M : \text{Type} \rightarrow \text{Type}$, there is a natural function $d[M] : (T \circ M) \Rightarrow (M \circ T)$. That is, for each $X : \text{Type}$ we have

¹⁴ `<*>` is left associative:
 $u <*> v <*> w = (u <*> v) <*> w$.

(2.12). Laws for applicative functors. I use the same laws as presented by McBride [MPO8] but other equivalent sets are available.

(2.13). Example inductive definition of `List` using a `nil : List X` and `cons : X → List X → List X` are the constructors.

(2.14). Example of the definition of a function `f` using pattern matching. The `inl` and `inr` constructors are suppressed in the pattern. Provocative spacing is used instead to suggest which case is being matched on.

¹⁵ Smyth and Plotkin are the first to place some conditions on when the fixpoint exists [SP82], see Adámek *et al* for a survey [AMM18].

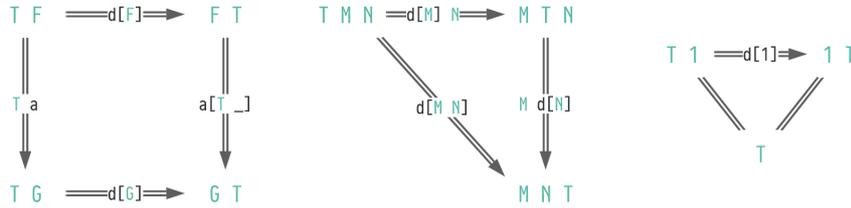
[SP82] **Smyth, Michael B; Plotkin, Gordon D** *The category-theoretic solution of recursive domain equations* (1982) SIAM Journal on Computing

[AMM18] **Adámek, Jiří; Milius, Stefan; Moss, Lawrence S** *Fixed points of functors* (2018) Journal of Logical and Algebraic Methods in Programming

(2.16). Definition of a 'monad map' for over lists for an applicative functor $M : \text{Type} \rightarrow \text{Type}$ and $A B : \text{Type}$.

(2.17). Inductive definition of binary trees and a definition of `mmap` to compare with (2.16).

$d[M][X] : T (M X) \rightarrow M (T X)$. In addition to being natural, d must obey the traversal laws given in (2.19) [JR12 Definition 3.3].



Given a traversable functor T and a monad M , we can recover $mmap : (A \rightarrow M B) \rightarrow T A \rightarrow M (T B)$ as $mmap f t := d[M][B] (T f t)$.

2.3.2. Functors with coordinates

Bird *et al* [BGM+13] prove that (in the category of sets) the traversable functors are equivalent to a class of functors called finitary containers. Their theorem states that there is a type $Shape\ T\ n : Type$ ¹⁶ for each traversable T and $n : \mathbb{N}$ such that that each $t : T\ X$ is isomorphic to an object called a **finitary container** on $Shape\ T$ shown in (2.20).

```
T X ≅
  (length  : N)
  × (shape  : Shape T length)
  × (children : Vec length X)
```

map and $traverse$ may be defined for the finitary container as map and $traverse$ over the $children$ vector. Since $t : T\ X$ has $t.length$ child elements, the children of t can be indexed by the numbers $\{k : \mathbb{N} \mid k < length\}$. We can then define operations to get and set individual elements according to this index k .

Usually, however, this numerical indexing of the children of $t : T\ X$ loses the semantics of the datatype. As an example; consider the case of a binary tree $Tree$ in (2.21). A tree $t : Tree\ X$ with n branch components will have length n and a corresponding $children : Vec\ n\ X$, but indexing via numerical indices $\{k \mid k < n\}$ loses information about where the particular child $x : X$ can be found in the tree.

```
TreeBase A X ::=
  | leaf  : TreeBase X
  | branch : TreeBase X → A → TreeBase X → TreeBase X

Tree A := Fix (TreeBase A)
```

Now I will introduce a new way of indexing the members of children for the purpose of reasoning about inductive datatypes. This idea has been used and noted before many times, the main one being *paths* in universal algebra [BN98 Dfn. 3.1.3]. However, I have not seen an explicit account of this idea in the general setting of traversable functors and later to general inductive datatypes (Section 2.3.3).

Definition 2.22 (coordinates): A traversable functor T has coordinates when equipped with a type $C : Type$ and a function $coords[n] : Shape\ T\ n \rightarrow Vec\ n\ C$. The $coords$ function amounts to a labelling of the n children of a particular shape with members of C .

Often when using traversals, working with the children list $Vec\ (length\ t)\ X$ for each shape of T can become unwieldy, so it is convenient to instead explicitly provide a pair of functions get and set (2.23) for manipulating particular children of a given $t : T\ X$.

[JR12] **Jaskelioff, Mauro; Rypacek, Ondrej** *An Investigation of the Laws of Traversals* (2012) Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia

(2.19). Commutative diagrams for the traversal laws. The leftmost diagram must hold for any natural function $a : F \Rightarrow G$.

[BGM+13] **Bird, Richard; Gibbons, Jeremy; Mehner, Stefan; et al.** *Understanding idiomatic traversals backwards and forwards* (2013) Proceedings of the 2013 ACM SIGPLAN symposium on Haskell

¹⁶ An explicit definition of $Shape\ T\ n$ is the pullback of $children[1] : T\ Unit \rightarrow List\ Unit$ and $!n : Unit \rightarrow List\ Unit$, the list with n elements.

(2.20). A finitary container is a count n , a **shape** $s : Shape\ T\ length$ and a vector $children : Vec\ length\ X$ is the type of lists in X with length $length$.

(2.21). Definition of binary trees using a base functor. Compare with the definition (2.17).

[BN98] **Baader, Franz; Nipkow, Tobias** *Term rewriting and all that* (1998) **publisher** Cambridge University Press

```

get : C → T X → Option X
set : C → T X → X → T X

get c t = if ∃ i, (coords t)[i] = c
  then some t.children[i]
  else none

set c t x = if ∃ i, (coords t)[i] = c
  then Vec.set i t.children x
  else t

```

(2.23). Getter and setter signatures and equations. Here $l[i]$ is the i th member of $l : \text{List } X$ and $\text{Vec.set } i v x$ replaces the i th member of the vector $v : \text{Vec } n X$ with $x : X$.

C is not unique, and in general should be chosen to have some semantic value for thinking about the structure of T . Here are some examples of functors with coordinates:

- List has coordinates \mathbb{N} . $\text{coords } l$ for $l : \text{List } X$ returns a list $[0, \dots, l.\text{length} - 1]$. $\text{get } i l$ is $\text{some } l[i]$ and $\text{set } i l x$ returns a new list with the i th element set to be x .
- $\text{Vec } n$, lists of length n , has coordinates $\{k : \mathbb{N} \mid k < n\}$ with the same methods as for List above.
- Option has coordinates Unit . $\text{coords } (\text{some } x) := [()]$ and $\text{coords } \text{none} := []$. $\text{get } _ o := o$ and set replaces the value of the option.
- Binary trees have coordinates $\text{List } D$ as shown in (2.24).

```
D ::= | left | right
```

```

coords
: Tree X      → List (List D)
| leaf        ↦ []
| branch l x r ↦
  [ ..[[left, ..c] for c in coords l]
  , []
  , ..[[right, ..c] for c in coords r]
  ]

```

```

get : List (List Bool) → Tree X      → Option X
| _                ↦ leaf          ↦ none
| []               ↦ branch l x r ↦ some x
| [left, ..c]     ↦ branch l x r ↦ get c l
| [right, ..c]    ↦ branch l x r ↦ get c r

```

(2.24). Defining the List Bool coordinates for binary trees. Here the left/right items in the $C = \text{List } D$ can be interpreted as a sequence of "take the left/right branch" instructions. set is omitted for brevity but follows a similar pattern to get .

2.3.3. Coordinates on initial algebras of traversable functors

Given a functor F with coordinates C , we can induce coordinates on the free monad $\text{Free } F : \text{Type} \rightarrow \text{Type}$ of F . The free monad is defined concretely in (2.25).

```

Free F X ::=
| pure : X → Free F X
| make : F (Free F X) → Free F X

join : (Free F (Free F X)) → Free F X
| pure x ↦ pure x
| (make f) ↦ make (F join f)

```

(2.25). Definition of a free monad $\text{Free } F X$ and join for a functor $F : \text{Type} \rightarrow \text{Type}$ and $X : \text{Type}$.

We can write $\text{Free } F X$ as the fixpoint of $A \mapsto X + F A$ ¹⁷. $\text{Free } F$ has coordinates $\text{List } C$ with methods defined in (2.26).

¹⁷ As mentioned in Section 2.2.3, these fixpoints may not exist. However for the purposes of this thesis the F s of interest are always polynomial functors.

```

coords : Free F X → List (List C)
  | pure x   ↦ []
  | make f   ↦
    [ [c, ..a]
      for a in coords (get c f)
      for c in coords f]

get : List C → Free F X → Option X
  | []       ↦ pure x   ↦ some x
  | [c, ..a] ↦ make f   ↦ (get c f) >=> get a
  | _       ↦ _        ↦ none

set : List C → Free F X → X → Free F X
  | []       ↦ pure _   ↦ x ↦ pure x
  | [c, ..a] ↦ make f   ↦ x ↦ (set c f)
  | _       ↦ _        ↦ none

```

In a similar manner, `List C` can be used to reference particular subtrees of an inductive datatype `D` which is the fixpoint of a traversable functor `D = F D`. Let `F` have coordinates `C`. `D` here is not a functor, but we can similarly define `coords : D → List (List C)`, `get : List C → Option D` and `set : List C → D → D → D`.

The advantage of using coordinates over some other system such as `optics` [FGM+07] or other apparatus for working with datatypes [LP03] is that they are much simpler to reason about. A coordinate is just an address of a particular subtree. Another advantage is that the choice of `C` can convey some semantics on what the coordinate is referencing (for example, `C = left | right` in (2.24)), which can be lost in other ways of manipulating datastructures.

2.4. Metavariables

Now with a way of talking about logical foundations, we can resume from Section 2.1.2 and consider the problem of how to represent partially constructed terms and proofs given a foundation. This is the purpose of a **development calculus**: to take some logical system \mathcal{L} and produce some new system $D\mathcal{L}$ such that one can incrementally build terms and proofs in a way that provides feedback at intermediate points and ensures that various judgements hold for these intermediate terms. In Chapter 3, I will create a new development calculus for building human-like proofs, and in Appendix A this system will be connected to Lean. First we look at how Lean's current development calculus behaves. Since I will be using Lean 3 in this thesis and performing various operations over its expressions, I will follow the same general setup as is used in Lean 3. The design presented here was first developed by Spiwack [Spi11] first released in Coq 8.5. It was built to allow for a type-safe treatment of creating tactics with metavariables in a dependently-typed foundation.

2.4.1. Expressions and types

In this section I will introduce the expression foundation language that will be used for the remainder of the thesis. The system presented here is typical of expression structures found in DTT-based provers such as Lean 3 and Coq. I will not go into detail on induction schema and other advanced features because the work in this thesis is independent of them.

Definition 2.27 (expression): A Lean expression is a recursive datastructure `Expr` defined in (2.28).

(2.26). Definitions of the coordinate methods for `Free F` given `F` has coordinates `C`. Compare with the concrete binary tree definitions (2.24).

[FGM+07] **Foster, J Nathan; Greenwald, Michael B; Moore, Jonathan T; et al.** *Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem* (2007) ACM Transactions on Programming Languages and Systems (TOPLAS)

[LP03] **Lämmel, Ralf; Peyton Jones, Simon** *Scrap Your Boilerplate* (2003) Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003, Proceedings

[Spi11] **Spiwack, Arnaud** *Verified computing in homological algebra, a journey exploring the power and limits of dependent type theory* (2011) PhD thesis (INRIA)

```

ExprBase X ::=
| lambda : Binder → X → ExprBase X  -- function abstraction
| pi : Binder → X → ExprBase X      -- dependent function type
| var : Name → ExprBase X            -- variables
| const : Name → ExprBase X          -- constants
| app : X → X → ExprBase X           -- function application
| sort : Level → ExprBase X          -- type universe

```

```
Binder := (name : Name) × (type : Expr)
```

```
Context := List Binder
```

```
Expr := Fix ExprBase
```

In (2.28), `Level` can be thought of as expressions over some signature that evaluate to natural numbers. They are used to stratify Lean's types so that one can avoid Girard's paradox [Hur95]. `Name` is a type of easily distinguishable identifiers; in the case of Lean `Names` are lists of strings or numbers. I sugar `lambda x α b` as `λ (x : α), b`, `pi x α b` as `Π (x : α), b`, `app f a` as `f a` and omit `var` and `const` when it is clear what the appropriate constructor is.

Using `ExprBase`, define pure expressions `Expr := Fix ExprBase` as in Section 2.2.3. Note that it is important to distinguish between the meta-level type system introduced in Section 2.2 and the object-level type system where the 'types' are merely instances of `Expr`¹⁸. That is, `t : Expr` is a meta-level statement indicating that `t` is an expression, but `⊢ t : α` is an object-level judgement about expressions stating that `t` has the type `α`, where `α : Expr` and `⊢ α : sort`.

Definition 2.29 (variable binding): Variables may be **bound** by `λ` and `Π` expressions. For example, in `λ (x : α), t`, we say that the expression **binds** `x` in `t`. If `t` contains variables that are not bound, these are called **free** variables. Now, given a partial map `σ : Name → Expr` and a term `t : Expr`, we define a **substitution** `subst σ t : Expr` as in (2.30). This will be written as `σ t` for brevity.

```

subst σ : Expr → Expr
| var x   ↦ if x ∈ dom σ then σ x else x
| e       ↦ ExprBase (subst σ) e

```

I will denote substitutions as a list of `Name → Expr` pairs. For example, `{x ↦ t, y ↦ s}` where `x y : Name` are the variables which will be substituted for terms `t s : Expr` respectively.

Substitution can easily lead to badly-formed expressions if there are variable naming clashes. I need only note here that we can always perform a renaming of variables in a given expression to avoid clashes upon substitution. These clashes are usually avoided within prover implementations with the help of de-Bruijn indexing [deB72].

2.4.2. Assignable datatypes

Given an expression structure `Expr` and `t : Expr`, we can define a traversal over all of the immediate subexpressions of `t`.

```

child_traverse (M : Monad) (f : Context → Expr → M Expr)
: Context → Expr → M Expr
| Γ      ↦ (Expr.var n)      ↦ (Expr.var n)
| Γ      ↦ (Expr.app l r)    ↦
  pure (Expr.app) <*> f Γ l <*> f Γ r
| Γ      ↦ (Expr.lambda n α b) ↦
  pure (Expr.lambda n) <*> f Γ α <*> f [..Γ, (n:α)] b

```

The function `child_traverse` defined in (2.31) is different from a normal traversal of a datatype because the mapping function `f` is also passed a context `Γ` indicating the current variable context of the subexpression. Thus when exploring a `λ`-binder, `f` can take into account the modified context. This means that we can define context-aware

(2.28). Definition of a base functor for pure DTT expressions as used by Lean.

[Hur95] **Hurkens, Antonius J. C.** *A simplification of Girard's paradox* (1995) International Conference on Typed Lambda Calculi and Applications

¹⁸This distinction can always be deduced from syntax, but to give a subtle indication of this distinction, object-level type assignment statements such as `(x : α)` are annotated with a slightly smaller variant of the colon `:` as opposed to `:` which is used for meta-level statements.

(2.30). Definition of substitution on an expression. Here, `ExprBase (subst σ) e` is mapping each child expression of `e` with `subst σ`; see Section 2.2.3.

(2.31). Illustrative code for mapping the immediate subexpressions of an expression using `child_traverse`.

expression manipulating tools such as counting the number of free variables in an expression (`fv` in (2.32)).

```

instantiate : Name → Expr → Context → Expr → Expr
| x ↦ r ↦ Γ ↦ (Expr.var n) ↦ if (x = n) then r else Expr.var n
| x ↦ r ↦ Γ ↦ t           ↦ child_traverse 1 (instantiate x r) Γ t

fv : Context → Expr → Set Name
| Γ ↦ (Expr.var n) ↦ if n ∈ Γ then ∅ else {n}
| Γ ↦ t           ↦ child_traverse Set (fv) Γ t

```

The idea here is to generalise `child_traverse` to include any datatype that may involve expressions. Frequently when building systems for proving, one has to make custom datastructures. For example, one might wish to create a 'rewrite-rule' structure (2.33) for modelling equational reasoning (as will be done in Chapter 4).

```
RewriteRule := (lhs : Expr) × (rhs : Expr)
```

Definition 2.34 (telescope): Another example might be a **telescope** of binders `Δ : List Binder` a list of binders is defined as a telescope in `Γ : Context` when each successive binder is defined in the context of the binders before it. That is, `[]` is a telescope and `[(x:α), ..Δ]` is a telescope in `Γ` if `Δ` is a telescope in `[..Γ, (x:α)]` and `Γ ⊢ x : α`.

But now if we want to perform a variable instantiation or count the number of free variables present in `r : RewriteRule`, we have to write custom definitions to do this. The usual traversal functions from Section 2.3.1 are not adequate for telescopes, because we may need to take into account a binder structure. Traversing a telescope as a simple list of names and expressions will produce the wrong output for `fv`, because some of the variables are bound by previous binders in the context.

Definition 2.35 (assignable): To avoid having to write all of this boilerplate, let's make a typeclass `assignable` (2.36) on datatypes that we need to manipulate the expressions in. The `expr_traverse` method in (2.36) traverses over the child expressions of a datatype (e.g., the lhs and rhs of a `RewriteRule` or the type expressions in a telescope). `expr_traverse` also includes a `Context` object to enable traversal of child expressions which may be in a different context to the parent datatype.

```

class assignable (X : Type) :=
  (expr_traverse :
    (M : Monad) →
    (Context → Expr → M Expr) →
    Context → X → M X
  )

expr_traverse M f
: Context → RewriteRule → RewriteRule
| Γ ↦ (l, r) ↦ do
  l' ← f Γ l;
  r' ← f Γ r;
  pure ⟨l, r⟩

expr_traverse M f
: Context → Telescope → Telescope
| Γ ↦ [] ↦ pure []
| Γ ↦ [(x:α), ..Δ] ↦ do
  α' ← f Γ α;
  Δ' ← expr_traverse M f [..Γ, (x:α)] Δ;
  pure [(x:α'), ..Δ']

```

Now, provided `expr_traverse` is defined for `X`: `fv`, `instantiate` and other expression-manipulating operations such as those in (2.32) can be modified to use `expr_traverse` instead of `child_traverse`. This `assignable` regime becomes useful when using de-Brujin indices to represent bound variables [deB72] because the length of `Γ` can be used to determine the binder depth of the current expression. Examples of implementations of `assignable` and expression-manipulating operations that can make use of `assignable` can be found in [my Lean implementation of this concept](#)¹⁹.

(2.32). Some example implementations of expression manipulating tools with the `child_traverse` construct. The monad structure on `Set` is `pure := x ↦ {x}` and `join (s : Set Set X) := ∪ s` and `map f s := f[s]`. `fv` stands for 'free variables'.

(2.33). Simple `RewriteRule` representation defined as a pair of `Expr`s, representing `lhs = rhs`. This is to illustrate the concept of assignable datatypes.

(2.36). Say that a type `X` is **assignable** by equipping `X` with the given `expr_traverse` operation. Implementations of `expr_traverse` for `RewriteRule` (2.33) and telescopes are given as examples.

2.4.3. Lean's development calculus

¹⁹ <https://github.com/leanprover-community/mathlib/pull/5719>

In the Lean source code, there are constructors for `Expr` other than those in (2.30). Some are for convenience or efficiency reasons (such as Lean 3 macros), but others are part of the Lean development calculus. The main development calculus construction is `mvar` or a **metavariable**, sometimes also called a **existential variable** or **schematic variable**. An `mvar` `?m` acts as a 'hole' for an expression to be placed in later. There is no kernel machinery to guarantee that an expression containing a metavariable is correct; instead, they are used for the process of building expressions.

As an example, suppose that we needed to prove `P ∧ Q` for some propositions `P Q : Prop`. The metavariable-based approach to proving this would be to **declare** a new metavariable `?t : P ∧ Q`. Then, a prover constructs a proof term for `P ∧ Q` in two steps; declare two new metavariables `?t1 : P` and `?t2 : Q`; and then **assign** `?t` with the expression `and.make ?t1 ?t2` where `and.make : P → Q → P ∧ Q` is the constructor for `∧`. After this, `?t1` and `?t2` themselves are assigned with `p : P` and `q : Q`. In this way, the proof term can be built up slowly as `?t ↦ and.make ?t1 ?t2 ↦ and.make p ?t2 ↦ and.make p q`. This process is more convenient for building modular programs that construct proofs than requiring that a pure proof term be made all in one go because a partially constructed proof is represented as a proof term where certain subexpressions are metavariables.

Lean comes with a development calculus that uses metavariables. This section can be viewed as a more detailed version of the account originally given by de Moura *et al* [MAKR15 §3.2] with the additional details sourced from inspecting the [Lean source code](#). Lean's metavariable management system makes use of a stateful global 'metavariable context' with carefully formed rules governing valid assignments of metavariables. While all automated provers make use of some form of metavariables, this specific approach to managing them for use with tactics was first introduced in Spiwack's thesis [Spi11], where the tactic monad for Coq was augmented with a stateful global metavariable context.

[MAKR15] **de Moura, Leonardo; Avigad, Jeremy; Kong, Soonho; et al.** *Elaboration in Dependent Type Theory* (2015) CoRR

The implementation of Lean allows another `Expr` constructor for metavariables:

(2.37). Redefining `Expr` with metavariables using the base functor given in (2.28).

```
Expr ::=
| ExprBase Expr
| ?Name
```

Metavariables are 'expression holes' and are denoted as `?x` where `x : Name`. They are placeholders into which we promise to substitute a valid pure expression later. Similarly to `fv(t)` being the free variables in `t : Expr`, we can define `mv(t)` to be the set of metavariables present in `t`. However, we still need to be able to typecheck and reduce expressions involving metavariables and so we need to have some additional structure on the context.

The idea is that in addition to a local context `Γ`, expressions are inspected and created within the scope of a second context called the **metavariable context** `M : MvarContext`. The metavariable context is a dictionary `MvarContext := Name → MvarDecl` where each metavariable declaration `d : MvarDecl` has the following information:

- `identifier : Name` A unique identifier for the metavariable.
- `type : Expr` The type of the metavariable.
- `context : Context` The local context of the metavariable. This determines the set of local variables that the metavariable is allowed to depend on.
- `assignment : Option Expr` An optional assignment expression. If `assignment` is not `none`, we say that the metavariable is **assigned**.

The metavariable context can be used to typecheck an expression containing metavariables by assigning each occurrence `?x` with the type given by the corresponding declaration `M[x].type` in `M`. The `assignment` field of `MvarDecl` is used to perform instantiation. We can interpret `M` as a substitution.

As mentioned in Section 2.1.2, the purpose of the development calculus is to represent a partially constructed proof or term. The kernel does not need to check expressions in the development calculus (which here means expressions containing metavariables), so there is no need to ensure that an expression using metavariables is sound in the sense that declaring and assigning metavariables will be compatible with some set of inference rules such as those given in (2.4). However, in Appendix A.1, I will provide some inference rules for typing expressions containing metavariables to assist in showing that the system introduced in Chapter 3 is compatible with Lean.

2.4.4. Tactics

A partially constructed proof or term in Lean is represented as a `TacticState` object. For our purposes, this can be considered as holding the following data:

```
TacticState :=
  (result : Expr)
  × (mctx : MvarContext)
  × (goals : List Expr)
(2.38).

Tactic (A : Type) := TacticState → Option (TacticState × A)
```

The `result` field is the final expression that will be returned when the tactic completes. `goals` is a list of metavariables that are used to denote what the tactic state is currently 'focussing on'. Both `goals` and `result` are in the context of `mctx`.

`Tactic`s may perform actions such as modifying the `goals` or performing assignments of metavariables. In this way, a user may interactively build a proof object by issuing a stream of tactics.

2.5. Understandability and confidence

This section is a short survey of literature on what it means for a mathematical proof to be understandable. This is used in Chapter 6 to evaluate my software and to motivate the design of the software in Chapter 3 and Chapter 4.

2.5.1. Understandability of mathematics in a broader context

What does it mean for a proof to be understandable? An early answer to this question comes from the 19th century philosopher Spinoza. Spinoza [Spi87] supposes 'four levels' of a student's understanding of a given mathematical principle or rule, which are:

1. **mechanical**: The student has learnt a recipe to solve the problem, but no more than that.
2. **inductive**: The student has verified the correctness of the rule in a few concrete cases.
3. **rational**: The student comprehends a proof of the rule and so can see why it is true generally.
4. **intuitive**: The student is so familiar and immersed in the rule that they cannot comprehend it not being true.

For the purposes of this thesis I will restrict my attention to type 3 understanding. That is, how the student digests a proof of a general result. If the student is at level 4, and treats the result like a fish treats water, then there seems to be little an ITP system can offer other than perhaps forcing any surprising counterexamples to arise when the student attempts to formalise it.

Edwina Michener's *Understanding Understanding Mathematics* [Mic78] provides a wide ontology of methods for understanding mathematics. Michener (p. 373) proposes that "understanding is a complementary process to problem solving" and incorporates

[Spi87] **Spinoza, Benedict** *The chief works of Benedict de Spinoza* (1887) **publisher** Chiswick Press

[Mic78] **Michener, Edwina Risland** *Understanding understanding mathematics* (1978) Cognitive science

Spinoza's 4-level model. She also references Poincaré's thoughts on understanding [Poi14 p. 118], from which I will take an extended quote from the original:

What is understanding? Has the word the same meaning for everybody? Does understanding the demonstration of a theorem consist in examining each of the syllogisms of which it is composed and being convinced that it is correct and conforms to the rules of the game? ...

Yes, for some it is; when they have arrived at the conviction, they will say, I understand. But not for the majority... They want to know not only whether the syllogisms are correct, but why there are linked together in one order rather than in another. As long as they appear to them engendered by caprice, and not by an intelligence constantly conscious of the end to be attained, they do not think they have understood.

In a similar spirit; de Millo, Lipton and Perlis [MUP79] write referring directly to the nascent field of program verification (here referred to 'proofs of software')

Mathematical proofs increase our confidence in the truth of mathematical statements only after they have been subjected to the social mechanisms of the mathematical community. These same mechanisms doom the so-called proofs of software, the long formal verifications that correspond, not to the working mathematical proof, but to the imaginary logical structure that the mathematician conjures up to describe his feeling of belief. Verifications are not messages; a person who ran out into the hall to communicate his latest verification would rapidly find himself a social pariah. Verifications cannot really be read; a reader can flay himself through one of the shorter ones by dint of heroic effort, but that's not reading. Being unreadable and - literally - unspeakable, verifications cannot be internalized, transformed, generalized, used, connected to other disciplines, and eventually incorporated into a community consciousness. They cannot acquire credibility gradually, as a mathematical theorem does; one either believes them blindly, as a pure act of faith, or not at all.

Poincaré's concern is that a verified proof is not sufficient for understanding. De Millo *et al* question whether a verified proof is a proof at all! Even if a result has been technically proven, mathematicians care about the structure and ideas behind the proof itself. If this were not the case, then it would be difficult to explain why new proofs of known results are valued by mathematicians. I explore the question of what exactly they value in Chapter 6.

Many studies investigating mathematical understanding within an educational context exist, see the work of Sierpinska [Sie90, Sie94] for a summary. See also Pólya's manual on the same topic [Pól62].

2.5.2. Confidence

Another line of inquiry suggested by Poincaré's quote is distinguishing *confidence* in a proof from a proof being *understandable*. By confidence in a proof, I do not mean confidence in the result being true, but instead confidence in the given script actually being a valid proof of the result.

As an illustrative example, I will give my own impressions on some proofs of the [Jordan curve theorem](#) which states that any non-intersecting continuous loop in the 2D Euclidean plane has an interior region and an exterior region. Formal and informal proofs of this theorem are discussed by Hales [Halo7]. I am confident that the proof of the Jordan curve theorem formalised by Hales in the HOL Light proof assistant is correct although I can't claim to understand it in full. Contrast this with the diagrammatic proof sketch (Figure 2.39) given in Hales' paper (originating with Thomassen [Tho92]). This sketch is more understandable to me but I am less confident in it being a correct proof (e.g., maybe there is some curious fractal curve that causes the diagrammatic proofs to stop being

[Poi14] **Poincaré, Henri** *Science and method* (1914) **publisher** Amazon (out of copyright)

[MUP79] **de Millo, Richard A; Upton, Richard J; Perlis, Alan J** *Social processes and proofs of theorems and programs* (1979) Communications of the ACM

[Sie90] **Sierpinska, Anna** *Some remarks on understanding in mathematics* (1990) For the learning of mathematics

[Sie94] **Sierpinska, Anna** *Understanding in mathematics* (1994) **publisher** Psychology Press

[Pól62] **Pólya, George** *Mathematical Discovery* (1962) **publisher** John Wiley & Sons

obvious...). In the special case of the curve C being a polygon, the proof involves "walking along a simple polygonal arc (close to C but not intersecting C)" and Hales notes:

Nobody doubts the correctness of this argument. Every mathematician knows how to walk without running in to walls. Detailed figures indicating how to "walk along a simple polygonal arc" would be superfluous, if not downright insulting. Yet, it is quite another matter altogether to train a computer to run around a maze-like polygon without collisions...

These observations demonstrate how one's confidence in a mathematical result is not merely a formal affair, but includes ostensibly informal arguments of correctness. This corroborates the attitude taken by De Millo *et al* in Section 2.5.1. Additionally, as noted in Section 1.1, confidence in results also includes a social component: a mathematician will be more confident that a result is correct if that result is well established within the field.

There has also been some empirical work on the question of confidence in proofs. Inglis and Alcock [IA12] performed an empirical study on eye movements in undergrads vs postgrads. A set of undergraduates and post-graduate researchers were presented with a set of natural language proofs and then asked to judge the validity of these proofs. The main outcomes they suggest from their work are that mathematicians can disagree about the validity of even short proofs and that post-graduates read proofs in a different way to undergraduates: moving their focus back and forth more. This suggests that we might expect undergraduates and postgraduates to present different reasons for their confidence in the questions.

2.5.3. Understandability and confidence within automated theorem proving.

The concepts of understandability and confidence have also been studied empirically within the context of proof assistants. This will be picked up in Chapter 6.

Stenning *et al*. [SCO95] used the graphical *Hyperproof* software (also discussed in Section 5.1) to compare graphical and sentence-based representations in the teaching of logic. They found that both representations had similar transferability²⁰ and that the best teaching representation (in terms of test scores) was largely dependent on the individual differences between the students. This suggests that in looking for what it means for a proof to be understandable, we should not forget that people have different ways of thinking about proofs, and so there is not going to be a one-size-fits-all solution. It also suggests that providing multiple ways of conceptualising problems should help with understandability.

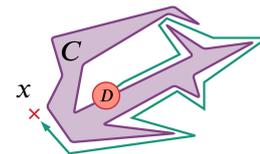
In Grebing's thesis [Gre19], a set of focus group studies are conducted to ask a set of users with a variety of experience-levels in Isabelle and KeY, to reflect on the user interfaces. One of her main findings was that due to the extensive levels of automation in the proving process, there can arise a 'gap' between the user's model of the proof state and the proof state created through the automation. Grebing then provides a bridge for this gap in the form of a proof scripting language and user interface for the KeY prover at a higher level of abstraction than the existing interface. Grebing also provides a review of other empirical studies conducted on the user interfaces of proof assistants [Gre19 §6.2.0].

2.6. Human-like reasoning

How should a prover work to produce human-like mathematical reasoning? The easiest answer is: however humans think it should reason!

The very earliest provers such as the Boyer-Moore theorem prover [BM73, BM90, BKM95] take this approach to some extent; the design is steered through a process of introspection on how the authors would prove theorems. Nevertheless, with their 'waterfall' architecture, the main purpose is to prove theorems automatically, rather than creating proofs that a

Figure 2.39. A cartoon illustrating a component of the proof of the Jordan curve theorem for polygons as described by Hales [Halo7]. Call the edge of the purple polygon C , then the claim that this cartoon illustrates is that given any disk D in red and for any point x not on C , we can 'walk along a simple polygonal arc' (here in green) to the disk D .



[Halo7] **Hales, Thomas C** *The Jordan curve theorem, formally and informally* (2007) The American Mathematical Monthly

[Tho92] **Thomassen, Carsten** *The Jordan-Schönflies theorem and the classification of surfaces* (1992) The American Mathematical Monthly

[IA12] **Inglis, Matthew; Alcock, Lara** *Expert and novice approaches to reading mathematical proofs* (2012) Journal for Research in Mathematics Education

[SCO95] **Stenning, Keith; Cox, Richard; Oberlander, Jon** *Contrasting the cognitive effects of graphical and sentential logic teaching: reasoning, representation and individual differences* (1995) Language and Cognitive Processes

²⁰ That is, do lessons learnt in one domain transfer to analogous problems in other domains? The psychological literature identifies this as a difficult problem in teaching.

[Gre19] **Grebing, Sarah Caecilia** *User Interaction in Deductive Interactive Program Verification* (2019) PhD thesis (Karlsruhe Institute of Technology)

[BM73] **Boyer, Robert S.; Moore, J. Strother** *Proving Theorems about LISP Functions* (1973) IJCAI

human could follow. Indeed Robinson's machine-like resolution method [BG01] was such a dominant approach that Bledsoe titled his paper *non-resolution theorem proving* [Ble81]. In this paper, Bledsoe sought to show another side of automated theorem proving through a review of alternative methods to resolution. A quote from this paper stands out for our current study:

It was in trying to prove a rather simple theorem in set theory by paramodulation and resolution, where the program was experiencing a great deal of difficulty, that we became convinced that we were on the wrong track. The addition of a few semantically oriented rewrite rules and subgoaling procedures made the proof of this theorem, as well as similar theorems in elementary set theory, very easy for the computer. Put simply: the computer was not doing what the human would do in proving this theorem. When we instructed it to proceed in a "human-like" way, it easily succeeded. Other researchers were having similar experiences.

This quote captures the concept of 'human-like' that I want to explore. Some piece of automation is 'human-like' when it doesn't get stuck in a way that a human would not.

Another early work on human-oriented reasoning is that of Nevins [Nev74]. Similar to this thesis, Nevins is motivated by the desire to make proofs more understandable to mathematicians. Some examples of prover automation that are designed to perform steps that a human would take are `grind` for PVS [SORS01] and the waterfall algorithm in ACL2 [KMM13].

All of the systems mentioned so far came very early in the history of computing, and had a miniscule proportion of the computing power available to us today. Today, the concern that a piece of automation may not find a solution in a human-like way or finds a circumlocutious route to a proof is less of a concern because computers are much more powerful. However I think that the resource constraints that these early pioneers faced provides some clarity on why building human-like reasoning systems matters. The designers of these early systems were forced to introspect carefully on how they themselves were able to prove certain theorems without needing to perform a large amount of compute, and then incorporated these human-inspired insights in to their designs.

My own journey into this field started with reading the work of Gowers and Ganesalingam (G&G) in their Robot prover [GG17]²¹. G&G's motivation was to find a formal system that better represented the way that a human mathematician would solve a mathematics problem, demonstrating this through the ability to generate realistic natural-language write-ups of these proofs. The system made use of a natural-deduction style hierarchical proof-state with structural sharing. The inference rules (which they refer to as 'moves') on these states and the order in which they were invoked were carefully chosen through an introspective process. The advantage of this approach is that the resulting proofs could be used to produce convincing natural language write-ups of the proofs. However, the system was not formalised and was limited to the domains hard-coded in to the system. The work in this thesis is a reimagining of this system within a formalised ITP system.

A different approach to exploring human-like reasoning is by modelling the process of mathematical discourse. Pease, Cornelli, Martin, *et al* [CMM+17, PLB+17] have investigated the use of graphical discourse models of mathematical reasoning. In this thesis, however I have restricted the scope to human-like methods for solving simple lemmas that can produce machine-checkable proofs.

Another key way in which humans reason is through the use of diagrams [Jam01] and alternative representations of mathematical proofs. A *prima facie* unintuitive result such as $1 + 2 + 3 + \dots + n = \frac{1}{2}n(n + 1)$ snaps together when presented with the appropriate representation in Figure 2.40. Jamnik's previous work explores how one can perform automated reasoning like this in the domain of diagrams Some recent work investigating and automating this process is the rep2rep project [RSS+20]. This is an important feature of general human-like reasoning, however in the name of scope management I will not explore representations further in this thesis.

[BM90] **Boyer, Robert S; Moore, J Strother** *A theorem prover for a computational logic* (1990) International Conference on Automated Deduction

[BKM95] **Boyer, Robert S; Kaufmann, Matt; Moore, J Strother** *The Boyer-Moore theorem prover and its interactive enhancement* (1995) Computers & Mathematics with Applications

[BG01] **Bachmair, Leo; Ganzinger, Harald** *Resolution theorem proving* (2001) Handbook of automated reasoning

[Ble81] **Bledsoe, Woodrow W** *Non-resolution theorem proving* (1981) Readings in Artificial Intelligence

[Nev74] **Nevins, Arthur J A** *human oriented logic for automatic theorem-proving* (1974) Journal of the ACM

[SORS01] **Shankar, Natarajan; Owre, Sam; Rushby, John M; et al.** *PVS prover guide* (2001) Computer Science Laboratory, SRI International, Menlo Park, CA

[KMM13] **Kaufmann, Matt; Manolios, Panagiotis; Moore, J Strother** *Computer-aided reasoning: ACL2 case studies* (2013) **publisher** Springer

[GG17] **Ganesalingam, Mohan; Gowers, W. T.** *A fully automatic theorem prover with human-style output* (2017) Journal of Automated Reasoning

²¹ A working fork of this can be found at <https://github.com/edaye/rs/robotone>.

[CMM+17] **Corneli, Joseph; Martin, Ursula; Murray-Rust, Dave; et al.** *Modelling the way mathematics is actually done* (2017) Proceedings of the 5th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design

[PLB+17] **Pease, Alison; Lawrence, John; Budzynska, Katarzyna; et al.** *Lakatos-style collaborative mathematics through dialectical, structured and abstract argumentation* (2017) Artificial Intelligence

2.6.1. Levels of abstraction

There have been many previous works which add higher-level abstraction layers atop an existing prover with the aim of making a prover that is more human-like.

Archer *et al.* developed the *TAME* system for the PVS prover [AH97]. Although they were focussed on proving facts about software rather than mathematics, the goals are similar: they wish to create software that produces proofs which are natural to humans. *TAME* makes use of a higher abstraction level. However, it is only applied to reasoning about **timed automata** and doesn't include a user study.

As part of the `auto2` prover tactic for Isabelle, Zhan [Zha16] developed a high-level proof script syntax to guide the automation of `auto2`. A script takes the form of asserting several intermediate facts for the prover to prove before proving the main goal. This script is used to steer the `auto2` prover towards proving the result. This contrasts with tactic-based proof and structural scripts (e.g. Isar [Wen99]) which are instead instructions for chaining together tactics. With the `auto2` style script, it is possible to omit a lot of the detail that would be required by tactic-based scripts, since steps and intermediate goals that are easy for the automation to solve can be omitted entirely. A positive of this approach is that by being implemented within the Isabelle theorem prover, the results of `auto2` are checked by a kernel. However it is not a design goal of `auto2` to produce proofs that a human can read.

2.6.2. Proof planning

Proof planning originated with Bundy [Bun88, Bun98] and is the application of performing a proof with respect to a high-level plan (e.g., I am going to perform induction then simplify terms) that is generated before low-level operations commence (performing induction, running simplification algorithms). The approach follows the general field of AI planning.

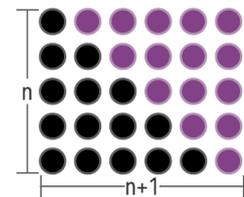
AI planning in its most general conception [KKY95] is the process of searching a graph \mathbb{G} using plan-space rather than by searching it directly. In a typical planning system, each point in plan-space is a DAG²² of objects called **ground operators** or **methods**, each of which has a mapping to paths in \mathbb{G} . Each ground operator is equipped with predicates on the vertices of \mathbb{G} called **pre/post-conditions**. Various AI planning methods such as GRAPHPLAN [BF97] can be employed to discover a partial ordering of these methods, which can then be used to construct a path in \mathbb{G} . This procedure applied to the problem of finding proofs is proof planning. The main issue with proof planning [Bun02] is that it is difficult to identify sets of conditions and methods that do not cause the plan space to be too large or disconnected. However, in this thesis we are not trying to construct plans for entire proofs, but just to model the thought processes of humans when solving simple equalities. A comparison of the various proof planners is provided by Dennis, Jamnik and Pollet [DJP06].

Proof planning in the domain of finding equalities frequently involves a technique called **rippling** [BSV+93, BBH105], in which an expression is annotated with additional structure determined by the differences between the two sides of the equation that directs the rewriting process. The rippling algorithm captures some human intuitions about which parts of a rewriting expression are salient. In the system for equational rewriting I introduce in Chapter 4, I avoid using rippling because the techniques are tied to performing induction.

Another technique associated with proof planning is the concept of **proof critics** [Ire92]. Proof critics are programs which take advantage of the information from a failed proof plan to construct a new, amended proof plan. An interactive version of proof critics has also been developed [LJR99]. In the work in Chapter 3, this concept of revising a proof based on a failure is used.

Another general AI system that will be relevant to this thesis is **hierarchical task networks** [MS99, Tat77] which are used to drive the behaviour of artificial agents such as

Figure 2.40. A visual representation of summing the first n integers with counters. The lower black triangle's rows comprise 1, 2, 3, 4, 5 from which a human can quickly see $\frac{1}{2}n(n+1)$.



[Jam01] **Jamnik, Mateja** *Mathematical Reasoning with Diagrams: From Intuition to Automation* (2001) **publisher** CSLI Press

[AH97] **Archer, Myla; Heitmeyer, Constance** *Human-style theorem proving using PVS* (1997) International Conference on Theorem Proving in Higher Order Logics

[Zha16] **Zhan, Bohua** *AUTO2, a saturation-based heuristic prover for higher-order logic* (2016) International Conference on Interactive Theorem Proving

[Bun88] **Bundy, Alan** *The use of explicit plans to guide inductive proofs* (1988) International conference on automated deduction

[Bun98] **Bundy, Alan** *Proof Planning* (1998) **publisher** University of Edinburgh, Department of Artificial Intelligence

[KKY95] **Kambhampati, Subbarao; Knoblock, Craig A; Yang, Qiang** *Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning* (1995) Artificial Intelligence

²² **Directed Acyclic Graph**

[BSV+93] **Bundy, Alan; Stevens, Andrew; Van Harmelen, Frank; et al.** *Rippling: A heuristic for guiding inductive proofs* (1993) Artificial Intelligence

[BBH105] **Bundy, Alan; Basin, David; Hutter, Dieter; et al.** *Rippling: meta-level guidance for mathematical reasoning* (2005) **publisher** Cambridge University Press

the ICARUS architecture [LCTo8]. In a hierarchical task network, tasks are recursively refined into subtasks, which are then used to find fine-grained methods for achieving the original tasks, eventually bottoming out in atomic actions such as actuating a motor. HTNs naturally lend themselves to human-like reasoning, and I will make use of these in designing a hierarchical algorithm for performing equational reasoning.

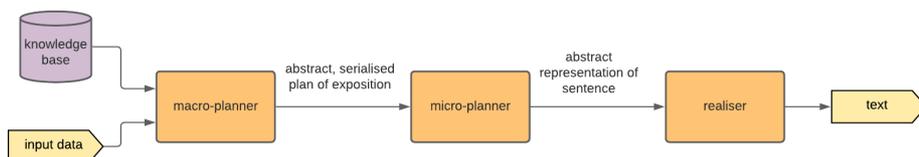
2.7. Natural language for formal mathematics

In this section I will survey the background and related work on using natural language to generate proofs. The material in this chapter will be used in Section 3.6 and Chapter 6.

2.7.1. Natural language generation in a wider context

Data-to-text natural language generation (NLG) is a subfield of natural language processing (NLP) that focusses on the problem of computing intelligible natural language discourses and text from some non-textual object (without a human in the loop!). An example is producing an English description of the local weather forecast from meteorological data. NLG techniques can range from simple 'canned text' and 'mail-merge' applications right up to systems with aspirations of generality such as modern voice recognition in smartphones.

There are a wide variety of architectures available for modern NLG [GK18], however they usually carry a modular structure, with a backbone [RD00] being split in to three pipeline stages as shown in Figure 2.41.



- **Macro-planner** or **discourse planner**: dictates how to structure the general flow of the text, that is, serialising the input data. These often take the form of 'expert systems' with a large amount of domain specific knowledge encoded.
- **Micro-planner**: determines how the stream of information from the macro-planner should be converted into individual sentences, how sentences should be structured and determining how the argument should 'flow'.
- **Realiser**: produces the final text from the abstracted output of the micro-planner, for example, applying punctuation rules and choosing the correct conjugations.

These choices of stages are mainly motivated through a desire to reuse code and to separate concerns (a realiser does not need to know the subject of the text it is correcting the punctuation from). I make use of this architecture in Section 3.6.

An alternative approach to the one outlined above is to use statistical methods for natural language generation. The advent of scalable machine learning (ML) and neural networks (NNs) of the 2010s has gained dominance in many NLG tasks such as translation and scene description. The system developed for this work in Section 3.6 is purely classical, with no machine learning component. In the context of producing simple write-ups of proofs, there will likely be some gains from including ML, but it is not clear that a statistical approach to NLG is going to assist in building understandable descriptions of proofs, because it is difficult to formally confirm that the resulting text generated by a black-box NLG component is going to accurately reflect the input.

2.7.2. Natural language generation for mathematics

[Ire92] **Ireland, Andrew** *The use of planning critics in mechanizing inductive proofs* (1992) International Conference on Logic for Programming Artificial Intelligence and Reasoning

[MS99] **Melis, Erica; Siekmann, Jörg** *Knowledge-based proof planning* (1999) Artificial Intelligence

[Tat77] **Tate, Austin** *Generating project networks* (1977) Proceedings of the 5th International Joint Conference on Artificial Intelligence.

[GK18] **Gatt, Albert; Krahmer, Emiel** *Survey of the state of the art in natural language generation: Core tasks, applications and evaluation* (2018) Journal of Artificial Intelligence Research

Figure 2.41. Outline of a common architecture for general NLG systems.

[RD00] **Reiter, Ehud; Dale, Robert** *Building natural language generation systems* (2000) **publisher** Cambridge University Press

The first modern study of the linguistics of natural language mathematics is the work of Ranta [Ran94, Ran95] concerning the translation between dependent type theory and natural language and I will use some of his insights in Section 3.6. Ganesalingam's thesis [Gan10] is an excellent reference for understanding the linguistics of mathematics in general, however it is more concerned with natural language input.

There have been numerous previous attempts at creating natural language output from a theorem prover: Felty-Miller [FM87], Holland-Minkley *et al* within the NuPrl prover [HBC99], and also in Theorema [BCJ+06]. A particularly advanced NLG for provers was Proverb [HF97] for the Ω mega theorem prover [BCF+97], this system's architecture uses the pipeline in Figure 2.41 and takes as input a proof term generated by the Ω mega toolchain and outputs a natural language sentence. An issue with these generation tools is that their text will often produce text that does not appear natural at the macro-level. That is, the general structure of the argument will be different to what would be found in a mathematical textbook. G&G illustrate some examples of this in their paper [GG17 §2].

The process of synthesising natural language is difficult in the general case. But as G&G [GG17] note, the language found in mathematical proofs is much more restricted than a general English text. At its most basic, a natural language proof is little more than a string of facts from the assumptions to the conclusion. There is no need for time-sensitive tenses or other complexities that arise in general text. Proofs are written this way because mathematical proofs are written to be *checked* by a human and so a uniformity of prose is used that minimises the chance of 'bugs' creeping in. This, combined with a development calculus designed to encourage human-like proof steps, makes the problem of creating mathematical natural language write-ups much more tenable. I will refer to these non-machine-learning approaches as 'classical' NLG.

A related problem worth mentioning here is the reverse process of NLG: parsing formal proofs and theorem statements from a natural language text. The two problems are interlinked in that they are both operating on the same grammar and semantics, but parsing raises a distinct set of problems to NLG, particularly around ambiguity [Gan10 ch. 2]. Within mathematical parsing there are two approaches. The first approach is **controlled natural language** [Kuh14] as practiced by ForTheL [Paso7] and Naproche/SAD [CFK+09]. Here, a grammar is specified to parse text that is designed to look as close to a natural language version of the text as possible. The other approach (which I will not make use of in this thesis) is in using machine learning techniques, for example the work on parsing natural mathematical texts is in the work of Stathopoulos *et al* [ST16, SBRT18].

In Section 3.6 I will make use of some ideas from natural language parsing, particularly the concept called *notion* by ForTheL and *non-extensional type* by Ganesalingam. A non-extensional type is a noun-phrase such as "element of a topological space" or "number" which is assigned to expressions, these types are not used by the underlying logical foundation but are used to parse mathematical text. To see why this is needed consider the syntax $\overline{x} \ y$. This is parsed to an expression differently depending on the types of \overline{x} and y (e.g., if \overline{x} is a function vs. an element of a group). Non-extensional types allow this parse to be disambiguated even if the underlying foundational language does not have a concept of a type.

2.8. Chapter summary

In this chapter I have provided the necessary background information and prior work needed to frame the rest of the thesis. I have explained the general design of proof assistants (Section 2.1). I have described a meta-level pseudolanguage for constructing algorithms (Section 2.2) and provided some gadgets for working with inductive types within it (Section 2.3). I have also presented the philosophy and social aspects of understandability in mathematics (Section 2.5); human-like automated reasoning (Section 2.6); and natural language generation of mathematical text (Section 2.7).

[Ran94] **Ranta, Aarne** *Syntactic categories in the language of mathematics* (1994) International Workshop on Types for Proofs and Programs

[Ran95] **Ranta, Aarne** *Context-relative syntactic categories and the formalization of mathematical text* (1995) International Workshop on Types for Proofs and Programs

[Gan10] **Ganesalingam, Mohan** *The language of mathematics* (2010) PhD thesis (University of Cambridge)

[FM87] **Felty, Amy; Miller, Dale** *Proof explanation and revision* (1987) Technical Report

[HBC99] **Holland-Minkley, Amanda M; Barzilay, Regina; Constable, Robert L** *Verbalization of High-Level Formal Proofs*. (1999) AAAI/IAAI

[BCJ+06] **Buchberger, Bruno; Crăciun, Adrian; Jebelean, Tudor; et al.** *Theorema: Towards computer-aided mathematical theory exploration* (2006) Journal of Applied Logic

[HF97] **Huang, Xiaorong; Fiedler, Armin** *Proof Verbalization as an Application of NLG* (1997) International Joint Conference on Artificial Intelligence

[BCF+97] **Benzmüller, Christoph; Cheikhrouhou, Lassaad; Fehrer, Detlef; et al.** *Omega: Towards a Mathematical Assistant* (1997) Automated Deduction - CADE-14

[Kuh14] **Kuhn, Tobias** *A survey and classification of controlled natural languages* (2014) Computational linguistics

[Paso7] **Paskevich, Andrei** *The syntax and semantics of the ForTheL language* (2007) <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.211.8865&rep=rep1&type=pdf>

[CFK+09] **Cramer, Marcos; Fisseni, Bernhard; Koepke, Peter; et al.** *The Naproche Project: Controlled Natural Language Proof Checking of Mathematical Texts* (2009) Controlled Natural Language, Workshop on Controlled Natural Language

[ST16] **Stathopoulos, Yiannos A; Teufel, Simone** *Mathematical information retrieval based on type embeddings and query expansion* (2016) COLING 2016

[SBRT18] **Stathopoulos, Yiannos; Baker, Simon; Rei, Marek; et al.** *Variable Typing: Assigning Meaning to Variables in Mathematical Text* (2018) NAACL-HLT 2018

Chapter 3

A development calculus

Now that we have reviewed the requisite background material, I can define the moving parts of a human-like theorem prover. The driving principle is to find ways of representing proofs at the same level of detail that a human mathematician would use to communicate to colleagues.

The contributions of this chapter are:

- The `Box` datastructure, a development calculus (Section 3.3) designed to better capture how humans reason about proofs while also being formally sound.
- A set of inference rules on `Box` which preserve this soundness (Section 3.5).
- A natural language write-up component converting proof objects created with this layer to an interactive piece of text (Section 3.6).
- In the supplementary Appendix A, an 'escape hatch' from the `Box` datastructure to a metavariable-oriented goal state system as used by Lean (Section 3.4.4, Appendix A). This enables compatibility between `Box`-style proofs and existing automation and verification within Lean.

HumanProof integrates with an existing proof assistant (in this case Lean). By plugging in to an existing prover, it is possible to gain leverage by utilising the already developed infrastructure for that prover such as parsers, tactics and automation. Using an existing prover also means that the verification of proofs can be outsourced to the prover's kernel.

The first research question of Section 1.2 was to investigate what it means for a proof to be human-like. I provided a review to answer this question in Section 2.6. Humans think differently to each other, and I do not wish to state that there is a 'right' way to perform mathematics. However, I argue that there are certain ways in which the current methods for performing ITP should be closer to the general cluster of ways in which humans talk about and solve problems.

In this chapter I investigate some ways in which the inference rules that provers use could be made more human-like, and then introduce a new proving abstraction layer, HumanProof, written in the Lean 3 theorem prover, implementing these ideas. Later, in Chapter 6, I gather thoughts and ratings from real mathematicians about the extent to which the developed system achieves these goals.

In Section 3.1, I first present an example proof produced by a human to highlight the key features of 'human-like' reasoning that I wish to emulate. Then in Section 3.2 I give an overview of the resulting designs and underline the primary design decisions and the evidence that drives them. In Section 3.3 I provide the details and theory of how the system works through defining the key `Box` structure and tactics on `Box`s. The theory behind creating valid proof terms from `Box`s is presented in Section 3.4 as well as how to run standard tactics within `Box`s (Section 3.4.4). This theoretical basis will then be used to define the human-like tactics in Section 3.5. Then, I will detail the natural language generation pipeline for HumanProof in Section 3.6.

3.1. Motivation

Building on the background where I explored the literature on the definition of 'human-like' (Section 2.6) and 'understandable' (Section 2.5.1) proofs, my goal in this section is find some specific improvements to the way in which computer aided mathematics is done. I use these improvements to motivate the design choices of the HumanProof system.

3.1.1. The need for human-like systems

In Section 1.1, I noted that non-specialist mathematicians have yet to widely accept proof assistants despite the adoption of other tools such as computer algebra systems. Section 1.1 presented three problems that mathematicians have with theorem provers: differing attitudes on correctness, a high learning cost to learning to use ITP and a low resulting reward -- learning the truth of something that they 'knew' was true anyway. One way in which to improve this situation is to reduce the cost of learning to use proof assistants through making the way in which they process proofs more similar to how a human would process proofs, making the proofs more closely match what the mathematician already knows. Making a prover which mimics a human's thought process also helps overcome the problem of differing attitudes of correctness.

Requiring a human-like approach to reasoning means that many automated reasoning methods such as SMT-solvers and resolution (see Section 2.6) must be ruled out. In these machine-oriented methods, the original statement of the proposition to be proved is first reduced to a normal form and mechanically manipulated with a small set of inference rules. The resulting proof is scarcely recognisable to a mathematician as a proof of the proposition, even if it is accepted by the kernel of a proof assistant. As discussed in Section 1.1, Section 2.5 and as will be confirmed in Chapter 6, mathematicians do not care just about a certificate that a statement is correct but also about the way in which the statement is correct.

Given some new way of creating proofs; how can we determine whether these created proofs are more 'human-like' than some other system? The way I propose here is to require that the program be able to imitate the reasoning of humans at least well enough to produce convincing natural language write-ups of the proofs that it generates, and then to test how convincing these write-ups are through asking mathematicians. This approach is shared by the previous work of Gowers and Ganesalingam [GG17]²³, where they use a similar framework to the HumanProof system presented in this thesis to produce natural language write-ups of proofs for some lemmas in the domain of metric space topology. The work presented in this thesis builds significantly on the work of G&G.

3.1.2. Modelling human-like reasoning

One of the key insights of Gowers and Ganesalingam is that humans reason with a different 'basis' of methods than the logical operations and tactics that are provided to the user of an ITP. For example, a hypothesis such as a function $f : X \rightarrow Y$ being continuous expands to a formula (3.39) with interlaced quantifiers.

$$\forall \varepsilon > 0, \forall x \in X, \exists \delta > 0, \forall y \in X, \mathbf{d}(x, y) < \delta \Rightarrow \mathbf{d}(f(x), f(y)) < \varepsilon$$

However in a mathematical text, if one needs to prove $\mathbf{d}(f(x), f(y)) < \varepsilon$, the hypothesis that f is continuous will be applied in one go. That is, a step involving (3.39) would be written as "Since f is continuous, there exists a $\delta > 0$ such that $\mathbf{d}(f(x), f(y)) < \varepsilon$ whenever $\mathbf{d}(x, y) < \delta$ ". Whereas in an ITP this process will need to be separated in to several steps: first show $x \in X$, then obtain δ , then show $\mathbf{d}(x, y) < \delta$.

Another example with the opposite problem is the automated tactics such as the tableaux prover `blast` [Pau99]. The issue with tactics is that their process is opaque and leaves little explanation for why they succeed or fail. They may also step over multiple stages that a human would rather see spelled out in full. The most common occurrence of this is in definition expansion; two terms may be identical modulo definition expansion but a proof found in a textbook will often take the time to point out when such an expansion takes place.

This points towards creating a new set of inference rules for constructing proofs that are better suited for creating proofs by corresponding better to a particular reasoning step as might be used by a human mathematician.

[GG17] **Ganesalingam, Mohan; Gowers, W. T.** *A fully automatic theorem prover with human-style output* (2017) *Journal of Automated Reasoning*

²³ Gowers and Ganesalingam is abbreviated G&G.

(3.1). Definition of a continuous function $f : X \rightarrow Y$ for metric spaces X, Y . Here \mathbf{d} is the distance metric for X or Y .

[Pau99] **Paulson, Lawrence C** *A generic tableau prover and its integration with Isabelle* (1999) *Journal of Universal Computer Science*

3.1.3. Structural sharing

Structural sharing is defined as making use of the same substructure multiple times in a larger structure. For example, a tree with two branches being the same would be using structural sharing if the sub-branches used the same object in memory. Structural sharing of this form is used frequently in immutable datastructures for efficiency. However here I am interested in whether structural sharing has any applications in human-like reasoning.

When humans reason about mathematical proofs, they often flip between forwards reasoning and backwards reasoning²⁴. The goal-centric proof state used by ITPs can make this kind of reasoning difficult. In the most simple example, suppose that the goal is $P \wedge Q \vdash Q \wedge P$ ²⁵. One solution is to perform a split on the goal to produce $P \wedge Q \vdash Q$ and $P \wedge Q \vdash P$. However, performing a conjunction elimination on the $P \wedge Q$ hypothesis will then need to be performed on both of the new goals. This is avoided if the elimination is performed before splitting $P \wedge Q$. In this simplified example it is clear which order the forwards and backwards reasoning should be performed. But in more complex proofs, it may be difficult to see ahead how to proceed. A series of backwards reasoning steps may provide a clue as to how forwards reasoning should be applied. The usual way that this problem is solved is for the human to edit an earlier part of the proof script with the forwards reasoning step on discovering this. I reject this solution because it means that the resulting proof script no longer represents the reasoning process of the creator. The fact that the forwards reasoning step was motivated by the goal state at a later point is lost.

The need to share structure among objects in the name of efficiency has been studied at least as far back as Boyer and Moore [BM72]. However, the motivation behind introducing it here is purely for the purpose of creating human-like proofs.

The solution that I propose here is to use a different representation of the goal state that allows for structural sharing. This alteration puts the proof state calculus more in the camp of OLEG [McBoo], and the G&G prover. The details of the implementation of structural sharing are presented later in Section 3.5.4.

Structural sharing can also be used to implement backtracking and counterfactuals. For example, suppose that we need to prove $A \vdash P \vee Q$, one could apply the \vee -left-introduction rule $P \Rightarrow P \vee Q$, but then one might need to backtrack later in the event that really the right-introduction rule $Q \Rightarrow P \vee Q$ should be used instead. Structural sharing lets us split a goal into two counterfactuals.

3.1.4. Verification

One of the key benefits of proof assistants is that they can rigorously check whether a proof is correct. This distinguishes the HumanProof project from the prior work of G&G, where no formal proof checking was present. While I have argued in Section 2.5 (and will later be suggested from the results of my user study in Section 6.6) that this guarantee of correctness is less important for attracting working mathematicians, there need not be a conflict between having a prover which is easy for non-specialists to understand *and* which is formally verified.

3.1.5. What about proof planning?

Proof planning is the process of creating proofs using abstract proof methods that are assembled with the use of classical AI planning algorithms²⁶. The concept of proof planning was first introduced by Bundy [Bun88]. A review of proof planning is given in Section 2.6.2. The advantage of proof planning is that it represents the way in which a problem will be solved at a much more abstract level, more like human mathematicians.

The primary issue with proof planning is that there is a sharp learning curve. In order to get started with proof plans, one must learn a great deal of terminology and a new way of thinking about formalised mathematics. The user has to familiarise themselves with the

²⁴ Broadly speaking, forwards reasoning is any mode of modifying the goal state that acts only on the hypotheses of the proof state. Whereas backwards reasoning modifies the goals.

²⁵ That is, given the hypothesis $P \wedge Q$, prove $Q \wedge P$ where P and Q are propositions and \wedge is the logical-and operation.

[BM72] **Boyer, R. S.; Moore, J. S.** *The sharing structure in theorem-proving programs* (1972) Machine intelligence

[McBoo] **McBride, Conor** *Dependently typed functional programs and their proofs* (2000) PhD thesis (University of Edinburgh)

[RN10] **Russell, Stuart J.; Norvig, Peter** *Artificial Intelligence - A Modern Approach* (2010) publisher Pearson Education

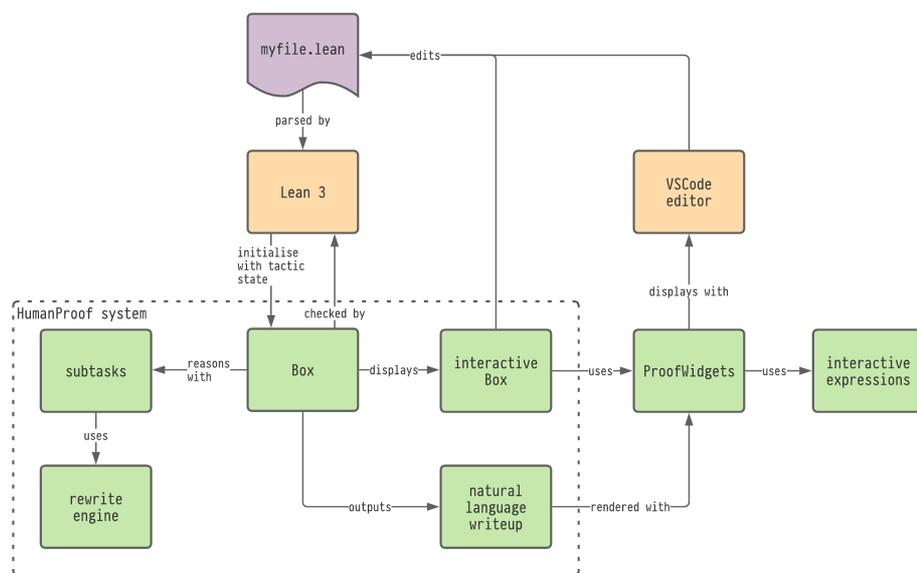
²⁶ An introduction to classical AI planning can be found in Russel and Norvig [RN10 Pt.III].

way in which *proof methods* are used to construct *proof plans* and how to diagnose malformed plans for their particular problems. Bundy presents his own critique of proof planning [Bun02] which goes in to more detail on this point.

The study of proof planning has fallen out of favour for the 21st century so far, possibly in relation to the rise of practical SMT solvers such as **E prover** [SCV19] and **Z3 prover** [MBo8] and their incorporation in to ITP through the use of 'hammer' software like Isabelle's Sledgehammer [BN10]. I share a great deal of the ideals that directed proof planning and the equational reasoning system presented in Chapter 4 is inspired by it. I take a more practical stance; the additional abstractions that are placed atop the underlying tactic system should be transparent, in that they are understandable without needing to be familiar with proof planning and with easy 'escape hatches' back to the tactic world if needed. This design goal is similar to that of the X-Barnacle prover interface [LD97] (discussed later in Section 5.1), where a GUI is used to present an explorable representation of a proof plan.

3.2. Overview of the software

The software implementation of the work presented in this thesis is called 'HumanProof' and is implemented using the **Lean 3 prover**. The source code can be found at <https://github.com/edayers/lean-humanproof-thesis>. In this section I give a high-level overview of the system and some example screenshots. A general overview of the system and how it relates to the underlying Lean theorem prover is shown in Figure 3.2.



Given a theorem to prove, HumanProof is invoked by indicating a special `begin [hp]` script block in the proof document (see Figure 3.3). This initialises HumanProof's `Box` datastructure with the assumptions and goal proposition of the proof. The initial state of the prover is shown in the goal view of the development environment, called the *Info View* (the right panel of Figure 3.3). Using the ProofWidgets framework (developed in Chapter 5), this display of the state is interactive: the user can click at various points in the document to determine their next steps. Users can then manipulate this datastructure either through the use of interactive buttons or by typing commands in to the proof script in the editor. In the event of clicking the buttons, the commands are immediately added to the proof script sourcefile as if the user had typed it themselves (the left panel of Figure 3.3). In this way, the user can create proofs interactively whilst still preserving the plaintext proof document as the single-source-of-truth; this ensures that there is no hidden state in the interactive view that is needed for the Lean to reconstruct a proof of the statement. While the proof is being created, the system also produces a natural language write-up (labelled 'natural language writeup' in Figure 3.2) of the proof (Section 3.6) that is

[Bun88] **Bundy, Alan** *The use of explicit plans to guide inductive proofs* (1988) International conference on automated deduction

[Bun02] **Bundy, Alan** *A critique of proof planning* (2002) Computational Logic: Logic Programming and Beyond

[SCV19] **Schulz, Stephan; Cruanes, Simon; Vukmirović, Petar** *Faster, Higher, Stronger: E 2.3* (2019) Proc. of the 27th CADE, Natal, Brasil

[MBo8] **de Moura, Leonardo; Björner, Nikolaj** *Z3: An efficient SMT solver* (2008) International conference on Tools and Algorithms for the Construction and Analysis of Systems

[BN10] **Böhme, Sascha; Nipkow, Tobias** *Sledgehammer: judgement day* (2010) International Joint Conference on Automated Reasoning

[LD97] **Lowe, Helen; Duncan, David** *XBarnacle: Making Theorem Provers More Accessible* (1997) 14th International Conference on Automated Deduction

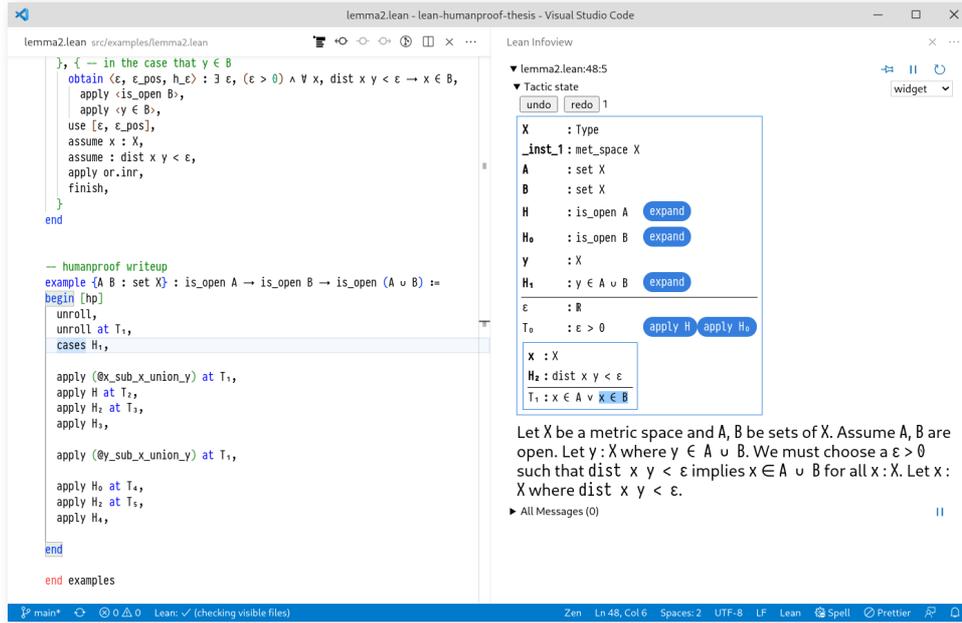
Figure 3.2. High-level overview of the main modules that comprise the HumanProof system and how these interface with Lean, ProofWidgets and the VSCode text editor. The green parts of the diagram are contributions given in this thesis. ProofWidgets (Chapter 5) was spun out from HumanProof for use as a general-purpose GUI system so that it could be used in other community projects (see Figure 5.18).

displayed alongside the proof state. As the proof progresses, users can see the incomplete natural language proof get longer too.

The system also comes equipped with a module for solving equalities using the 'subtasks algorithm' (Chapter 4); labelled 'subtasks' on Figure 3.2. The subtasks algorithm uses a hierarchical planning (see Section 2.6.2) system to produce an equality proof that is intended to match the way that a human would create the proof, as opposed to a more machine like approach such as E-matching [BN98 Ch. 10]. The output of this subsystem is a chain of equations that is inserted into the natural language writeup.

[BN98] **Baader, Franz; Nipkow, Tobias** *Term rewriting and all that* (1998) **publisher** Cambridge University Press

Figure 3.3. Screenshot of HumanProof in action on a test lemma. To the left is the code editor. The user invokes HumanProof with the `begin [hp]` command. The blue `apply H` button can be clicked to automatically insert more proofs.



3.3. The `Box` datastructure

At the heart of HumanProof is a development calculus using a datastructure called `Box`. The considerations from Section 3.1.3 led to the development of an 'on-tree' development calculus. Rather than storing a flat list of goals and a metavariable context alongside the result, the entire development state is stored in a recursive tree structure which I call a `Box`. The box tree, to be defined in Section 3.3.2, stores the proof state as an incomplete proof tree with on-tree metavariable declarations which is then presented to the user as a nested set of boxes.

3.3.1. An example of `Box` in action.

Before defining boxes in Section 3.3.2, let's look at a simple example. Boxes are visualised as a tree of natural-deduction-style goal states. Let's start with a minimal example to get a feel for the general progression of a proof with the `Box` architecture. Let's prove $P \vee Q \rightarrow Q \vee P$ using `Box`es. The initial box takes the form (3.4).

(3.4).

$$\boxed{?t : P \vee Q \rightarrow Q \vee P}$$

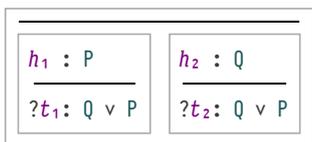
And we can read (3.4) as saying "we need to show $P \vee Q \rightarrow Q \vee P$ ". The `?t` is the name of the metavariable that the proof of this will be assigned to. The first action is to perform an `intro` step to get (3.5).

(3.5).

$$\boxed{\begin{array}{l} h : P \vee Q \\ \hline ?t : Q \vee P \end{array}}$$

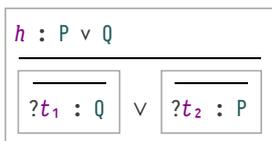
To be read as "Given $P \wedge Q$, we need to show $Q \vee P$ ". So far the structure is the same as would be observed on a flat goal list structure. The idea is that everything above a

horizontal line is a *hypothesis* (something that we have) and everything below is a *goal* (something we want). When all of the goals are solved, we should have a valid proof of the original goal. At this point, we would typically perform an elimination step on `h` (e.g., cases `h` in Lean) (3.6).



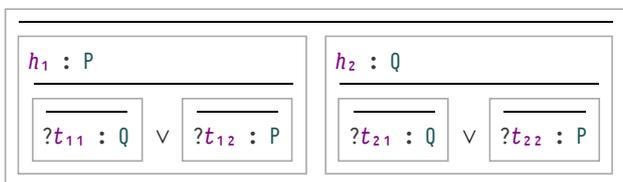
(3.6).

Here in (3.6) we can see *nested* boxes, each nested box below the horizontal line must be solved to solve the parent box. However, in the box architecture there is an additional step available; a branching on the goal (3.7).



(3.7).

If a pair of boxes appear with a `∨` between them, then *either* of the boxes can be solved to solve the parent box. And then we can eliminate `h` on the branched box:



(3.8).

Now at this point, we can directly match `h1` with `?t12` and `h2` with `?t21` to solve the box. Behind the scenes, the box is also producing a `result` proof term that can be checked by the proof assistant's kernel.

3.3.2. Definition of `Box`

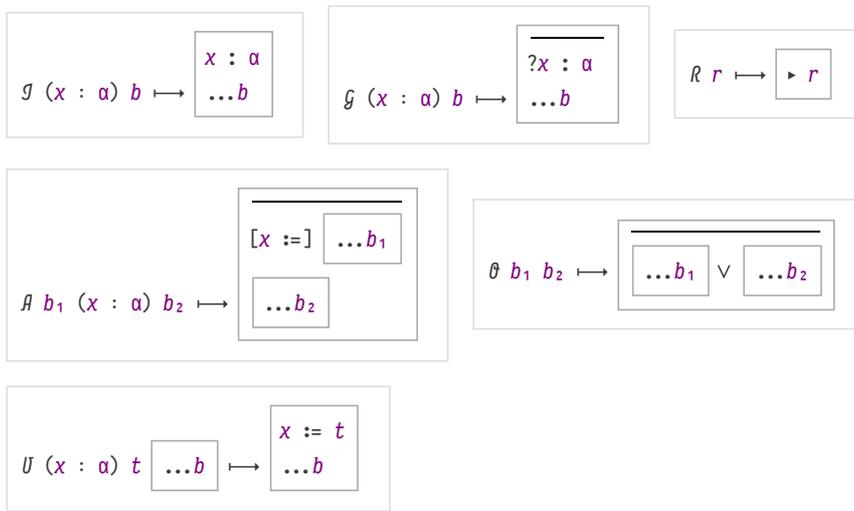
The above formulation is intended to match with the architecture designed in G&G, so that all of the same proof-steps developed in G&G are available. Unlike G&G, the system also interfaces with a flat goal-based development calculus, and so it is possible to use both G&G proof-steps and Lean tactics within the same development calculus. To do this, let's formalise the system presented above in Section 3.3.1 with the following `Box` datatype (3.9). Define a `Binder := (name : Name) × (type : Expr)` to be a name identifier and a type with notation `(name : type)`, using a smaller colon to keep the distinction from a meta-level type annotation.

```
Box ::=
| G (x : Binder) (b : Box) : Box
| g (m : Binder) (b : Box) : Box
| R (r : Expr) : Box
| H (b1 : Box) (r : Binder) (b2 : Box) : Box
| θ (b1 : Box) (b2 : Box) : Box
| U (x : Binder) (t : Expr) (b : Box) : Box
```

(3.9). Inductive definition of `Box`.

I will represent instances of the `Box` type with a 2D box notation defined in (3.10) to make the connotations of the datastructure more apparent.

(3.10). Visualisation rules for the `Box` type. Each visualisation rule takes a pair $L \mapsto R$ where L is a constructor for `Box` and R is the visualisation. Everything above the horizontal line in the box is called a *hypothesis*. Everything below a line within a box is a \mathcal{G} -box, called a *goal*. This visualisation is also implemented in Lean using the widgets framework presented in Section 5.8.



These visualisations are also presented directly to the user through the use of the widgets UI framework presented in Chapter 5. The details of this visualisation are given in Section 5.8.

To summarise the roles for each constructor:

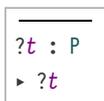
- $\mathcal{G} x b$ is a **variable introduction binder**, that is, it does the same job as a lambda binder for expressions and is used to introduce new hypotheses and variables.
- $\mathcal{G} m b$ is a **goal binder**, it introduces a new metavariable $?m$ that the child box depends on.
- $R r$ is the **result box**, it depends on all of the variables and goals that are declared above it. It represents the proof term that is returned once all of the goal metavariables are solved. Extracting a proof term from a well-formed box will be discussed in Section 3.4.
- $A b_1 (x : \alpha) b_2$ is a **conjunctive pair** of boxes. Both boxes have to be solved to complete the proof. `Box` b_2 depends on variable x . When b_1 is solved, the x value will be replaced with the resulting proof term of b_1 .
- $O b_1 b_2$ is a **disjunctive pair**, if *either* of the child boxes are solved, then so is the total box. This is used to implement branching and backtracking.
- $U x b$ is a **value binder**. It introduces a new assigned variable.

`Box`s also have a set of well-formed conditions designed to follow the typing judgements of the underlying proof-assistant development calculus. This will be developed in Section 3.4.

3.3.3. Initialising and terminating a `Box`

Given an expression representing a theorem statement $P : \text{Expr}$, $\phi \vdash P : \text{Prop}$, we can initialise a box to solve P as $b_0 := \mathcal{G} (t : P) (R t)$ (3.11).

(3.11). Initial $b_0 : \text{Box}$ given $\vdash P : \text{Prop}$.



In the case that P also depends on a context of hypotheses $\Gamma \vdash P : \text{Prop}$, these can be incorporated by prepending to the initial b_0 in (3.11) with an \mathcal{G} box for each $h \in \Gamma$. For example, if $\Gamma = [(x:\alpha), (y:\beta)]$ then send b_0 to $\mathcal{G} (x:\alpha), \mathcal{G} (y:\beta), b_0$.

Say that a `Box` is **solved** when there are no \mathcal{G} -binders remaining in the `Box`. At this point, the proving process ceases and a proof term and natural language proof may be generated.

3.3.4. Transforming a `Box`

The aim is to solve a box through the use of a sequence of sound transformations on it. Define a **box-tactic** is a partial function on boxes `BoxTactic := Box → Option Box`. Box-tactics act on `Box`es in the same way that tactics act on proof states. That is, they are functions which act on a proof-state (i.e., a representation of an incomplete proof) in order to prove a theorem. This is to make it easier to describe how box-tactics interface with tactics in Section 3.4 and Appendix A.

In Section 3.3.1 we saw some examples of box-tactics to advance the box state and eventually complete it. A complete set of box-tactics that are implemented in the system will be given in Section 3.5.

As with tactics²⁷, there is no guarantee that a particular box-tactic will produce a sound reasoning step; some box-tactics will be nonsense (for example, a box-tactic that simply deletes a goal) and not produce sound proofs. In Section 3.4 I will define what it means for a box-tactic to be sound and produce a correct proof that can be checked by the ITP's kernel.

3.3.5. Relation to other development calculi

The `Box` calculus's design is most similar to McBride's OLEG [McBoo] and G&G's prover [GG17]. A more abstract treatment can be found in the work of Sterling and Harper [SH17], implemented within the [RedPRL theorem prover](#).

The novel contribution of the `Box` calculus developed here is that it works *within* a Spiwack-style [Spi11]²⁸ flat metavariable context model as is used in Lean. That is, it is a layer atop the existing metavariable context system detailed in Section 2.4.3. This means that it is possible for the new calculus to work alongside an existing prover, rather than having to develop an entirely new one as was required for OLEG and the G&G prover. This choice opens many possibilities: now one can leverage many of the advanced features that Lean offers such as a full-fledged modern editor and metaprogramming toolchain [EUR+17]. This approach also reduces some of the burden of correctness pressed upon alternative theorem provers, because we can outsource correctness checking to the Lean kernel. Even with this protection, it is still frustrating when a development calculus produces an incorrect proof and so I will also provide some theoretical results in Section 3.4 and Appendix A on conditions that must be met for a proof step to be sound. The design of the `Box` calculus is independent of any particular features of Lean, and so a variant of it may be implemented in other systems.

The central datatype is the `Box`. This performs the role of holding a partially constructed proof object and a representation of the goals that remain to be solved. As discussed in Section 3.1.3, the purpose is to have a structurally shared tree of goals and assumptions that is also compatible with Lean tactics.

McBride's OLEG [McBoo] is the most similar to the design presented here. OLEG 'holes' are functionally the same as metavariables. That is, they are specially tagged variables that will eventually be assigned with expressions. OLEG provides an additional constructor for expressions called 'hole-bindings' or '?-bindings'. Because OLEG is a ground-up implementation of a new theorem prover, hole-bindings can be added directly as constructors for expressions. This is not available in Lean (without reimplementing Lean expressions and all of the algorithms)²⁹. These hole-bindings perform the same role as the `g` constructor in that they provide the context of variables that the hole/metavariable is allowed to depend on. But if the only purpose of a hole-binding is to give a context, then why not just explicitly name that context as is done in other theorem provers? The `Box` architecture given above is intended to give the best of both worlds, in that you still get a shared goal-tree structure without needing to explicitly bind metavariables within the expression tree. Instead they are bound in a structure on top of it.

Lean and Coq's proof construction systems make use of the metavariable context approach outlined in Section 2.4. The metavariable context here performs the same role as the `g` goal boxes, however this set of goals is flattened in to a list structure rather than stored in a

²⁷ At least, tactics in a 'checker' style proof assistant such as Lean. See Section 2.1 for more information.

[McBoo] **McBride, Conor** *Dependently typed functional programs and their proofs* (2000) PhD thesis (University of Edinburgh)

[GG17] **Ganesalingam, Mohan; Gowers, W. T.** *A fully automatic theorem prover with human-style output* (2017) Journal of Automated Reasoning

[SH17] **Sterling, Jonathan; Harper, Robert** *Algebraic Foundations of Proof Refinement* (2017) CoRR

[Spi11] **Spiwack, Arnaud** *Verified computing in homological algebra, a journey exploring the power and limits of dependent type theory* (2011) PhD thesis (INRIA)

²⁸ See Section 2.4 for more background information.

[EUR+17] **Ebner, Gabriel; Ullrich, Sebastian; Roesch, Jared; et al.** *A metaprogramming framework for formal verification* (2017) Proceedings of the ACM on Programming Languages

[McBoo] **McBride, Conor** *Dependently typed functional programs and their proofs* (2000) PhD thesis (University of Edinburgh)

²⁹ It might be possible to use Lean's expression macro system to implement hole-bindings, but doing so would still require reimplementing a large number of type-context-centric algorithms such as unification [SB01].

[SB01] **Snyder, Wayne; Baader, Franz** *Unification theory* (2001) Handbook of automated reasoning

tree as in `Box`. This makes many aspects such as unification easier but means that structural sharing (Section 3.1.3) is lost. In Section 3.4.4 I show that we do not have to forgo use of the algorithms implemented for a flat metavariable structure to use `Box`s.

In Isabelle, proofs are constructed through manipulating the proof state directly through an LCF-style [Mil72] kernel of available functions³⁰. Schematic variables are used to create partially constructed terms.

Sterling and Harper [SH17] provide a category-theoretical theory of partially constructed proofs and use these principles in the implementation of `RedPRL`. They are motivated by the need to create a principled way performing refinement of proofs in a dependently-typed foundation. They develop a judgement-independent framework for describing development calculi within a category-theoretical setting.

Another hierarchical proof system is HiProof [ADL10]. HiProof makes use of a tree to write proofs. The nodes of a tree are invocations of inference rules and axioms and an edge denotes the flow of evidence in the proof. These nodes may be grouped to provide varying levels of detail. These hierarchies are used to describe a proof, whereas a `Box` here describes a partially completed proof and a specification of hypotheses and goals that must be set to construct the proof.

[Mil72] **Milner, Robin** *Logic for computable functions description of a machine implementation* (1972) Technical Report

³⁰ As can be seen in the source <https://isabelle-dev.sketis.net/source/isabelle/browse/default/src/Pure/thm.ML>.

[SH17] **Sterling, Jonathan; Harper, Robert** *Algebraic Foundations of Proof Refinement* (2017) CoRR

[ADL10] **Aspinall, David; Denney, Ewen; Lüth, Christoph** *Tactics for hierarchical proof* (2010) Mathematics in Computer Science

3.4. Creating valid proof terms from a

`Box`

Note that because we are using a trusted kernel, the result of producing an invalid proof with `Box` is a mere inconvenience because the kernel will simply reject it. However, in order for the `Box` structure defined in Section 3.3.2 to be useful within a proof assistant such as Lean as motivated by Section 3.1.4, it is important to make sure that a solved `Box` produces a valid proof for the underlying trusted kernel. To do this, I will define a typing judgement $M; \Gamma \vdash b : \alpha$ and then present a method for extracting a proof term $M; \Gamma \vdash r : \alpha$ from `b` with the same type provided `b` is solved.

3.4.1. Assignability for `Box`

In Section 2.4.2, I introduced the concept of an *assignable* datastructure for generalising variable-manipulation operations to datatypes other than expressions. We can equip a datatype containing expressions with an assignability structure `assign` (3.12). This is a variable-context-aware traversal over the expressions present for the datatype. For `Box`, this traversal amounts to traversing the expressions in each box, while adding to the local context if the subtree is below a binder. The definition of `assign` induces a definition of variable substitution and abstraction over `Box`s.

```
assign (f : Context → Expr → M Expr) (Γ : Context)
  : Box → M Box
| G x b   ↪ pure G <*> assign f Γ x <*> assign f [..Γ, x] b
| G m b   ↪ pure G <*> assign f Γ m <*> assign f [..Γ, m] b
| R r     ↪ pure R <*> assign f Γ r
| A b1 x b2 ↪ pure A <*> assign f Γ b1 <*> assign f Γ x <*> assign f [..Γ, x] b2
| Θ b1 b2 ↪ pure Θ <*> assign f Γ b1 <*> assign f Γ b2
| U x t b ↪ pure U <*> assign f Γ x <*> assign f Γ t <*> assign f [..Γ, x:=t] b
```

(3.12) Definition of `assign` for `Box`. See Section 2.4.2 for a description of assignability. The `<*>` operator is the applicative product for some applicative functor `M` (see Section 2.2.2). Note that goal `g` declarations are bound, so for the purposes of assignment they are treated as simple variable binders.

3.4.2. Typing judgements for `Box`

In Section 2.4, I defined contexts Γ , metavariable contexts M . As covered in Carneiro's thesis [Car19], Lean's type theory affords a set of inference rules on typing judgements $\Gamma \vdash t : \alpha$, stating that the expression `t` has the type α in the context Γ . However, these inference rules are only defined for expressions $t : \text{Expr}$ that do not contain metavariables. In Appendix A.1, I extend these judgements (A.10), (A.11) to also include expressions containing metavariable contexts $M; \Gamma \vdash t : \alpha$.

[Car19] **Carneiro, Mario** *Lean's Type Theory* (2019) Masters' thesis (Carnegie Mellon University)

In a similar way, we can repeat this for `Box`: given contexts `M` and `Γ` we can define a typing judgement $M; \Gamma \vdash b : \beta$ where $b : \text{Box}$ and β is a type. The inference rules for this are given in (3.13). These have been chosen to mirror the typings given in Section 2.4.3.

$$\begin{array}{c}
\frac{M; \dots, \Gamma, x:\alpha \vdash b : \beta}{M; \Gamma \vdash (\mathcal{G} (x:\alpha), b) : (\Pi (x:\alpha), \beta)} \mathcal{G}\text{-typing} \qquad \frac{M; \Gamma \vdash t : \alpha}{M; \Gamma \vdash R t : \alpha} R\text{-typing} \\
\\
\frac{[..M, \langle m, \alpha, \Gamma \rangle]; \Gamma \vdash b : \beta}{M; \Gamma \vdash (\mathcal{G} (?x:\alpha), b) : \beta} \mathcal{G}\text{-typing} \qquad \frac{M; \Gamma \vdash b_1 : \alpha \quad M; [\dots, \Gamma, (x:\alpha)] \vdash b_2 : \beta}{M; \Gamma \vdash (\mathcal{A} b_1 (x:\alpha) b_2) : \beta} \mathcal{A}\text{-typing} \qquad \frac{M; \Gamma \vdash b_1 : \alpha \quad M; \Gamma \vdash b_2 : \alpha}{M; \Gamma \vdash (\mathcal{O} b_1 b_2) : \alpha} \mathcal{O}\text{-typing} \\
\\
\frac{M; \Gamma \vdash v : \alpha \quad M; [\dots, \Gamma, (x:\alpha)] \vdash b : \beta}{M; \Gamma \vdash (U (x:\alpha=v), b) : \beta} U\text{-typing}
\end{array}$$

(3.13). Typing inference rules for `Box`. Compare with (A.10) and (A.11) in Appendix A.1.

These typing rules have been designed to match the typing rules (A.10) of the underlying proof terms that a `Box` produces when solved, as I will show next.

3.4.3. Results of a `Box`

The structure of `Box` is designed to represent a partially complete expression without the use of unbound metavariables. `Box`es can be converted to expressions containing unbound metavariables using `results : Box → Set Expr` as defined in (3.14).

```

results
: Box      → Set Expr
|  $\mathcal{G} (x:\alpha) b$   → {(Expr.λ (x:α) r[x]) for r in results b}
|  $\mathcal{G} (x:\alpha) b$   → results b
|  $R t$            → {t}
|  $\mathcal{A} b_1 (x:\alpha) b_2$  →
  {s for s in results {x → r} b2
   for r in results b1}
|  $\mathcal{O} b_1 b_2$      → results b1 ∪ results b2
|  $U (x:\alpha) b$   → {(Expr.let x b r) for r in results b}

```

(3.14). Definition of `results`. `r[x]` denotes a delayed abstraction (Appendix A.3.1) needed in the case that `r` contains metavariables.

A $b : \text{Box}$ is **solved** when there are no remaining `g` entries in it. When `b` is solved, the set of results for `b` does not contain any metavariables and hence can be checked by the kernel. In the case that `b` is unsolved, the results of `b` contain unbound metavariables. Each of these metavariables corresponds to a `g`-binder that needs to be assigned.

Lemma 3.15 (compatibility): Suppose that $M; \Gamma \vdash b : \alpha$ for $b : \text{Box}$ as defined in (3.13). Then $[..M, ..goals b]; \Gamma \vdash r : \alpha$. (Say that `b` is **compatible** with $r \in \text{results } b$.) Here, `goals b` is the set of metavariable declarations formed by accumulating all of the `g`-binders in `b`. (3.16) shows a formal statement of Lemma 3.15.

$$\frac{M; \Gamma \vdash b : \alpha \quad r \in \text{results } b}{[..M, ..goals b]; \Gamma \vdash r : \alpha}$$

(3.16). Statement of Lemma 3.15. That is, take a $b : \text{Box}$ and $\alpha : \text{Expr}$, then if $b : \alpha$ in the context $M; \Gamma$ and $r : \text{Expr}$ is a result of `b` (3.14); then $r : \alpha$ in the context $M; \Gamma$ with additional metavariables added for each of the goals in `b`.

Lemma 3.15 states that given a box `b` and an expression `r` that is a result of `b`, then if `b` is a valid box with type `α` then `r` will type to `α` too in the metavariable context including all of the goals in `b`.

Lemma 3.15 is needed because it ensures that our `Box` will produce well-typed expressions when solved. Using Lemma 3.15, we can find **box-tactics** `m : Box → Option Box` - partial functions from `Box` to `Box` - such that $M; \Gamma \vdash b : \alpha \Rightarrow M; \Gamma \vdash m b : \alpha$ whenever $b \in \text{dom } m$.

Hence a chain of such box-tactic applications will produce a result that satisfies the initial goal.

Proof: Without loss of generality, we only need to prove Lemma 3.15 for a $b : \text{Box}$ with no \emptyset boxes and a single result $[r] = \text{results } b$. To see why, note that any box containing an \emptyset can be split as in (3.17) until each Box has one result. Then we may prove Lemma 3.15 for each of these in turn.

$$\text{results} \left(\boxed{\begin{array}{c} \dots p \\ \hline \dots b_1 \vee \dots b_2 \end{array}} \right) = \text{results} \left(\boxed{\begin{array}{c} \dots p \\ \dots b_1 \end{array}} \right) \cup \text{results} \left(\boxed{\begin{array}{c} \dots p \\ \dots b_2 \end{array}} \right) \quad (3.17).$$

Write $\text{result } b$ to mean this single result $[r]$. Performing induction on the typing judgements for boxes, the most difficult is $\#$ -typing, where we have to show (3.18).

$$\begin{array}{l} M; \Gamma \vdash b_1 : \alpha \\ M; [\dots \Gamma, (x:\alpha)] \vdash b_2 : \beta \\ \hline M'; \Gamma \vdash \text{result } b_1 : \alpha \\ M'; [\dots \Gamma, (x:\alpha)] \vdash \text{result } b_2 : \beta \\ \hline M'; \Gamma \vdash \text{result } (\# b_1 (x:\alpha) b_2) : \beta \end{array}$$

(3.18). The induction step that must be proven for the $\#$ -box case of Lemma 3.15.

where $M' := [\dots M, \dots \text{goals } (\# b_1 (x:\alpha) b_2)]$. To derive this it suffices to show that result is a 'substitution homomorphism':

$$\begin{array}{l} M; \Gamma \vdash \sigma \text{ ok} \\ \hline M; \Gamma \vdash \sigma (\text{result } b) \equiv \text{result } (\sigma b) \end{array}$$

(3.19). result is a substitution homomorphism.

where σ is a substitution³¹ in context Γ and \equiv is the definitional equality judgement under Γ . Then we have

³¹ See Section 2.4.1. A substitution is a partial map from variables to expressions.

$$\begin{array}{l} M'; \Gamma \vdash \\ \text{result } (\# b_1 (x:\alpha) b_2) \\ \equiv \text{result } (\{\!|x \mapsto \text{result } b_1\!\} b_2) \\ \equiv \{\!|x \mapsto \text{result } b_1\!\} (\text{result } b_2) \\ \equiv (\lambda (x:\alpha), \text{result } b_2) (\text{result } b_1) \end{array}$$

(3.20). Here, $\{\!|x \mapsto e\!\} b$ is used to denote substitution applied to b . That is, replace each occurrence of x in b with e .

We can see the substitution homomorphism property of result holds by inspection on the equations of result , observing that each LHS expression behaves correctly. Here is the case for \mathcal{G} :

$$\begin{array}{l} M'; \Gamma \vdash \\ \text{result } (\sigma (\mathcal{G} (x:\alpha) b)) \\ \equiv \text{result } \$ \mathcal{G} (x:(\sigma \alpha)) (\sigma b) \\ \equiv (\lambda (x:(\sigma \alpha)), (\text{result } (\sigma b))[x]) \\ \equiv (\lambda (x:(\sigma \alpha)), (\sigma (\text{result } b))[x]) \text{ -- } \textit{induction hypothesis} \\ \equiv \sigma (\lambda (x:\alpha), (\text{result } b)) \\ \equiv \sigma (\text{result } (\mathcal{G} (x:\alpha) b)) \end{array}$$

(3.21). result and σ obey the 'substitution homomorphism' property on the case of \mathcal{G} . Here λ is used to denote the internal lambda constructor for expressions. Note here we are assuming $\text{dom } \sigma \subseteq \Gamma$, so $x \notin \text{dom } \sigma$, otherwise $\text{dom } \sigma$.

This completes the proof of Lemma 3.15. By using compatibility, we can determine whether a given box-tactic $m : \text{Box} \rightarrow \text{Option Box}$ is *sound*. Define a box-tactic m to be **sound** when for all $b \in \text{dom } m$ we have some α such that $M; \Gamma \vdash (m b) : \alpha$ whenever $M; \Gamma \vdash b : \alpha$.

Hence, to prove a starting proposition³² P , start with an initial box $b_0 := \mathcal{G} (?t_0:P) (R ?t_0)$. Then if we only map b_0 with sound box-tactics to produce a solved box b_n , then each of $\text{results } b_n$ always has type α and hence is accepted by Lean's kernel.

³² Or, in general, a type α .

Given a box-tactic m that is sound on b , then we can construct a sound box-tactic on $\mathcal{G} (x:\alpha) b$ too that acts on the nested box b .

3.4.4. Escape-hatch to tactics

As discussed in Section 2.4.4, many provers, including Lean 3, come with a tactic combinator language to construct proofs through mutating an object called the `TacticState` comprising a metavariable context and a list of metavariables called the goals. In Section 3.1 I highlighted some of the issues of this approach, but there are many built-in and community-made tactics which can still find use within a HumanProof proof. For this reason, it is important for HumanProof to provide an 'escape hatch' allowing these tactics to be used within the context of a HumanProof proof seamlessly. I achieve this compatibility system between `Box`s and tactics through defining a zipper [Hue97] structure on `Box`s (Appendix A.2) and then a set of operations for soundly converting an underlying `TacticState` to and from a `Box` object. The details of this mechanism can be found in Appendix A.2. It is used to implement some of the box-tactics presented next in Section 3.5, since in some cases the box-tactic is the same as its tactic-based equivalent.

[Hue97] **Huet**,
G erard *Functional Pearl: The Zipper* (1997) *Journal of functional programming*

3.4.5. Summary

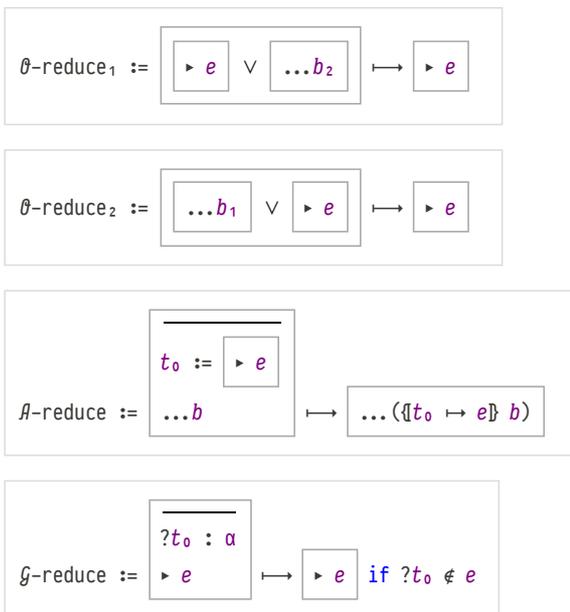
In this section, I defined assignability on `Box`s and the valid typing judgement inference rules on `Box`. I used these to define the soundness of a box-tactic and showed that for a box-tactic to be sound, it suffices to show that its typing judgement is preserved through the use of Lemma 3.15. I also briefly review Appendix A, which presents a mechanism for converting a tactic-style proof to a box-tactic.

3.5. Human-like-tactics for `Box`.

Using the framework presented above we can start defining sound tactics on `Box`s and use `Box` to actualise the kinds of reasoning discussed in Section 3.1. Many of the box-tactics here are similar to inference rules that one would find in a usual system, and so I do not cover these ones in great detail. I also skip many of the soundness proofs, because in Appendix A I instead provide an 'escape hatch' for creating sound box-tactics from tactics in the underlying metavariable-oriented development calculus.

3.5.1. Simplifying box-tactics

We have the following box-tactics for reducing `Box`s, these should be considered as tidying box-tactics.

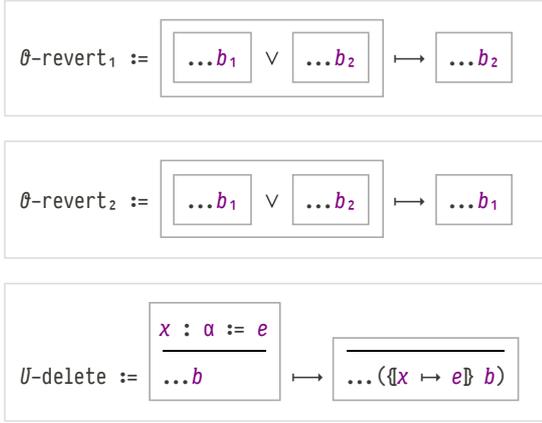


(3.22). Reduction box-tactics for `Box`. These are box-tactics which should always be applied if they can and act as a set of reductions to a box. Note that these are not congruent; for example $\theta\text{-reduce}_1$ and $\theta\text{-reduce}_2$ on $\theta \langle R e_1 \rangle \langle R e_2 \rangle$ produce different terminals.

3.5.2. Deleting tactics

These are box-tactics that cause a `Box` to become simpler, but which are not always 'safe' to do, in the sense that they may lead to a `Box` which is impossible to solve. That is, the `Box` may still have a true conclusion but it is not possible to derive this from the information given on the box. For example, deleting a hypothesis $p : P$, may prevent the goal $?t : P$ from being solved. The rules for deletion are presented in (3.23).

To motivate `θ-revert` tactics, recall that an `θ`-box $b_1 \vee b_2$ represents the state that either b_1 or b_2 needs to be solved, so `θ`-reversion amounts to throwing away one of the boxes. This is similar to `θ-reduce` in (3.22) with the difference being that we do not need one of the boxes to be solved before applying. These are useful when it becomes clear that a particular `θ`-branch is not solvable and can be deleted.

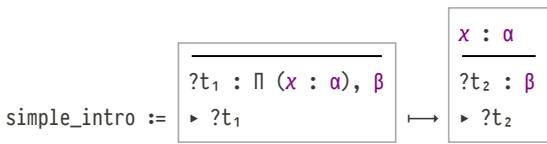


(3.23). Deletion box-tactics. `θ-revert1` and `θ-revert2` take an `θ`-box and remove one of the branches of the `θ`-box. `U-delete` removes a `U`-box and replaces each reference to the variable bound by the `U`-box with its value.

3.5.3. Lambda introduction

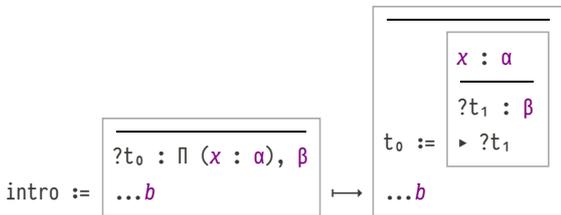
In tactics, an `intro` tactic is used to introduce `Π`-binders³³. That is, if the goal state is $\vdash \Pi (x : \alpha), \beta[x]$ the `intro` tactic produces a new state $(x : \alpha) \vdash \beta[x]$. To perform this, it assigns the goal metavariable $?t_1 : \Pi (x : \alpha), \beta[x]$ with the expression $\lambda (x : \alpha), ?t_2$ where $?t_2 : \beta[x]$ is the new goal metavariable with context including the additional local variable $x : \alpha$.

The `intro` tactic on `Box` is analogous, although there are some additional steps required to ensure that contexts are preserved correctly. The simplified case `simple_intro` (3.24), performs the same steps as the tactic version of `intro`.



³³ `Π`-binders $\Pi (x : \alpha), \beta$ are the dependent generalisation of the function type $\alpha \rightarrow \beta$ where the return type β may depend on the input value α .

The full version (3.25) is used in the case that the `λ`-box is not immediately followed by an `R`-box. In this case, a conjunctive `λ`-box must be created in order to have a separate context for the new $(x : \alpha)$ variable.



(3.24). A simple variable introduction box-tactic. Note that the new goal $?t_2$ is not wrapped in a lambda abstraction because it is abstracted earlier by the `λ` box.

(3.25). The full version of the lambda introduction box-tactic. The box on the rhs of \mapsto is an `λ` box: $\lambda (g \ x, g \ ?t, R \ ?t_1) \ t_0 \ b$.

The fact that `intro` is sound follows mainly from the design of the definitions of `λ`:

Structural sharing is defined as making use of the same substructure multiple times in a larger structure. For example, a tree with two branches being the same would be using structural sharing if the sub-branches used the same object in memory. Define b' to be $\lambda (x : \alpha), g (?t_1 : \beta), R ?t_1$, represented graphically in (3.26). The typing judgement (3.26) follows from the typing rules (3.13).

$$\frac{x : \alpha}{?t_1 : \beta} \vdash \triangleright ?t_1 : \Pi (x : \alpha), \beta$$

By the definition of a sound box-tactic we may assume $\vdash (g \ ?t_0, \ b) : \gamma$ for some type γ . From the g typing rule (3.13) we then have $[?t_0]; \phi \vdash b : \gamma$. Then it follows from β typing (3.13) that $\vdash \beta \ b' \ (t_0 : \Pi (x : \alpha), \ \beta) \ b : \gamma$ where $b' := g \ (x : \alpha), \ g \ (?t_1 : \beta), \ R \ ?t_1$.

3.5.4. Split and cases tactics

Here I present some box-tactics for performing introduction and elimination of the \wedge type. The `Box` version of `split` performs the same operation as `split` in Lean: introducing a conjunction. A goal $?t_0 : P \wedge Q$ is replaced with a pair of new goals $(?t_1, ?t_2)$. These can be readily generalised to other inductive datatypes with one constructor³⁴. In the implementation, these are implemented using the tactic escape-hatch described in Appendix A.

$$\text{split} := \frac{?t_0 : P \wedge Q}{\dots b} \mapsto \frac{?t_1 : P \quad ?t_2 : Q}{\dots (\lambda ?t_0 \mapsto \langle ?t_1, ?t_2 \rangle) b}$$

Similarly we can eliminate a conjunction with `cases`.

$$\text{cases} := \frac{h_0 : P \wedge Q}{\dots b} \mapsto \frac{h_0 : P \wedge Q \quad h_1 : P := \text{fst } h_0 \quad h_2 : Q := \text{snd } h_0}{\dots b}$$

3.5.5. Induction box-tactics

\wedge -elimination (3.28) from the previous section can be seen as a special case of induction on datatypes. Most forms of dependent type theory use inductive datatypes (see Section 2.2.3) to represent data and propositions, and use induction to eliminate them. To implement induction in CIC³⁵, each inductive datatype comes equipped with a special constant called the **recursor**. This paradigm broadens the use of the words 'recursion' and 'induction' to include datastructures that are not recursive.

For example, we can view conjunction $A \wedge B : \text{Prop}$ as an inductive datatype with one constructor $\text{mk} : A \rightarrow B \rightarrow A \wedge B$. Similarly, a disjunctive $A \vee B$ has two constructors $\text{inl} : A \rightarrow A \vee B$ and $\text{inr} : B \rightarrow A \vee B$. Interpreting \rightarrow as implication, we recover the basic introduction axioms for conjunction and disjunction. The eliminators for \wedge and \vee are implemented using recursors given in (3.29).

$$\begin{aligned} \wedge\text{-rec} &: \forall (A \ B \ C : \text{Prop}), (A \rightarrow B \rightarrow C) \rightarrow (A \wedge B) \rightarrow C \\ \vee\text{-rec} &: \forall (A \ B \ C : \text{Prop}), (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow (A \vee B) \rightarrow C \end{aligned}$$

Performing an *induction step* in a CIC theorem prover such as Lean amounts to the application of the relevant recursor. Case analysis on a disjunctive hypothesis makes for a good example of recursion, the recursor $\vee\text{-rec} : (P \rightarrow C) \rightarrow (Q \rightarrow C) \rightarrow (P \vee Q) \rightarrow C$ is used. Given a box $g \ (h_0 : P \vee Q), \ b$ where $h_0 \vdash b : \alpha$, the `v-cases` box-tactic sends this to the box defined in (3.30). This is visualised in (3.31).

(3.26). The judgement that b' has type $\Pi (x : \alpha), \beta$ may possibly depend on x .

³⁴ One caveat is that the use of \exists requires the use of a non-constructive axiom of choice with this method. This is addressed in Section 3.5.8

(3.27). Box-tactic for introducing conjunctions.

(3.28). Box-tactic for eliminating conjunctions. $\text{fst} : P \wedge Q \rightarrow P$ and $\text{snd} : P \wedge Q \rightarrow Q$ are the \wedge -projections. In the implementation, h_0 is hidden from the visualisation to give the impression that the hypothesis h_0 has been 'split' in to h_1 and h_2 .

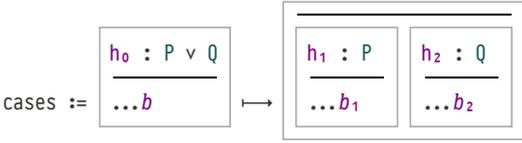
³⁵ Calculus of Inductive Constructions. The foundation used by Lean 3 and Coq (Section 2.1.3). See [Car19 §2.6] for the axiomatisation of inductive types within Lean 3's type system.

(3.29). Recursors for conjunction and disjunction.

```

A (G (h1:P), b1) (c1:P → α) (
  A (G (h2:Q), b2) (c2:Q → α) (
    R (v-rec c1 c2 h0)
  )
)
where b1 := {h0 ↦ inl h1} b
       b2 := {h0 ↦ inr h2} b

```



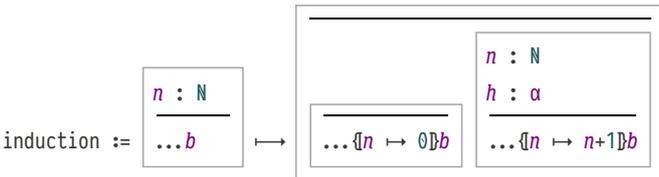
Note that the $b : \text{Box}$ in (3.31) may contain multiple goals. When the `cases` box-tactic is applied to $G (h_0 : P \vee Q), b$, the resulting `Box` on the rhs of (3.31) results in two copies of these goals. This implements structural sharing of goals as motivated in Section 3.1.3. Structural sharing has a significant advantage over the goal-state style approach to tactics, where the equivalent `cases` tactic would have to be applied separately to each goal if there were multiple goals.

This structurally-shared induction step also works on recursive datastructures such as lists and natural numbers. These datatypes' recursors are more complicated than non-recursive datastructures such as those in (3.29) in order to include induction hypotheses. The recursor for natural numbers is shown in (3.32). (3.33) is the corresponding box-tactic that makes use of (3.32). (3.34) is the detailed `Box` structure for the right-hand side of (3.33).

```

N-rec :
  (ℓ : N → Type)           -- motive
  → (ℓ 0)                  -- zero case
  → ((i : N) → ℓ i → ℓ (i + 1)) -- successor case
  → (i : N) → ℓ i

```



```

A ({n ↦ 0} b) (c1 : {n ↦ 0} α) (
  A (G (n : N), G (h : α), {n ↦ n+1} b) (c2 : {n ↦ n+1} α) (
    R (N-rec (n ↦ α) c1 c2 n)
  )
)

```

In general, finding the appropriate motive ℓ for an induction step amounts to a higher order unification problem which was shown to be undecidable [Dow01 §3]. However, in many practical cases ℓ can be found and higher-order provers come equipped with heuristics for these cases, an early example being Huet's semidecidable algorithm. Rather than reimplementing these heuristics, I implement induction box-tactics on `Box` by using the 'escape hatch' feature (Section 3.4.4).

3.5.6. Introducing θ boxes

The purpose of θ boxes is to enable backtracking and branches on `Box`es that enables structural sharing. The G&G prover [GG17] takes a similar approach. For example, suppose that we had a goal $x \in A \cup B$ for some sets A, B . We might have some lemmas of the form $h_1 : P \rightarrow x \in A$ and $h_2 : Q \rightarrow x \in B$ but we are not yet sure which one to use. In a goal-based system, if you don't yet know which injection to use, you have to guess and manually backtrack. However, there may be some clues about which lemma is correct that only become apparent after applying an injection. In the above example, if only $h_3 : P$ is present as a hypothesis, it requires first performing injection before noticing that h_1 is the correct lemma to apply. In [#future-work] I discuss more advanced, critic-like workflows that θ -boxes also enable.

(3.30). Explicit datastructure showing the resulting `Box` after performing `v-cases` on $G (h_0 : P \vee Q), b$.

(3.31). Special case of recursion for eliminating \vee statements. The right-hand side of \mapsto is simplified for the user, but is represented as a nested set of `A` boxes as explicitly written in (3.30). b_1 and b_2 are defined in (3.30).

(3.32). Recursor for natural numbers. `N-rec` can be seen to have the same signature as mathematical induction on the natural numbers.

(3.33). Induction box-tactic on natural numbers. Implemented using the 'escape hatch' detailed in Appendix A. Here, α is the result type of b (Section 3.4.2). That is, $(n:N) \vdash b : \alpha$.

(3.34). Detail on the rhs of (3.33). The signature for `N-rec` is given in (3.32).

[Dow01] **Dowek, Giles** *Higher-order unification and matching* (2001) Handbook of automated reasoning

[GG17] **Ganesalingam, Mohan; Gowers, W. T.** *A fully automatic theorem prover with human-style output* (2017) Journal of Automated Reasoning

The θ box allows us to explore various counterfactuals without having to perform any user-level backtracking (that is, having to rewrite proofs). The primitive box-tactic that creates new θ -boxes is shown in (3.35). This is used to make more 'human-like' box-tactics such as `v-split` (3.36).

$$\theta\text{-intro} := \boxed{\dots b} \mapsto \boxed{\dots b} \vee \boxed{\dots b}$$

$$v\text{-intro} := \boxed{\frac{?t : P \vee Q}{\dots b}} \mapsto \boxed{\frac{?t : P}{\dots b}} \vee \boxed{\frac{?t : Q}{\dots b}}$$

(3.35). Box-tactic for introducing an θ -box by duplication.

(3.36). Box-tactic for introducing an θ -box by duplication.

3.5.7. Unification under a `Box`

Unification is the process of taking a pair of expressions $\mathcal{L} \ r : \text{Expr}$ within a joint context $M; \Gamma$ and finding a valid set of assignments of metavariables σ in M such that $(M + \sigma); \Gamma \vdash \mathcal{L} \equiv r$. Rather than develop a whole calculus of sound unification for the `Box` system, I can use the 'escape hatch' tactic compatibility layer developed in Appendix A to transform a sub-`Box` to a metavariable context and then use the underlying theory of unification used for the underlying development calculus of the theorem prover (in this case Lean). This is a reasonable approach because unifiers and matchers for theorem provers are usually very well developed in terms of both features and optimisation, so I capitalise on a unifier already present in the host proof assistant has a perfectly good one already.

3.5.8. Apply

In textbook proofs of mathematics, often an application of a lemma acts under \exists binders. For example, let's look at the application of `fs n` being continuous from earlier.

$$h_1 : \forall (x : X) (\varepsilon : \mathbb{R}) (h_0 : \varepsilon > 0), \exists (\delta : \mathbb{R}) (h_1 : \delta > 0), \forall (y : X) (h_2 : \text{dist } x \ y < \delta), \text{dist } (f \ x) \ (f \ y) < \varepsilon$$

(3.37). An example lemma h_1 to apply. h_1 is a proof that `fs n` is continuous.

In the example the application of h_1 with N, ε, h_3 , and then eliminating an existential quantifier δ and then applying more arguments y , all happen in one step and without much exposition in terms of what δ depends on. A similar phenomenon occurs in backwards reasoning. If the goal is `dist (f x) (f y) < ε`, in proof texts the continuity of `f` is applied in one step to replace this goal with `dist x y < δ`, where δ is understood to be an 'output' of applying the continuity of `f`.

Contrast this with the logically equivalent Lean tactic script fragment (3.38):

```
...
show dist (f x) (f y) < ε,
obtain <δ, δ_pos, h1> : ∃ δ, δ > 0 ∧ ∀ y, dist x y < δ → dist (f x) (f y) < ε,
  apply <continuous f>,
  apply h1,
show dist x y < δ,
...
```

(3.38). A Lean tactic-mode proof fragment that is usually expressed in one step by a human, but which requires two steps in Lean. The `show` lines can be omitted but are provided for clarity to show the goal state before and after the `obtain` and `apply` steps. The `obtain <_,_,_> : P` tactic creates a new goal $t : P$ and after this goal is solved, performs case-elimination on t . Here, `obtain <δ, δ_pos, h1>` introduces $\delta : \mathbb{R}, \delta_pos : \delta > 0$ and h_1 to the context.

In order to reproduce this human-like proof step, we need to develop a theory for considering 'complex applications'. A further property we desire is that results of the complex application must be stored such that we can recover a natural language write-up to explain it later (e.g., creating "Since `f` is continuous at x , there is some δ ...").

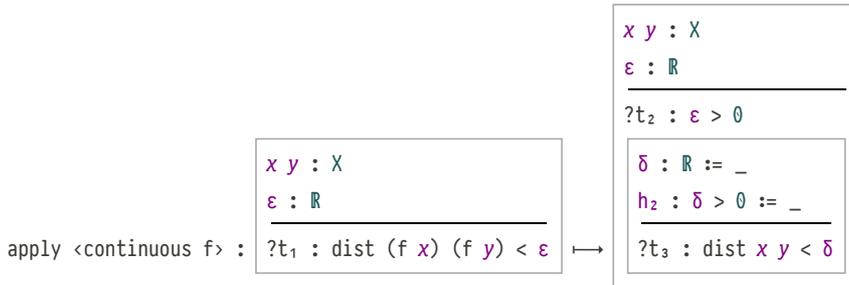
The `apply` subsystem works by performing a recursive descent on the type of the assumption being applied. For example, applying the lemma given in (3.37) to a goal $t : P$ attempts to unify P with `dist (f ?x) (f ?y) < ?ε` with new metavariables $?x ?y : X, \varepsilon : \mathbb{R}$. If the match is successful, it will create a new goal for each variable in a Π -binder³⁶ above the matching expression and a new \forall -binder for each introduced \exists -

³⁶ Note that \forall is sugar for Π .

variable and each conjunct. These newly introduced nested boxes appear in the same order as they appear in the applied lemma.

This apply system can be used for both forwards and backwards reasoning box-tactics. Above deals with the backwards case, in the forwards case, the task is reversed, with now a variable bound by a λ -binder being the subject to match against the forwards-applied hypothesis.

An example of applying (3.37) to the goal $\text{dist } (f \ x) \ (f \ y) < \epsilon$ can be seen in (3.39).



(3.39). An example of applying (3.37) to t_1 . It produces a set of nested goals in accordance with the structure of the binders in (3.37). Result Boxes are omitted.

3.5.8.1. A note on using apply with existential statements

One complication with this approach to apply is performing many logical inference steps when applying a lemma in one go. There is a technical caveat with applications of existential statements such as $\exists (\delta : \mathbb{R}), d(x, y) < \delta$: by default, Lean is a non-classical theorem prover, which here amounts to saying that the axiom of choice is not assumed automatically. Without the axiom of choice, it is not generally possible to construct a projection function $\epsilon : \exists (x : \alpha), P [x] \rightarrow \alpha$ such that $P[\epsilon \ h]$ is true for all $h : \exists (x : \alpha), P$. There are two ways to overcome this limitation:

1. Assume the axiom of choice and make use of the nonconstructive projector ϵ .
2. When an apply step encounters an existential quantifier, wrap the entire proof in an existential quantifier recursor³⁷

$\exists\text{-rec } (C : \text{Prop}) : (\forall (x : \alpha), P \ x \rightarrow C) \rightarrow (\exists (x : \alpha), P \ x) \rightarrow C$ using λ -boxes. This is performed in exactly the same manner that induction box-tactics are applied in Section 3.5.5.

³⁷ Recursors are discussed in Section 3.5.5.

HumanProof, as it is currently implemented, uses strategy 1. This prevents proofs from being constructive, but is otherwise not so great a concession, since mathematicians regularly make use of this in fields outside logic. There was some effort to also implement strategy 2, but I dropped it.

3.5.9. Summary

This section introduced a set of sound box-tactics that are implemented for the HumanProof system. In the next section we will see how these box-tactics can be used to create natural language write-ups of proofs.

3.6. Natural language generation of proofs

In this section I detail how the above box architecture is used to produce natural language writeups as the proof progresses. The general field is known as Natural Language Generation (NLG). You can find a background survey of NLG both broadly and within the context of generating proofs in Section 2.7.

Here I lean on the work of Ganesalingam, who in his thesis [Gan10] has specified a working theory of the linguistics of natural language mathematics. As well as generating a formally verifiable result of a proof, I also extend on G&G by providing some new mechanisms for converting Lean predicates and typeclasses in to English language

[Gan10] **Ganesalingam, Mohan** *The language of mathematics* (2010) PhD thesis (University of Cambridge)

sentences. That is, in the implementation of the G&G theorem prover, many natural language constructs such as " X is a metric space" were hard-coded in to the system. In this work I provide a general framework for attaching verbalisations of these kinds of statements to typeclasses and predicates within Lean. I also make the resulting write-up interactive; emitting a partial proof write-up if the proof-state is not yet solved and also inspecting the natural language write-up through the widgets system are possible. In contrast G&G's output was a static \LaTeX file.

The goal of this section is to demonstrate that the `Box` architecture above is representative of human-like reasoning by constructing natural language writeups of the proofs created using `Box`s. As such the NLG used here is very simple compared to the state of the art and doesn't make use of any modern techniques such as deep learning. The output of this system is evaluated by real, human mathematicians in Chapter 6. An example of a proof generated by the system is shown below in Output 3.40. There are some challenges in converting a `Box` proof to something that reads like a mathematical proof that I will detail here.

Let X, Y and Z be metric spaces, let f be a function $X \rightarrow Y$ and let g be a function $Y \rightarrow Z$. Suppose f is continuous and g is continuous. We need to show that $g \circ f$ is continuous. Let $\varepsilon > 0$ and let $x \in X$. We must choose $\delta > 0$ such that $\forall (y : X), d(x, y) \leq \delta \Rightarrow d((g \circ f)(x), (g \circ f)(y)) \leq \varepsilon$. Since g is continuous, there exists a $\eta > 0$ such that $d((g \circ f)(x), (g \circ f)(y)) \leq \varepsilon$ whenever $d(f(x), f(y)) \leq \eta$. Since f is continuous, there exists a $\theta > 0$ such that $d(f(x), f(y)) \leq \eta$ whenever $d(x, y) \leq \theta$. Since $d(x, y) \leq \delta$, we are done by choosing δ to be θ .

3.6.1. Overview

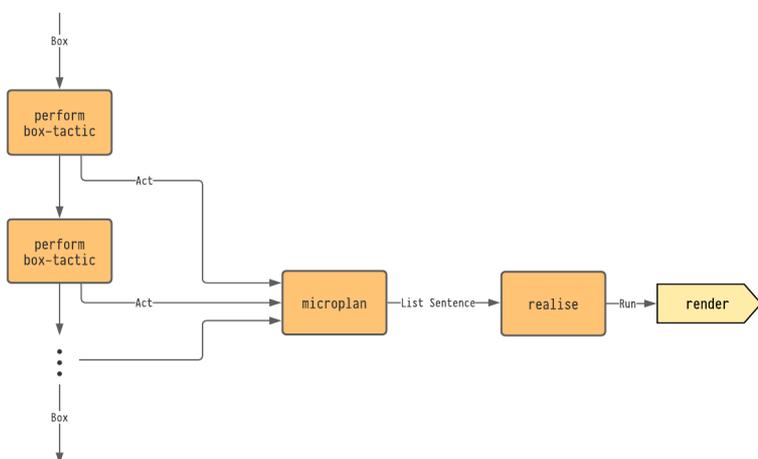
The architecture of the NLG component is given in Figure 3.41. The design is similar to the standard architecture discussed in Section 2.7.1. In Section 3.1.2 I explained the decision to design the system to permit only a restricted set of box-tactics on a `Box` representing the goal state of the prover. To create the natural language write-up from these box-tactics, each box-tactic also emits an `Act` object. This is an inductive datatype representing the kind of box-tactic that occurred. So for example, there is an `Intro : List Binder \rightarrow Act` that is emitted whenever the `intro` box-tactic is performed, storing the list of binders that were introduced. A list of `Act`s is held in the state monad for the interactive proof session. This list of acts is then fed to a micro-planner, which converts the list of acts to an abstract representation of sentences³⁸. These sentences are converted to a realised sentence with the help of `Run` which is a form of S-expression [McC60] containing text and expressions for interactive formatting. This natural language proof is then rendered in the output window using the widgets system (Chapter 5).

Output 3.40. Output from the HumanProof natural language write-up system for a proof that the composition of continuous functions is continuous.

³⁸ Sometimes referred to as a *phrase specification*

[McC60] **McCarthy, John** *Recursive functions of symbolic expressions and their computation by machine, Part I* (1960) Communications of the ACM

Figure 3.41. Overview of the pipeline for the NLG component of HumanProof. A `Box` has a series of box-tactics performed upon it, each producing an instance of `Act`, an abstract representation of what the box-tactic did. A list of all of the `Act`s from the session is then converted in to a list of sentences, which is finally converted to an S-expression-like structure called `Run`. Compare this with the standard architecture given in Figure 2.41; the main difference being that the macroplanning phase is performed by the choice of box-tactics performed on boxes as detailed in Section 3.5.



3.6.2. Grice's laws of implicature

One resource that has proven useful in creating human-like proofs is the work of the Grice on implicature in linguistics [Gri75]. To review, Grice states that there is an unwritten rule in natural languages that one should only provide as much detail as is needed to convey the desired message. For example, the statement "I can't find my keys" has the implicature "Do you know where my keys are?", it implies that the keys may have been lost at the current location and not in a different part of town and so on. If superfluous detail is included, the reader will pick this up and try to use it to infer additional information. Saying "I can't find my keys at the moment" interpreted literally has the same meaning as "I can't find my keys", but implicitly means that I have only just realised the key loss or that I will be able to find them soon. Grice posits four maxims that should be maintained in order for a sentence or phrase to be useful:

[Gri75] **Grice, Herbert**
P *Logic and conversation*
 (1975) Speech acts

1. **Quantity** The contribution should contain no more or less than what is required.
 Examples: "Since $x > 5$ and x is prime, $x > 6$ ". "Let x be a positive real such that $x > 0$."
2. **Quality** Do not say things for which you don't have enough evidence or things that are not true. An example here would be a false proof.
3. **Relation** The contributed sentence should be related to the task at hand. Example; putting a true but irrelevant statement in the middle of the proof is bad.
4. **Manner** The message should avoid being obscure, ambiguous and long-winded.

Mathematical texts are shielded from the more oblique forms of implicature that may be found in general texts, but Grice's maxims are still important to consider in the construction of human-readable proofs and serve as a useful rule-of-thumb in determining when a generated sentence will be jarring to read.

With respect to the quantity maxim, it is important to remember also that what counts as superfluous detail can depend on the context of the problem and the skill-level of the reader. For example, one may write:

Suppose A and B are open subsets of X . Since f is continuous, $f^{-1}[A \cup B]$ is open.

A more introductory text will need to also mention that X is a topological space and so $A \cup B$ is open. Generally these kinds of implicit lemma-chaining can become arbitrarily complex, but it is typically assumed that these implicit applications are entirely familiar to the reader. Mapping ability level to detail is not a model that I will attempt to write explicitly here. One simple way around this is to allow the proof creator to explicitly tag steps in the proof as 'trivial' so that their application is suppressed in the natural language write-up. Determining this correct level of detail may be a problem in which ML models may have a role to play.

3.6.3. *Microplanning symbolic mathematics*

From a linguistic perspective, a remarkable property of mathematical texts is the interlacing of mathematical symbols and natural language. In the vast majority of cases, each symbolic construct has a natural language equivalent (else verbalising that symbol in conversation would be difficult). For example: " $x + y$ " versus " x plus y ". Sometimes multiple verbalisations are possible: $P \Rightarrow Q$ can be " P implies Q " or " Q whenever P ". Sometimes the the symbolic form of a statement is not used as frequently: " p is prime" versus $p \in \mathbb{P}$. In making text flow well, the decision of when to move between symbolic and textual renderings of a mathematical proof is important. The rule-of-thumb that I have arrived at is to render the general flow of the proof's reasoning using text and to render the objects that are being reasoned about using symbols. The idea here is that one should be able to follow the rough course of argument whilst only skimming the symbolic parts of the proof.

3.6.4. *Microplanning binders with class predicate collections*

In mathematics, it is common that a variable will be introduced in a sentence and then referenced in later sentences. For example, one will often read sentences such as "Let X be a metric space and let x and y be points in X ". This corresponds to the following telescope³⁹ of binders: $(X : \text{Type}) (_ : \text{metric_space } X) (x \ y : X)$. These effectively act as 'linguistic variable binders'.

In this subsection I will highlight how to convert lists of binders to natural language phrases of this form. To the best of my knowledge this is an original contribution so I will explain this mechanism in more detail. This approach is inspired by the idea of 'notions' as first used in the *ForTheL* controlled natural language parser for the SAD project [VLP07, Paso7, VLPA08] also used by Naproche/SAD [DKL20]. Ganesalingam [Gan10] refers to these as *non-extensional types* and Ranta [Ran94] as *syntactic categories*. The act of The PROVERB system [HF97] and the G&G system [GG17] provide a mechanism for generating natural language texts using a similar technique for aggregating assumptions, however these approaches do not allow for the handling of more complex telescopes found in dependent type theory. Table 3.42 presents some examples of the kinds of translations in question.

Telescope	Generated text
$(X : \text{Type}) [\text{metric_space } X] (x \ y : X)$	Let X be a metric space and let x and y be points in X .
$(G : \text{Type}) [\text{group } G] (x \ y : G)$	Let G be a group and let x and y be elements of G .
$(G : \text{Type}) [\text{group } G] (H : \text{set } G) (h_1 : \text{subgroup.normal } G \ H)$	Let G be a group and H be a normal subgroup of G .
$(a \ b : \mathbb{Z}) (h_1 : \text{coprime } a \ b)$	Let a and b be coprime integers.
$(f : X \rightarrow Y) (h_1 : \text{continuous } f)$	Let $f : X \rightarrow Y$ be a continuous function.
$(T : \text{Type}) [\text{topological_space } T] (U : \text{set } T) (h_1 : \text{open } U)$	Let T be a topological space and let U be an open set in T .
$(\epsilon : \mathbb{R}) (h_1 : \epsilon > 0)$	Let $\epsilon > 0$.

The variable introduction sentences in Table 3.42 take the role of a variable binder for mathematical discourse. This variable is then implicitly 'in scope' until its last mention in the text. Some variables introduced in this way can remain in scope for an entire book. For example, the choice of underlying field k in a book on linear algebra. As Ganesalingam notes [Gan10 §2.5.2], "If mathematicians were not able to use variables in this way, they would need to write *extremely* long sentences!"

Let's frame the problem as follows: take as input a *telescope* of binders (e.g., $[(a : \mathbb{Z}), (b : \mathbb{Z}), (h_1 : \text{coprime } a \ b)]$) and produce a 'variable introduction text' string as shown in the above table. The problem involves a number of challenges:

- There is not a 1-1 map between binders and pieces of text: in "Let a , b be coprime", the binder $h_1 : \text{coprime } a \ b$ is not named but instead treated as a property of a and b .
- The words that are used to describe a variable can depend on which typeclass [HHPW96]⁴⁰ their type belongs to. For instance, we write "let x and y be **points**" or "let x and y be **elements of G** " depending on whether the type of x and y is an instance of `group` or `metric_space`.
- Compare " x and y are *prime*" versus " x and y are *coprime*". The first arises from $(x \ y : \mathbb{N}) (h_1 : \text{prime } x) (h_2 : \text{prime } y)$ whereas the second from $(x \ y : \mathbb{N}) (h_1 : \text{coprime } x \ y)$. Hence we need to model the adjectives "prime" and "coprime" as belonging to distinct categories.

To solve this I introduce a schema of **class predicate collections**. Each binder in the input telescope is converted to two pieces of data; the **subject expression** x and the **class predicate** cp ; which is made from one of the following constructors.

³⁹ A **telescope** is a list of binders where the type of a binder may depend on variables declared earlier in the list. Telescopes are equivalent to a well-formed context (see Section 2.1.3) but the term telescope is also used to discuss lists of binders that appear in expressions such as lambda and forall bindings.

[VLP07] **Verchinine, Konstantin; Lyaletski, Alexander; Paskevich, Andrei** *System for Automated Deduction (SAD): a tool for proof verification* (2007) International Conference on Automated Deduction

[Paso7] **Paskevich, Andrei** *The syntax and semantics of the ForTheL language* (2007) <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.211.8865&rep=rep1&type=pdf>

[VLPA08] **Verchinine, Konstantin; Lyaletski, Alexander; Paskevich, Andrei; et al.** *On correctness of mathematical texts from a logical and practical point of view* (2008) International Conference on Intelligent Computer Mathematics

[DKL20] **De Lon, Adrian; Koepke, Peter; Lorenzen, Anton** *Interpreting Mathematical Texts in Naproche-SAD* (2020) Intelligent Computer Mathematics

[Gan10] **Ganesalingam, Mohan** *The language of mathematics* (2010) PhD thesis (University of Cambridge)

[Ran94] **Ranta, Aarne** *Syntactic categories in the language of mathematics* (1994) International Workshop on Types for Proofs and Programs

[HF97] **Huang, Xiaorong; Fiedler, Armin** *Proof Verbalization as an Application of NLG* (1997) International Joint Conference on Artificial Intelligence

[GG17] **Ganesalingam, Mohan; Gowers, W. T.** *A fully automatic theorem prover with human-style output* (2017) Journal of Automated Reasoning

- `adjective`: "continuous", "prime", "positive"
- `fold_adjective`: "coprime", "parallel"
- `symbolic_postfix`: " $\in A$ ", " > 0 ", " $: X \rightarrow Y$ "
- `class_noun`: "number", "group", "points in X ", "elements of G ", "function", "open set in T "
- `none`: a failure case. For example, if the binder is just for a proposition that should be realised as an assumption instead of a predicate about the binder.

The subject expression and the class predicate for a given binder in the input telescope are assigned by consulting a lookup table which pattern-matches the binder type expressions to determine the subject expression and any additional parameters (for example T in "open set in T "). Each pair $\langle x, cp \rangle$ is mapped to $\langle [x], [cp] \rangle : \text{List Expr} \times \text{List ClassPredicate}$. I call this a **class predicate collection** (CPC). The resulting list of CPCs is then reduced by aggregating [DH93] adjacent pairs of CPCs according to (3.43).

$$\begin{aligned} \langle xs, cps \rangle, \langle ys, cps \rangle &\rightsquigarrow \langle xs ++ ys, cps \rangle \\ \langle xs, cps_1 \rangle, \langle xs, cps_2 \rangle &\rightsquigarrow \langle xs, cps_1 ++ cps_2 \rangle \end{aligned}$$

In certain cases, the merging operation can also delete class predicates that are superseded by later ones. An example is that if we have $(x : X) (h_1 : x \in A)$, this can be condensed directly to $\langle [x], [\text{symbolic_postfix } "\in A"] \rangle$ which realises to "Let $x \in A$ " instead of the redundant "Let $x \in A$ be an element of X " which violates Grice's maxim of quantity (Section 3.6.2).

Additionally, the resulting class predicate collection list is partitioned into two lists so that only the first mention of each subject appears in the first list. For example; $x : X$ and $h : x \in A$ both have the subject x , but "Let x be a point and let $x \in A$ "

These class predicate collections can then be realised for a number of binder cases:

- Let: "Let U be open in X "
- Forall: "For all U open in X "
- Exists: "For some U open in X "

`class_noun` can be compared to the concept of a 'notion' in ForTheL and Naproche/SAD and a 'non-extensional type' in Ganesalingam [Gan10]. It takes the role of a noun that the introduced variable belongs to, and is usually preceded with an indefinite article: "let x be an element of G ".

Will some mechanism like CPCs be necessary in the future, or are they a cultural artefact of the way that mathematics has been done in the past? When designing mathematical definitions in formalised mathematics, one must often make a choice in how new datatypes are defined: should there be a new type 'positive real number' or just use the real numbers \mathbb{R} and add a hypothesis $\varepsilon > 0$? In natural language mathematics, one is free to move between these bundled and unbundled representations without concern. The CPC structure reflects this; " ε is a positive real" can be interpreted as either a "real that is positive" or as a single semantic type "positive real". Natural mathematics does not disambiguate between these two because they are both equivalent within its informal rules, similar to how the representation of $a + b + c$ does not need to disambiguate between $(a + b) + c$ and $a + (b + c)$ since they are equal.

3.6.5. Handling 'multi-apply' steps

The specialised `apply` box-tactic discussed in Section 3.5.8 requires some additional processing. The `apply` box-tactic returns a datatype called `ApplyTree` that indicates how a given lemma was applied, resulting in parameters, goals and values obtained through eliminating an existential statement. These are converted in to "since" sentences:

Table 3.42. Examples of generating natural language renderings of variable introductions from type-theory telescopes. Square brackets on a binder such as `[group G]` denote a typeclass binder. This typeclass binder is equivalent to the binder `(g : group G)` where the binder name `g` is omitted. Typeclasses were first introduced by Hall *et al* for use with the Haskell programming language [HHPW96]. Typeclasses are used extensively in the Lean 3 theorem prover. A description of their implementation can be found in [MAKR15 §2.4].

[HHPW96] **Hall, Cordelia V; Hammond, Kevin; Peyton Jones, Simon L; et al.** *Type classes in Haskell* (1996) ACM Transactions on Programming Languages and Systems (TOPLAS)

[MAKR15] **de Moura, Leonardo; Avigad, Jeremy; Kong, Soonho; et al.** *Elaboration in Dependent Type Theory* (2015) CoRR

[Gan10] **Ganesalingam, Mohan** *The language of mathematics* (2010) PhD thesis (University of Cambridge)

⁴⁰ See the caption of Table 3.42 for more information on typeclasses.

[DH93] **Dalianis, Hercules; Hovy, Eduard** *Aggregation in natural language generation* (1993) European Workshop on Trends in Natural Language Generation

(3.43). Rules for aggregating class predicate collections.

[Gan10] **Ganesalingam, Mohan** *The language of mathematics* (2010) PhD thesis (University of Cambridge)

"Since f is continuous, there exists some $\delta > 0$ such that $d(f(x), f(y)) < \epsilon$ whenever $d(x, y) < \delta$ "

The code that produces this style of reasoning breaks down in to a `Reason` component indicating where the fact came from and a restatement of the fact with the parameters set to be relevant to the goal. In most cases, the `Reason` can simply be a restatement of the fact being used. However, it is also possible to produce more elaborate reasons. For example, `apply h` for some hypothesis h will also match preconditions on h if they appear in context. That is, if $h_0 : P \rightarrow Q$, then `apply h0` in the box in (3.44) will automatically include the propositional assumption $h_1 : P$ to solve the `Box`, instead of resulting in a new goal $?t_2 : P$. This will produce the reason "Since $P \rightarrow Q$ and P , we have Q ".

(3.44).

$\frac{h_0 : P \rightarrow Q \quad h_1 : P}{?t_1 : Q}$
--

3.6.6. Multiple cases

Some problems branch into multiple cases. For example, the $A \cup B$ problem. Here, some additional macroplanning needs to occur, since it usually makes sense to place each of the cases in their own paragraph. When `cases` is performed, the resulting $\#$ -box contains two separate branches for each case as discussed in (3.28).

When a new box-tactic is performed to create an `Act`, box-tactics that are performed within one of these case blocks causes the `Act` to be tagged with the case. This is then used to partition the resulting rendered string into multiple paragraphs.

3.6.7. Realisation

As shown in Figure 3.41, the set of `Acts` is compiled to a sequence of `Sentence` objects and these are converted to a run of text. As detailed in Section 2.7 this last step is called *realisation*. In the realisation phase, each sentence is converted to a piece of text containing embedded mathematics. Each statement is constructed through recursively assembling canned phrases representing each sentence. This means that longer proofs can become monotonous but the application of synonymous phrases could be used to add variation. However, the purpose of this NLG system is to produce 'human-like' reasoning and so if the proofs read as too monotonous, it suggests that less detail should have been included in the `Act` list structure.

When realising logical statements, the prose would become unnatural or ambiguous after a certain depth. After a depth of two these statements switch to being entirely symbolic. For example: $(P \rightarrow Q) \rightarrow X \rightarrow Y$ would recursively render in natural language naïvely as " Y whenever X and Q whenever P ", even with some more sophisticated algorithm to remove the clunkiness, writing " Y whenever X and $P \rightarrow Q$ " is just much clearer.

Mathematical expressions were pretty printed using Lean's pretty printing engine. However, the Lean 3 pretty printer needs a metavariable context in order to render, so it was necessary to add a tactic state object alongside the `Act` objects. It was necessary to store this context separately for each act because some metavariables would become solved through the course of the proof and cause confusing statements such as "by setting ϵ to be ϵ ", where it should read "by setting η to be ϵ ". Another printing issue was in the printing of values created through destructuring existential variables, which would be rendered as `classical.some`.

3.6.8. Summary

In this section, I detailed the workings of the natural language write-up component of the HumanProof system. I gave an overview of the standard architecture pipeline and then

discussed the areas of novelty, namely the approach to producing suitable noun-phrase string from type-theoretical telescopes and on the verbalisations multi-apply steps.

3.7. Conclusion

In this chapter, I have introduced a new `Box` development calculus for human-like reasoning and demonstrated its compatibility (Section 3.4, Appendix A) with the development calculus of the Lean theorem prover. I have outlined the structure of a set of box-tactics within this calculus that allow for the creation of both formal and natural-language proofs of this output.

I then detailed the natural language generation component of HumanProof. The component can produce readable proofs of simple lemmas. Supporting larger projects is left for future work.

In the next chapter, we will discuss a new component to enhance the `Box` system for use with equational reasoning. I will make use of the work presented in this chapter in the evaluation (Chapter 6).

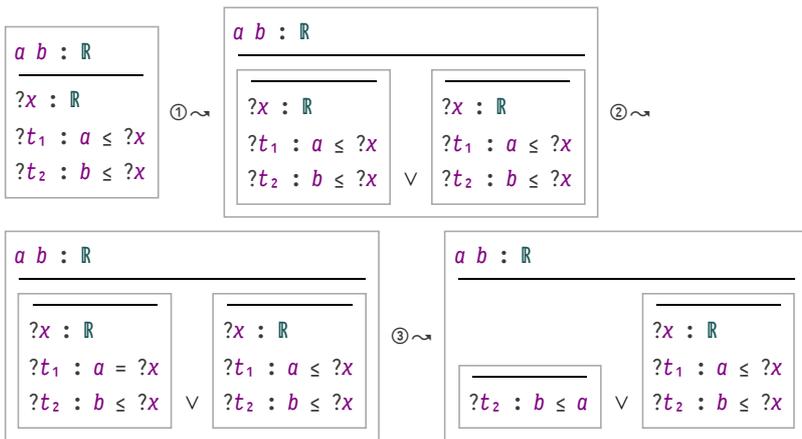
I will finish this chapter with some thoughts on future directions for the `Box` datastructure. A more general outlook on future work can be found in Section 7.2, where I also discuss potential future directions in applying deep learning to natural language generation.

3.7.1. Future work: θ -critics

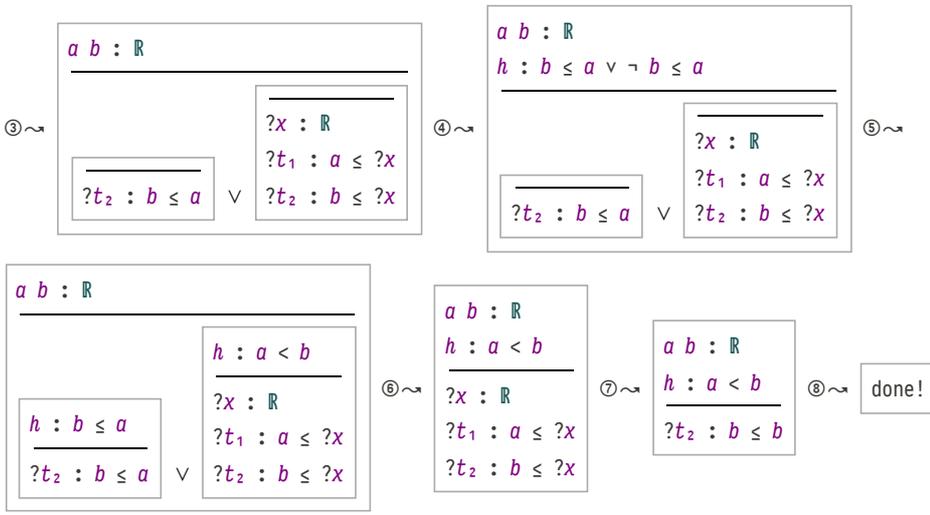
An avenue for future research is the definition of some additional box-tactics for the `Box` datastructure that allow it to work in a similar fashion to Ireland's *proofcritics* [Ire92]. Recall from Section 2.6.2 that proof critics (broadly speaking) are a proof planning technique that can revise a proof plan in light of information gained from executing a failed plan. θ -boxes can support a similar idea as I will now exemplify in (3.45), where the statement to prove is $\forall a b : \mathbb{R}, \exists x : \mathbb{R}, (a \leq x) \wedge (b \leq x)$. The proof requires spotting the trichotomy property of real numbers: $\forall x y : \mathbb{R}, x \leq y \vee y < x$, however it is difficult to see whether this will apply from the goal state.

[Ire92] **Ireland, Andrew** *The use of planning critics in mechanizing inductive proofs* (1992) International Conference on Logic for Programming Artificial Intelligence and Reasoning

(3.45). Sketch of some future work making use of θ -boxes to perform a speculative application of the lemma $a = x \rightarrow a \leq x$ (highlighted). The box-tactics are: ① θ -intro (3.36); ② apply $a = x \rightarrow a \leq x$ to the left instance of $?t_1$; ③ apply reflexivity to the left $?t_1$, causing a and $?x$ to be unified (see Section 3.5.7).



At this point, one can spot that the lefthand `Box` is no longer possible to solve unless one assumes $b \leq a$. However, rather than deleting the left-hand box, we can instead use this information as in (3.46).



(3.46). Continuation of (3.45) to perform an 'informed backtracking'. The key step is $\textcircled{4}$, the inclusion of an instance of the LEM axiom triggered by the insolubility of the goal $?t_2 : b \leq a$ on the left-hand branch of the θ box. $\textcircled{5}$ is an amalgamation of two box-tactics; \vee -cases (3.31) and θ -hoisting (A.42) as described in Definition A.39. $\textcircled{6}$ is application of $h : a \leq b$ in the left-hand box and θ -reduce₁ (3.22). $\textcircled{7}$ is an application of $\neg(b \leq a) \rightarrow a \leq b$ and $\textcircled{8}$ is an application of $b \leq b$.

The remaining research question for putting (3.45) and (3.46) into practice is to determine some heuristics when it is appropriate to perform θ -intro (step $\textcircled{2}$) and step $\textcircled{4}$, where an instance $h : b \leq a \vee \neg b \leq a$ is introduced. What is an appropriate trigger for suggesting the manoeuvre in $\textcircled{4}$, $\textcircled{5}$, $\textcircled{6}$ to the user?

Chapter 4

Subtasks

4.1. Equational reasoning

Equality chains are ubiquitous in mathematics. Here, by **equality chain**, I mean parts of proofs found in mathematical literature that consist of a list of expressions separated by $=$ or some other transitive relation symbol. The chains are chosen such that each pair of adjacent expressions are clearly equal to the reader, in the sense that the equality does not need to be explicitly justified. And hence, by transitivity, the chain shows that the first expression is equal to the last one.

For example, take some vector space V . Suppose that one wishes to prove that given a linear map $A : V \rightarrow V$, its adjoint $A^\dagger : V \rightarrow V$ is linear⁴¹. To do so one typically provides the equality chain (4.1) for all vectors $x, u, v : V$.

$$\begin{aligned} & \langle A^\dagger (u + v), x \rangle \\ &= \langle u + v, A x \rangle \\ &= \langle u, A x \rangle + \langle v, A x \rangle \\ &= \langle A^\dagger u, x \rangle + \langle A^\dagger v, x \rangle \\ &= \langle A^\dagger u, x \rangle + \langle A^\dagger v, x \rangle \\ &= \langle A^\dagger u + A^\dagger v, x \rangle \end{aligned}$$

The equations that one can compose the reasoning chain from (e.g., $\langle A^\dagger a, b \rangle = \langle a, A b \rangle$) are called **rewrite rules**. For the example (4.1), there are a large number of axiomatic rewrite rules available (4.2) and still more rules derived from these. We can formulate the equation rewriting problem for two expressions $\Gamma \vdash l = r$ as finding a path in the graph E whose vertices are expressions in Γ and whose edges are generated by a set of rewrite rules R (such as those in (4.2)). Any free variables in R are substituted with correctly typed expressions to produce **ground** rewrite rules that are then closed under symmetry, transitivity, congruence⁴².

$a + b = b + a$	$a + (b + c) = (a + b) + c$	$0 + a = a$	$a + -a = 0$	$-a + a = 0$
$a * b = b * a$	$a * (b * c) = (a * b) * c$	$1 * a = a$	$a \neq 0 \rightarrow a * a^{-1} = 1$	
$a \neq 0 \rightarrow a^{-1} * a = 1$	$a * (b + c) = a * b + a * c$	$y + x = x + y$		
$x + (y + z) = (x + y) + z$	$x + 0 = 0$	$1 \cdot x = x$	$(a + b) \cdot x = a \cdot x + b \cdot x$	
$(a * b) \cdot x = a \cdot (b \cdot x)$	$a \cdot (x + y) = a \cdot x + a \cdot y$			
$\langle u + v, x \rangle = \langle u, x \rangle + \langle v, x \rangle$	$\langle u, x + y \rangle = \langle u, x \rangle + \langle u, y \rangle$			
$a * \langle u, x \rangle = \langle a \cdot u, x \rangle$	$a * \langle u, x \rangle = \langle u, a \cdot x \rangle$	$A(x + y) = A x + A y$		
$a \cdot A x = A(a \cdot x)$	$\langle A^\dagger u, x \rangle = \langle u, A x \rangle$			

A central part of automated theorem proving (ATP) is constructing equality proofs such as (4.1) from (4.2) automatically. This can be done with well-researched techniques from the field of **term rewriting systems** [BN98]. These techniques take advantage of the fact that computers can perform many operations per second, and large search spaces can be explored quickly, though heuristic functions are still needed to prevent a combinatorial explosion. Many domains - such as checking that two expressions are equal using the ring axioms - also have specialised decision procedures available for them. I'll call these approaches to solving equalities **machine-oriented**; this contrasts with human-oriented as discussed in Section 2.6.

⁴¹ In general, the adjoint should act on the **dual space** $A^\dagger : V^* \rightarrow V^*$.

(4.1). The running example problem for this chapter. Here, $\langle _, _ \rangle : V \times V \rightarrow \mathbb{C}$ is the inner product taking a pair of vectors to a complex number.

⁴² A relation \sim is **congruent** when $x \sim y$ implies $t\{z \mapsto x\} \sim t\{z \mapsto y\}$ for all valid expressions x, y and t where t has a free variable z .

(4.2). A possible set of rewrite rules relevant to (4.1). Where $a, b, c : \mathbb{C}$ for some field \mathbb{C} ; $x, y, z : V$ for some \mathbb{C} -vector space V ; and $A : V \rightarrow V$ is a linear map in V . Note that the vector space laws are over an arbitrary vector space and so can also apply to the dual space V . This list is for illustrative purposes rather than being exhaustive: the details within an ITP can vary. For example, in Lean, there is a single commutativity rule $\forall (a, b : \alpha) [comm_monoid\ \alpha], a * b = b * a$ which applies to any type with an instance of the `comm_monoid` typeclass.

[BN98] **Baader, Franz; Nipkow, Tobias** *Term rewriting and all that* (1998) **publisher** Cambridge University Press

In accordance with the research goals of this thesis (Section 1.2), the purpose here is to investigate alternative, human-like ways of producing equality proofs. As motivated in Section 1.1, this serves the purpose of improving the usability of proof assistants by making the proofs generated more understandable (Section 2.5). The goal of this chapter is *not* to develop methods that compete with machine-oriented techniques to prove more theorems or prove them faster. Instead, I want to focus on the abstract reasoning that a human mathematician typically carries out when they encounter an equality reasoning problem such as (4.1).

With this in mind, the goal of this chapter is to create an algorithm which:

- can solve simple equality problems of the kind that an undergraduate might find easy;
- does not encode any domain-specific knowledge of mathematics. That is, it does not invoke specialised procedures if it detects that the problem lies in a particular domain such as [Presburger arithmetic](#);
- is efficient in the sense that it does not store a large state and does not perform a significant search when a human would not.

Typically, existing ATP methods do not scale well with the number of competing rules introduced, as one would expect of algorithms that make use of significant amounts of brute-force search. If we can devise new architectures that solve simple equalities with less search, then it may be possible to scale up these techniques to larger problems and improve the efficiency of established ATP methods.

This chapter presents the `subtask` algorithm which has some success with respect to the above goals. The algorithm is written in Lean 3 [MKA+15] and can be found [on GitHub](#). The work in this chapter also appears as a paper published in KI 2019 [AGJ19]. In the remainder of the chapter I give a motivating example (Section 4.2) followed by a description of the algorithm (Section 4.3). The algorithm is then contrasted with existing approaches (Section 4.4) and evaluated against the above goals (Section 4.5).

4.2. Example

Let us begin with a motivating example (4.1) in elementary linear algebra. We have to solve the goal of the `Box` (4.3) using the rewrite rules given in (4.2).

$$\begin{array}{l}
 V : \text{VectorSpace } \mathbb{C} \\
 x \ u \ v : V \\
 A : V \rightarrow V \\
 \hline
 \langle A \dagger (u + v), x \rangle = \langle A \dagger u + A \dagger v, x \rangle
 \end{array}$$

To do this, a human's proving process might proceed as follows:

[MKA+15] **de Moura, Leonardo; Kong, Soonho; Avigad, Jeremy; et al.** *The Lean theorem prover (system description)* (2015) International Conference on Automated Deduction

[AGJ19] **Ayers, E. W.; Gowers, W. T.; Jamnik, Mateja** *A human-oriented term rewriting system* (2019) KI 2019: Advances in Artificial Intelligence - 42nd German Conference on AI

(4.3). The `Box` representing the task to solve in this instance. Full detail on `Box` is given in Chapter 3. For the purposes of this chapter, a `Box` represents a theorem to prove with a list of variables and hypotheses above the line and a goal proposition to prove below the line.

- ① I need to create the expression $\langle A^\dagger u + A^\dagger v, x \rangle$.
- ② In particular, I need to make the subexpressions $A^\dagger u$ and $A^\dagger v$. Let's focus on $A^\dagger u$.
- ③ The only sensible way I can get this is to use the definition $\langle u, A^\dagger z \rangle = \langle A^\dagger u, z \rangle$, presumably with $z = x$.
- ④ In particular, I'll need to make the subterm $A^\dagger z$ for some z .
- ⑤ I can do that straight away: $\langle A^\dagger (u + v), x \rangle = \langle u + v, A^\dagger x \rangle$ using the rewrite rule $\forall w z, \langle A^\dagger w, z \rangle = \langle w, A^\dagger z \rangle$.
- ⑥ Now I'm in a position to obtain the subexpression $\langle u, A^\dagger x \rangle$ I wanted in step 3, so let me do that using bilinearity: $\langle u + v, A^\dagger x \rangle = \langle u, A^\dagger x \rangle + \langle v, A^\dagger x \rangle$.
- ⑦ And now I can get the subexpression $A^\dagger u$ I wanted even earlier in step 2, so let me do that: $\langle u, A^\dagger x \rangle + \langle v, A^\dagger x \rangle = \langle A^\dagger u, x \rangle + \langle v, A^\dagger x \rangle$.
- ⑧ In step 2 I also wanted to create $A^\dagger v$, which I can now get too: $\langle A^\dagger u, x \rangle + \langle v, A^\dagger x \rangle = \langle A^\dagger u, x \rangle + \langle A^\dagger v, x \rangle$.
- ⑨ And with one application of bilinearity I'm home: $\langle A^\dagger u, x \rangle + \langle A^\dagger v, x \rangle = \langle A^\dagger u + A^\dagger v, x \rangle$.

List 4.4. A sketch of a human's possible thought process when constructing an equality proof for (4.3).

The key aspect of the thought process in List 4.4 is the setting of intermediate aims, such as obtaining certain subexpressions when one does not immediately see how to obtain the entire expression. Let's do this by creating a tree of subtasks Figure 4.5.

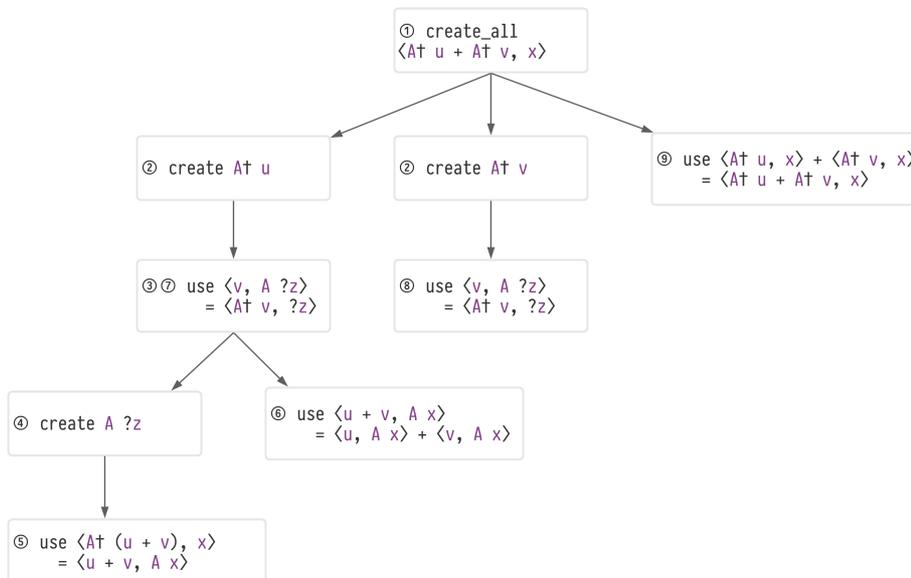


Figure 4.5. The subtask tree for solving (4.3):

$\langle A^\dagger (u + v), x \rangle = \langle A^\dagger u + A^\dagger v, x \rangle$. Circled numbers correspond to steps in List 4.4, so the 'focus' of the algorithm travels around the tree as it progresses. Details on how this tree is generated will follow in Section 4.3.

The tree in Figure 4.5 represents what the algorithm does with the additivity-of-adjoint problem (4.3). It starts with the subtask $\text{create_all } \langle A^\dagger u + A^\dagger v, x \rangle$ at ①. Since it cannot achieve that in one application of an available rule, it creates a set of subtasks and then chooses the one that is most promising: later in Section 4.3, I will explain how it generates and evaluates possible subtasks. In this case the most promising subtask is $\text{create } A^\dagger u$, so it selects that in ② and identifies a rewrite rule - the definition of adjoint: $\forall w z, \langle A^\dagger w, z \rangle = \langle w, A^\dagger z \rangle$ - that can achieve it; adding $\text{use } \langle u, A^\dagger z \rangle = \langle A^\dagger u, z \rangle$ to the tree at ③. The z that appears at ③ in Figure 4.5 is a metavariable⁴³ that will in due course be assigned to x . Now the process repeats on ③, a set of subtasks are again created for the lhs of $\langle u, A^\dagger z \rangle = \langle A^\dagger u, z \rangle$ and the subtask $\text{create } A^\dagger z$ is selected (④). Now, there does exist a rewrite rule that will achieve $\text{create } A^\dagger z$: $\langle A^\dagger (u + v), x \rangle = \langle u + v, A^\dagger x \rangle$, so this is applied and now the algorithm iterates back up the subtask tree, testing whether the new expression $\langle u + v, A^\dagger x \rangle$ achieves any of the subtasks and whether any new rewrite rules can be used to achieve them.

⁴³ That is, a placeholder for an expression to be chosen later. See Section 2.4 for background information on metavariables.

In the next section, I will provide the design of an algorithm that behaves according to these motivating principles.

4.3. Design of the algorithm

The subtasks algorithm may be constructed as a search over a directed graph \mathbb{S} .

The `subtask` algorithm's state $s : \mathbb{S}$ has three components $\langle t, f, c \rangle$:

- a tree `ts : Tree(Task)` of **tasks** (as depicted in Figure 4.5)
- a 'focussed' node `f : Address(ts)` in the tree⁴⁴.
- an expression `c : Expr` called the **current expression** (CE) which represents the current left-hand-side of the equality chain. The subtasks algorithm provides the edges between states through sound manipulations of the tree and current expression. Each task in the tree corresponds to a predicate or potential rule application that could be used to solve an equality problem.

⁴⁴ `ts` and `f` are implemented as a zipper [Hue97]. Zippers are described in Appendix A.

Given an equational reasoning problem $\Gamma \vdash l = r$, the initial state $s_0 : \mathbb{S}$ consists of a tree with a single root node `CreateAll r` and a CE `l`. We reach a goal state when the current expression `c` is definitionally equal⁴⁵ to `r`.

⁴⁵ That is, the two expressions are equal by reflexivity.

The first thing to note is that if we project $s : \mathbb{S}$ to the current expression `c`, then we can recover the original equational rewriting problem \mathbb{E} by taking the edges to be all possible rewrites between terms. One problem with searching this space is that the number of edges leaving an expression can be infinite⁴⁶. The typical way that this problem is avoided is to first *ground* all available rewrite rules by replacing all free variables with relevant expressions. The subtasks algorithm does not do this, because this is not a step that humans perform when solving simple equality problems. Even after grounding, the combinatorial explosion of possible expressions makes \mathbb{E} a difficult graph to search without good heuristics. The subtasks algorithm makes use of the subtasks tree `t` to guide the search in \mathbb{E} in a manner that is intended to match the process outlined in List 4.4 and Figure 4.5.

⁴⁶ For example, the rewrite rule $\forall a b, a = a - b + b$ can be applied to any expression with any expression being substituted for `b`.

A task `t : Task` implements the following three methods:

- `refine : Task → (List Task)` creates a list of **subtasks** for a given task. For example, a task `create (x + y)` would refine to subtasks `create x`, `create y`. The refinement may also depend on the current state s of the tree and CE. The word 'refinement' was chosen to reflect its use in the classic paper by Kambhampati [KKY95]; a refinement is the process of splitting a set of candidate solutions that may be easier to check separately.
- `test : Task → Expr → Bool` which returns true when the given task `t : Task` is **achieved** for the given current expression `e : Expr`. For example, if the current expression is `4 + x`, then `create x` is achieved. Hence, each task may be viewed as a predicate over expressions.
- Optionally, `execute : Task → Option Rewrite` which returns a `Rewrite` object representing a rewrite rule from the current expression `ci` to some new expression `ci+1` (in the context Γ) by providing a proof `ci = ci+1` that is checked by the prover's kernel. Tasks with `execute` functions are called **strategies**. In this case, `test` must return true when `execute` can be applied successfully, making `test` a *precondition* predicate for `execute`. As an example, the `use (x = y)` task performs the rewrite `x = y` whenever the current expression contains an instance of `x`.

[KKY95] **Kambhampati, Subbarao; Knoblock, Craig A; Yang, Qiang** *Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning* (1995) Artificial Intelligence

This design enables the system to be modular, where different sets of tasks and strategies can be included. Specific examples of tasks and strategies used by the algorithm are given in `{#the-main-subtasks}`. Given a state $s : \mathbb{S}$, the edges leading from s are generated using the flowchart shown in Figure 4.6.

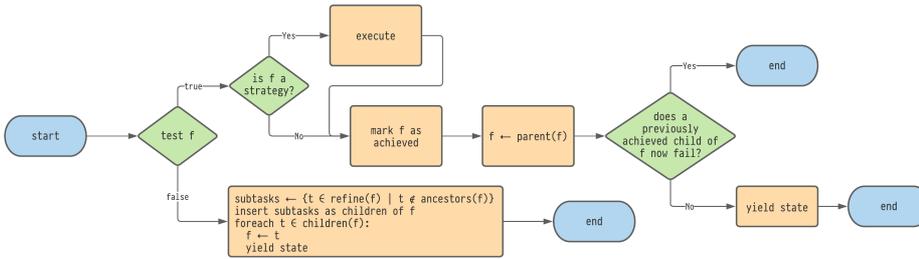
Let `f` be the focussed subtask for s . In the case that `test(f)` is true the algorithm 'ascends' the task tree. In this branch, `f` is tagged as 'achieved' and the new focussed task is set as parent of `f`. Then, it is checked whether any siblings of `f` that were marked as achieved are no longer achieved (that is, there is a sibling task `t` tagged as achieved but

$\text{test}(t)$ is now false). The intuition behind this check on previously achieved subtasks is that once a sibling task is achieved, it should not be undone by a later step because the assumption is that all of the child subtasks are needed before the parent task can be achieved.

In the case that $\text{test}(f)$ is false, meanwhile, the algorithm 'explores' the task tree by finding new child subtasks for f . To do this, $\text{refine}(f)$ is called to produce a set of candidate subtasks for f . For each $t \in \text{refine}(f)$, t is inserted as a child of f provided that $\text{test}(t)$ is false and t does not appear as an ancestor of f . Duplicate children are removed. Finally, for each subtask t , a new state is yielded with the focus now set to t . Hence s 's outdegree⁴⁷ in the graph will be the number of children that f has after refining.

⁴⁷ The outdegree of a vertex v in a directed graph is the number of edges leaving v .

Figure 4.6. Flowchart for generating edges for a starting state $s : S$. Here, each call to `yield state` will produce another edge leading from s to the new state.



Now that the state space S , the initial state s_0 , the goal states and the edges on S are defined, we can perform a search on this graph with the help of a heuristic function $h : S \rightarrow [0, \infty]$ to be discussed in Section 4.3.2. The subtasks algorithm uses greedy best-first search with backtracking points. However, other graph-search algorithms such as A* or hill-climbing may be used.

4.3.1. The defined subtasks

In this section I will provide a list of the subtasks that are implemented in the system and some justification for their design.

4.3.1.1. `create_all e`

The `create_all : Expr → Task` task is the root task of the subtask tree.

- **Refine:** returns a list of `create b` subtasks where each b is a minimal subterm of e not present in the current expression.
- **Test:** true whenever the current expression is e . If this task is achieved then the subtasks algorithm has succeeded in finding an equality chain.
- **Execute:** none.

The motivation behind the refinement rule is that since b appears in e but not in the current expression, then it must necessarily arise as a result of applying a rewrite rule. Rather than include every subterm of e with this property, we need only include the minimal subterms with this property since if $b < b'$, then $\text{test}(\text{create } b) \Leftarrow \text{test}(\text{create } b')$. In the running example (4.3), the subtasks of `create_all <A† u + A† v, x>` are `create (A† u)` and `create (A† v)`.

4.3.1.2. `create e`

The `create` task is achieved if the current expression contains e .

- **Refine:** returns a list `use (a = b)` subtasks where e overlaps with b (see further discussion below). It can also return `reduce_distance` subtasks in some circumstances.
- **Test:** true whenever the current expression is e .
- **Execute:** none.

Given a rewrite rule $r : \forall (..xs), a = b$, say that an expression e overlaps with the right hand side of r when there exists a most-general substitution σ on r 's variables xs such that

- e appears in $\sigma(b)$;
- e does not appear in $\sigma(a)$;
- e does not appear in σ . This last condition ensures that the expression comes about as a result of the term-structure of the rule r itself rather than as a result of a substitution. The process of determining these rules is made efficient by storing the available rewrite rules in a term indexed datastructure [SRV01].

Additionally, as mentioned, `create e` can sometimes refine to yield a `reduce_distance` subtask. The condition for this depends on the distance between two subterms in a parent expression $c : Expr$, which is defined as the number of edges between the roots of the subterms -- viewing c 's expression tree as a graph. If two local variables x, y are present exactly once in both the current expression and e , and the distance between them is greater in the current expression, then `reduce_distance x y` is included as a subtask.

In order to handle cases where multiple copies of e are required, `create` has an optional count argument that may be used to request an n th copy of e .

4.3.1.3. use (a = b)

This is the simplest strategy. It simply represents the subtask of using the given rewrite rule.

- **Refine:** Returns a list of `create e` subtasks where each e is a minimal subterm of a not present in the current expression. This is the same refinement rule that is used to create subtasks of the `create_all` task.
- **Test:** True whenever the rule $a = b$ can be applied to the current expression.
- **Execute:** Apply $a = b$ to the current expression. In the event that it fails (for example if the rule application causes an invalid assignment of a metavariable) then the strategy fails.

4.3.1.4. reduce_distance (x, y)

`reduce_distance` is an example of a greedy, brute-force strategy. It will perform any rewrite rule that moves the given variables closer together and then terminate.

- **Refine:** returns the empty list. That is, there are no subtasks available.
- **Test:** True whenever there is only one instance of x and y in the current expression and there exists a rewrite rule that will move x closer to y .
- **Execute:** repeatedly applies rewrite rules greedily moving x and y closer together, terminating when they can move no further together.

4.3.2. Heuristics

In this section I present the heuristic function developed for the subtasks algorithm. The ideas behind this function are derived from introspection on equational reasoning and some degree of trial and error on a set of equality problems.

There are two heuristic functions that are used within the system, an individual strategy heuristic and an 'overall-score' heuristic that evaluates sets of child strategies for a particular task. Overall-score is used on tasks which are not strategies by performing a lookahead of the child strategies of the task. The child strategies $S_1, S_2 \dots$ are then scored individually through a scoring system, scoring higher if they:

- achieve a task higher in the task tree;
- achieve a task on a different branch of the task tree;

[SRV01] Sekar, R;
Ramakrishnan, I.V.;
Voronkov, Andrei *Term
Indexing* (2001) Handbook of
automated reasoning

- have a high degree of term overlap with the current expression. This is measured using symbol counting and finding largest common subterms;
- use local variables and hypotheses;
- can be achieved in one rewrite step from the current expression.

The intuition behind all of these is to give higher scores to strategies that are salient in some way, either by containing subterms that are present in the current expression or because other subtasks are achieved.

From these individual scores, the score for the parent task of $S_1, S_2 \dots$ is computed as follows: If there is only one strategy then it scores 10. If there are multiple strategies, it discards any scoring less than -5. If there are positive-scoring strategies then all negative-scoring strategies are discarded. The overall score is then set to be 5 minus the number of strategies in the list. The intention of this procedure is that smaller sets of strategies should be preferred, even if their scores are bad because it limits choice in what to do next.

The underlying idea behind the overall-scoring heuristic is that often the first sensible strategy found is enough of a signpost to solve simple problems. That is, once one has found one plausible strategy of solving a simple problem it is often fruitful to stop looking for other strategies which achieve the same thing and to get on with finding a way of performing the new strategy.

4.3.3. Properties of the algorithm

The subtasks algorithm is sound provided sound rewrite rules are produced by the function `execute : Task → Option Rewrite`. That is, given an equation to solve $\Gamma \vdash l = r$ and given a path $s_0 \rightsquigarrow s_1 \rightsquigarrow \dots \rightsquigarrow s_n$ in S where s_0 is the initial state defined in `By` forgetting the subtask tree, a solution path in S can be projected to a solution path in E , the equational rewriting graph. This projected path is exactly a proof of $l = r$; it will be composed of a sequence $l \equiv c_0 = c_1 = \dots = c_n \equiv r$ where c_i is the current expression of s_i . Each equality in the chain holds either by the assumption of the proofs returned from `execute` being sound or by the fact that the current expression doesn't change between steps otherwise.

The next question to ask is whether S is complete with respect to E . That is, does S contain a path to the solution whenever E contains one? The answer to this depends on the output of `refine`. If `refine` always returns an empty list of subtasks then S is not complete, because no subtasks will ever be executed. The set of subtasks provided in Section 4.3.1 are not complete. For example the problem $1 - 1 = x + -x$ will not solve without additional subtasks since the smallest non-present subterm is x , so `create x` is added which then does not refine further using the procedure in Section 4.3.1. In Section 4.6 I will discuss some methods to address this.

4.4. Qualitative comparison with related work

There has been a substantial amount of research on the automation of solving equality chain problems over the last decade. The approach of the subtasks algorithm is to combine these rewriting techniques with a hierarchical search. In this section I compare subtasks which with this related work.

4.4.1. Term Rewriting

One way to find equality proofs is to perform a graph search using a heuristic. This is the approach of the `rewrite-search` algorithm by Hoek and Morrison [HM19], which uses the heuristic of string edit-distance between the strings' two pretty-printed expressions. The `rewrite-search` algorithm does capture some human-like properties in the heuristic, since

the pretty printed expressions are intended for human consumption. The subtasks algorithm is different from `rewrite-search` in that the search is guided according to achieving sequences of tasks. Since both subtasks and `rewrite-search` are written in Lean, some future work could be to investigate a combination of both systems.

A term rewriting system (TRS) R is a set of oriented rewrite rules. There are many techniques available for turning a set of rewrite rules in to procedures that check whether two terms are equal. One technique is **completion**, where R is converted into an equivalent TRS R' that is **convergent**. This means that any two expressions a, b are equal under R if and only if repeated application of rules in R' to a and b will produce the same expression. Finding equivalent convergent systems, if not by hand, is usually done by finding decreasing orderings on terms and using Knuth-Bendix completion [KB70]. When such a system exists, automated rewriting systems can use these techniques to quickly find proofs, but the proofs are often overly long and needlessly expand terms.

Another method is rewrite tables, where a lookup table of representatives for terms is stored in a way that allows for two terms to be matched through a series of lookups.

Both completion and rewrite tables can be considered machine-oriented because they rely on large datastructures and systematic applications of rewrite rules. Such methods are certainly highly useful, but they can hardly be said to capture the process by which humans reason.

Finally, there are many normalisation and decision procedures for particular domains, for example on rings [GM05]. Domain specific procedures do not satisfy the criterion of generality given in Section 4.1.

4.4.2. Proof Planning

Background information on proof planning is covered in Section 2.6.2.

The subtasks algorithm employs a structure that is similar to a **hierarchical task network** (HTN) [Sac74, Tat77, MS99]. The general idea of a hierarchical task network is to break a given abstract task (e.g., "exit the room") in to a sequence of subtasks ("find a door" then "go to door" then "walk through the door") which may themselves be recursively divided into subtasks ("walk through the door" may have a subtask of "open the door" which may in turn have "grasp doorhandle" until bottoming out with a ground actuation such as "move index finger 10°"). This approach has found use for example in the ICARUS robotics architecture [CL18, LCT08]. HTNs have also found use in proof planning [MS99].

The main difference between the approach used in the subtasks algorithm and proof planning and hierarchical task networks is that the subtasks algorithm is greedier: the subtasks algorithm generates enough of a plan to have little doubt what the first rewrite rule in the sequence should be, and no more. I believe that this reflects how humans reason for solving simple problems: favouring just enough planning to decide on a good first step, and then planning further only once the step is completed and new information is revealed.

A difference between HTNs and subtasks is that the chains of subtasks do not necessarily reach a ground subtask (for subtasks this is a rewrite step that can be performed immediately). This means that the subtasks algorithm needs to use heuristics to determine whether it is appropriate to explore a subtask tree or not instead of relying on the task hierarchy eventually terminating with ground tasks. The subtasks algorithm also inherits all of the problems found in hierarchical planning: the main one being finding heuristics for determining whether a subtask should be abandoned or refined further. The heuristics given in Section 4.3.2 help with this but there are plenty more ideas from the hierarchical task planning literature that could be incorporated also. Of particular interest for me are the applications of hierarchical techniques from the field of reinforcement learning⁴⁸.

[HM19] **Hoek, Keeley; Morrison, Scott** *lean-rewrite-search GitHub repository* (2019) <https://github.com/semorri/lean-rewrite-search>

[KB70] **Knuth, Donald E; Bendix, Peter B** *Simple word problems in universal algebras* (1970) *Computational Problems in Abstract Algebra*

[GM05] **Grégoire, Benjamin; Mahboubi, Assia** *Proving equalities in a commutative ring done right in Coq* (2005) *International Conference on Theorem Proving in Higher Order Logics*

[Sac74] **Sacerdoti, Earl D** *Planning in a hierarchy of abstraction spaces* (1974) *Artificial intelligence*

[Tat77] **Tate, Austin** *Generating project networks* (1977) *Proceedings of the 5th International Joint Conference on Artificial Intelligence*

[MS99] **Melis, Erica; Siekmann, Jörg** *Knowledge-based proof planning* (1999) *Artificial Intelligence*

[CL18] **Choi, Dongkyu; Langley, Pat** *Evolution of the ICARUS cognitive architecture* (2018) *Cognitive Systems Research*

[LCT08] **Langley, Pat; Choi, Dongkyu; Trivedi, Nishant** *Icarus user's manual* (2008)

[MS99] **Melis, Erica; Siekmann, Jörg** *Knowledge-based proof planning* (1999) *Artificial Intelligence*

4.5. Evaluation

The ultimate motivation behind the subtasks algorithm is to make an algorithm that behaves as a human mathematician would. I do not wish to claim that I have fully achieved this, but we can evaluate the algorithm with respect to the general goals that were given in Section 4.1.

- Scope: can it solve simple equations?
- Generality: does it avoid techniques specific to a particular area of mathematics?
- Reduced search space: does the algorithm avoid search when finding proofs that humans can find easily without search?
- Straightforwardness of proofs: for easy problems, does it give a proof that an experienced human mathematician might give?

The method of evaluation is to use the algorithm implemented as a tactic in Lean on a library of thirty or so example problems. This is not large enough for a substantial quantitative comparison with existing methods, but we can still investigate some properties of the algorithm. The source code also contains many examples which are outside the abilities of the current implementation of the algorithm. Some ways to address these issues are discussed in Section 4.6.

Table 4.7 gives some selected examples. These are all problems that the algorithm can solve with no backtracking.

Problem	Steps	Location
$\begin{array}{l} h : a \\ l s : \text{List } a \\ \text{rev}(l ++ s) = \text{rev}(s) ++ \text{rev}(l) \\ \forall a, \text{rev}(a :: l) = \text{rev}(l) ++ [a] \\ \hline \text{rev}(h :: l ++ s) = \text{rev}(s) ++ \text{rev}(h :: l) \end{array}$	5	<code>datatypes.lean/rev_app_rev</code>
$\begin{array}{l} A : \text{Monoid} \\ a : A \\ m n : \mathbb{N} \\ a^{\wedge} (m + n) = a^{\wedge} m * a^{\wedge} n \\ \hline a^{\wedge} (\text{succ}(m) + n) = a^{\wedge} \text{succ}(m) * a^{\wedge} n \end{array}$	8	<code>groups.lean/my_pow_add</code>
$\begin{array}{l} R : \text{Ring} \\ a b c d e f : R \\ a * d = c * d \\ c * f = e * b \\ \hline d * (a * f) = d * (e * b) \end{array}$	9	<code>rat.lean</code>
$\begin{array}{l} R : \text{Ring} \\ a b : R \\ \hline (a + b) * (a + b) = a * a + 2 * (a * b) + b * b \end{array}$	7	<code>rings.lean/sumsq_with_equate</code>
$\begin{array}{l} B C X : \text{set} \\ \hline X \setminus (B \cup C) = (X \setminus B) \setminus C \end{array}$	4	<code>sets.lean/example_4</code>

From this Table 4.7 we can see that the algorithm solves problems from several different domains. I did not encode any decision procedures for monoids or rings. In fact I did not even include reasoning under associativity and commutativity, although I am not in principle against extending the algorithm to do this. The input to the algorithm is a list of over 100 axioms and equations for sets, rings, groups and vector spaces which can be found in the file `equate.lean` in the source code⁴⁹. Thus, the algorithm exhibits considerable generality.

⁴⁸ A good introductory text to modern reinforcement learning is *Reinforcement Learning: An Introduction* by Sutton and Barto [SB18b]. Readers wishing to learn more about hierarchical reinforcement learning may find [this survey article by Flet-Berliac](#) to be a good jumping-off point [Fle19].

[SB18b] **Sutton, Richard S; Barto, Andrew G** *Reinforcement learning: An introduction* (2018) **publisher** MIT press

[Fle19] **Flet-Berliac, Yannis** *The Promise of Hierarchical Reinforcement Learning* (2019) The Gradient

Table 4.7. `subtask`'s performance on some example problems. **Steps** gives the number of rewrite steps in the final proof. **Location** gives the file and declaration name of the example in the source code.

⁴⁹ <https://github.com/EdAyers/lean-subtask>

All of the solutions to the above examples are found without backtracking, which adds support to the claim that the subtasks algorithm requires less search. There are, however, other examples in the source where backtracking occurs.

The final criterion is that the proofs are more straightforward than those produced by machine-oriented special purpose tactics. This is a somewhat subjective measure but there are some proxies that indicate that `subtasks` can be used to generate simpler proofs.

To illustrate this point, consider the problem of proving

$(x + y)^2 + (x + z)^2 = (z + x)^2 + (y + x)^2$ within ring theory. I choose this example because it is easy for a human to spot how to do it with three uses of commutativity, but it is easy for a program to be led astray by expanding the squares. `subtask` proves this equality with 3 uses of commutativity and with no backtracking or expansion of the squares. This is an example where domain specific tactics do worse than `subtask`, the `ring` tactic for reasoning on problems in commutative rings will produce a proof by expanding out the squares. The built-in tactics `ac_refl` and `blast` in Lean which reason under associativity and commutativity both use commutativity 5 times. If one is simply interested in verification, then such a result is perfectly acceptable. However, I am primarily interested in modelling how humans would solve such an equality, so I want the subtasks algorithm not to perform unnecessary steps such as this.

It is difficult to fairly compare the speed of `subtask` in the current implementation because it is compiled to Lean bytecode which is much slower than native built-in tactics that are written in C++. However it is worth noting that, even with this handicap, `subtask` takes 1900ms to find the above proof whereas `ac_refl` and `blast` take 600ms and 900ms respectively.

There are still proofs generated by `subtask` that are not straightforward. For example, the lemma $(x * z) * (z^{-1} * y) = x * y$ in group theory is proved by `subtask` with a superfluous use of the rule $e = x * x^{-1}$.

4.6. Conclusions and Further Work

In this chapter, I introduced a new, task-based approach to finding equalities in proofs and provided a demonstration of the approach by building the `subtask` tactic in Lean. I show that the approach can solve simple equality proofs with very little search. I hope that this work will renew interest in proof planning and spark interest in human-oriented reasoning for at least some classes of problems.

In future work, I wish to add more subtasks and better heuristics for scoring them. The framework I outlined here allows for easy experimentation with such different sets of heuristics and subtasks. In this way, I also wish to make the subtask framework extensible by users, so that they may add their own custom subtasks and scoring functions.

Another possible extension of the subtasks algorithm is to inequality chains. The subtasks algorithm was designed with an extension to inequalities in mind, however there are some practical difficulties with implementing it. The main difficulty with inequality proofs is that congruence must be replaced by appropriate monotonicity lemmas. For example, 'rewriting' $x + 2 \leq y + 2$ using $x < y$ requires the monotonicity lemma $\forall x y z, x \leq y \rightarrow x + z \leq y + z$. Many of these monotonicity lemmas have additional goals that need to be discharged such as $x > 0$ in $y \leq z \rightarrow x * y \leq x * z$, and so the subtasks algorithm will need to be better integrated with a prover before it can tackle inequalities.

There are times when the algorithm fails and needs guidance from the user. I wish to study further how the subtask paradigm might be used to enable more human-friendly interactivity than is currently possible. For example, in real mathematical textbooks, if an equality step is not obvious, a relevant lemma will be mentioned. Similarly, I wish to investigate ways of passing 'hint' subtasks to the tactic. For example, when proving

$x * y = (x * z) * (z^{-1} * y)$, the algorithm will typically get stuck (although it can solve the flipped problem), because there are too many ways of creating z . However, the user - upon seeing subtask get stuck - could steer the algorithm with a suggested subtask such as create $(x * (z * z^{-1}))$.

Using subtasks should help to give better explanations to the user. The idea of the subtasks algorithm is that the first set of strategies in the tree roughly corresponds to the high-level actions that a human would first consider when trying to solve the problem. Thus, the algorithm could use the subtask hierarchy to determine when no further explanation is needed and thereby generate abbreviated proofs of a kind that might be found in mathematical textbooks.

Another potential area to explore is to perform an evaluation survey where students are asked to determine whether an equality proof was generated by the software or a machine.

Chapter 5

A graphical user interface framework for formal verification

In this chapter I present the 'ProofWidgets'⁵⁰ framework for implementing general user interfaces within an interactive theorem prover. The framework uses web technology and functional reactive programming (FRP), as well as the metaprogramming features of advanced interactive theorem proving (ITP) systems such as Lean [EUR+17] to allow arbitrary interactive graphical user interfaces (GUIs) to represent the goal state of a theorem prover. Users of the framework can create user interfaces declaratively within the ITP's metaprogramming language, without having to develop in multiple languages and without coordinated changes across multiple projects, which improves development time for new designs. The ProofWidgets framework also allows GUIs to make use of the full context of the theorem prover and the specialised libraries that ITP systems offer, such as methods for dealing with expressions and tactics. The framework also includes an extensible structured pretty-printing engine that enables advanced interaction with expressions such as interactive term rewriting.

I have created an implementation of this framework: the ProofWidgets framework for the [leanprover-community fork of Lean 3](#). Throughout the chapter I will use this implementation as an illustration of the principles of the framework. I also provide a practical tutorial to get started with the ProofWidgets framework in Appendix B.

The research contributions of this chapter are:

- A new and general framework for creating portable, web-based, graphical user interfaces within a theorem prover.
- A functional API⁵¹ for creating widgets within the meta-programming framework of a theorem prover.
- An implementation of this framework for the Lean theorem prover.
- A new representation of structured expressions for use with ProofWidgets.
- A description and implementation of a goal-state widget used to interactively show and explore goal states within the Lean theorem prover.

A paper based on the content of this chapter has been published in [ITP 2021](#). I would like to thank Gabriel Ebner, Brian Gin-ge Chen, and Daniel Fabian for helping with and reviewing the changes to Lean 3 and the [VSCode extension](#) needed to make the implementation possible. I would also like to thank Robert Y. Lewis; Markus Himmel; Minchao Wu; Kendall Frey; Patrick Massot; and Angela Li for providing feedback and creating their own ProofWidgets (shown in Figure 5.18).

Section 5.1 provides a survey of existing user interfaces for theorem provers. Section 5.2 provides some additional information on the structure of web-apps and GUI frameworks. I state the research goals of ProofWidgets and how the system addresses research questions of the thesis in Section 5.3. Section 5.4 details the abstract architecture of the framework. Section 5.5 presents a closer look at the mechanism for interactive pretty printing. Section 5.6 compares ProofWidgets to related systems introduced in Section 5.1. Section 5.7 provides an overview of some of the practical considerations to do with implementation within a theorem prover, and in particular Lean. Section 5.8 details how

⁵⁰ In general software parlance, a [widget](#) is a graphical component from which to build GUIs. It can also sometimes mean a small piece of interactive user interface that is embedded in another application, for example iPhone widgets.

[EUR+17] **Ebner, Gabriel; Ullrich, Sebastian; Roesch, Jared; et al.** *A metaprogramming framework for formal verification* (2017) Proceedings of the ACM on Programming Languages

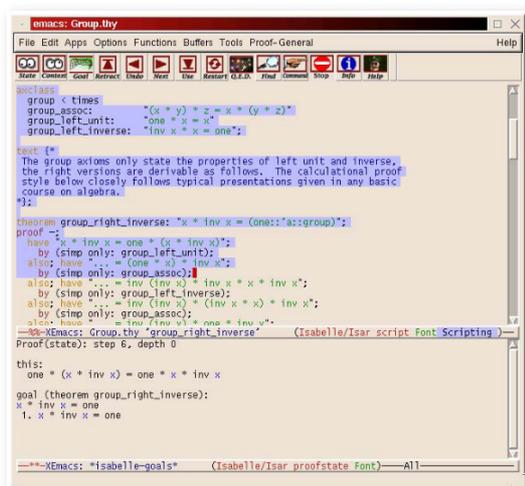
⁵¹ Application programming interface. A set of protocols to allow two applications to communicate with each other.

ProofWidgets are applied to the task of creating a user interface for HumanProof. Section 5.9 looks at ways in which the system may be extended in the future.

5.1. Survey of user interfaces for provers

Here, I review user interfaces of theorem provers. One research question in Section 1.2 is to investigate how human-like reasoning can be enabled through the use of interactive graphical user interfaces (GUIs). The field of ITP has a rich history of using graphical user interfaces to represent and interact with proofs and expressions. Here I will provide a brief review of these. The background covered in this section will then be picked up in Chapter 5, where I introduce my own GUI framework for ITP.

An early general user interface for interactive proving was Aspinall's *Proof General* [Asp00, ALW07]. This took the form of an Emacs extension that offered a general purpose API⁵² for controlling proof assistants such as Isabelle. A typical Proof General session would make use of two text buffers: the proof script buffer and the goal state buffer (see Figure 5.1). Users type commands in to the script buffer, and observe changes in the goal state buffer. This two-panel setup remains the predominant workflow for proof assistants today. The two-buffers method has stood the test of time, and so I keep this design in pursuit of my research goals. However, I modernise the goal state buffer and make it able to render non-textual user interfaces such as graphs and plots. Proof General also offers the ability to perform interaction with the goal state, for example 'proof-by-pointing' with subexpressions in the output window.



[Asp00] **Aspinall, David** *Proof General: A generic tool for proof development* (2000) International Conference on Tools and Algorithms for the Construction and Analysis of Systems

[ALW07] **Aspinall, David; Lüth, Christoph; Winterstein, Daniel** *A framework for interactive proof* (2007) Towards Mechanized Mathematical Assistants

⁵² Application programming interface. A set of protocols to allow two applications to communicate with each other.

Figure 5.1. Proof General with an Isabelle/Isar file open. The top buffer is the theory sourcefile and the lower buffer is the goal state. Image source: [Asp00].

The idea of proof-by-pointing will play a key role in Section 5.5. It was first described by Bertot and Théry [BT98]. Proof-by-pointing preserves the semantics of pretty-printed expressions so that a user may inspect the tree structure of the expression through pointing to different parts of the string. A pretty-printed expression is a string of characters that represents an expression in the underlying foundation of a prover. For example the string $x + y$ is the pretty printed form of the expression `app (app (const "plus") (var "x")) (var "y")`. This form of interaction, where the user can interact graphically with expressions, is a powerful tool. For example, it enables 'interactive rewriting' of expressions, where equations can be manipulated by applying rewrite rules (for example, commutativity $x + y = y + x$) at exactly the subexpression where they are needed, all with the click of a mouse. I incorporate this tool and adapt it to a more modern, web-based situation.

[BT98] **Bertot, Yves; Théry, Laurent** *A generic approach to building user interfaces for theorem provers* (1998) Journal of Symbolic Computation

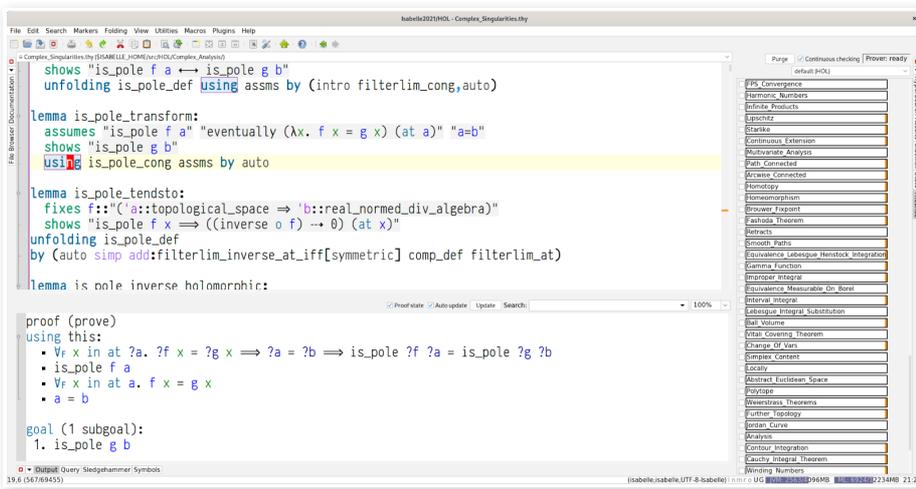


Figure 5.2. Screenshot of Isabelle2021. The main text buffer is adorned with the goal state window below and a sidebar with the current prover status of the open theories. All aspects of the IDE can be modified from within Isabelle through writing Scala code. Image source: own screenshot.

The most advanced specially-created IDE⁵³ for proving is Isabelle's *Prover IDE* (PIDE) [Wen12] (see Figure 5.2), developed primarily by Makarius Wenzel in *Scala* and based on the *JEdit* text editor. PIDE richly annotates Isabelle documents and proof states to provide inline documentation; interactive and customisable commands; and automatic insertion of text among other features. PIDE uses a Java GUI library called *Swing*. Isabelle's development environment allows users to code their own GUI in *Scala*. There have been some recent efforts to support *VSCode* as a client editor for Isabelle files. A web-based client for Isabelle, called *Clide* [LR13] was developed, although it provided only a subset of the functionality of the *JEdit* version. The design of PIDE also facilitates the development of new GUI designs from within its inbuilt *Scala* framework with instant changes in the GUI in response to changes in code. In Chapter 5, I adapt this idea to provide this same 'hot-reloading' functionality for *Lean*.

53 Integrated Development Environment

[Wen12] **Wenzel, Makarius** *Isabelle/jEdit-A Prover IDE within the PIDE Framework*. (2012) Intelligent Computer Mathematics - 11th International Conference

SerAPI [Gal16] is a library for machine-machine interaction with the Coq theorem prover. The project supports some web-based IDE projects such as *jsCoq* [GPJ17] and *PeaCoq*. Very recently, a framework called *Alectryon* [Pit20] has been released for Coq that enables users to embed web-based representations of data (see the link for more details). *Alectryon* offers the polish of a modern, graphical UI that I am aiming for, however it only produces a page showing the state of the proof script after the fact, and doesn't implement any interactive proof-creation system.

[Gal16] **Gallego Arias, Emilio Jesús** *SerAPI: Machine-Friendly, Data-Centric Serialization for Coq* (2016) Technical Report

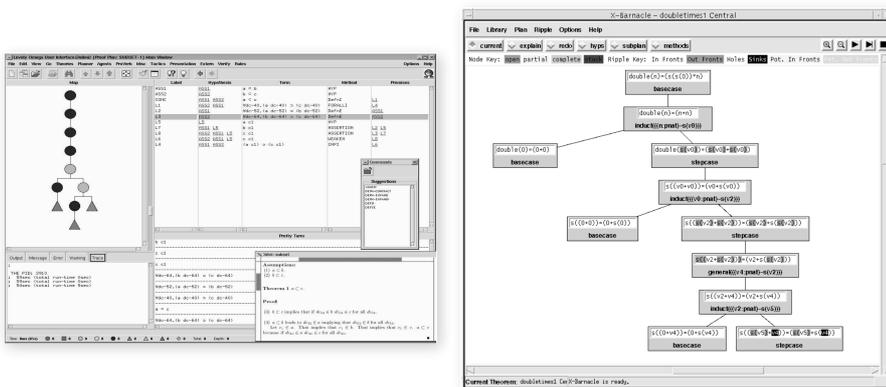


Figure 5.3. Screenshots for LQUI and X-Barnacle respectively. Note the use of graphical and multimodal representations of proofs. Image source: [SHB+99] [Low97].

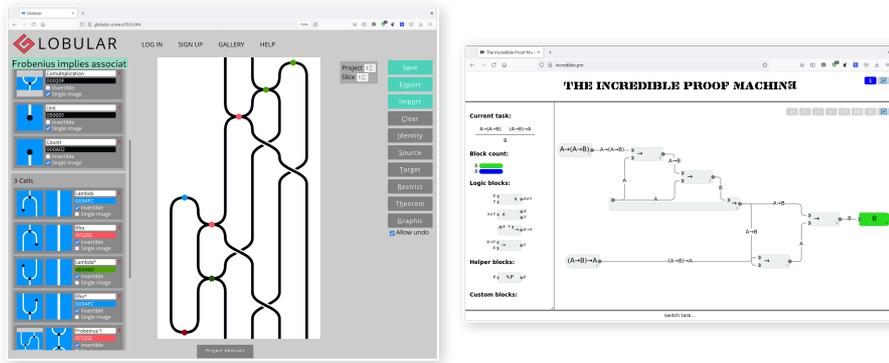
There are some older GUI-centric theorem provers that have fallen out of use: *LQUI* [SHB+99], *HyperProof* [BE92] and *XBarnacle* [LD97]. These tools were all highly innovative for including graphical and multimodal representations of proofs, however the code for these seems to have been lost, paywalled or succumbed to *bit rot*, to the extent that I can only view them through the screenshots (Figure 5.3) that are included with the papers. Source code for *Omega* and *CLAM* (which *LQUI* and *XBarnacle* use respectively) can be found in the *Theorem Prover Museum*⁵⁴. In my work I hope to recapture some of the optimism and experimental verve of these early systems by providing a GUI framework that makes it simple and reproducible to re-create these graphical and multimodal representations. (To peek ahead: some examples of community made representations can be seen in Section 5.7.4).

54 <https://theoremprover-museum.github.io/>

Other contemporary proof assistants with specially made GUIs are *Theorema* [BJK+16] and *KeY* [ABB+16]. *Theorema* is built upon the computer algebra system [Wolfram Mathematica](#) and makes use of its inbuilt GUI framework. However a problem is that it is tied to proprietary software. *KeY* is a theorem prover for verifying Java applications. *KeY* embraces multimodal views of proofs and offers numerous interactive proof discovery features and interactive proof-by-pointing inspection of subexpressions. In her thesis, Grebing investigates the usability of *KeY* [Gre19] through the use of focus groups, an approach relevant for my evaluation study in Chapter 6. I was particularly inspired by the interactivity made by *KeY*, however I want this to work in a more general ITP whereas *KeY* is more geared to verifying Java programs.

[Gre19] **Grebing, Sarah Caecilia** *User Interaction in Deductive Interactive Program Verification* (2019) PhD thesis (Karlsruhe Institute of Technology)

Figure 5.4. Screenshots of *Globular* (left) and *The Incredible Proof Machine* (right). Image source: own screenshots.



Another source of inspiration for me are the theorem prover web-apps: Vicary's *Globular* [VKB18] and Breitner's *Incredible Proof Machine* [Bre16] (see Figure 5.4). These tools are natively web-based and offer a visual representation of the proof state for users to manipulate. However they are both limited to particular domains of reasoning: *Globular* categories and simple problems in first order logic. They also do not offer anything in the way of automation, whereas I am interested in GUIs that assist in directing automation.

[VKB18] **Vicary, Jamie; Kissinger, Aleks; Bar, Krzysztof** *Globular: an online proof assistant for higher-dimensional rewriting* (2018) *Logical Methods in Computer Science*

These tools all demonstrate an ongoing commitment by the ITP community to produce graphical user interfaces which explore new ways of representing and interacting with proof assistants. It is with these previous works in mind that I design a new kind of general purpose approach to a GUI framework for a prover. Further comparison of the system that I have developed with the systems discussed here is given in Section 5.6.

[Bre16] **Breitner, Joachim** *Visual theorem proving with the Incredible Proof Machine* (2016) *International Conference on Interactive Theorem Proving*

5.2. Background on web-apps and functional GUIs

In this section I give some background information on web-apps (Section 5.2.1), functional UI frameworks (Section 5.2.2) and code-editing language servers Section 5.2.3 that is needed to frame the design considerations discussed in the remainder of the chapter.

5.2.1. Anatomy of a web-app

Web-apps are ubiquitous in modern software. By a web-app, we mean any software that uses modern browser technology to implement a graphical application. Web-apps are attractive targets for development because they are platform independent and can be delivered from a server on the internet using a browser or be packaged as an entirely local app using a packaging framework such as [Electron](#). Many modern desktop and mobile applications such as VSCode are thinly veiled browser windows.

The structure of a web-page is dictated by a tree structure called the **Document Object Model (DOM)**, which is an abstract representation of the tree structure of an XML or HTML document with support for event handling as might occur as a result of user interaction.

A **fragment** is a valid subtree structure that is not the entire document (Figure 5.5). So for example, an 'HTML fragment' is used to denote a snippet of HTML that could be embedded within an HTML document.

With the help of a CSS style sheet, the web browser paints this DOM to the screen in a way that can be viewed and interacted with by a user (see Figure 5.6). Through the use of JavaScript and event handlers, a webpage may manipulate its own DOM in response to events to produce interactive web-applications. The performance bottleneck in web-apps is usually the layout and painting stages of the browser rendering pipeline; the process by which the abstract DOM is converted to pixels on a screen.⁵⁵

Modern browsers support W3C standards for many advanced features: video playback, support for touch and ink input methods, drag and drop, animation, 3D rendering and many more. HTML also has a set of widely supported accessibility features called ARIA which can be used to ensure that apps are accessible to all. The power of web-apps to create portable, fully interactive user interfaces has clear applications for ITP some of the systems studied in Section 5.1 such as *Globular* and *Incredible Proof Machine* already make use of web technologies.

⁵⁵ See this chromium documentation entry for more information on critical paths in browser rendering. <https://www.chromium.org/developers/the-rendering-critical-path>

Figure 5.5. Anatomy of an HTML fragment.

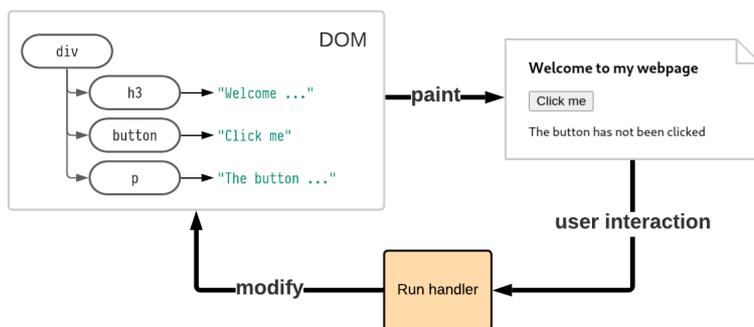


Figure 5.6. Interaction loop for a typical web-app. A DOM tree is painted using a CSS file to produce a viewable webpage. User interaction invokes *event handlers* which manipulate the DOM, causing a repaint.

The ProofWidgets framework places an ITP system within this interaction loop by providing a metaprogramming interface to create DOM fragments and to update the DOM in response to user interaction. Here, a **fragment** is any potential subtree of a full document. These fragments are then sent to the client editor which mounts this fragment within an embedded web-browser for viewing. If the user interacts with the resulting view, these interactions are sent back to the server and used to compute a new DOM fragment and update the interface.

5.2.2. Functional GUI frameworks

Most meta-level programming languages for ITPs are functional programming languages⁵⁶. However GUIs are inherently mutable objects that need to react to user interaction. Fortunately, there is a long tradition of user interface frameworks for pure-functional programming. Reactive programming [BCC+13]⁵⁷ enables the control of the inherently mutating GUI within a pure functional programming interface. The ideas of reactive programming have achieved a wide level of adoption in web-app development

⁵⁶ ML and Scala for Isabelle, OCaml for Coq, Lean for Lean.

thanks to the popularity of tools such as the [React JavaScript library](#)⁵⁸ and the [Elm programming language](#) [CC13].

Elm and React are UI frameworks for creating web-apps. The programming model used by these reactive frameworks is to model a user interface as a pure **view** function from a data source (e.g., a shopping list is rendered to an `` HTML fragment) to a representation of the DOM called the **Virtual DOM** (VDOM). The VDOM is a tree that represents the target state of the DOM.

To modify the UI's data in response to user interactions, an **update** function for converting user input events to a new version of the data is defined. For example, an update function for a shopping list defines how the list should be updated in response to receiving an 'action' such as deleting and adding a new item. Once the update function is applied and the data has been updated, the system reevaluates the view function on the new data to create a new VDOM. The browser's real DOM is then mutated to match this updated VDOM tree. This may sound inefficient - recomputing the entire VDOM tree each time - but there is an optimisation available: if the view function contains nested view functions, one can memoise these functions and avoid updating the parts of the VDOM that have not changed.

The VDOM is used because directly updating the browser's DOM is costly: as mentioned in Section 5.2.1, a bottleneck in performance for websites can be the repainting phase. When the data updates, the view function creates a new VDOM tree. This tree is then diffed with the previous VDOM tree to produce a minimal set of changes to the real DOM. A **diff** between a pair of trees t_1 and t_2 is a list of tree editing operations (move, add, delete) that transforms t_1 to t_2 . General tree diffing is known to be NP-hard [Bil05], so a simplified algorithm is used. In React, this diffing algorithm is called **reconciliation**.

In turn, Elm and React are inspired by ideas from **Functional Reactive Programming** (FRP). FRP was first invented by Elliot [EH97]. FRP is distinguished from general reactive programming by the explicit modelling of time. FRP has found use in UI programming but also more broadly in fields such as robotics and signal processing. A modern example of a FRP framework is [netwire](#)⁵⁹.

I have elected to use an API for creating user interfaces that is closer to the features of a functional user interface framework. I found that the full-FRP paradigm, where the programmer has to explicitly create programs with time, was too complex for the purposes of making a simple UI framework. In any case, the design requirement that the UI logic take place in the Lean VM but rendered by the client means that the modes of iteration are limited to the point where FRP offers no advantages over simpler paradigms. If not used carefully, full-FRP can also introduce 'time-leaks': a cycle of events trigger each other, causing the program to max out CPU and lock up. As is investigated in Lemma C.14, ProofWidgets use a weaker algebra than FRP which prevents time-leaks from occurring.

5.2.3. Code editors and client-server protocols

Some modern code editors such as [Atom](#) and [VSCode](#) are built using web technology. In order to support tooling features such as go-to-definition and hover information, these editors act as the client in a client/server relationship with an independently running process called a **language server**. As the user modifies the code in the client editor, the client communicates with the server: notifying it when the document changes and sending requests for specific information based on the user's interactions. In ITP this communication is more elaborate than in a normal programming language, because the process of proving is inherently interactive: the user is constantly observing the goal state of the prover and using this information to inform their next command in the construction of a proof.

The most important thing to note here is that changing the communication protocol between the client and the server is generally hard, because the developer has to update the protocol in both the server and the client. There may even be multiple clients. This

[BCC+13] **Bainomugisha, Engineer; Carreton, Andoni Lombide; Cutsem, Tom van; et al.** *A survey on reactive programming* (2013) ACM Computing Surveys (CSUR)

⁵⁷ The term 'reactive programming' generally refers to a kind of [declarative programming](#) where calculated values are automatically updated when its dependant values update. The classic example is an Excel spreadsheet, where value changes in cells propagate to dependent cells.

⁵⁸ <https://reactjs.org>

[CC13] **Czaplicki, Evan; Chong, Stephen** *Asynchronous functional reactive programming for GUIs* (2013) ACM SIGPLAN Conference on Programming Language Design and Implementation

[Bil05] **Bille, Philip** *A survey on tree edit distance and related problems* (2005) Theoretical computer science

[EH97] **Elliott, Conal; Hudak, Paul** *Functional reactive animation* (1997) Proceedings of the second ACM SIGPLAN international conference on Functional programming

⁵⁹ <http://hackage.haskell.org/package/netwire>

makes it difficult to quickly iterate on new user interface designs. A way of solving this protocol problem is to offer a much tighter integration by combining the codebases for the editor and the ITP. This is the approach taken by Isabelle/PIDE/jEdit [Wen12] and has its own trade-offs as discussed in Section 5.1.

5.3. Research goals

In terms of the broader research questions for the thesis given in Section 1.2, this chapter is mainly concerned with enabling *Question 3: presenting human-like reasoning interactively*. Specifically, ProofWidgets enables the interactive presentation of the `Box` calculus developed in Chapter 3. The application of ProofWidgets to HumanProof is covered in Section 5.8.

5.3.1. Architecture design goals

APIs between ITP systems and code editors such as `Emacs` or `VSCoDe` are large and difficult to extend and port. The ProofWidgets protocol addresses this problem by allowing an ITP user to develop new interfaces for users to interact with provers without having to learn specialised knowledge of a particular editor or technology other than basic HTML and CSS⁶⁰.

Modern ITP systems such as `Isabelle`, `Coq` and `Lean` use advanced language servers and protocols to interface with text editors to produce a feature-rich proving experience. These systems feature standard helpers such as syntax highlighting and type hover information as would be found in normal programming language tooling. They additionally include prover-specific features such as displaying the goal state and providing interactive suggestions of tactics to apply. ITP offers some additional UI challenges above what one might find in developing an editor extension for a standard programming language, because the process of proving is inherently interactive: the user is constantly observing the goal state of the prover and using this information to inform their next command.

The original motivation for ProofWidgets was to create a specific user interface for `Box`. However, while developing I became frustrated with the development workflow for prototyping the user interface in `Lean`: each time the interface changed, I would need to coordinate changes across three different codebases; the `Lean` core, the `VSCoDe` editor extension and the repository for `Box`. It became clear that any approach to creating user interfaces in which the editor code needed to be aware of the datatypes used within the ITP metalogic was doomed to require many coordinated changes across multiple codebases. This inspired my alternative approach; write a full-fledged GUI⁶¹ framework in the metalogic of the ITP itself. This approach has the advantage of tightening the development loop and has more general use outside of my particular project (as I will show in Section 5.7.4).

As Bertot and Théry [BT98] note, there are two approaches available to create a reusable user interface for a theorem prover:

- A deep integration between a code editor and a specific prover. This is the approach taken by Isabelle/PIDE/jEdit [Wen12], although more recently `VSCoDe` support has become available [Wen18 §3]. I call this the **monolithic approach**.
- Stipulate a protocol between client editors and provers. Here we can either have many clients to one prover (e.g., the `Lean 3` server protocol is supported by both `Emacs` and `VSCoDe extensions`); or one client to many provers, as Proof General [Asp00] achieved. I call this the **protocol approach**.

Focussing on the second approach, the protocol used is typically high level. That is, the protocol is stated in terms of concepts in the prover; for example, the `SerAPI` protocol [Gal16] used by `Coq` provides a high level of granularity for extracting goal states and inspecting `Coq` expressions and proof objects. While such an API enables fine control over

[Wen12] **Wenzel, Makarius** *Isabelle/jEdit-A Prover IDE within the PIDE Framework*. (2012) Intelligent Computer Mathematics - 11th International Conference

⁶⁰ Hypertext Markup Language and Cascading Style Sheets. These are the languages that control the content and styling of webpages.

⁶¹ Graphical User Interface, pronounced 'gooey'.

[BT98] **Bertot, Yves; Théry, Laurent** *A generic approach to building user interfaces for theorem provers* (1998) Journal of Symbolic Computation

[Wen12] **Wenzel, Makarius** *Isabelle/jEdit-A Prover IDE within the PIDE Framework*. (2012) Intelligent Computer Mathematics - 11th International Conference

[Wen18] **Wenzel, Makarius** *Isabelle/PIDE after 10 years of development* (2018) UITP workshop: User Interfaces for Theorem Provers.

the prover through an external tool such as an editor, the API is large and any changes to the API or additional features require changes in multiple places and potentially introduce incompatibilities.

I argue here that having a wide protocol such as this is detrimental to the agility of prover development. In contrast, the ProofWidgets framework provides a "prover \leftrightarrow editor" protocol that removes the need for an editor to be aware of the internal representations of the prover. This protocol works by reworking the "client \leftrightarrow server" API to instead support the rendering of *arbitrary* interactive user interfaces. In the ProofWidgets protocol, the code responsible for the layout and event handling of these interfaces is moved to the core ITP, instead of being the responsibility of the editor. This has the effect of creating a full, general purpose GUI framework within a theorem prover.

Here is a concrete example to motivate this design choice: in the development of the Lean VSCode extension, it was requested that it should be possible to filter some of the variables in the goal state to declutter the output window (see Figure 5.7). The Lean community originally achieved this by reparsing the textual goal state emitted by the Lean server component and removing the filtered items using regular expressions. This worked, but it required adding some specific code for the VSCode client -- supporting this feature in other editors would require rewriting this filtering code. Additionally, if the Lean server changes how the goal state is formatted, this filtering code would need to be rewritten. Even if an API which allows more semantic access to the expression structure is used such as SerAPI [Gal16], we still have the problem that the filtering code has to be written multiple times for each supported editor. Using ProofWidgets, this filtering code can be written once *in Lean itself* and it works in any editor that supports the ProofWidgets API (at the time of writing VSCode and a prototype version of the web editor). Furthermore, Lean users are free to make any custom tweaks to the UI without needing to make any changes to editor code.



Another motivation for ProofWidgets was to add 'structural pretty printing' or 'proof-by-pointing' as Théry and Bertot call it [BT98]. This is where pretty-printed strings have information attached to them that provides detail on the structure of the original expression that produced the string. In other frameworks that implement proof-by-pointing such as KeY [ABB+16] and later versions of Proof General [Asp00], a tight integration between the code editor used by the ITP and the pretty-printing system is required. As is shown in Section 5.5, the design of ProofWidgets means that all of the code for creating and interacting with these complex structures can be handled within the ITP system's metalegic.

The three enabling technologies of the ProofWidgets framework are:

- the introduction of web-based code editors such as [Atom](#) and [Microsoft Visual Studio Code](#) (VSCode);
- metaprogramming frameworks for creating programs that manipulate and inspect expressions and tactic states from within theorem provers. The primary example here is the Lean metaprogramming framework [EUR+17]. However other metaprogramming systems such as those found in Isabelle are also available.
- modern, functional, reactive user interface frameworks for the web such as [Elm](#) and [React](#).

5.3.2. Design goals

[Asp00] **Aspinall, David** *Proof General: A generic tool for proof development* (2000) International Conference on Tools and Algorithms for the Construction and Analysis of Systems

[Gal16] **Galleo Arias, Emilio Jesús** *SerAPI: Machine-Friendly, Data-Centric Serialization for Coq* (2016) Technical Report

[Gal16] **Galleo Arias, Emilio Jesús** *SerAPI: Machine-Friendly, Data-Centric Serialization for Coq* (2016) Technical Report

Figure 5.7. Demonstration of hypothesis filtering in Lean. Selecting the items from the dropdown menu with show or hide the hypotheses of the goal state according to their type. The original version of this feature was implemented in JavaScript as part of the VSCode Lean. Now the same thing is implemented within Lean as a ProofWidget. The effect of this choice is that the menu is now implemented entirely within Lean and without needing to update the VSCode extension.

[BT98] **Bertot, Yves; Théry, Laurent** *A generic approach to building user interfaces for theorem provers* (1998) Journal of Symbolic Computation

[ABB+16] **Ahrendt, Wolfgang; Beckert, Bernhard; Babel, Richard; et al.** *Deductive Software Verification - The KeY Book* (2016) **publisher** Springer

[Asp00] **Aspinall, David** *Proof General: A generic tool for proof development* (2000) International Conference on Tools and Algorithms for the Construction and Analysis of Systems

The ProofWidgets framework has the following design goals. The principle behind these goals is ease-of-use for people creating their own ProofWidgets, as well as ensuring that as much code as possible is present in Lean itself.

[EUR+17] **Ebner, Gabriel; Ullrich, Sebastian; Roesch, Jared; et al.** *A metaprogramming framework for formal verification* (2017) Proceedings of the ACM on Programming Languages

- Programmers write GUIs using the metaprogramming framework of the ITP.
- Programmers are given an API that can produce arbitrary DOM fragments, including inline CSS styles.
- No cross-compilation to JavaScript or WebAssembly: the GUI-generating code must run in the same environment as the tactic system. This ensures that the user interaction handlers have full access to the tactic execution context, including the full database of definitions and lemmas, as well as all of the metaprogramming library. In a cross-compilation based approach (implementation difficulty notwithstanding), the UI programmer would have to choose which parts of this context to export to the client.
- To support interactively discoverable tactics, the system needs to be able to command the client text editor to modify its sourcetext. I'll expand on this point in Section 5.3.3.
- The pretty printer must be extended to allow for 'interactive expressions': expressions whose tree structure may be explored interactively. I'll expand on this point in Section 5.3.4.
- Programmers should be able to create visualisations of their data and proofs.
- It should be convenient for programmers to be able to style their GUIs in a consistent manner.
- The GUI programming model should include some way of managing local UI state, for example, whether or not a tooltip is open.
- The GUI should be presented in the same output panel that the plaintext goal state was presented in. This ensures that the new features are discoverable and do not change the workflow of existing users.
- The framework should be backwards compatible with the plaintext goal state system. Users should be able to opt out of the GUI if they do not like it or want to use a non web-app editor such as Emacs.

While all of these could be implemented by suitably extending the Lean server and client, this would cause the size of the API to balloon significantly as discussed in Section 5.3.1. These features require the context of Lean's goal state and tactics engine. To produce a UI would need this context to be propagated from the server to the client. The server/client API would become *wide*. Any new idea for improving the interface would require inventing a new part of the server/client API and an implementation spanning many different languages: Lean, C++, JavaScript, and possibly other dependants such as the [web-editor](#) and the [Python API](#). Further, the implementation would have to occur in the Lean 3 core codebase, not an external library.

Instead, the goal of the ProofWidgets system is to sidestep this by giving the ITP system's metaprogramming framework full control over the UI of the goal state. By choosing this lower level of abstraction, the time required to create new and experimental interactive features is drastically reduced, because ProofWidgets can be developed in real time by just editing the code in a single code file within the prover's project. This has already proved useful several times in the Lean implementation of this protocol, for example, a go-to-definition button for expressions in the type information viewer was added by just changing [a few lines of code](#) in the mathlib codebase.

5.3.3. Discoverable tactics

Often a beginner to ITP is confronted with a goal state where they don't know which tactic could be used to progress in the proof. A key tenet of making a user interface intuitive to use is to make available actions *discoverable* (also called *learnability* within the HCI

literature [GFA09]). As shown in Figure 5.8, ProofWidgets allow one to make a goal state which actively suggests available tactics to the user.

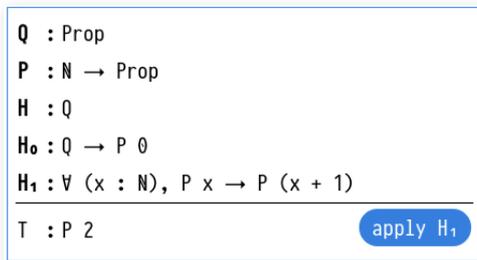


Figure 5.8. Example of a suggested tactic widget for a goal. Clicking the `apply H1` button will insert `apply H1` into the proof document and advance the tactic state.

5.3.4. Interactive term rewriting

Currently in Lean, if one wants to solve an equality problem such as

`(x + 3) * (x + 1) = x * x + 4 * x + 3` without resorting to a specialised tactic such as

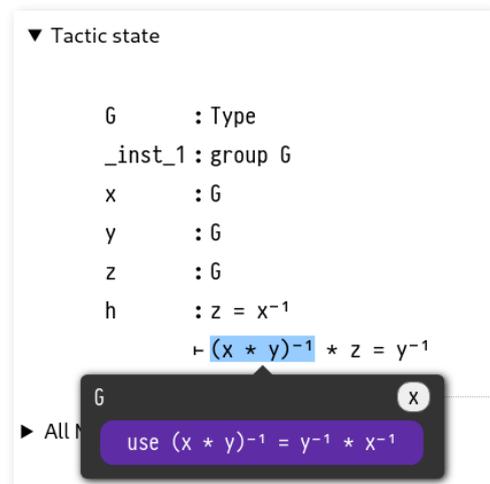
`ring` or a full-blown solver, you have the option of using a general equational rewriting

tactic `rewrite` or `simp`. The tactic `rewrite h` where `h : Π ..xs, lhs = rhs` finds a subterm

of the goal which the `lhs`⁶² of `h` matches with and replaces it with the `rhs` with

appropriate substitutions of the variables `xs`. Similarly, `simp` repeatedly performs rewrites of the goal from a curated set of lemmas.

These tactics do not give an easy way to explore subterms of an expression to interactively find where to apply rewrites. With ProofWidgets, one can build a tool where one can click directly on a subterm of a goal state and see the available rewrites at that point, or to see what an expression will look like after applying `simp`. An example of this is shown in Figure 5.9.



⁶² Left Hand Side

Figure 5.9. A widget goal view with subterm-sensitive suggestions of rewriting lemmas. Clicking on the subterm `(x * y)-1` suggests the rewrite rule shown in purple.

Since the widgets system is also able to influence the code editor, it is also possible to build user interfaces that interactively build the proof script in the Lean document.

5.3.5. Non-goals

In order to scope the design of the system, I specify some features that are not a requirement for the design of ProofWidgets.

No animation and time-continuous interactions: I don't address the task of creating a framework that is capable of complex reactive animation or continuous, mouse and touch driven interactions like pinching and dragging. Although many existing FRP⁶³ frameworks are built for these, it seems unlikely that Lean 3's VM⁶⁴ is going to be efficient enough to support them. This does not mean that no animations are possible in ProofWidgets, however. Complex animations are possible through [CSS transitions](#) without needing to involve a programmatic event at all.

No concurrency and asynchronous tasks: By this I mean the ability for the Lean server to 'push' events to the client after some long-running task has completed. This is not

⁶³ Functional Reactive Programming, see Section 5.2.2 for more details.

⁶⁴ Virtual Machine. Lean 3 compiles to bytecode which is run in its VM.

a requirement mainly in the name of keeping the implementation simple enough for a first proof-of-concept version. In Section 5.9 I offer some thoughts on how this could be implemented.

No compilation to javascript: This may become possible in Lean 4 with the help of Lean 4's compiler and [WebAssembly](#), but for now the UI logic of ProofWidgets should run entirely on the (local) Lean server. This has a number of advantages: it is much less complex than a cross-compilation approach and it allows for the entirety of Lean's proof state and prover apparatus to be available.

Don't support all user interface modalities: Modern browser user interfaces offer many different methods of interaction; drag and drop, touch gestures such as pinch to zoom. For now, the ProofWidgets framework only offers a small subset of these, namely mouse events and text input. However [HTML ARIA attributes](#) are supported by ProofWidgets so that they can be made accessible to differently-abled people.

Performance should be 'good enough': As mentioned in Section 5.2.1, the performance bottleneck for web apps is typically the layout and painting stages of the browser rendering pipeline. Using a client-side framework such as ReactJS to minimise the number of changes to the browser's DOM gives acceptable performance for most use cases. In Section 5.9 I also provide some ideas on how to improve the performance of ProofWidgets.

5.4. System description

The design goals discussed in Chapter 1 led me to design ProofWidgets to use a declarative VDOM-based architecture (see Section 5.2.2) similar to that used in the Elm programming language [CC13] and the [React](#) JavaScript library as discussed in Section 5.2.2. By using the same programming model, I can leverage the familiarity of potential users with commonly used React and Elm APIs. In the following sections I detail the design of ProofWidgets, starting with the UI programming model Section 5.4.1 and the client/server protocol Section 5.4.2.

5.4.1. UI programming model

This section is about the API that users of the ITP system can use to implement user interfaces created with the protocol given in Section 5.4.2.

Most meta-level programming languages for ITPs are functional programming languages⁶⁵. So the mutable DOM paradigm shown in Section 5.2.1 is going to not be suitable for our purposes because functional programming languages act predominantly on immutable datastructures. Fortunately, as discussed in Section 5.2.2, there are a number of functional paradigms available for building user interfaces in an immutable way. I summarise their operation here and a more detailed overview of the API is given in Appendix C. The design of the UI building API is inspired by the design used in the Elm programming language [CC13].

New user interfaces are created using the `Html` and `Component` types. A user may define an HTML fragment by constructing a member of the inductive datatype `Html`, which is either an element (e.g., `<div></div>`), a string or a `Component` object to be discussed shortly.

These fragments can have **event handlers** attached to them. For example, in (5.10), a button is created by defining a **handler** `h : Unit → α` sending the unit type to a member of some type `α`. When this interface is rendered in the client and the button is clicked, the server is notified and causes the node to **emit** the element `h() : α`. In (5.10), when the button is pressed it will emit `4 : ℕ`. The value of `h()` is then propagated towards the root of the `Html` tree until it reaches a component.

[CC13] **Czaplicki, Evan; Chong, Stephen** *Asynchronous functional reactive programming for GUIs* (2013) ACM SIGPLAN Conference on Programming Language Design and Implementation

⁶⁵ ML and Scala for Isabelle, OCaml for Coq, Lean for Lean

[CC13] **Czaplicki, Evan; Chong, Stephen** *Asynchronous functional reactive programming for GUIs* (2013) ACM SIGPLAN Conference on Programming Language Design and Implementation

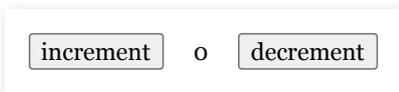
```
button : (h : Unit → α) → String → Html α
exampleButton : Html N := button (λ () ↦ 4) "click me"
```

Now we need to provide a mechanism for doing something with this emitted object. A **component** is an inductive datatype taking two type parameters: π (the props type) and α (the action type)⁶⁶. It represents a stateful object in the user interface tree where the state $s : \sigma$ can change as a result of a user interaction event. By 'stateful' we mean an object which holds some mutating state for the lifetime of the user interface. Through the use of components, it is possible to describe the behaviour of this state without having to leave the immutable world of a pure functional programming language. Three functions determine the behaviour of a component:

- $init : \pi \rightarrow \sigma$ initialises the state.
- $view : \pi \rightarrow \sigma \rightarrow \text{Html } \alpha$ maps the state to a VDOM tree.
- $update : \pi \rightarrow \alpha \rightarrow \sigma \rightarrow \sigma \times \text{Option } \beta$ is run when a user event is triggered in the child HTML tree returned by $view$. The emitted value $a : \alpha$ is used to produce a tuple $\sigma \times \text{Option } \beta$ consisting of a new state $s : \sigma$ and optionally, a new event $b : \beta$ to emit. If the new event is provided, it will propagate further towards the root of the VDOM tree and be handled by the next component in the sequence.

A simple example of a counter component is shown in (5.11) and Figure 5.12. In (5.11), the component has an integer $s : \mathbb{Z}$ for a state, and updating the state is done through clicking on the 'increment' and 'decrement' buttons which will emit 1 and -1 when clicked. The values a are used to update the state to $a + s$. Creating stateful components in this way has a variety of practical uses when building user interfaces for inspecting and manipulating the goal state. We will see in Section 5.5 that a state is used to represent which expression the user has clicked. Indeed, an entire tactic state can be stored as the state of the component. Then the update function runs various tactics to update the tactic state and output the new result.

```
< σ      := ℤ
, s₀     := 0
, view   := s ↦
  <div>
    button (λ () ↦ 1) "increment"
    <span>{to_string s}</span>
    button (λ () ↦ -1) "decrement"
  </div>
, update := (a : ℤ) ↦ (s : ℤ) ↦ a + s
>
```



5.4.2. The server protocol

The communication protocol between the client editor and the ITP server is illustrated in Figure 5.13. A more detailed overview on the specifics for the Lean implementation can be found in [the leanprover-community documentation](#).

(5.10). Simple example of an event handler. The function `button` takes an event handler h and some text for the button content. The value the handler returns will be emitted in the event of a button click.

⁶⁶This is designed to be familiar to those who use React components: <https://reactjs.org/docs/components-and-props.html>.

(5.11). Code for a simple counter app showcasing statefulness. The output is shown in Figure 5.12

Figure 5.12. The resulting view of a simple counter component.

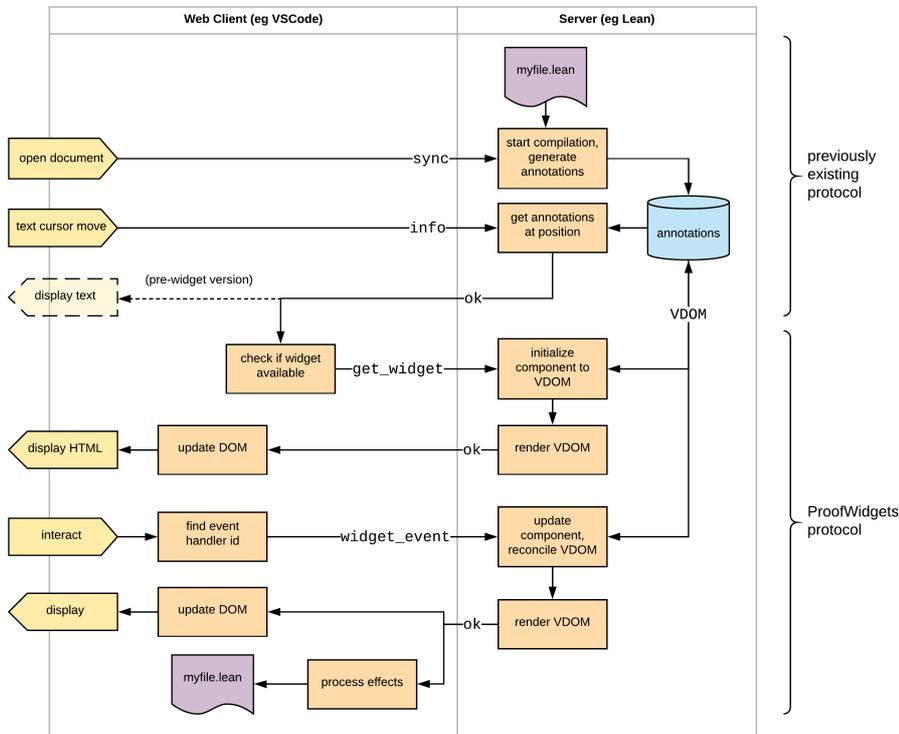


Figure 5.13. The architecture of the ProofWidgets client/server communication protocol. My contribution is present in the section marked 'ProofWidgets protocol'. Arrows that span the dividing lines between the client and server components are API requests and responses. The arrows crossing the boundary between the client and server applications are sent in the form of JSON messages. Rightward arrows are **requests** and leftward arrows are **responses**.

Once the programmer has built an interface using the API introduced in Section 5.4.1, it needs to be rendered and delivered to the browser output window. ProofWidgets extends the architecture discussed in Section 5.2.3 with an additional protocol for controlling the life-cycle of a user interface rendered in the client editor. When a sourcefile for the prover is opened (in Figure 5.13, `myfile.lean`), the server begins parsing, elaborating and verifying this sourcefile as usual. The server incrementally annotates the sourcetext as it is processed and these annotations are stored in memory. The annotations include tracing diagnostics messages as well as `thunks`⁶⁷ of the goal states at various points in a proof. When the user clicks on a particular piece of sourcecode in the editor ('text cursor move' in Figure 5.13), the client makes an `info` request for this position to the server, which responds with an `ok` response containing the logs at that point.

The ProofWidgets protocol extends the `info` messages to allow the prover to similarly annotate various points in the document with VDOM trees (see Section 5.2.2) created from components. These annotating components (see Section 5.4.1) have the type `Component TacticState Empty` where `TacticState` is the current state of the prover and `Empty` is the uninhabited type. A default component for rendering goals of proof scripts is provided, but users may override this with their own components. The VDOM trees are derived from this component, where the VDOM has the same tree structure as the `Html` datatype (i.e., a tree of elements, strings and components), but the components in the VDOM tree also contain the current state and the current child subtree of the component. This serves the purpose of storing a model of the current state of the user interface. These VDOMs can be **rendered** to HTML fragments that are sent to the client editor and presented in the editor's output window.

There are two ways to create a VDOM tree from a component: from scratch using **initialisation** or by updating an existing VDOM tree using **reconciliation**.

Initialisation is used to create a fresh VDOM tree. To initialise a component, the system first calls `init` to produce a new state `s`. `s` is fed to the `view` method to create an `Html` tree `t`. Any child components in `t` are recursively initialised.

The inputs to reconciliation are an existing VDOM tree `v` and a new `Html` tree `t`. `t` is created when the `view` function is called on a parent component. The goal of reconciliation is to create a new VDOM tree matching the structure of `t`, but with the component states from `v` transferred over. The tree diffing algorithm that determines whether a state should

⁶⁷ A `Thunk` is a lazily evaluated expression.

be transferred is similar to the [React reconciliation algorithm](#) and so I will omit a discussion of the details here. The main point is that when a user interface changes, the states of the components are preserved to give the illusion of a mutating user interface.

For interaction, the HTML fragment returned from the server may also contain event handlers. Rather than being calls to JavaScript methods as in a normal web-app, the client editor intercepts these events and forwards them to the server using a `widget_event` request. The server then **updates** the component according to the event to produce a new `Html` tree that is reconciled with the current VDOM tree. The ProofWidgets framework then responds with the new HTML fragment derived from the new VDOM tree. In order to ensure that the correct event handler is fired, the client receives a unique identifier for each handler that is present on the VDOM and returns this identifier upon receiving a user interaction. So, in effect, the ITP server performs the role of an event handler: processing a user interaction and then updating the view rendered to the screen accordingly. In addition to updating the view, the response to a `widget_event` request may also contain **effects**. These are commands to the editor, for example revealing a certain position in the file or inserting text at the cursor position. Effects are used to implement features such as go-to definition and modifying the contents of sourcefiles in light of a suggested modification to advance the proof state. If a second user interaction event occurs while the first is being handled, the server will queue these events.

The architecture design presented above is a different approach to how existing tools handle the user interface. It offers a much smaller programming API consisting of `Component` and `Html` and a client/server protocol that supports the operation of arbitrary user interfaces controlled by the ITP server. Existing tools (Section 5.1) instead give fixed APIs for interaction with the ITP, or support rendering of custom HTML without or with limited interactivity.

To implement ProofWidgets for an ITP system, it is necessary to implement the three subsystems that have been summarised in this section: a programming API for components; the client editor code (i.e., the VSCode extension) that receives responses from the server and inserts HTML fragments to the editors output window; and the server code to initialise, reconcile and render these components.

5.5. Interactive expressions

This section is about using ProofWidgets to perform 'interactive pretty printing' where expressions are rendered to HTML with explicit structure information. As discussed in Section 5.1, structural pretty printing is not a novel feature, however the way in which it is designed here makes structural pretty printing extensible and accessible to the metaprogramming framework. The ability to interactively pretty print expressions is a critical part of implementing the design goal of interactive term rewriting discussed in Section 5.3.4.

An example of the system in operation is given in Figure 5.14: as one hovers over the various expressions and subexpressions in the infoview, one gets semantically correct highlighting for the expressions, and when you click on a subexpression, a tooltip appears containing type information. This tooltip is itself a widget and so can also be interacted with, including opening a nested tooltip.

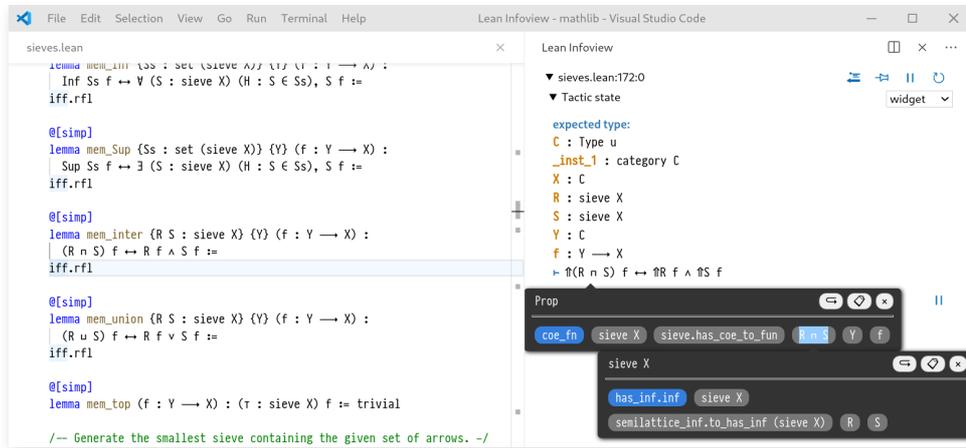


Figure 5.14. Screenshot showing the interactive expression view in action within the Lean theorem prover. The left-hand pane is the Lean source document and the right-hand pane is the infoview showing the context and expected type at the editor's cursor. There are two black tooltips giving information about an expression in the infoview.

A number of other features are demonstrated in Figure 5.14:

- Hovering over subterms highlights the appropriate parts of the pretty printed string.
- The buttons in the top right of the tooltip activate effects including a "go to definition" button and a "copy type to clipboard" button.
- Expressions within the tooltip can also be explored in a nested tooltip. This is possible thanks to the state tracking system detailed in the previous section.

Note that the Lean editor already had features for displaying type information for the source document with the help of hover info, however this tooltip mechanism is only textual (not interactive) and only works on expressions that have been written in the source document. Prior to ProofWidgets there was no way to inspect expressions as they appeared in the infoview.

All of the code which dictates the appearance and behaviour of the infoview widget is written in Lean and reloads in real time when its code is changed. This means that users can produce their own custom tooltips and improve upon the infoview experience without needing to leave the project.

5.5.1. Tagged strings

Before ProofWidgets, the Lean pretty-printer would take an expression and a context for the expression and produce a member of the `format` type. This is implemented as a symbolic expression (shortened to 'sexpr') *a la* LISP [McC60].

For ProofWidgets, I modified Lean's C++ pretty printer so that it would also tag certain sexprs with two pieces of data: the subexpression that produced the substring and an **expression address** indicating where the subexpression lies in the parent expression. The expression address is a list of **expression coordinates** used to reference subterms of an expression. An expression coordinate is a number that indexes the recursive arguments in the constructors for an expression. In this sense it is doing the same job as the coordinates defined in Section 2.3.2. That is, it parametrises the lenses that are available for subexpressions. A simplified example of the pseudocode is shown in (5.15).

[McC60] **McCarthy, John** *Recursive functions of symbolic expressions and their computation by machine, Part I* (1960) Communications of the ACM

```

Expr ::=
| var  : String → Expr
| app  : Expr → Expr → Expr
| lam  : String → Expr → Expr
| const : String → Expr

Coord ::=
| f | a | lam_body

Address := List Coord

get : Address      → Expr      → Option Expr
| []              → e         → some e
| [f, ..t]       → app l r   → get t l
| [a, ..t]       → app l r   → get t r
| [lam_body, ..t] → lam s b  → get t b
| _              → _         → none

set : Address → Expr → Expr → Expr := ...

coords : Expr → List Address := ...

TaggedString ::=
| tag : (Expr × Address) → TaggedString → TaggedString
| append : TaggedString → TaggedString → TaggedString
| of_string : String → TaggedString

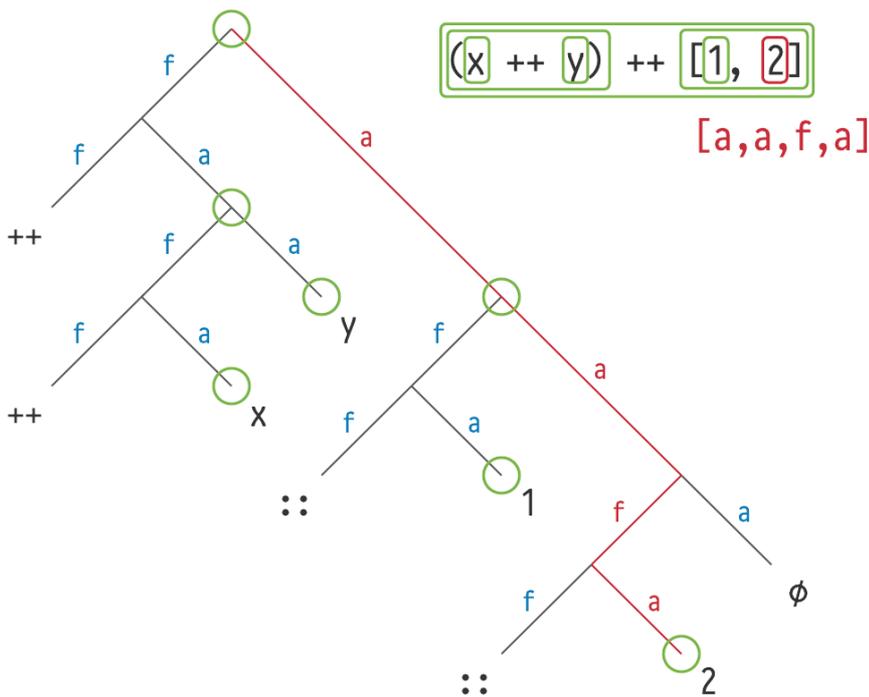
pretty_print : Expr → Tactic TaggedString

```

In this way, the `TaggedString` acts as a reversed source-map⁶⁸ between the resulting `sexpr` and the original expression, even when using specialised syntax such as lists `[1,2,3]` and `set` comprehensions. This tagged string is used to create widgets that allow users to interactively inspect various parts of the expression in the infoview.

In the case of a subexpression being below a binder (e.g., in the body of a lambda expression) the pretty printer instantiates the de-Brujin variable with a dummy local variable, so the given subexpression doesn't contain free de-Brujin variables and still typechecks without having to know the binders above the subexpression.

Below are some diagrams to illustrate the relationship between a `TaggedString` and an `Expr`.



(5.15). Pseudocode for implementing tagged strings from expressions. The `TaggedString` datastructure expands on a Wadler-style formatting tree by including 'tagged' portions of the expression. A tag includes the subexpression that the string represents and an `Address` object allowing one to determine which subtree of the parent expression is being represented.

⁶⁸ In the context of compilers, a source-map is a file that identifies parts of the compiler-output with the source code. This enables the use of diagnostic tools such as debuggers.

Figure 5.16. An expression tree for $(x + y) + [1, 2]$. Each `f` or `a` above the lines is an expression coordinate. The red `[a, a, f, a]` example of an expression address, corresponding to the red line on the tree. Each green circle in the tree will pretty-print to a string independent of the expression tree above it. While the pretty-printed expression involves infix operators, these expression trees are stored internally as function application trees. The pretty-printer is responsible for determining whether these trees should be printed in infix form or not.

"(x ++ y) ++ [1 , 2]"

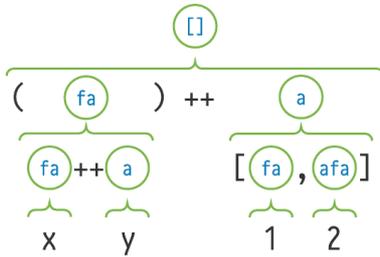


Figure 5.17. The `TaggedString` tree produced by pretty-printing the expression `(x ++ y) ++ [1,2]`. The green circles are `TaggedString.tag` constructors and the blue address text within is the relative address of the `tag` in relation to the `tag` above it. So that means that the full expression address for a subterm can be recovered by concatenating the `Addresses` above it in the tree. E.g., the `2` subexpression is at `[] ++ [a] ++ [a,f,a] = [a,a,f,a]`

There are two versions of the code for interactive expression rendering: the [original core Lean version](#) and the more experimental [mathlib version](#).

To render an interactive expression given a `TaggedString`, define a stateful `Component (TacticState × Expr) Empty`. The `TacticState` object includes the metavariable context and a local context in which the given expression is valid. The state of the component includes an optional `Address` of the subexpression. When the user hovers over a particular point in the printed `TaggedString`, the expression address corresponding to that part of the string is calculated using the `tags` and this address is set as the state of the component. This address is then used to colour in the portion of the string that is currently hovered over by the user's mouse cursor which gives the semantic-aware highlighting effect.

When the user clicks on a subexpression, a tooltip appears containing type information as well as some details on it such as the type and the available explicit arguments. Users can create their own tooltips using `attr.popper`. The stateful component framework developed in the last section means that these expressions can themselves be interactive expressions and we can recursively expand the selection, as shown in Figure 5.14 earlier.

5.6. Related work

In Section 5.1 I covered other graphical user interfaces for proof assistants. Here I will relate them to ProofWidgets. As discussed in Section 5.3.1, the main differentiating feature of ProofWidgets is its use of web technology for rendering and allowing the metaprogramming language of the theorem prover to take full responsibility for constructing the DOM of the GUI and handling user interactions. This contrasts with the two other approaches to constructing GUIs for theorem provers; which I dubbed the monolithic approach and protocol approach in Section 5.3.1.

The first related architecture is Isabelle's *Prover IDE* (PIDE). An advantage of the ProofWidgets approach compared to PIDE's is that the API between the editor and the prover can be smaller since, in ProofWidgets, the appearance and behaviour is entirely dictated by the server. In contrast, the implementation of PIDE is tightly coupled to the bundled `jEdit` editor, which has some advantages over ProofWidgets in that it gives more control to the developer to create new GUIs. The downside of PIDE's approach here is that one must maintain this editor and so supporting any other editor with feature-parity becomes difficult. ProofWidgets also makes use of modern web technology which is ubiquitously supported. In contrast, PIDE uses a Java GUI library called `Swing`. Creating custom UIs in PIDE requires coding in both `Scala` and `StandardML`. The result does not easily generalise to the `VSCoq Isabelle extension` because `VSCoq` is based on web-technology instead of the `Swing` framework, so if the custom UI is to also support the `VSCoq extension`, some JavaScript must also be written.

The example of the protocol approach that I will elect to compare ProofWidgets with is the `SerAPI` protocol for `Coq`. `SerAPI` is a library for machine-machine interaction with the `Coq` theorem prover. `SerAPI` contrasts to ProofWidgets in that it expects another program to be

responsible for displaying graphical elements such as goal states and visualisations; in the ProofWidgets architecture all of the UI appearance and behaviour code is also written in Lean, and the web-app client can render general GUIs emitted by the system.

Theorema [BJK+16] is a tool integrated into [Wolfram Mathematica](#), a proprietary computer algebra system. Mathematica comes with its own widget system, which can also be used in a web setting, and so by allowing Mathematica do the heavy-lifting, Theorema is able to have fine-grained control over its GUI whilst remaining portable. However, this approach means that it is tied to the proprietary Mathematica ecosystem, whereas ProofWidgets only depends on web standards which are open.

As discussed in Section 5.1, there is a cohort of now extinct theorem provers that had a great deal of focus on graphical, multimodal representations of data; LΩUI for Ωmega [BCF+97, SHB+99], HyperProof [BE92] and XBarnacle [LD97] for the CLAM prover. I hope that ProofWidgets can enable a rekindling of research into these more inventive and visual representations of proof states.

5.7. Implementation of Widgets in Lean

This section explains the underlying model for how ProofWidgets are created and mutated as the user interacts with them.

As discussed in Section 5.4.2 Lean has a server mode in which Lean works with a code editor such as VSCode or Emacs to provide an interactive editing environment. In server mode, Lean monitors open sourcefiles in the open project and maintains a structure called the log tree which attaches information providers to locations in the sourcefiles. For instance, the log tree holds the goal state of an interactive-mode proof and logging information which is revealed to the user when they navigate their cursor to that position.

Thanks to Lean's extensive meta-programming features [EUR+17], the Lean programmer can attach their own messages with the

```
tactic.save_info_thunk : thunk format → tactic unit
```

 function. `save_info_thunk` attaches a thunk to the log tree containing a procedure to generate a logging message. When the user clicks over that particular part of the document in the client, the Lean server evaluates the thunk and produce a formatted piece of text to display to the user in the infoview. This process is shown in the dashed part of Figure 5.13.

The ProofWidgets framework adds an additional kind of object to the log tree using `tactic.save_widget : (tactic_state ⇒ empty) → tactic unit` with a component as its argument. As discussed in Section 5.4.1, the log tree entry that is created through `save_widget` attaches some state to the object which *does* mutate. In this way, it is possible to model ephemeral UI state such as whether the user has opened a tooltip. The implementation code can be found in the [widget.cpp file in the leanprover-community/lean GitHub repository](#).

5.7.1. Reconciliation of HTML

Informally, the purpose of the reconciliation step is to compare an old tree and a new tree and try to find a way of matching up the new tree to the old tree. This is required to make sure that the internal states of sub-components in the tree are preserved even if the ordering of the components is changed. For example, suppose we had a list of counter subcomponents each with an integer state. Then if we reorder these components in the list we should expect that these counter states are not lost or scrambled. The general version of this problem is a tree-editing problem, which is known to be NP-hard [Bilo5], however as noted by ReactJS, we can get acceptable performance with heuristics and by allowing the created UI to add an identifying attribute called a `key` to elements.

In the Lean implementation, if the system is reconciling a list of child attributes, it will reconcile pairs of elements from the old and the new tree that share a key. Behaviour in the

[BJK+16] **Buchberger, Bruno; Jebelean, Tudor; Kutsia, Timur; et al.** *Theorema 2.0: computer-assisted natural-style mathematics* (2016) Journal of Formalized Reasoning

[BCF+97] **Benzmüller, Christoph; Cheikhrouhou, Lassaad; Fehrer, Detlef; et al.** *Omega: Towards a Mathematical Assistant* (1997) Automated Deduction - CADE-14

[SHB+99] **Siekman, Jörg; Hess, Stephan; Benzmüller, Christoph; et al.** *LOUI: Lovely OMEGA user interface* (1999) Formal Aspects of Computing

[BE92] **Barwise, Jon; Etchemendy, John** *Hyperproof: Logical reasoning with diagrams* (1992) Working Notes of the AAAI Spring Symposium on Reasoning with Diagrammatic Representations

[LD97] **Lowe, Helen; Duncan, David** *XBarnacle: Making Theorem Provers More Accessible* (1997) 14th International Conference on Automated Deduction

[EUR+17] **Ebner, Gabriel; Ullrich, Sebastian; Roesch, Jared; et al.** *A metaprogramming framework for formal verification* (2017) Proceedings of the ACM on Programming Languages

[Bilo5] **Bille, Philip** *A survey on tree edit distance and related problems* (2005) Theoretical computer science

unrecommended case of more than a pair having the same key is handled by pairing off the first in the list. Any remaining pairs of elements are reconciled in the order they appear in the list.

5.7.2. CSS

The Lean ProofWidgets system *per se* only emits JSON that is converted to HTML. But for the web-client and viewer implementations it also necessary to include a style sheet to make it look visually appealing. Rather than providing a mechanism for including stylesheets, the implementations load a stylesheet library called [Tachyons](#).

Tachyons is a 'functional CSS' library, which means that CSS classes each perform one very precise job. So for example, if you want to render a purple button with rounded corners and padding you would include the attribute `className "link pa2 br2 white bg-purple"`. Tachyons then has some small CSS class selectors (e.g., `.pa2 { padding: .5rem; }`) that style the DOM element appropriately. These CSS classes have a couple of benefits over inline styles: they are terser and they enforce a set of standardised colours and spacing that make it easier to provide a more consistent appearance. By using this approach to styling one can remove the need for a specially tailored stylesheet, which is perfect for the use case of ProofWidgets.

One drawback of rendering expressions in HTML is that the CSS typesetting paradigm is very different to the Wadler-style linebreaking algorithm [Wad03] that Lean's normal pretty printer uses. [Daniel Fabian discovered a CSS trick](#) to automatically linebreak expressions properly if the content box is too narrow.

[Wad03] **Wadler, Philip A** *prettier printer* (2003) The Fun of Programming, Cornerstones of Computing

5.7.3. Supported Effects

There is an additional hook for dealing with side-effects: changes to the client editor document state in response to widget events. Currently the supported effects are: highlighting a portion of the document; inserting text into the document; putting text into the paste buffer; opening a file (to implement go-to-definition).

This is the final piece of the puzzle to produce an interactive proof production experience: allowing ProofWidgets to affect the proof script. In the context of the Lean implementation, ProofWidgets allows a Lean programmer to embed an interactive GUI at any point in the Lean document. Thanks to Lean's extensive metaprogramming features [EUR+17], the user can write their GUI code in Lean itself. Widgets are already being used in `mathlib`, the Lean mathematics library [Com20].

[EUR+17] **Ebner, Gabriel; Ullrich, Sebastian; Roesch, Jared; et al.** *A metaprogramming framework for formal verification* (2017) Proceedings of the ACM on Programming Languages

5.7.4. Community-built ProofWidgets

ProofWidgets has already found use within the wider Lean community. See Figure 5.18 for a quilt of projects that people have made using ProofWidgets. Of particular note is the *Mathematica Bridge* by Lewis and Wu [Lew17], which connects Lean to Wolfram Mathematica and uses the ProofWidgets framework to show Lean functions plotted by Mathematica.

[Com20] **The Mathlib Community** *The Lean Mathematical Library* (2020) Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs

[Lew17] **Lewis, Robert Y.** *An Extensible Ad Hoc Interface between Lean and Mathematica* (2017) Proceedings of the Fifth Workshop on Proof eXchange for Theorem Proving, PxTP 2017, Brasília, Brazil, 23-24 September 2017

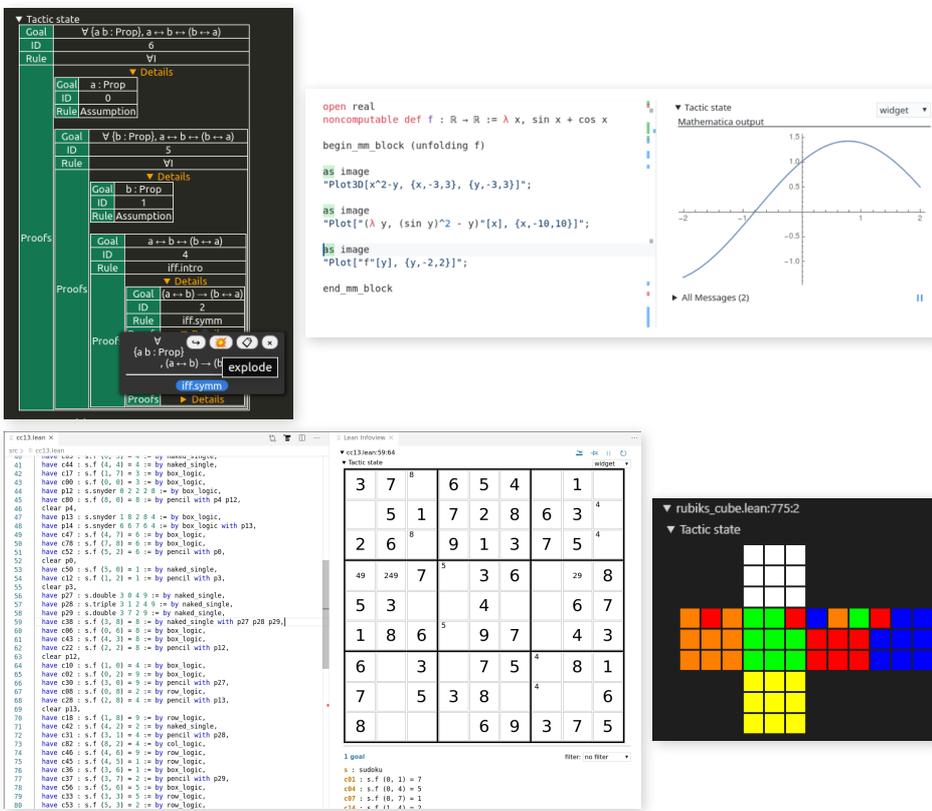


Figure 5.18. Community-made projects using ProofWidgets, in order these are: *explode proof viewer* by Minchao Wu. *Mathematica bridge* by Robert Y Lewis and Minchao Wu. *Sudoku solver and visualiser* by Markus Himmel. *Rubik's cube formalisation* by Kendall Frey.

5.8. Visualising Boxes

In this section, I am going to discuss how ProofWidgets are used to create an interactive `Box` element. `Box`s were introduced in Chapter 3⁶⁹ and are the development calculus of the HumanProof system. The first component, the visualisation system, follows the same visual rules as defined in (3.10). This is used to create a visualisation of a box.

In order to correctly print bound variables, the visualisation code has a metavariable context as an input. This is updated with new metavariables and local contexts as `g` and `g` `Box`s are traversed. This is simple to implement in ProofWidgets because all of the usual theorem proving apparatus are available whilst constructing the user interface. In contrast, implementing the same would be cumbersome to achieve with the 'wide API' approach discussed in Section 5.3.1.

To implement interactivity, the main component for rendering the `Box` is stateful, with the state being a `Box` and an undo stack of `Box`s indicating the box-tactics that have been applied so far. Interactivity is implemented by defining a precondition test for each box-tactic. For example, the `intro` box-tactic (Section 3.5.3) has a precondition of the goal type being a `∀` binder. If the precondition holds, the view function renders a button next to the selected goal indicating that a box-tactic is available for application on that goal or hypothesis. The button's event handler (Section 5.4.1) then performs the box-tactic and produces a new `Box` which is emitted to be picked up by the main component and added to the undo stack. An example of this can be seen in Figure 5.8.

The precondition system is needed because in some cases the box-tactic applications can take a long enough time that a noticeable lag will appear in the system⁶⁹.

The contextual bookkeeping of performing box-tactics on a `Box Zipper` as worked out in Appendix A is needed to ensure that the box-tactic is sound.

5.9. Future work

In the future, I wish to improve Lean ProofWidgets in a number of ways. One simple way is in improving documentation, Appendix B provides a tutorial on using ProofWidgets in

⁶⁹ The rule of thumb that I use is that any delay of more than half a second without a visual cue after clicking a button is jarring.

Lean that I hope to expand to more examples later.

In terms of performance, in order to produce responsive interfaces that use long-running tactics (e.g., searching a library or running a solver) it will be necessary to provide a mechanism for allowing concurrency. At the moment, if a long-running operation is needed to produce output, this blocks the rendering process and the UI becomes unresponsive for the length of the operation. Currently Lean has a `task` type which represents a 'promise' to the result of a long-running operation, which could be used to solve this problem. This could be cleanly integrated with ProofWidgets by providing an additional hook `with_task`:

```
component.with_task
  (get_task :  $\pi \rightarrow \text{task } \tau$ )
  : (Component ((option  $\tau$ )  $\times \pi$ )  $\alpha$ )  $\rightarrow$  (Component  $\pi$   $\alpha$ )
```

Here (5.19), `get_task` would return a long-running `task` object and the props for the inner component would transition from `none` to `some t` upon the completion of the task.

Cancelling a task is implemented simply by causing a rerender.

There are also many features of web-browsers that would be worth implementing such as drag-and-drop, sliders and mouse position events. There is also currently no support for adding third party libraries such as the data visualisation library `D3`. Allowing support for including arbitrary JavaScript libraries to run on the client would allow this, but making such a system portable and robust is more challenging, because they would require JavaScript glue code in order to work correctly with the system. Another aesthetic consideration is finding a principled way of implementing a Wadler-style pretty printer [Wado03] within CSS and HTML.

Currently, the server sends an entire DOM tree in every event loop, this could be replaced with a JSON patch file to save bandwidth.

I wish to reimplement ProofWidgets in Lean 4. Lean 4 has a bootstrapped compiler, so the reconciling code can be written in Lean 4 itself without having to modify the core codebase as was necessary for Lean 3. I hope that the pseudocode written in Appendix C will assist in this project. Lean 4 has an overhauled, extensible parser system [UM20] which should allow a `JSX-like` HTML syntax to be used directly within Lean documents.

(5.19). Adding concurrency to components with the help of `with_task`.

[Wado03] **Wadler, Philip** *A prettier printer* (2003) *The Fun of Programming, Cornerstones of Computing*

[UM20] **Ullrich, Sebastian; de Moura, Leonardo** *Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages* (2020) *Automated Reasoning*

Chapter 6

Evaluation

Up to this point, I have developed the various parts of the HumanProof software. The focus of this chapter is my study to investigate how mathematicians think about the understandability of proofs and to evaluate whether the natural language proofs generated by HumanProof aid understanding.

6.1. Objectives

This chapter addresses the third objective of *Question 2* in my set of research questions (Section 1.2): evaluating HumanProof by performing a study on real mathematicians. Let me break this objective down into a number of hypotheses and questions.

The hypotheses that I wish to test are:

1. Users will (on average) rank HumanProof natural language proofs as more understandable than Lean proofs.
2. Users will (on average) rank HumanProof natural language proofs as less understandable than proofs from a textbook. This hypothesis is a control: a textbook proof has been crafted by a human for teaching and so I would expect it to be more understandable than automatically-generated proofs.
3. Users will (on average) be more confident in the correctness of a Lean proof.

In addition to testing these hypotheses, I wish to gather qualitative answers to the following two questions:

1. What properties of a proof contribute to, or inhibit, understandability?
2. How does having a computer-verified proof influence a mathematician's confidence in the correctness of the proof?

6.2. Methodology

The goal of this study is to evaluate the hypotheses and gain answers to the questions above.

To do this, I found a set of mathematicians and showed them sets of proof scripts written using Lean, HumanProof and a proof script taken from a mathematics textbook. I recorded the sessions and performed a qualitative analysis of the verbal responses given by participants. I also asked participants to score the proof scripts on a [Likert scale](#) (that is, rank from 1 to 5) according to two 'qualities':

1. How understandable are the proofs for the purpose of verifying correctness? This is to assess hypotheses 1 and 2.
2. How confident is the participant that the proofs are correct? Note that a proof assistant gives a guarantee of correctness. This is to assess hypothesis 3.

In Section 2.5, I discussed some of the literature in the definitions of understandability and distinguished between a person finding a proof *understandable* and a person being *confident* that a proof is correct. In this experiment, instead of using these definitions I asked the participants themselves to reflect on what understandable means to them in the context of the experiment.

A similar methodology of experimentation to the one presented here is the work of Jackson, Ireland and Reid [IJR99] in developing interactive proof critics⁷⁰. Their study

sought to determine whether users could productively interact with their software, and they adopted a co-operative evaluation methodology [MH93] where participants are asked to 'think aloud' when using their software.

In overview, each experiment session lasted about an hour and was split into three phases:

1. Participants were given a training document and a short presentation on how to read Lean proofs. They were also told the format of the experiment. (10 minutes)
2. Over 4 rounds, the participant was given a statement of a lemma in Lean code and then shown three proofs of the same lemma in a random order. They were asked to rate these and also to 'think aloud' about their reasons for choosing these ratings. (40 minutes)
3. A debrief phase, where 3 demographic questions were asked as well as a brief discussion on what understandability means to them. (5 minutes)

Due to the COVID19 pandemic, each experiment session was conducted using video conferencing software with the participants submitting ratings using an online form. As well as the data that the participants filled in the form with, the audio from the sessions was recorded and transcribed to methodically extract the explanations for why the participants assigned the ratings they did.

The study was ethically approved by the Computer Laboratory at the University of Cambridge before commencing. The consent form that participants were asked to sign can be found in Appendix D.4.

6.2.1. Population

I wish to understand the population of working mathematicians, which here means people who work professionally as mathematicians or students of mathematics or mathematical sciences such as physics. That is, people who are mathematically trained but who do not necessarily have experience with proof assistants.

Postgraduates and undergraduates studying mathematics at the University of Cambridge were recruited through an advert posted to the university's mathematics mailing list, resulting in 11 participants. The first participant was used in a pilot study, however the experiment was not changed in light of the results of the pilot and so was included in the main results. All participants were rewarded with a £10 gift card.

6.2.2. Choice of theorems to include

I will include proofs that involve all of the aspects of HumanProof discussed in Chapter 3 and Chapter 4. These are:

- Natural language write-up (Section 3.6)
- Branching on disjunctive goals (Section 3.5.6).
- Advanced rule application automatically unrolling existential quantifiers (Section 3.5.8).
- Subtasks engine for solving equalities (Chapter 4).

Note that the GUI developed in Chapter 5 was not included in this study. This is primarily due to the difficulty of performing user-interface studies remotely.

The proofs should be drawn from a context with which undergraduates and postgraduates are familiar and be sufficiently accessible that participants have ample time to read and consider the proofs. The four problems are:

1. The composition of group homomorphisms is a group homomorphism.
2. If A and B are open sets in a metric space, then $A \cup B$ is also open.
3. The kernel of a group homomorphism is a normal subgroup.

[IJR99] **Ireland, Andrew; Jackson, Michael; Reid, Gordon** *Interactive proof critics* (1999) Formal Aspects of Computing

⁷⁰ Proof critics are discussed in Section 2.6.2.

[MH93] **Monk, Andrew; Haber, Jeanne** *Improving your human-computer interface: a practical technique* (1993) **publisher** Prentice Hall

4. If A and B are open sets in a metric space, then $A \cap B$ is also open.

Lemma 1 ($g \circ h$ is a group homomorphism) is simple but will be a 'teaching' question whose results will be discarded. This serves to remove any 'burn-in' effects to do with participants misunderstanding the format of the experiment and to give the participant some practice.

6.2.3. Choice of proofs

The main part of the experiment is split into 4 tasks, one for each of the lemmas given above. Each task consists of

- A theorem statement in natural language and in Lean. For example, Lemma 1 appears as "Lemma 1: the composition of a pair of group homomorphisms is a group homomorphism" and also `is_hom f → is_hom g → is_hom (g ∘ f)`.
- A brief set of definitions and auxiliary propositions needed to understand the statement and proof.
- Three proof scripts;
 - A Lean tactic sequence, written so that it is not necessary to see the tactic state to understand the proof.
 - A HumanProof generated natural language write-up.
 - A natural language proof taken from a textbook.

These proofs can be viewed in Appendix D.3.

Both of the metric space problems use the same definitions, so this will save some time in training the participant with notation and background material. They also produce different HumanProof write-ups.

In Lemma 4 (Appendix D.3.4), the statement to prove is that the intersection of two open sets is open. The HumanProof proof for this differs from the Lean and textbook proofs in that an existential variable $\varepsilon : \mathbb{R}$ needs to be set to the minimum of two other variables. In the Lean and textbook proofs, ε is explicitly assigned when it is introduced. But HumanProof handles this differently as discussed in Section 3.5.8: ε is transformed into a metavariable and is assigned after it can be deduced from unification later.

Two group theory problems (Lemmas 1 and 3) are given which are performed using a chain of equalities. These use the subtasks engine detailed in Chapter 4.

There are usually many ways of proving a theorem. Each of the three proofs for a given theorem have been chosen so that they follow the same proof structure, although the level of detail is different, and in the equality chains different sequences are used. This should help to ensure that the participant's scores are informed by the presentation of the proof, rather than the structure of the proof.

For all proofs, variable names were manually adjusted to be the same across all three proofs. So for example, if HumanProof decides to label a variable ε but it is η in the textbook, ε is changed to be η . Stylistic choices like this were arbitrated by me to be whichever choice I considered to be most conventional.

The study was originally intended to be face-to-face, showing users an on-screen Lean proof with the help of the Lean infoview to give information on the intermediate proof state. However, due to the study being remote, the material was presented using static images within a web-form instead. This meant that all of the proofs needed to be read without the help of any software. So I wrote the Lean proofs in a different style to the one that would be found in the Lean mathematics library. The following list gives these stylistic changes. I will discuss the validity implications of this in Section 6.7.

- All variable introductions had their types explicitly written even though Lean infers them.

- Before and after a complex tactic I insert a `show G` statement with the goal being shown.
- French quotes `<x ∈ A>` to refer to proofs are used wherever possible (as opposed to using `assumption` or the variable name for the proof).
- When proving `A ∪ B` is open, one needs to perform case analysis on `y ∈ A ∪ B` where the reasoning for either case is very similar. While it is possible to use a `wlog` tactic to reduce the size of the proof, I decided against using it because one must also provide an auxiliary proof for converting the proof on the first case to the second case.

In the textbook proof all $\text{T}_{\text{E}}\text{X}$ was replaced with Lean-style pretty printed expressions. This is to make sure that the participants are not biased towards the human-made proof due to their familiarity with $\text{T}_{\text{E}}\text{X}$ -style mathematical typesetting. Another issue with the metric space proofs is that the original textbook proofs that I found were written in terms of open balls around points, but I wanted to keep the definitions used by the proofs as similar as possible so I used \forall, \exists -style arguments. Thus the parts of the textbook proofs that mention open balls were replaced with the equivalent \forall, \exists definition. This might have caused the textbook proofs to become less understandable than the original. This concern is revisited in Section 6.7.

6.3. Agenda of experiment

In this subsection I provide more detail on the design and structure of my empirical study. The session for each participant was designed to take less than an hour and be one-to-one over [Zoom](#), a video conferencing application. Zoom was also used to schedule and record the sessions.

6.3.1. Pre-session

Before the session starts, a time is scheduled and participants are asked to sign a consent form which may be viewed in Appendix D.4. This consent form makes sure that the participants are aware of and accept how the data generated from the session will be used. Namely, I record their names and emails, however only the anonymised answers to the forms and some selected quotes are publicly released. All recording data and identification data were deleted after the course of the experiment.

6.3.2. Preparatory events

Once the session starts, I greet the participant, double-check that they have signed the consent form and give them an overview of the agenda of the study. I also remind them that their mathematical ability is not being tested.

6.3.3. Training phase

In this phase, the training document given in Appendix D.2 is sent to the participant as a PDF, and the participant is asked to share their screen with me so that I can see the training document on their desktop. Participants were told that the purpose of the training document was to get them familiar enough with the new syntax so that they could read through a Lean proof and understand what is being done, even if they might not be able to write such a proof.

Then I walk the participant through the training document, answering any questions they have along the way. The purpose of the training document is to get the participant used to the Lean syntax and tactic framework.

For example, the right associativity of function/implication arrows `f : P → Q → R` and the curried style of function application (`f p q` versus `f(p,q)`). The final part of the training document is a reference table of tactics. Participants are informed that this table is

only for reference and that it is covered in the training phase. The training phase is designed to take around 10 to 15 minutes.

The participant is informed that they can refer back to this training document at any time during the experiment, although in the name of reducing browser-tab-juggling, I would also offer to verbally explain any aspects of the material that participants were unsure about.

6.3.4. Experiment phase

After the training phase, I ask the participant if I can start recording their screen and send them a link to a Google Form containing the material and questions for phases 2 and 3.

The first page of the form is a landing page with some checkboxes to double check that the participant is ready, has signed the consent form, and understands the format of the experiment.

The participant then proceeds to the first in a sequence of four pages. Each page contains a lemma, some necessary definitions, a set of three proofs and two sets of Likert scale radio buttons for the participant to rate their understandability and confidence scores for each proof. The lemmas are presented in a random order, although the first lemma - the composition of two group homomorphisms is a group homomorphism - always appears first because it is used to counter the learning effect as discussed in Section 6.2.2. The proofs are also presented in a random order on the page.

The form page given for each task is presented in Figure 6.1, each page is laid out in a vertical series of boxes, the first box contains the lemma number and the statement of the lemma to be proven. The next box contains the definitions, written in both natural language and Lean syntax. Then come three boxes containing the proof scripts in a random order. Each proof script is presented as a PNG image to ensure that the syntax and line-breaks render the same on all browsers and screen sizes.

Figure 6.1. Annotated overview of a task page in Google forms. The content present in the form can be read in full in Appendix D.3.

Lemma 1

The composition of two group homomorphisms is a group homomorphism.

Definitions

Here we write the group product on two group elements $x, y : G$ as $x * y$ and write $x^{-1} : G$ as the inverse element of x . The identity is written as $e : G$.

Given a pair of groups G, H , a function $f : G \rightarrow H$ is a **group homomorphism** whenever $f(x * y) = f(x) * f(y)$ for all $x, y \in G$.

```

variables {G H I : Type} [group G] [group H] [group I]
variables {f : G → H} {g : H → I}

def is_hom (f : G → H) := ∀ (x y : G), f (x * y) = f x * f y

(g ∘ f) x := g (f (x))
                    
```

Proof A

Let $f : G \rightarrow H$ and $g : H \rightarrow I$ be group homomorphisms and let x and y be elements of G . Then we have

```

(g ∘ f)(x * y) = g(f(x * y))
                = g(f(x) * f(y))
                = g(f(x)) * g(f(y))
                = (g ∘ f)(x) * (g ∘ f)(y)
                    
```

and we are done.

Proof B

Let G, H and I be groups and let $f : G \rightarrow H$ and $g : H \rightarrow I$ where f and g are group homomorphisms. Let x and y be elements of G . Since f and g are group homomorphisms, we have

```

(g ∘ f)(x * y) = g(f(x * y))
                = g(f(x) * f(y))
                = g(f(x)) * g(f(y))
                = (g ∘ f)(x) * g(f(y))
                = (g ∘ f)(x) * (g ∘ f)(y)
                    
```

We are done.

Proof C

```

theorem hom_composition : is_hom f → is_hom g → is_hom (g ∘ f) :=
begin
  assume hf : is_hom f,
  assume hg : is_hom g,
  assume x y : G,
  calc g (f (x * y)) = g (f x * f y)      : by rewrite hf
    ... = (g (f x)) * (g (f y)) : by rewrite hg,
end
                    
```

Title and natural language lemma statement

Prerequisite definitions provided in both natural language and in Lean code.

Proof script images. One each of Lean, HP and Textbook in a random order. They are always titled "Proof A", "Proof B", "Proof C" in order.

At the bottom of the page, participants were presented with two more boxes containing a set of radio buttons allowing the participant to rate each proof as shown in Figure 6.2.

Figure 6.2. A filled out form.

How understandable do you find each of the above proofs?

	very difficult to understand	difficult	ok	easy	very easy to understand
Proof A	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Proof B	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
Proof C	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>

Clear selection

How confident are you that the above proofs are correct?

	not confident at all	less confident	moderately confident	quite confident	certain
Proof A	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Proof B	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
Proof C	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>

Clear selection

Back Next

The title questions for these sets of buttons are "How understandable do you find each of the above proofs?" and "How confident are you that the above proofs are correct?". Each column of the radio buttons has a phrase indicating the Likert scoring level that that column represents.

The participants were reminded that:

- They could assume that the Lean proof was valid Lean code and had been checked by Lean.
- That the confidence measure should not be about the confidence in the result itself being true but instead confidence that the proof given actually proves the result.
- The interpretation of the concept of 'understandable' is left to the participant.

Once the participant filled out these ratings, I would prompt them to give some explanation for why they had chosen these ratings - although usually they would volunteer this information without prompting. In the case of a tie, I also asked how they would rank the tied items if forced to choose.

If I got the impression that the participant was mixing up the labels during the rating phase, I planned to interject about the label ordering. This did not occur in practice though.

6.3.5. Debriefing phase

In this final phase, the user is presented with a series of multiple choice questions. Namely;

- What is the level of education that you are currently studying? (Undergraduate 1st year, Undergraduate 2nd year, Undergraduate 3rd year, Masters student, PhD student, Post-doc and above, Other)
- Which of the below labels best describes your area of specialisation? (Pure Mathematics, Applied Mathematics, Theoretical Physics, Statistics, Computer Science, Other)
- How much experience with automated reasoning or interactive theorem proving do you have? (None, Novice, Moderate, Expert)

Finally, there is a text-box question entitled "What does it mean for a proof to be 'understandable' to you?". At this point, I would tell the participant that they could also answer verbally and I would transcribe their answer.

After the participant has submitted the form, I thanked them and stopped recording. In some cases, the participant wanted to know more about interactive theorem proving and Lean, and in this case I directed them towards [Kevin Buzzard's Natural Numbers Game](#).

6.4. Results

Full results are given in Table 6.3. In the remainder of this section the proof scripts are coded as follows: **L** is the Lean proof; **H** is the natural language HumanProof-generated proof; **T** is the textbook proof. Similarly, the lemmas are shortened to their number (i.e., "Lemma 2" is expressed as "2"). The 'Question ordering' and 'Ordering' columns give the order in which the participant saw the lemmas and proofs respectively. So an ordering of 1432 means that they saw Lemma 1, then Lemma 4, then 3, then 2. And within a lemma, an ordering of HLT means that they saw the HumanProof proof followed by textbook proof, followed by the Lean proof. The columns are always arranged in the LHT order, so the L column is always the rating that they assigned to the Lean proof. 'Unders.' and 'Conf.' are the understandability and confidence qualities, respectively. Figure 6.4 and Figure 6.5 plot these raw results as bar charts with various groupings. These results are interpreted and analysed in Section 6.5.

Table 6.3. The full (anonymised) results table of the quantitative data collected from the experiment. Print note: a full, selectable version of this table can be found at <https://www.edayers.com/thesis/evaluation#full-results-table>.

No	Education	Area	:abbr[ITP] experience	Question ordering	Lemma 1: composition of group homomorphisms is a group homomorphism						Lemma 2: $A \cup B$ is open						Lemma 3: kernel is normal						Lemma 4: $A \cap B$ is open									
					Ordering	Unders.			Conf.			Ordering	Unders.			Conf.			Ordering	Unders.			Conf.			Ordering	Unders.			Conf.		
						L	H	T	L	H	T		L	H	T	L	H	T		L	H	T	L	H	T		L	H	T	L	H	T
1	PhD	Statistics	None	1324	HTL	4	5	5	5	5	5	HLT	3	2	5	4	4	5	LTH	5	5	5	5	5	5	HLT	3	5	5	5	5	5
2	PhD	Physics	None	1234	HLT	4	3	4	4	4	5	THL	4	4	3	2	4	4	TLH	5	4	5	5	5	5	THL	3	3	4	4	3	4
3	Undergrad	Pure	None	1423	THL	3	5	5	4	5	5	HTL	2	5	4	2	5	5	HTL	5	5	4	5	5	5	LTH	3	4	5	2	5	5
4	Post-doc	Pure	None	1423	THL	5	5	5	5	5	5	HTL	3	5	5	4	5	1	TLH	5	5	4	5	5	5	LHT	3	5	4	4	5	5
5	PhD	Pure	None	1324	LHT	5	4	5	5	5	5	HTL	4	4	5	5	5	5	HTL	5	4	5	5	5	5	HTL	4	5	4	5	5	5
6	Masters	Pure	None	1432	LHT	2	4	4	5	5	5	THL	4	4	5	5	5	5	TLH	5	4	5	5	5	5	THL	4	4	5	5	5	5
7	Post Doc	Applied	None	1234	TLH	4	5	5	5	5	5	LHT	3	5	4	5	5	3	LHT	5	5	5	5	5	5	LHT	3	5	4	4	5	5
8	Undergrad	Pure	Novice	1324	HLT	3	5	5	3	5	5	HLT	3	3	4	3	3	3	LTH	4	5	4	4	5	5	LTH	3	3	5	4	4	4
9	PhD	Physics	None	1432	THL	5	5	5	5	5	5	LTH	3	5	5	5	5	4	TLH	5	4	4	5	5	5	LHT	2	4	5	4	5	5
10	PhD	Pure	None	1243	LHT	5	4	5	5	5	5	TLH	3	4	4	5	4	4	HLT	5	5	5	5	5	5	LTH	4	5	5	5	4	4
11	Masters	Pure	None	1432	LHT	3	4	4	4	4	4	TLH	2	4	3	3	4	2	LTH	4	3	3	4	4	4	THL	3	5	4	3	4	4

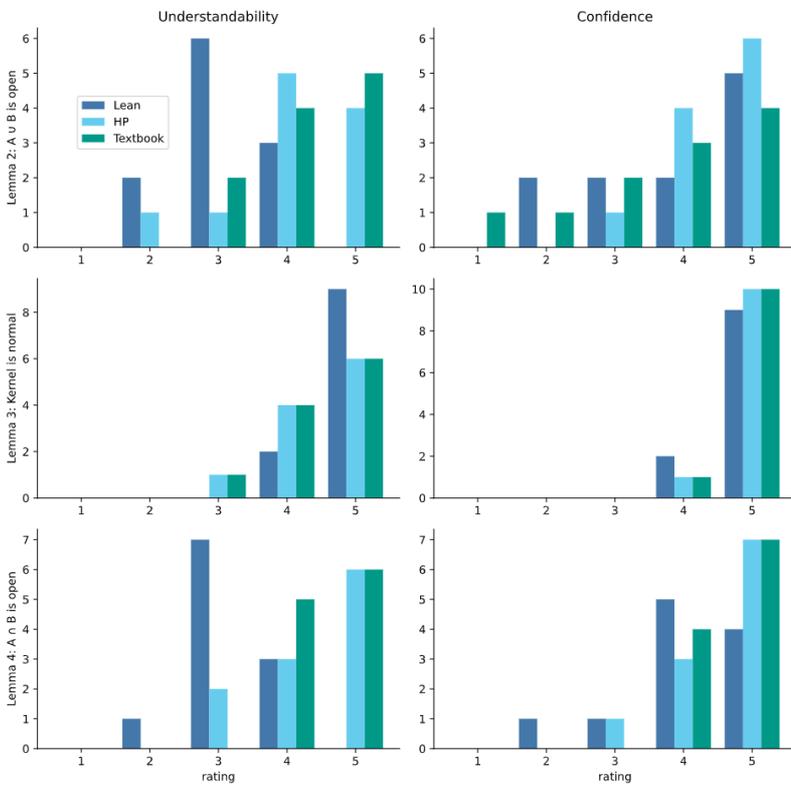


Figure 6.4. Bar chart of the ratings grouped by lemma. That is, each row is a different lemma and for each of the values [1,5] the number of people who assigned that rating are plotted as a bar, with a different colour for each of the proofs. The columns correspond to the two 'qualities' that the participants rated understandability and confidence.

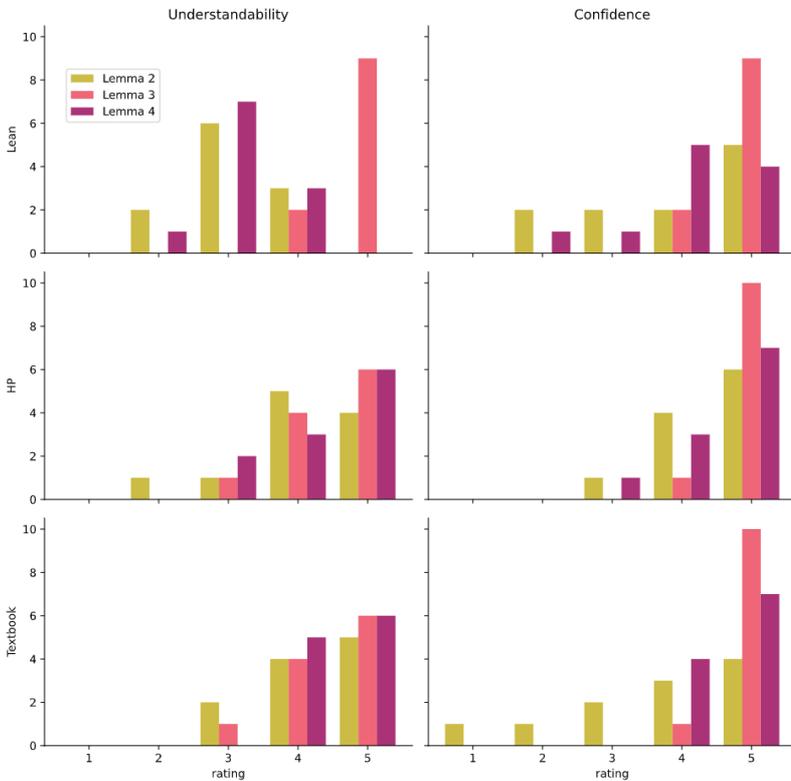


Figure 6.5. Bar chart of the ratings grouped by proof. The same data is used as in Figure 6.4 but with separate vertical plots for each of the types.

6.5. Quantitative analysis of ratings

6.5.1. Initial observations

On all the lemmas, people on average ranked the understandability of HumanProof and Textbook proofs to be the same. On Lemma 3, people on average ranked the Lean proof as more understandable than the HumanProof and Textbook proofs. But on Lemmas 2 and 4, they on average ranked understandability of the Lean proof to be less than the HumanProof and Textbook proofs. For participants' confidence in the correctness of the proof, the participants were generally certain of the correctness of all of the proofs. On Lemma 2, confidence was ranked Textbook < HumanProof < Lean. On Lemma 3 the

confidence was almost always saturated at 5 for all three proofs, meaning that this result had a ceiling effect. On Lemma 4, confidence was ranked Lean < HumanProof < Textbook.

6.5.2. Likelihood of preferences

Now consider the hypotheses outlined in Section 6.1. There are not enough data to use advanced models like [ordinal regression](#), but we can compute some simple likelihood curves for probabilities of certain propositions of interest for a given participant.

Let's find a likelihood curve for the probability that some participant will rank proof A above proof B , where A and B are from the set {Lean, HumanProof, Textbook}. Fixing a statistical model with parameters θ and dataset \mathbf{x} , define the likelihood $L(\mathbf{x}|\theta)$ as a function of θ proportional to the probability of seeing the data \mathbf{x} given parameters θ .

Take a pair of proofs $A, B \in \{\text{Lean, HumanProof, Textbook}\}$ and a fixed lemma $T \in \{\text{Lemma 2, Lemma 3, Lemma 4}\}$ and quality $Q \in \{\text{Unders., Conf.}\}$. Write $\#(A < B)$ to be the number of data in the dataset that evaluate $A < B$ to true. Hence from the data we get three numbers $\#(A < B)$, $\#(B < A)$ and $\#(A = B)$. In the case of a tie $A = B$, let's make the assumption that the participants 'true preference' is either $A < B$ or $B < A$, but that them reporting a tie means that the result could go either way. So let's model the proposition that $A < B$ for a random mathematician being drawn from a Bernoulli distribution with parameter π . For example, if the true value for π was 0.4 for Lean < Textbook on Lemma 2, we would expect a new participant to rank Lean below Textbook 40% of the time and Textbook below Lean 60% of the time. Our goal is to find a likelihood function for π given the data. We can write this as

$$\begin{aligned} L(\mathbf{x}|\pi) &= \prod_{x \in \mathbf{x}} L(x|\pi) \\ &= \pi^{\#(A < B)} (1 - \pi)^{\#(B < A)} P(\text{tie}|\pi)^{\#(A=B)} \end{aligned} \tag{6.6}$$

How to implement $P(\text{tie}|\pi)$? The assumption above tells us to break ties randomly. So that means that if there is one tie, there is a 50% chance of that being evidence for $A < B$ or 50% for $B < A$. Hence

$$L(\text{tie}|\pi) = \frac{1}{2}(\pi + (1 - \pi)) = \frac{1}{2} \tag{6.7}$$

Plotting the normalised likelihood curves for $L(\mathbf{x}|\pi)$ for each lemma, quality and pair of proofs is shown in Figure 6.8.

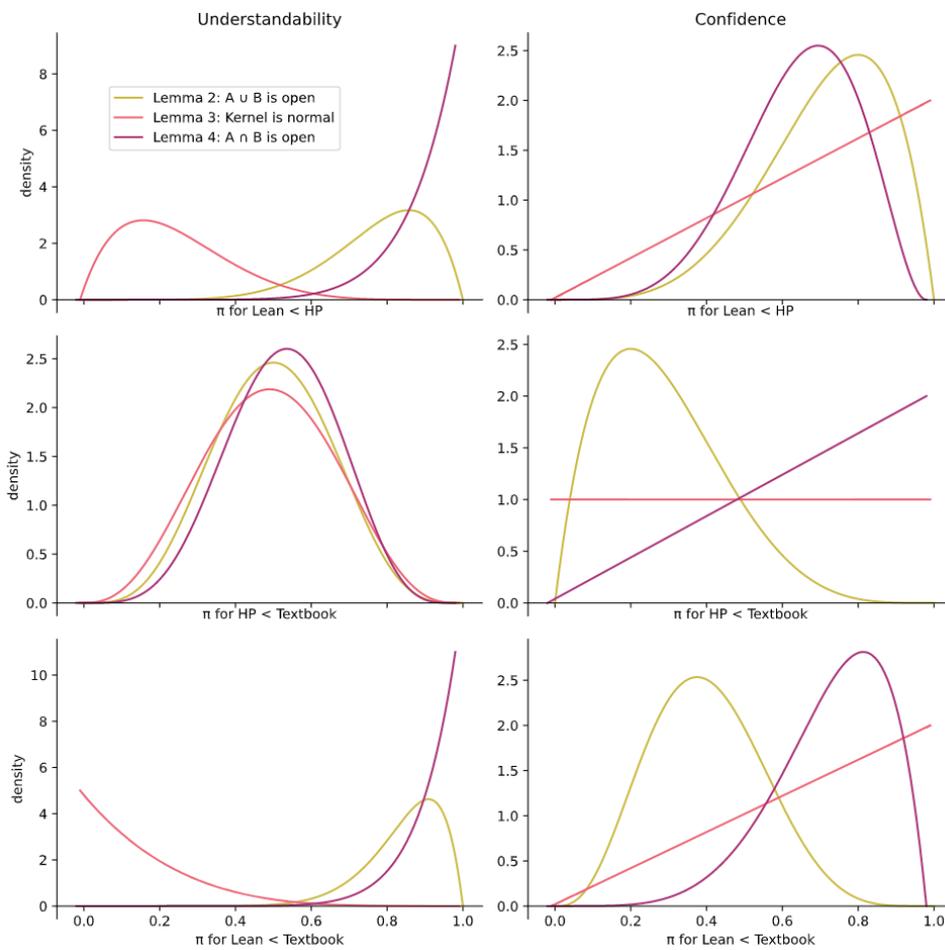


Figure 6.8. Normalised likelihood curves for the probability that a random mathematician will rate A less than B for a given lemma and quality. Note that each plot has a different scale, what matters is where the mass of the distribution is situated. This plot is analysed in Section 6.5.3.

Each curve has been scaled to have an area of 1. However each of the six plots have a different y-axis scaling to make the shapes as clear as possible. These graphs encapsulate everything that the data can tell us about each independent π . We can interpret them as telling us how to update our prior $p(\pi)$ to a posterior $p(\pi|\mathbf{x}) \propto p(\pi)L(\mathbf{x}|\pi)$. A curve skewed to the right for $A < B$ means that it is more likely that a participant will rank B above A. A curve with most of the mass in the center means that participants could rank A and B either way.

In the confidence plot for HumanProof < Textbook (mid right of Figure 6.8), the horizontal line for "kernel is normal" means that the given data tell us nothing about whether users prefer HumanProof or the Textbook proof. Consulting the raw data in Table 6.3, we can see that all 11 participants ranked these proofs equally, so the data can't tell us anything about which one they would really prefer if they had to choose a strict ordering.

6.5.3. Testing the hypotheses

We can interpret the likelihood curves in Figure 6.8 to test the three hypotheses given in Section 6.1. Each of the hypotheses takes the form of a comparison between how the participants ranked pairs of proof script types. The external validity of the conclusions (i.e., do the findings tell us anything about whether the hypothesis is true in general?) is considered in Section 6.7.

For the first hypothesis, I ask whether HumanProof natural language proofs are rated as more understandable than Lean proofs. The top left plot comparing the ratings of Lean to HumanProof proofs in Figure 6.8 contains the relevant π likelihoods. To convert each of these likelihood functions on π in to a posterior distribution $p(\pi|\mathbf{x})$, first multiply by a prior $p(\pi)$ since $p(\pi|\mathbf{x}) \propto p(\pi)L(\mathbf{x}|\pi)$. In the following analysis I choose a uniform prior $p(\pi) \propto 1$.

One way to answer the hypothesis is to take each posterior $p(\pi|\mathbf{x})$ and compute the area under the curve where $\pi > 0.5$. This will be the prior probability that the statement "HumanProof natural language proofs is preferred" for that particular lemma. The full set of these probabilities are tabulated in Table 6.9.

1. The participants ranked HumanProof natural language proofs as more understandable than Lean proofs. (The top-left plot in Figure 6.8.) Reading this gives a different conclusion for Lemma 3⁷¹ versus Lemmas 2 and 4⁷². For Lemma 3, we can see that users actually found the Lean proof to be *more* understandable than the natural language proofs $P(\pi > \frac{1}{2}) = 6\%$, we will see some hints as to why in Section 6.6.2. For Lemmas 2 and 4: 96% and 99.8% of the mass is above $\pi = \frac{1}{2}$.

2. The participants rank HumanProof natural language proofs as less understandable than proofs from a textbook. (The middle-left plot in Figure 6.8.) For all three lemmas a roughly equal amount of the area is either side of $\pi = 0.5$, suggesting that participants do not have a rating preference of textbook proofs versus HumanProof proofs for understandability. Hence there is no evidence to support hypothesis 2 from this experiment. As will be discussed in Section 6.7, I suspect that finding evidence for hypothesis 2 requires longer, more advanced proofs.

3. The participants are more confident in the correctness of a Lean proof to the natural language-like proofs. (The top-right and bottom-right plots in Figure 6.8.) For Lemma 3: the straight line indicates that only one participant gave a different confidence score for the proofs. As such a significant amount of the area lies on either side of $\pi = 0.5$, and so the evidence is inconclusive. For Lemma 2: we have an 89% chance that they are more confident in HumanProof over Lean and a 25% chance that they are more confident in Textbook over Lean. So this hypothesis resolves differently depending on whether it is the HumanProof or Textbook proofs. Finally with Lemma 4: the numbers are 85% and 94%, so the hypothesis resolves negatively; on Lemma 4 we should expect mathematicians to be more confident in the natural-language-like proofs, even though there is a guarantee of correctness from the Lean proof. The verbal reasons that participants gave when justifying this choice are explored in Section 6.6.1.

		Understandability	Confidence
p(Lean < HumanProof)	Lemma 2	96%	89%
	Lemma 3	6%	75%
	Lemma 4	99.8%	85%
p(HumanProof < Textbook)	Lemma 2	50%	11%
	Lemma 3	50%	50%
	Lemma 4	62%	75%
p(Lean < Textbook)	Lemma 2	99.7%	25%
	Lemma 3	3%	75%
	Lemma 4	99.95%	94%

So here, we can see that how the hypotheses resolve are dependent on the lemma in question. In order to investigate these differences, let us now turn to the verbal comments and responses that participants gave during each experiment session.

6.6. Qualitative analysis of verbal responses

In this section, I seek answers to the questions listed in Section 6.1. That is, what do the participants interpret the word 'understandable' to mean and what are their reasons for scoring the proofs as they did?

⁷¹ Lemma 3: the kernel of a group homomorphism is a normal subgroup (Appendix D.3.3).

⁷² Lemma 2: If A and B are open then $A \cup B$ is open (Appendix D.3.2). Lemma 4: If A and B are open then $A \cap B$ is open (Appendix D.3.4).

Table 6.9. A table of probabilities for whether a new participant will rank $X < Y$ for different pairs of proof scripts. As detailed in Section 6.5.2 and Section 6.5.3, these numbers are found by computing $p(\pi > \frac{1}{2}|\mathbf{x})$ on a uniform prior $p(\pi)$. For a fixed X, Y and quality q : π is the parameter of a Bernoulli distribution modelling the probability that a particular participant will rate $X < Y$ for q .

Schreier's textbook on qualitative content analysis [Sch12] was used as a reference for determining the methodology here. There are a few different techniques in sociology for qualitative analyses: 'coding', 'discourse analysis' and 'qualitative content analysis'. The basic idea in these is to determine the kinds of responses that would help answer ones research question (coding frame) and then systematically read the transcripts and categorise responses according to this coding frame.

[Sch12] **Schreier, Margrit** *Qualitative content analysis in practice* (2012) **publisher** SAGE Publications

To analyse the verbal responses, I transcribed the recordings of each session and *segmented* them into sentences or sets of closely related sentences. The sentences that expressed a reason for a preference between the proofs or a general comment about the understandability were isolated and categorised. If the same participant effectively repeated a sentence for the same Lemma, then this would be discarded.

6.6.1. Verbal responses on understandability

In the debriefing phase, participants gave their opinions on what it means for a proof to be understandable. I have coded these responses into four categories.

- **Syntax:** features pertaining to the typesetting and symbols used to present the proof.
- **Level of detail (LoD):** features to do with the amount of detail shown or hidden from the reader, for example, whether or not associativity steps are presented in a calculation. Another example is explicitly *vs.* implicitly mentioning that $A \subseteq A \cup B$.
- **Structure:** features to do with the wider layout of the arguments and the ordering of steps in the proof. I also include here the structure of exposition. So for example, when picking a value for introducing an exists statement, is the value fixed beforehand and then shown to be correct or is it converted to a metavariable to be assigned later?
- **Signposting:** features to do with indicating how the proof is going to progress without actually performing any steps of the proof, for example, explaining the intuition behind a proof or pausing to restate the goal.
- **Other:** anything else.

The presence of the 'other' category means that these codings are exhaustive. But are they mutually exclusive? We can answer this by comparing the categories pairwise:

- **Syntax/LoD:** An overlap here would be a syntactic device which also changes the level of detail. There are a few examples of this which are supported by Lean: implicit arguments⁷³ and casting⁷⁴. Both casting and implicit arguments hide unnecessary detail from the user to aid understanding. However since these devices are used by both Lean and natural language proofs they shouldn't be the reason that one is preferred over the other.
- **Syntax/Structure:** the larger layout of a proof document could be mediated by syntax such as `begin/end` blocks, but here a complaint on syntax would be a complaint against the choices of token used to represent the syntax. If the comment is robust to changing the syntax for another set of tokens, then it is not a comment about the syntax.
- **Syntax/Signposting:** similarly to Syntax/Structure, signposting has syntax associated with it, but we should expect any issue with signposting to be independent of the syntax used to denote it.
- **LoD/Structure:** LoD is different to structure in that if a comment is still valid after adding or removing detail (e.g., steps in an equality chain or an explanation for a particular result) then it is not a comment about LoD.
- **LoD/Signposting:** There is some overlap between LoD and signposting, because if a signpost is omitted, then this could be considered changing the level of detail. However I distinguish them by stipulating that signposts are not strictly necessary to complete the proof, whereas the omitted details *are* necessary but are not mentioned.

⁷³ For example, for the syntax representing a group product $a * b$, the carrier group is implicit.

⁷⁴ In `0.5 + 1`, the `1` is implicitly cast from a natural number to a rational number.

- **Structure/Signposting:** Similarly to LoD/Signposting, signposts can be removed while remaining a valid proof.

The verbal answers to the meaning of understandability are coded and tabulated in Table 6.10.

Table 6.10. Verbal responses prompted by 'What does it mean for a proof to be understandable to you?'. Each phrase in the first column is paraphrased to be a continuation of 'A proof is understandable when...'

A proof is understandable when ...	count	category
"it provides intuition or a sketch of the proof"	6	signposting
"it clearly references lemmas and justifications"	4	level of detail
"it emphasises the key difficult steps"	4	signposting
"each step is easy to check"	3	level of detail
"it is aimed at the right audience"	3	other
"it allows you to easily reconstruct the proof without remembering all of the details"	2	structure
"it hides bits that don't matter" (for example, associativity rewrites)	2	level of detail
"it explains the thought process for deriving 'inspired steps' such as choosing ϵ "	2	structure
"it is clear"	1	syntax, structure
"it is concise"	1	syntax, structure
"it has a balance of words and symbols"	1	syntax

Additionally, 4 participants claimed that just being easy to check does *not* necessarily mean that a proof is understandable. So the data show that signposting and getting the right level of detail are the most important aspects of a proof that make it understandable. The syntax and structure of the proof mattered less in the opinion of the participants.

6.6.2. Reasons for rating proofs in terms of understandability

Now I turn to the two qualitative questions asked in Section 6.1:

1. What properties of a proof contribute to, or inhibit, understandability?
2. How does having a computer-verified proof influence a mathematician's confidence in the correctness of the proof?

In order to answer these, we need to find general reasons why respondents ranked certain proofs above others on both the understandability and confidence quality.

The main new category to introduce here is whether the participants' reasons for choosing between the proofs are an intrinsic part of the proof medium or a presentational choice that can be fixed easily. For example, it may be preferred to skip a number of steps in an equality calculation, but it might be awkward to get this to verify in Lean in a way that might not be true for smaller proofs.

Since the numerical results for the metric space lemmas tell a different story to the group theory lemma, I separate the analysis along these lines.

For the group homomorphism question, the Lean proof was usually preferred in terms of understandability to the HumanProof and textbook proofs. The main reasons given are tabulated in Table 6.11. I write $\boxed{H+}$ or $\boxed{L-}$ and so on to categorise whether the comment is a negative or positive opinion on the given proof. So $\boxed{T+}$ means that the quote is categorised as being positive about the textbook proof. Write $\boxed{=}$ to mean that the statement applies to all of the proofs.

Table 6.11. Table of reasons participants gave for their ranking of the understandability of proofs for Lemma 3.

	paraphrased quote	count	category	judgement
1	"I would rather that $f(g * k * g^{-1}) = f g * f k * (f g)^{-1}$ be performed in one step"	2	level of detail	L-
2	"I would rather that $f(g * k * g^{-1}) = f g * f k * (f g)^{-1}$ be performed in two steps"	2	level of detail	L+
3	"I like how the HumanProof proof does not need to apply $(f g)^{-1} = f (g^{-1})$ "	2	structure	H+
4	"I prefer the Lean proof because it explains exactly what is happening on each step with the <code>rewrite</code> tactics"	2	level of detail	L+
5	"I don't like how none of the proofs explicitly state that proving $f(g * k * g^{-1}) = e$ implies that the kernel is therefore normal"	2	level of detail	=
6	Express a lack of difference between the proofs of Lemma 3	2	structure	=
7	"The textbook proof is too hard to read because the equality chain is all placed on a single line"	2	syntax, level of detail	T-
8	"In the textbook proof, I dislike how applying $f(g * k * g^{-1}) = f g * f k * (f g)^{-1}$ together with $(f g)^{-1} = f (g^{-1})$ is performed in one step"	1	level of detail	T-
9	"It is not clear in the equality chain exactly where the kernel property is used in the HumanProof proof"	1	level of detail	H-

Rows 2, 4 and 7 in Table 6.11 suggest that the main reason why Lean tended to rank higher in understandability for Lemma 3 is because the syntax of Lean's `calc` block requires that the proof terms for each step of the equality chain be included explicitly. Participants generally found these useful for understandability, because it meant that each line in the equality chain had an explicit reason instead of having to infer which rewrite rule was used by comparing expressions (this is a case where the Lean proof's higher level of detail was actually a good thing rather than getting in the way). This suggests that future versions of HumanProof should include the option to explicitly include these rewrite rules.

Rows 1 and 2 in Table 6.11 show that there is also generally disagreement on what the most understandable level of detail is in terms of the number of steps that should be omitted in the equality chain.

Now let us turn to the metric space lemmas in Table 6.12.

Table 6.12. Table of reasons participants gave in their ranking of the understandability of proofs for Lemmas 2 and 4.

	paraphrase quote	count	category	judgement
1	"The textbook proof of Lemma 2 is too terse"	5	level of detail	T-
2	"The textbook proof is what I would see in a lecture but if I was teaching I would use HumanProof"	5	structure, level of detail	H+
3	Expressing shock or surprise upon seeing the Lean proof	5	level of detail	L-
4	"I like the use of an explicit 'we must show' goal statement in the HumanProof proof"	3	structure	H+
5	"The Lean proof includes too much detail"	3	level of detail	L-
6	"In Lemma 2, the last paragraph of HumanProof is too wordy / useless"	3	level of detail, structure	H-
7	"It is difficult to parse the definition of <code>is_open</code> "	2	syntax	=
8	"I prefer ' $x \in A$ whenever $\text{dist } y x < \epsilon$ for all x ' to ' $\forall x, \text{dist } y x < \epsilon \rightarrow x \in A$ '"	2	syntax	=
9	"In the Lean proof, it is difficult to figure out what each tactic is doing to the goal state"	2	structure, syntax	L-
10	"Lean gives too much detail to gain any intuition on the proof"	2	level of detail	L-
11	"I prefer HumanProof's justification of choosing ϵ , generally"	2	structure	H+
12	"I prefer HumanProof's justification of choosing ϵ , but only for the purposes of teaching"	2	structure	H+
13	"I prefer the lack of justification of choosing ϵ "	2	structure	T+
14	"It is difficult to parse the definition of <code>is_open</code> "	2	syntax	=
15	"I prefer ' $\forall x, \text{dist } y x < \epsilon \rightarrow x \in A$ ' to ' $x \in A$ whenever $\text{dist } y x < \epsilon$ for all x '"	1	syntax	=
16	"Knowing that the 'similarly' phrase in Textbook proof of Lemma 2 requires intuition"	1	level of detail	T
17	"Both HumanProof and textbook proofs of Lemma 4 are the same"	1	structure	H=T

Here, most of the criticisms of Lean are on the large level of detail that the proof needs (rows 1, 2, 3, 5, 6, 10, 16 of Table 6.12). Now, to some extent the amount of detail included has been inflated by my representation of the lemma to ensure that the participants can read through the proof without having an interactive window open, so this might be more of a complaint about how I have written the proof than an intrinsic problem with Lean.

Another common talking point (rows 12, 13, 14 in Table 6.12) was the way in which HumanProof structured proofs by delaying the choice of ϵ to when it is clear what the value should be, rather than the Lean and Textbook proofs which choose ϵ up-front. There was not agreement among the participants about which structure of proof is better. One participant noted that they preferred proofs where ' ϵ is pulled out of a hat and checked later'.

Row 1 of Table 6.12 shows that participants generally struggled with the textbook proof of Lemma 2 (Appendix D.3.2). This might be because the original proof was stated in terms of open ϵ -balls which I replaced with a $\forall \exists$ expression, and this change unfairly marred the understandability of the textbook proof. This was done to ensure that the same definitions were used across all versions of the proof.

I also provide below in Table 6.13 some general remarks on the proofs that weren't specific to a particular lemma.

Table 6.13. Table of general reasons participants gave in their ranking of the understandability of proofs that apply to all of the Lemmas.

	Paraphrase quote	count	category	judgement
1	"I am finding Lean difficult to read mostly because I am not used to the syntax rather than because the underlying structure is hard to follow"	6	syntax	L-
2	"The Lean proof makes sense even though I don't know the tactic syntax well"	5	syntax	L+
3	"I am getting used to reading Lean", "This is much easier now that I am familiar with the syntax"	4	syntax	L+
4	"Lean really focusses the mind on what you have to prove"	2	level of detail	L+
5	"In HumanProof and textbook proofs I find reading the Lean-like expressions to be difficult"	1	syntax	=

The trend in Table 6.13 is syntax. Many participants were keen to state that they found the Lean tactic syntax difficult to read, but they also stated that this shouldn't be a reason to discount a proof as being difficult to understand because it is just a matter of not being used to the syntax rather than an intrinsic problem with the proof. When asked to give a reason why the Lean proof was found to be less understandable for Lemmas 2 and 4, the reason was usually about level of detail and syntax rather than about the structure of the proof. The take-home message here is that while newcomers are likely to find the new syntax of Lean strange, they are willing to learn it and do not see it as a problem with formal provers.

6.6.3. Verbal responses for confidence

When the participants gave a rating of their confidence in the correctness of the proofs, I reminded them that the Lean proofs had been verified by Lean's kernel. Looking at the numerical results in the right column of Figure 6.8 and the results of Section 6.5.3, we can see that participants are more likely to be confident in the HumanProof proofs than the Lean proofs, and more likely to be confident in the Textbook proofs over the Lean proofs (with the exception of Lemma 2). As discussed, the signal is not very strong but this suggests that knowing that a proof is formally verified does not necessarily make mathematicians more confident that the proof is correct.

Three participants volunteered that they were less certain of Lean because they don't know how it works and it might have bugs. Meanwhile one participant, after ranking their confidence in Lean lower, stated

"I can't tell if the reason I said that I am less confident was just an irrational suspicion or something else... I can't figure out what kind of mistake... in my mind there might exist some kind of mistake where it would logically be fine but somehow the proof doesn't work, but I don't know if that's a real thing or not."

This suggests a counterintuitive perspective; convincing mathematicians to have more confidence in formal methods such as provers is not a problem of verifying that a given proof is correct. Instead they need to be able to see why a proof is true.

6.6.4. Summary of verbal responses

To wrap up, in this section we have explored the evidence that the verbal responses of participants provide for the research questions laid out in Section 6.1.

What properties of a proof contribute to, or inhibit, the understandability of a proof? The most commonly given property is on signposting a proof by providing an intuition or sketch of the proof, followed by getting the level of detail right; skipping over parts of the proof that are less important while remaining easy to check. Syntactic clarity was less important.

Does having a formal guarantee of correctness increase a mathematician's confidence in the correctness of the proof? The answer seems to be no, but perhaps this will change as the mathematicians have more experience with an interactive theorem prover.

6.7. Threats to validity and limitations

Here, I list some of the ways in which the experiment could be invalidated. There are two kinds of threats: internal and external. An internal threat is one which causes the experiment to not measure what it says it is measuring. An external threat is about generalisation; do the results of the experiment extend to broader claims about the whole system?

Below I list the threats to validity that I have identified for this study.

Confounding – Some other aspect of how the proofs are presented might be causing participants to be biased towards one or the other (e.g., if one uses \TeX typeset mathematics vs monospace code, or dependent on the choice of variable names). Because of this I have changed the human-written natural language proof scripts to use Lean notation for mathematical expressions instead of \TeX . Another defence against this threat is to simply *ask* the participants about why they chose to rate the proofs as they did. However, it is possible that this bias is subconscious and therefore would not be picked up in the verbal responses ("I don't know why I prefer this one"...).

Selection bias – Participants are not drawn randomly from the population but are drawn from people who answer an advert for a study. This could cause a bias towards users who are more interested in ITP and ATP⁷⁵. I defend against this with the question in the debrief phase asking what their prior experience is with ITP and ATP. Additionally, it is not mentioned that the HumanProof-generated proof is generated by a machine, so being biased towards ITP would cause the bias to be towards the Lean proof rather than towards the generated proof.

Maturation/ learning effects – During the course of the experiment, the participants may become used to performing the rankings and reading the two Lean-based proofs. The training phase and throwing away the results of the first task should help remove any effects from the participants being unfamiliar with the material. The lemmas and proofs are presented in a randomised order so at least any learning effect bias is distributed evenly over the results.

⁷⁵ Interactive Theorem Proving and Automated Theorem Proving, see Section 2.1.2.

'Cached' scoring – Similar to maturation, participants may remember their scores to previous rounds and use that as a shortcut to thinking about the new problem presented to them. This is partly the purpose of asking the participants to explain why they ranked as they do, which should cause them to think about their rating more than relying on cached ratings.

Experimenter bias – I have a bias towards the HumanProof system because I built it. This could cause me to inadvertently behave differently when the participant is interacting with the HumanProof proof script. In order to minimise this, I tried to keep to the experimenter script as much as possible. I have also introduced bias in how I have selected the example questions. The textbook proofs were chosen from existing mathematical textbooks, but I perhaps introduced bias when I augmented them to bring them in line with the other two proofs. To some extent the Lean proofs are not representative of a Lean proof that one would find 'in the wild'. I couldn't use an existing mathlib⁷⁶ proof of the same theorem because Lean proofs in mathlib can be difficult to read without the Lean goal-state window visible and without first being familiar with the definitions and conventions which are highly specific to Lean. Although this bias may cause participants' perceptions to be changed, an impression I got during the sessions was that the participants were generally sympathetic to the Lean proofs, possibly because they assumed that I was primarily interested in the Lean proof scripts as opposed to the HumanProof proof which was not declared to be computer generated.

⁷⁶ Mathlib is Lean's mathematical library.

Generalisation - The lemmas that have been selected are all elementary propositions from the Cambridge undergraduate mathematics course. This was necessary to make sure that the proofs could be understood independently of the definitions and lemmas that are required to write down proofs of later results and to ensure that the participants could review the proofs quickly enough to complete each session in under an hour. However this choice introduces a threat to external validity, in that it is not clear whether any results found should generalise to more advanced lemmas and proofs. More experiments would need to be done to confirm whether the results and responses generalise to a more diverse range of problems and participants. Do we have a strong prior reason to expect that participants would answer differently on more advanced lemmas and proofs? My personal prior is that they would rank Lean even worse, most notably because the result would rely on some unfamiliar lemmas and concepts such as filters and type coercions. I also suspect that HumanProof would perform worse on more advanced proofs, since the proofs would be longer and hence a more sophisticated natural language generation system would be needed to generate the results.

Ceiling effects - Some of the ratings have ceiling effects, where the ratings data mostly occurs around the extrema of the scale. This can be seen to occur for the understandability and confidence scores for Lemma 3.

Sample size - There were only 11 participants in the study. This low sample size manifests in the results as the fat likelihood curves in Figure 6.8. However, the study was designed to be fairly informal, with the quantitative test's purpose mainly to spark discussion, so it is not clear to me that any additional insight would be gained from increasing the study size.

6.8. Conclusions and future work

In this chapter I have evaluated the natural language write-up component of HumanProof by asking mathematicians to compare it against Lean proofs and proofs derived from textbooks. The results give different answers for equational reasoning proofs as opposed to the more structural natural language proofs and Lean proofs. This suggests that we should come to two conclusions from this experiment.

In the case of structural proofs, users generally prefer HumanProof and textbook proofs to Lean proofs for understandability. The participants reported that to some extent it was

because they found the natural language proofs more familiar, but also because the Lean proofs were considered too detailed.

However the qualitative, verbal results showed (Section 6.6.2) that participants were usually sympathetic towards the Lean structure, only wishing that the proofs be a little more terse, provide signposts and hide unnecessary details.

In the wider sense, we can use the results from this study to see that mathematicians are willing to move to different syntax, but that the high-level structure and level of detail of formalised proofs needs to be improved in order for them to adopt ITP. The study confirms that producing natural language proofs at a more human-like level of detail is useful for mathematicians. It also provides some surprising complaints about how proofs are typically laid out in textbooks where a more formal and detailed approach would actually be preferred, as we saw for the group homomorphisms question, where the Textbook version of the proof was considered to be too terse.

Within the wider scope of the thesis and the research questions in Section 1.2, the study presented in this chapter seeks to determine whether software can produce formalized, understandable proofs. The study shows that HumanProof system developed in Chapter 3 and Chapter 4 can help with understandability (as shown in Section 6.5 for the case of Lemmas 2 and 4).

Chapter 7

Conclusion

In this thesis, I have presented the design and evaluation for HumanProof and ProofWidgets. In this chapter I will conclude the thesis by reflecting on the research questions given in Chapter 1 and talking about further work.

7.1. Revisiting the research questions

Let's review the research questions that I set out to answer in Section 1.2 and outline the contributions that I have made towards them.

7.1.1. What constitutes a human-like, understandable proof?

Identify what 'human-like' and 'understandable' mean to different people.

In Section 2.5, I investigated the work of other mathematicians and mathematics educators on the question of what it means to understand a proof of a theorem. This review took me from educational research to the works of Spinoza, but yielded little in the way of explicit answers to this question. In Chapter 6, I asked some students of mathematics at the University of Cambridge what features of a proof made it understandable to them. The participants remarked that a proof being understandable is a function of numerous factors: providing the intuition or motivation of a proof first, signposting the purpose of various sections of a proof and providing the right level of detail. One thing that was frequently stressed, however, was that syntax and notation of proofs only played a minor role in how understandable a proof is; while unfamiliar syntax only hinders understanding temporarily and may be overcome by becoming familiar with the notation.

Distinguish between human-like and machine-like in the context of ITP. A similar review was undertaken in Section 2.6 for what constitutes 'human-like' reasoning. The topic has received attention from early efforts to create proof assistants and automated theorem provers up to the present day. My conclusion from this review is that 'human-like' is best understood as referring to a general approach to ATP algorithm design, in contrast to 'machine-like'. Human-like proving techniques emphasise reasoning methods that are compatible with how humans reason, in the sense that a proof is intelligible for a human. Pre-80s, for example, Robinson's *resolution* theorem proving was the dominant architecture of provers [BG01]. However as noted by Bledsoe [Ble81], repeated application of the resolution rule $(A \vee B) \wedge (\neg A \vee C) \vdash B \vee C$ can hardly be called 'human-like', and such a proof would not be found in a mathematical textbook. Since human-like is defined more in terms of what it is not, there are a wide variety of approaches which may all be described as human-like: proof planning, diagrammatic reasoning, and graphical discourse models. I chose to focus on human-like logical reasoning and modelling how an undergraduate may approach writing a proof.

Merge these strands to create and determine a working definition of human-like. In Section 3.1, I decided that I would deem the design of the system as human-like if it was similar enough to the reasoning of humans and could produce natural language write-ups that were convincing enough for mathematicians. I also restricted myself to only look at elementary 'follow your nose' proofs. That is, simple proofs where at each step of the proof the number of sensible steps (according to a human) is highly restricted.

[BG01] **Bachmair, Leo; Ganzinger, Harald** *Resolution theorem proving* (2001) Handbook of automated reasoning

[Ble81] **Bledsoe, Woodrow W** *Non-resolution theorem proving* (1981) Readings in Artificial Intelligence

7.1.2. How can human-like reasoning be represented within an interactive theorem prover to produce formalised, understandable proofs?

Form a calculus of representing goal states and inference steps that acts at the abstraction layer that a human uses when solving proofs. In Chapter 3 I detail a development calculus created for the purpose of representing human-like goal states in accordance with the working definition given in Section 7.1.2. The calculus (defined in Section 3.3.2) makes use of a hierarchical proof state structure that also incrementally constructs a formal proof term in dependent type theory. The calculus is compared with other designs (Section 3.3.5), of which the closest is the design to Gowers and Ganesalingam's *Robot* prover [GG17] and McBride's *OLEG* [McBoo]. I then provided a set of tactics for manipulating these proof states and provide proofs that these tactics are sound (Section 3.4 and Appendix A).

In Chapter 4, I also introduce a new algorithm for creating human-like equational reasoning proofs. This algorithm makes use of a hierarchical set of 'subtasks' to find equality proofs. This system can solve simple equational reasoning tasks in a novel way. It works well as a prototype but the subtasks system will need more improvement before it can be used in a practical setting.

Create a system for also producing natural language proofs from this calculus.

The component that performs this is detailed in Section 3.6. The system for verbalising HumanProof-constructed proofs to natural language made use of a classical NLG pipeline. This component need only be good enough to demonstrate that the calculus of HumanProof can create human-like proofs, and so I did not focus on innovating beyond existing work in this field, notably Gowers & Ganesalingam [GG17] and earlier systems such as PROVERB [HF97]. However I did contribute a system for verbalising sequences of binders in dependent type theory in Section 3.6.4.

Evaluate the resulting system by performing a study on real mathematicians.

This was discussed in Chapter 6. The study found that mathematicians generally prefer HumanProof proofs to Lean proofs with the exception of equality proofs, where the additional information required to specify each step in an equality chain was preferred. A surprising result of the study was that non-specialist mathematicians do not trust proofs backed with a formal guarantee more than natural language proofs. This result suggests that - considering a proof assistant as a tool for working mathematicians - formalisation can't be a substitute for an understanding of the material.

I made some significant progress towards this research goal, however the solution that I have implemented can be found to stumble upon given harder examples, both in terms of automation and in the write-ups getting progressively clunkier upon growing in size. The implementation as it stands also does not extend to more difficult domains where some detail must be hidden in the name of brevity. I will outline some specific solutions to these issues in Section 7.2. In the end, I chose to focus less on extending the automation of tactics beyond what was available in *Robot* and instead focus on subtasks described in Chapter 4 and interactive theorem proving through a graphical user interface.

I believe that these defects could be fixed with more research, however one has to ask whether such a human-research-intensive approach is going to be a good long-term solution. This question becomes particularly salient when faced with the advent of large-scale deep learning language models:

Very recently, we are starting to see applications of attention based models [VSP+17] (also known as transformers) to the problem of predicting human-written proofs of mathematics with promising results: Li *et al* direct transformer models towards predicting steps in Isabelle/Isar⁷⁷ proofs [LYWP21]. See also Lample and Charton's work on applying transformers to algebraic reasoning and integration [LC20]. Some work in this space that I have been involved with is with Han, Rute, Wu and Polu [HRW+21] on training GPT3

[GG17] **Ganesalingam, Mohan; Gowers, W. T.** *A fully automatic theorem prover with human-style output* (2017) Journal of Automated Reasoning

[McBoo] **McBride, Conor** *Dependently typed functional programs and their proofs* (2000) PhD thesis (University of Edinburgh)

[GG17] **Ganesalingam, Mohan; Gowers, W. T.** *A fully automatic theorem prover with human-style output* (2017) Journal of Automated Reasoning

[HF97] **Huang, Xiaorong; Fiedler, Armin** *Proof Verbalization as an Application of NLG* (1997) International Joint Conference on Artificial Intelligence

[VSP+17] **Vaswani, Ashish; Shazeer, Noam; Parmar, Niki; et al.** *Attention is All you Need* (2017) Neural Information Processing Systems

⁷⁷ [Wen99], see Section 2.6.1

[BMR+20] to predict Lean 3 tactics. The success of this approach strongly suggests that deep learning methods will play a critical role to the future of human-like automated reasoning. Through the use of statistical learning, the nuances of generating natural language and determining a precise criterion for what counts as 'human-like' can be avoided by simply providing a corpus of examples of human-like reasoning. Deep learning models are notorious for being data-hungry, and so there are still many questions remaining on how the data will be robustly extracted from our mathematical texts and formal proof archives. Perhaps few-shot techniques (see [BMR+20]) will help here. The research touched on above indicates that this method is not incompatible with also producing formalised proofs, although some care will need to be taken to be sure that the formal proofs and the human-readable accounts correspond correctly to each other.

7.1.3. How can this mode of human-like reasoning be presented to the user in an interactive, multimodal way?

Investigate new ways of interacting with proof objects. The result of working on this subgoal was the interactive expression engine of the ProofWidgets framework as discussed in Chapter 5. This system follows a long history of research on 'proof-by-pointing' starting with Bertot and Théry [BT98], and my approach mainly follows similar work in other systems, for example, the system found in KeY [ABB+16]. My approach is unique in the coupling of the implementation of proof-by-pointing with the general purpose ProofWidgets framework.

Make it easier to create novel GUIs for interactive theorem provers. This was the primary mission of Chapter 5. As noted in Section 5.1, there are many existing GUI systems that are used to create user interfaces for interactive theorem proving. In Chapter 5 I contribute an alternative paradigm for creating user interfaces where the metalanguage of the prover itself is used to create proofs. The ProofWidgets system as implemented in Lean 3 is already in use today.

Produce an interactive interface for a human-like reasoning system. In Section 5.8, I connected the ProofWidgets framework to the `Box` datastructure developed in Chapter 3 to create an interactive, formalised human-like proof assistant. This serves as a prototype to achieve the research goal. There are many more implementation improvements that could be made and future directions are provided in Section 7.2.

I hope that this work will be viewed as a modern revival of the spirit and approach taken by the older, proof-planning-centric provers such as LQUI for Ω [SHB+99, BCF+97] and XBarnacle for CLAM [LD97]. This spirit was to make proof assistants accessible to a wider userbase through the use of multi-modal user interfaces that could represent the proofs in many different ways. I want to rekindle some of this optimism that better user interfaces can lead to wider adoption of ITP.

7.2. Future work and closing remarks

There are still many things that I want to do to HumanProof and to investigate the world of human-like automated reasoning. Some of the more technical and chapter-specific ideas for future work are covered in their respective chapters (Section 3.7, Section 4.6, Section 5.9, Section 6.8), in this section I restrict my attention to future research directions in a broader sense.

In this thesis, the purpose of the natural language generator was to demonstrate that the system was human-like. However, one question that arises from the evaluation study in Chapter 6 is whether natural language generation is useful for creating accessible ITPs. Participants were generally willing to learn to use a new syntax and language for mathematics⁷⁸, which suggests that the main hurdle to adoption is not the use of a technical language. An additional focus group or study investigating whether natural language proofs play a role in easing the use of a theorem prover would be helpful in

[Wen99] **Wenzel, Markus** *Isar - A Generic Interpretative Approach to Readable Formal Proof Documents* (1999) Theorem Proving in Higher Order Logics

[LYWP21] **Li, Wenda; Yu, Lei; Wu, Yuhuai; et al.** *IsarStep: a Benchmark for High-level Mathematical Reasoning* (2021) International Conference on Learning Representations

[LC20] **Lample, Guillaume; Charton, François** *Deep Learning For Symbolic Mathematics* (2020) ICLR

[HRW+21] **Han, Jesse Michael; Rute, Jason; Wu, Yuhuai; et al.** *Proof Artifact Co-training for Theorem Proving with Language Models* (2021) arXiv preprint arXiv:2102.06203

[BMR+20] **Brown, Tom B.; Mann, Benjamin; Ryder, Nick; et al.** *Language Models are Few-Shot Learners* (2020) NeurIPS

[BT98] **Bertot, Yves; Théry, Laurent** *A generic approach to building user interfaces for theorem provers* (1998) Journal of Symbolic Computation

[ABB+16] **Ahrendt, Wolfgang; Beckert, Bernhard; Bubel, Richard; et al.** *Deductive Software Verification - The KeY Book* (2016) publisher Springer

[SHB+99] **Siekman, Jörg; Hess, Stephan; Benzmüller, Christoph; et al.** *LOUI: Lovely OMEGA user interface* (1999) Formal Aspects of Computing

[BCF+97] **Benzmüller, Christoph; Cheikhrouhou, Lassaad; Fehrer, Detlef; et al.** *Omega: Towards a Mathematical Assistant* (1997) Automated Deduction - CADE-14

[LD97] **Lowe, Helen; Duncan, David** *XBarnacle: Making Theorem Provers More Accessible* (1997) 14th International Conference on Automated Deduction

⁷⁸ Although note that this may be due to sample selection bias (see Section 6.7).

determining whether natural language generation of mathematics should be pursued in the future.

The evaluation showed that mathematicians value signposting⁷⁹, motivation and the right level of detail. I didn't focus on these aspects so much in the design of HumanProof. Is there a way of automating these more expository aspects of human-written proofs? The question of determining the right level of exposition has some subjective and audience-specific component to it, however I suspect that it is still possible to make some progress in this direction: the gap in comprehensibility between a human-written proof and a generated proof for any non-trivial example is undeniable. Rather than trying to build an 'expert system' of determining the right level of exposition, I think that the right approach is to use modern machine learning approaches as touched on in Section 7.1.2.

⁷⁹ As discussed in Section 6.6.1, signposting here means an indication of how the proof is going to progress without actually performing any inference

7.2.1. Closing remarks

HumanProof is a prototype and thus not production-ready software for day-to-day formalisation. I do hope that HumanProof will provide some ideas and inspiration in how the theorem provers of the future are designed.

Appendix A

Zippers and tactics for boxes

This is a technical appendix on soundly running Lean tactics within a `Box`. It also provides some of the more technical background omitted from Section 2.4, such as the inference rules for the development calculus of Lean. In this I appendix I describe an 'escape-hatch' to use Lean tactics within a `Box` proof, meaning that we don't have to forgo using any preexisting tactic libraries by using the `Box` framework. Ultimately, these meta-level proofs are not critical to the thesis because the resulting object-level proofs produced through these box-tactics is also checked by the Lean kernel.

A.1. Typing of expressions containing metavariables

In this section I provide a set of formal judgements describing a theory of the metavariable system of Lean. When Lean typechecks a proof or term (described in Section 2.1), it is checked with respect to a dependent type theory called the calculus of constructions [CH88]. However, Lean also allows terms to contain special variables called metavariables for producing partially constructed proofs. Background on metavariables is provided in Section 2.4. Although Lean's kernel does not check expressions containing metavariables, it is nevertheless important to have an understanding of the theory of metavariables to assist in creating valid expressions.

In this section I extend the Lean typing rules presented by Carneiro [Car19] to also handle typing judgements over expressions containing metavariables. These definitions are used in Appendix A.2 and Appendix A.3 to run Lean's native tactics within a `Box` context. I used the knowledge written in this section to [implement an interface](#) in the Lean 3 metaprogramming framework [EUR+17] for fine-grained control over the metavariable context.

The work in this section is not an original contribution, because de Moura and the other designers of Lean 3 had to produce this theory to create the software in the first place. To my knowledge, however, there is currently no place where the theory is written down in the same manner as Carneiro's work. There are also many accounts of the theories of development calculi similar to this [SH17, McBoo, Spi11]. I also do not offer a comprehensive account of the theory of Lean's development calculus, instead only including the parts that are needed to prove later results. The information in this section is gleaned from [EUR+17] and [MAKR15], the sourcecode of the [Lean 3 theorem prover](#) and through many informal chats with the community on the [leanprover Zulip server](#).

I start by repeating some definitions (A.1) that were given in Section 2.4. An explanation of the notation used in this appendix can be found in Section 2.2.

[CH88] **Coquand, Thierry; Huet, Gérard P.** *The Calculus of Constructions* (1988) Information and Computation

[Car19] **Carneiro, Mario** *Lean's Type Theory* (2019) Masters' thesis (Carnegie Mellon University)

[EUR+17] **Ebner, Gabriel; Ullrich, Sebastian; Roesch, Jared; et al.** *A metaprogramming framework for formal verification* (2017) Proceedings of the ACM on Programming Languages

[SH17] **Sterling, Jonathan; Harper, Robert** *Algebraic Foundations of Proof Refinement* (2017) CoRR

[McBoo] **McBride, Conor** *Dependently typed functional programs and their proofs* (2000) PhD thesis (University of Edinburgh)

[Spi11] **Spiwack, Arnaud** *Verified computing in homological algebra, a journey exploring the power and limits of dependent type theory* (2011) PhD thesis (INRIA)

[MAKR15] **de Moura, Leonardo; Avigad, Jeremy; Kong, Soonho; et al.** *Elaboration in Dependent Type Theory* (2015) CoRR

Binder := Name × Expr

Context := List Binder

```
Expr ::=
| app : Expr → Expr → Expr
| lam : Binder → Expr → Expr
| pi  : Binder → Expr → Expr
| var : Name → Expr
| mvar : Name → Expr
| const : Name → Expr
| sort : {Level} → Expr
```

```
MvarDecl :=
  (name      : Name)
  × (type    : Expr)
  × (ctx     : Context)
  × (assignment : Option Expr)
```

MvarContext := List MvarDecl

Let $M : \text{MvarContext}$ and $\Gamma : \text{Context}$. Binder s are sugared as $(x : \alpha)$ ⁸⁰ for $\langle x, \alpha \rangle : \text{Binder}$. Unassigned MvarDecl s $(\langle x, \alpha, \Gamma, \text{none} \rangle : \text{MvarDecl})$ are sugared as $(\Gamma \vdash ?x : \alpha)$ and assigned MvarDecl s $(\langle x, \alpha, \Gamma, \text{some } t \rangle : \text{MvarDecl})$ are sugared as $(\Gamma \vdash ?x : \alpha = t)$ (the $\Gamma \vdash$ may be omitted if not relevant). I use the convention that metavariable names always begin with question marks. Given $x : \text{Name}$, write $x \in \Gamma$ when x appears in $\Gamma : \text{Context}$.

To simplify analysis, I assume that all contexts do not include variable clashes. That is to say, there are no two binders with the same name in Γ or M . Since there are infinitely many variable names to choose from, these kinds of variable clashes can be avoided through renaming.

Each of the definitions in (A.1) are assignable as defined in Definition 2.35 in Section 2.4.2.

Definition A.2: Also define the following for $r : R$ where R is assignable.

- $\text{fv}(r)$ are the free variables in r .
- $\text{mv}(r)$ are the metavariables in r .
- $\text{umv}(M, r)$ are the unassigned metavariables in r (according to context M).
- $\text{amv}(M, r)$ are the assigned metavariables in r .

Definition A.3 (substitution): In Definition 2.29, a substitution σ is defined as a partial map $\text{Sub} := \text{Name} \rightarrow \text{Expr}$ sending variable names to expressions. Given $\sigma : \text{Sub}$ and an $r : R$ where R is assignable, σr replaces each variable in r with the corresponding expression in σ ⁸¹. The notation for substitutions I use is $\{x \mapsto t\}$ where $x : \text{Name}$ is the variable to be substituted and $t : \text{Expr}$ is the replacement expression⁸². I extend this definition to include metavariable substitutions; $\{?m \mapsto t\}$ is the substitution replacing each instance of $?m$ with the expression t .

A metavariable context M can be viewed as a substitution mapping $?m$ to t for each assigned declaration $\langle ?m, \alpha, \Gamma, \text{some } t \rangle \in M$. That is, M acts by replacing each instance of an assigned metavariable with its assignment.

Definition A.4 (instantiation): Given an assignable e , write $M e$ to be the M -instantiation of e , created by performing this substitution to e . e is M -instantiated when all of the metavariables present in e are unassigned with respect to M .

Definition A.5 (flat): Say that a metavariable context M is **flat** when $\text{amv}(M, M) = \emptyset$. That is, when there are no assigned metavariables in the expressions found in M .

Definition A.6 (dependency ordering): Given an assignable r , say that r **depends on** $?m$ when $?m \in \text{mv}(r)$. Given a pair of declarations $d_1 = \langle ?m_1, \alpha_1, \Gamma_1, o_1 \rangle$ and $d_2 = \langle ?m_2, \alpha_2, \Gamma_2, o_2 \rangle$ in M , write $d_1 \triangleright d_2$ when d_2 depends on $?m_1$. That is $d_1 \triangleright d_2$ when α_2 or Γ_2 or o_2 depend on $?m_1$. I will write $?m_1 \triangleright ?m_2$ as a shorthand for $d_1 \triangleright d_2$ when it is clear what the declarations are from context.

(A.1). Recap of the definitions of contexts. A Name is a list of strings and numbers and acts as an identifier. In Lean 3, a distinction is made between free and bound var s, but this is simplified here. Under the hood, MvarContext is implemented as a dictionary keyed on Name instead of as a List . The sort expression represents the type of types or propositions depending on the value of the Level parameter.

⁸⁰ Note the use of a smaller colon $:$ for typing judgements vs $:$ for meta-level type assignments.

⁸¹ With the usual caveats for variable clashes as noted in Definition 2.29

⁸² The reader may enjoy a list of substitution notations collected by Steele: <https://youtu.be/dCuZkaou00Q?t=1916>. [Ste17]. Link to slides: <http://groups.csail.mit.edu/mac/users/gjs/6.945/readings/Steele-MIT-April-2017.pdf>.

[Ste17] **Steele Jr., Guy L.** *It's Time for a New Old Language* (2017) <http://2017.clojure-conj.org/guy-steele/>

Given the list of declarations $M : \text{MvarContext}$ as vertices, \triangleright forms a directed graph. Assuming that this graph is acyclic, there exists a **topological ordering** of the declarations. That is, there is an ordering of the declarations in M such that each declaration only depends on earlier declarations.

Definition A.7 (well-formed context): This substitution operation on M helps motivate the constraints that make a metavariable context well-formed. In particular, define M to be **well-formed** when

- M 's dependency graph is acyclic. For example, the metavariable declaration $\langle ?m, \alpha, \Gamma, \text{some } ?m \rangle$ assigning $?m$ to itself would cause a loop. More perniciously, the assignment could cause an infinitely growing term as in $\{\{?m \mapsto ?m + ?m\}\}$. The no-loop property depends on the entire M , as we may have a multi-declaration dependency cycle such as $\{\{?m \mapsto f(?n), ?n \mapsto g(?n)\}\}$.
- Performing M on an expression $t : \text{Expr}$ (or other assignable object) should preserve the type of t in a suitable context Γ . This requirement will be formalised in Appendix A.1.1 when typing judgements are introduced for expressions. To illustrate with some examples:
 - Performing $\{\{?m \mapsto \text{"hello"}\}\}$ to $?m + 4$ would produce a badly typed expression $\text{"hello"} + 4$, so assignments must have the same type as their metavariables.
 - Performing $\{\{?m \mapsto x + 2\}\}$ to $?m + (\lambda x, x - ?m) 5$ will produce $(x + 2) + (\lambda x, x - (x + 2)) 5$. But this is badly formed because the variable x escapes the scope of its lambda binder. Hence there needs to be a way of making sure that a metavariable assignment can't depend on variables that would cause these malformed expressions. This is why the `MvarDecl` definition includes a context Γ for each declaration.

Definition A.8 (assign): Given a metavariable context M with an unassigned metavariable $?m$ and a candidate expression $t : \text{Expr}$, we need a way of updating M so that $?m$ is assigned to t . Call this function

`assign : MvarContext → Name → Expr → Option MvarContext`. The procedure of `assign M ?m t` is as follows⁸³:

1. Find the corresponding declaration $\langle ?m, \alpha, \Gamma, \text{none} \rangle \in M$. If it doesn't exist in M or it is already assigned, fail by returning `none`.
2. Assert that instantiating $?m$ with v does not introduce dependency cycles. That is, for each $?x \in \text{mv}(M t)$ ($M t$ is the M -instantiation of t), adding $?x \triangleright ?m$ does not introduce a cycle to M 's dependency graph.
3. Assert that typings and contexts are correct with $M; \Gamma \vdash t : \alpha$ (to be defined in Appendix A.1.1).
4. Delete $?m$ from M .
5. Update M to be $\{\{?m \mapsto M t\}\} M$. That is, each occurrence of $?m$ in M is replaced with the M -instantiation of t . Now $?m \notin \text{mv}(M)$.
6. Insert $\langle ?m, \alpha, \Gamma, \text{some } t \rangle$ into M .

Note that performing the assignment operation introduced in Definition A.8 causes the dependency ordering to change: for example supposing $d_1 \triangleright d_2$ in M and then if d_1 is assigned with a term t not containing a metavariable, then d_2 will no longer depend on $?m_1$ and so $d_1 \not\triangleright d_2$. An assignment may also cause a declaration to depend on metavariables that it did not previously depend on. As such, when an assignment is performed it may be necessary to reorder the declarations to recover the topological ordering. A reordering always exists, because step 2 of Definition A.8 ensures that the resulting metavariable context has no cycles.

Given $M : \text{MvarContext}$, we will often be adding additional declarations and assignments to M to make a new $M + \Delta : \text{MvarContext}$. Let's define $\Delta : \text{MvarContextExtension}$ as in (A.9).

⁸³The implementation in core Lean can be found at https://github.com/leanprover-community/lean/blob/05dd36d1717649932fccaafa0868321fb87f916d/src/library/type_context.cpp#L2175.

```

MvarContextExtension ::=
| declare (mvar_name : Name) (type : Expr) (context : Context)
| assign (mvar_name : Name) (assignment : Expr)

extend : MvarContext → MvarContextExtension → MvarContext
| M ↦ declare ?m α Γ ↦ [..M, <?m, α, Γ, none>]
| M ↦ assign ?m v ↦ assign M ?m v

```

In order for a `declare ?m α Γ` to be valid for `M`, require that `M;Γ ⊢ α : Type` and that `?m ∉ M`. Then we have that performing a valid declaration preserves the acyclicity of `M` with respect to `▷`. An assignment extension also preserves acyclicity; since step 2 of the procedure in Definition A.8 explicitly forbids dependency loops.

Hence, given a sequence of extensions to `M`, the `▷` relation is still acyclic and hence there exists a topological ordering of the declarations in `M` for `▷`.

A.1.1. Judgements and inference rules for metavariables

Now let's define the following judgements (in the same sense as in Section 2.1.3):

- `M;Γ ⊢ s : α` when `s` has type `α` under `M` and `Γ`.
- `M;Γ ⊢ s ≡ t` when `s : Expr` and `t : Expr` are definitionally equal (see [Car19 §2.2]).
- `ok M` when the metavariable context `M` is well-formed.
- `M ⊢ ok Γ` when the given local context `Γ` is well-formed under `M`.

The inference rules for these are given in (A.10).

I'll reproduce the list of (non-inductive) typing axioms here for completeness, but please see Carneiro's thesis [Car19] for a more comprehensive version, including a full set of inference rules for let binders, reductions, definitional equality and inductive constructions among others.

$$\begin{array}{c}
M; \Gamma \vdash \alpha : \text{sort} \\
M; \Gamma \vdash s : \beta \\
\hline \Gamma\text{-widening} \\
M; [.. \Gamma, (x:\alpha)] \vdash s : \beta
\end{array}
\qquad
\begin{array}{c}
M; \Gamma \vdash \alpha : \text{sort} \\
\hline \text{var-typing} \\
M; [.. \Gamma, (x:\alpha)] \vdash x : \alpha
\end{array}
\qquad
\begin{array}{c}
\hline \text{sort-typing} \\
\phi; \phi \vdash \text{sort} : \text{sort}'
\end{array}$$

$$\begin{array}{c}
M; \Gamma \vdash s : \Pi (x:\alpha), \beta \\
M; \Gamma \vdash t : \alpha \\
\hline \text{app-typing} \\
M; \Gamma \vdash s t : \{x \mapsto t\} \beta
\end{array}
\qquad
\begin{array}{c}
M; \Gamma \vdash \alpha : \text{sort} \\
M; [.. \Gamma, (x:\alpha)] \vdash s : \beta \\
\hline \lambda\text{-typing} \\
M; \Gamma \vdash (\lambda (x:\alpha), s) : (\Pi (x:\alpha), \beta)
\end{array}$$

$$\begin{array}{c}
M; \Gamma \vdash \alpha : \text{sort} \\
M; [.. \Gamma, (x:\alpha)] \vdash \beta : \text{sort} \\
\hline \Pi\text{-typing} \\
M; \Gamma \vdash (\Pi (x:\alpha), \beta) : \text{sort}
\end{array}
\qquad
\begin{array}{c}
M; \Gamma \vdash e : \alpha \\
M; \Gamma \vdash \alpha \equiv \beta \\
\hline \text{defeq-typing} \\
M; \Gamma \vdash e : \beta
\end{array}
\qquad
\begin{array}{c}
\hline \text{empty-ctx-ok} \\
M \vdash \phi \text{ ok}
\end{array}$$

$$\begin{array}{c}
M; \Gamma \vdash \alpha : \text{sort} \\
\hline \text{cons-ctx-ok} \\
M \vdash [.. \Gamma, x:\alpha] \text{ ok}
\end{array}$$

I now extend the above analysis to include an account of the metavariable development calculus that Lean uses to represent partially constructed proofs.

(A.9). Definition of `MvarContextExtension`. That is, an extension is either a declaration or an assignment.

(A.10). Non-development typing rules for Lean 3 CIC. Rules relating to inductive datatypes are omitted, see [Car19 §2.6] for the full set. The rules here differ from those in [Car19] through the addition of a spectating metavariable context `M`. In all cases, it is assumed that there are no variable clashes, so for example writing `[.. Γ, (x:α)]` implicitly assumes that `x ∉ Γ`. Note that in the rule `sort-typing`, one of the `sort s` is primed. This is because the presentation given here introduces a Russel-style paradox called Girard's paradox [Hur95] unless the `sort` expressions are parameterised by a natural number such that `sort n : sort (n + 1)`, but these are omitted here for brevity.

[Hur95] **Hurkens, Antonius J. C.** *A simplification of Girard's paradox* (1995) International Conference on Typed Lambda Calculi and Applications

(A.11). Metavariable typing rules.

$$\frac{M; \Gamma \vdash \alpha : \text{sort}}{[..M, \langle ?x, \alpha, \Gamma \rangle]; \Delta \vdash s : \beta} \text{H-widening}_1 \quad \frac{M; \Gamma \vdash t : \alpha}{[..M, \langle ?x, \alpha, \Gamma, t \rangle]; \Delta \vdash s : \beta} \text{H-widening}_2$$

$$\frac{M; \Gamma \vdash \alpha : \text{sort}}{[..M, \langle ?x, \alpha, \Gamma, \text{none} \rangle]; \Gamma \vdash ?x : \alpha} \text{metavariable}_1 \quad \frac{M; \Gamma \vdash t : \alpha}{[..M, \langle ?x, \alpha, \Gamma, t \rangle]; \Gamma \vdash ?x : \alpha} \text{metavariable}_2$$

$$\frac{M; \Gamma \vdash t : \alpha}{[..M, \langle ?x, \alpha, \Gamma, t \rangle]; \Gamma \vdash ?x \equiv t} \text{assignment-eq}$$

$$\frac{}{\vdash \emptyset \text{ ok}} \text{empty-mctx-ok} \quad \frac{M; \Gamma \vdash \alpha : \text{sort}}{\vdash [..M, \langle ?x, \alpha, \Gamma \rangle] \text{ ok}} \text{declare-ok} \quad \frac{M; \Gamma \vdash t : \alpha}{\vdash [..M, \langle ?x, \alpha, \Gamma, t \rangle] \text{ ok}} \text{assign-ok}$$

(A.12). Context well-formedness rules.

A.1.2. Properties of the Lean development calculus

In this subsection I note some regularity lemmas for the extended development calculus similarly to Carneiro [Car19 §3.2]. The first thing to note is that a judgement $M; \Gamma \vdash J$ is invariant under a reordering of the declarations of Γ or M that preserves the dependency ordering.

Lemma A.13 (Γ -regularity): Using (A.10) and some additional rules for definitional equality (\equiv , not printed here) and inductive datatypes, Carneiro proves various properties of the type system, of which the following regularity lemmas are relevant for the analysis here:

- $(\Gamma \vdash e : \alpha) \Rightarrow \vdash \Gamma \text{ ok}$. If the context is not well formed, then we can't make any typing judgements.
- $\Gamma \vdash e : \alpha \Rightarrow \text{fv}(e) \subseteq \Gamma \wedge \text{fv}(\alpha) \subseteq \Gamma$. If a term is well typed in Γ then all of the free variables are present.

Proof: these lemmas are proven by induction on the premiss judgments; any $h : (\Gamma \vdash e : \alpha)$ must be constructed from one of the judgements in (A.10) and (A.11).

Lemma A.14 (M -regularity): These regularity lemmas can be extended to include metavariables and a metavariable context M .

- $(M; \Gamma \vdash t : \alpha) \Rightarrow (\vdash M \text{ ok})$
- $(M; \Gamma \vdash t : \alpha) \Rightarrow (M \vdash \Gamma \text{ ok})$
- $(M; \Gamma \vdash t : \alpha) \Rightarrow \text{mv}(t) \subseteq M$

Proof: by applications of induction in a similar way to Lemma A.13.

Lemma A.15: The metavariables in M are topologically ordered on \triangleright (Definition A.6)

Proof: induction on $\vdash M \text{ ok}$. Each successive declaration can't depend on those that precede it.

Lemma A.16: A well-formed (Definition A.7) metavariable context M preserves typing judgements.

$$\frac{M; B \vdash b : \beta}{M; B \vdash (M b) \equiv b}$$

(A.22). Formal statement of Lemma A.16.

Proof: This follows from the congruence rule for \equiv ⁸⁴ and the `assignment-eq` rule in (A.11).

Lemma A.18: $\vdash M \text{ ok} \Rightarrow \vdash (M M) \text{ ok}$ where $M M$ is the instantiation (Definition A.4) of M with itself.

⁸⁴ I.e., $f_1 \equiv f_2$ and $a_1 \equiv a_2$ imply $f_1 a_1 \equiv f_2 a_2$, see [Car19 §2.6].

Proof: $M M$ is defined to be M with every occurrence of the M -assigned metavariables replaced with itself. Hence through repeated applications of Lemma A.16, we have $\vdash (M M) \text{ ok}$.

$\vdash M \text{ ok}$ does not imply that M is flat (Definition A.5), however any such M with $\vdash M \text{ ok}$ can be flattened through repeated instantiation of M on itself.

Lemma A.19: If $\vdash M \text{ ok}$, then a finite number of self-instantiations of M will be flat.

Proof: This follows from M 's declarations being a topological ordering on \triangleright (Lemma A.15). Let $\langle ?x, \alpha, \Gamma, \text{some } t \rangle$ be an assignment in M . Then $?m \triangleright ?x$ for all $?m \in \text{mv}(t)$ by Definition A.6. Now, for any declaration $d \in M$ where $?x \triangleright d$, we have $?x \not\vdash M d$ and $?m \triangleright M d$ for all $?m \in \text{mv}(t)$ since each occurrence of $?x$ in d has been replaced with t . Hence after each instantiation of M , all declarations that depend on an assigned metavariable $?x$ will be replaced with declarations that depend on strictly earlier metavariables in the dependency ordering, and so by well-founded induction on \triangleright , eventually there will be no assigned metavariables in M_n .

Is it possible to create an M such that there is a dependency (Definition A.6) cycle among the metavariable declarations in M ? For example, can we declare a recursive pair of metavariables $?n : \{i : \mathbb{N} \mid i \leq ?m\}$ and $?m : \{i : \mathbb{N} \mid i \leq ?n\}$? This follows from the typing rules (A.11) because a metavariable context is only ok when there is an explicit ordering on the metavariables such that each does not depend on the last⁸⁵.

⁸⁵ In Lean's actual implementation, it is possible to do this through the `tactic.unsafe.type_context` monad using an unsafe assignment, in this case an infinite-descending expression will form which will not typecheck (because Lean's typechecker has a finite depth).

However, in the definition of `assign` (Definition A.8), step 5 is to perform an update $M \mapsto \llbracket ?m \mapsto M v \rrbracket M$. Note that $\vdash M \text{ ok} \not\Rightarrow \vdash \llbracket ?m \mapsto M v \rrbracket M \text{ ok}$; set $M = [(\alpha : ?m_1), (\beta : ?m_2), (\alpha : ?m_3)]$; the resulting substitution $\llbracket ?m_1 \mapsto ?m_3 \rrbracket$ sends M to $M' := [(\alpha : ?m_1), (\beta : ?m_3), (\alpha : ?m_3)]$ where now $?m_3 \triangleright ?m_2$ and so $\not\vdash M \text{ ok}$. Fortunately, as noted after Definition A.8, there is a reordering π of declarations in M' which keeps the dependency ordering.

Lemma A.20: Assuming the conditions of assignment for `assign M ?m v` hold (Definition A.8), there is a permutation π such that $\vdash \pi (\llbracket ?m \mapsto M v \rrbracket M) \text{ ok}$.

Proof: Let π be the topological ordering of $(\llbracket ?m \mapsto M v \rrbracket M)$ with respect to \triangleright . This ordering exists by the 'no loops' assumption in Definition A.8. We can show $\vdash \pi (\llbracket ?m \mapsto M v \rrbracket M) \text{ ok}$ by noting that every declaration $d \in M$ has a corresponding $\llbracket ?m \mapsto M v \rrbracket d \in M'$. We can then perform induction on M' . Assuming that $\vdash M' \text{ ok}$, we have $\vdash [..M', \llbracket ?m \mapsto M v \rrbracket \langle ?b, \beta, B \rangle] \text{ ok}$ and need to show $M'; \llbracket ?m \mapsto M v \rrbracket B \vdash \llbracket ?m \mapsto M v \rrbracket \beta : \text{sort}$. By `assign`'s second condition, we have $M'; \Gamma \vdash v : \alpha$, and so by a similar argument to Lemma A.16, we have that $\llbracket ?m \mapsto M v \rrbracket$ preserves typing judgements. It is also clear that typing judgements are preserved by reordering, provided that dependencies are respected.

Lemma A.21: The function `assign M ?m v` (Definition A.8) with valid arguments preserves typing judgements. That is, the inference (A.22) holds.

$$\frac{\langle ?m, \alpha, \Gamma, \text{none} \rangle \in M \quad M; \Gamma \vdash v : \alpha \quad \text{no loops} \quad M; B \vdash b : \beta}{(\text{assign } M \text{ ?m } v); B \vdash b : \beta}$$

(A.22). Formal statement of Lemma A.21.

Proof: WLOG we may assume that M is flat. This is because instantiating preserves typing judgements by Lemma A.16 and repeated instantiation has a fixpoint by Lemma A.19. The result of performing the steps in Definition A.8 is that `assign M ?m v` returns M with the $?m$ declaration removed, each instance of $?m$ substituted with $M v$ and appended with

$\langle ?m, \alpha, \Gamma, v \rangle$. I will prove this by first showing that $\vdash (\text{assign } M \ ?m \ v) \text{ ok}$. We have that $M_1 := \llbracket ?m \mapsto M \ v \rrbracket M$ does not introduce any dependency cycles by the 'no loops' condition, so reorder. And so as noted in Lemma A.20, there exists a permutation $M_2 := \pi M_1$ of the declarations in M_1 such that $\vdash M_2 \text{ ok}$. We have that $?x \notin \text{mv}(M_2)$ (since M_2 is flat and $?m \notin \text{mv}(M \ v)$), so we can remove the declaration for $?x$ in M_2 and append $\langle ?m, \alpha, \Gamma, v \rangle$ without changing the validity. Hence $\vdash (\text{assign } M \ ?m \ v) \text{ ok}$. Finally, we can show $(\text{assign } M \ ?m \ v); B \vdash b : \beta$ by induction on the declarations in $M_3 := (\text{assign } M \ ?m \ v)$. The metavariables in M_3 are the same as those in M , so it suffices to show that $M; E \vdash t : \gamma \Rightarrow M_3; E \vdash t : \gamma$ for all t, γ, E . Which can be shown by noting that the substitution $\llbracket ?m \mapsto M \ v \rrbracket$ preserves typing judgements.

A.2. Zippers on Boxes

In order to run Lean tactics at various points in the `Box` structure defined in Chapter 3, we need to navigate to a certain point in a `Box`, and build up a metavariable context M containing all of the metavariables from the \mathcal{G} -binders and a local context Γ comprising the variables given in the \mathcal{J} -binders above the given point in the `Box`. The way to implement this plan is to define a context-aware **zipper** [Hue97] on a `Box`.

Let's first create a coordinate system for `Box`. Coordinates for functors and inductive datatypes were introduced in Section 2.3.2.

```
Coord ::=
| J | G | U | A1 | A2 | O1 | O2
Address := List Coord
```

Where here A_1 is the coordinate for the b_1 argument of the A constructor. That is, $\text{get } A_1 (A \ b_1 \ x \ b_2) = \text{some } b_1$. Similarly for the other coordinates. **Definition A.24:** A list of `Coord` instances can be interpreted as an **address** to a certain point in an expression (see Section 2.3.3).

Definition A.25 (zipper): Next, define a datastructure called a **path** (using the same constructor names as `Box`s) as shown in (A.26). A **zipper** is a tuple consisting of a `Path` and a `Box`.

```
PathItem ::=
| J : Binder → PathItem
| G : Binder → PathItem
| U : Binder → Expr → PathItem
| A1 : Unit → Binder → Box → PathItem
| A2 : Box → Binder → Unit → PathItem
| O1 : Unit → Box → PathItem
| O2 : Box → Unit → PathItem

Path := List PathItem

Zipper :=
(path : Path)
× (cursor : Box)
```

We can visualise the zipper as in Figure A.27: a `Box` is annotated with an additional blob of paint ● at some point in the tree. The ancestors of ● are in the zipper's `path`, and everything below is the `cursor`.

[Hue97] **Huet, Gérard** *Functional Pearl: The Zipper* (1997) *Journal of functional programming*

(A.23). Coordinate type for `Box`. Each constructor of `Coord` corresponds to a recursive argument for a constructor in `Box` (3.9). Hence there is no `R` coordinate.

(A.26). Type definitions for `Zipper`s and `Path`s. Definition of `Zipper` and `Path` over an expression. See Figure A.27 for a visualisation. The constructors of `Path` are created to match the signatures of the constructors in `Box` (3.9). `Unit` is used as a placeholder.

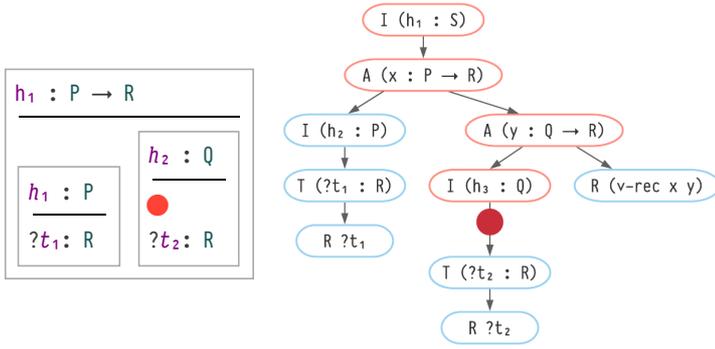


Figure A.27. Visualisation of a zipper. The box to the left shows an example `Box` with a red blob indicating the position of the zipper's cursor. The right figure shows the underlying tree of `Box` constructors. All blue nodes are `Box` constructors and all orange nodes are `Path` constructors.

On the zipper, we can perform the `up : Zipper → Option Zipper` and `down : Coord → Zipper → Option Zipper` operations as defined in (A.28) and (A.29). Applying `up` to this zipper will move the ● up to the next node in the `Box` tree (or return `none` otherwise). Similarly, `down c z` (A.26) will inspect the cursor of `z` and move the ● down on its `c`th recursive argument. The use of a zipper datastructure is used over an address for ● because the zipper system allows us to modify the cursor in place and then 'unzip' the zipper to perform an operation on a sub-box.

```
unwrap : Coord → Box → Option (PathItem × Box)
| g → g (x:α) b ↦ some (g (x:α) , b )
| g ?m ↦ g (?m:α) b ↦ some (g ?m , b )
| U ↦ U (x:α) v b ↦ some (U (x:α) v , b )
| A1 ↦ A b1 (x:α) b2 ↦ some (A1 () (x:α) b2 , b1)
| A2 ↦ A b1 (x:α) () ↦ some (A2 b1 (x:α) (), b2)
| θ1 ↦ θ b1 b2 ↦ some (θ1 () b2 , b1)
| θ2 ↦ θ b1 () ↦ some (θ2 b1 (), b2)
```

```
wrap : PathItem → Box → Box
| g (x:α) ↦ b ↦ g (x:α) b
| g ?m ↦ b ↦ g (?m:α) b
| U (x:α) v ↦ b ↦ U (x:α) v b
| A1 () (x:α) b2 ↦ b1 ↦ A b1 (x:α) b2
| A2 b1 (x:α) () ↦ b2 ↦ A b1 (x:α) b2
| θ1 () b2 ↦ b1 ↦ θ b1 b2
| θ2 b1 () ↦ b2 ↦ θ b1 b2
```

```
up : Zipper → Option Zipper
| <[], b> ↦ none
| <[..p, i], b> ↦ some <p, wrap i b>
```

```
down : Coord → Zipper → Option Zipper
| c ↦ <p, b> ↦ do
  <i, b2> ← (unwrap c b)
  pure <[..p, i], b2>
```

```
down : List Coord → Zipper → Option Zipper
| [] ↦ z ↦ some z
| [c, ..a] ↦ z ↦ down c z ≫ down a
```

```
unzip : Zipper → Box
| <[], b> ↦ b
| <[..p, i], b> ↦ unzip <p, wrap i b>
```

```
zip : Box → Zipper
| b ↦ <φ, [], b>
```

Motivated by Figure A.27, it may be readily verified that `down c (up z) = some z` provided `z`'s path is not empty and if `c` is the coordinate corresponding to `z`'s rightmost path entry. With similar conditions: `up (down c z) = some z`.

Definition A.30 (zipper contexts): Define the context of a zipper `z.ctx` as:

```
ctx : PathItem → Context
| g x ↦ [x]
| A2 b1 x _ ↦ [x]
| _ ↦ []

ctx : Zipper → Context
| <_, p, _> ↦ [..(ctx x) for x in p]
```

(A.28). Helper definitions `wrap` and `unwrap` for converting between `Box`s and `PathItem`s. These follow the standard schema found in [Hue97].

(A.29). Definitions for `up`, `down` and some helper methods for navigating `Zipper`. The definitions for `wrap` and `unwrap` are given in (A.28).

(A.31).

That is, `z.ctx` returns the list of the variables that are bound in the path. For the example `z` in Figure A.27, `z.ctx = [h1]`.

Similarly, define the metavariable context `z.mctx` of a zipper as

```
mctx : PathItem → List Binder
  | g m      ↦ [m]
  | _       ↦ []

mctx : Zipper → MvarContext
  | <_,p,_> ↦ {..(mctx x) for x in p}
```

(A.32). Defining the induced metavariable context `M` for a zipper.

So `z.mctx` is a metavariable context containing all of the goals defined above the cursor of `z`.

Now, given a zipper `z`, write `z ⊢ t : α` to mean `z.mctx; z.ctx ⊢ t : α`. Similarly for `p : Path`, `p ⊢ t : α`.

Lemma A.33 (zipping is sound): Suppose that

`f : MvarContext → Context → Box → Option Box` is a sound box tactic parametrised by the contexts `M` and `Γ`, then given `b : Box` and a valid address `a : List Coord`, we get another sound box-tactic `f@a` defined in (A.34).

```
f@a : Box → Option Box
  | b1 ↦ do
    <p, b2> ← down a b1
    b3 ← f p.mctx p.ctx b2
    pure (unzip <p, b3>)
```

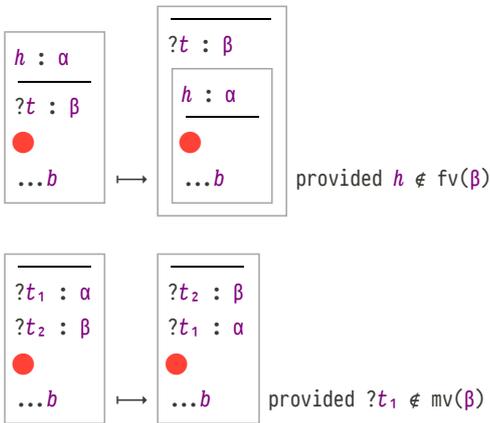
(A.34). Operation to perform the box-tactic `f M Γ : Box → Option Box` 'under' the address `a : List Coord`. `do` notation is used.

Proof: Suppose that `⊢ b1 : α`, then by induction on the typing laws for `Box` given in Section 3.4.2, we can show `p ⊢ b2 : ζ` for some `ζ : Expr`. Since `f` is sound and assuming `f(b2)` doesn't fail, we also have `p ⊢ b3 : ζ`. Then finally the typing laws Section 3.4.2 can be applied in reverse to recover `⊢ (unzip <p, z>) : α`.

Here we are working towards being able to soundly run a tactic in the context provided by a `Box` zipper `z` by finding ways to manipulate zippers that preserve the inference rules given in (A.10) and (A.11). We also need to perform some modifications to the paths of `Zipper`s.

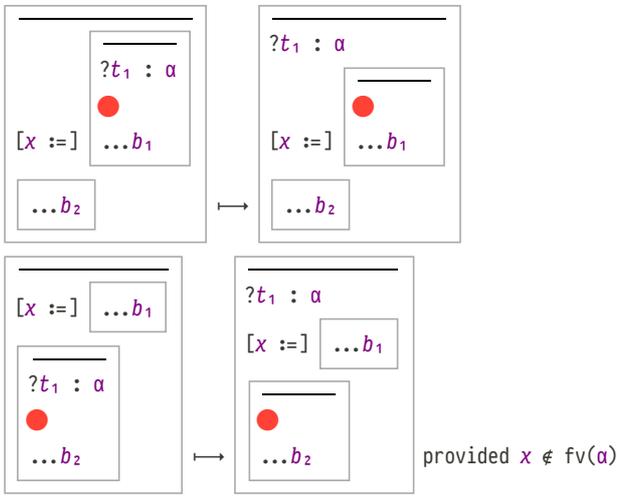
Definition A.35 (path soundness): A path modification `ρ : p ↦ p'` is *sound* on a zipper `z = <p, b>` if `M; Γ ⊢ unzip <p, b> : β` and `M; Γ ⊢ unzip <p', b> : β`. Hence to show soundness, one simply needs to show that corresponding box-tactic `unzip ∘ ρ` is sound.

We have the following sound path-based box-tactics:



(A.36). Restriction. Note that the context of `?t` has changed.

(A.38). Goal swapping.



A.3. Running tactics in Boxes

Now that we have the inference rules for metavariables and a definition of a zipper over a `Box`, we can define how to run a tactic within a `Box`.

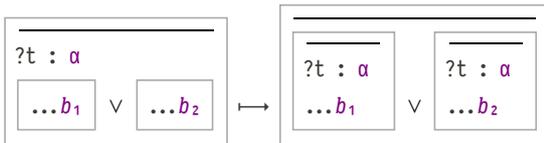
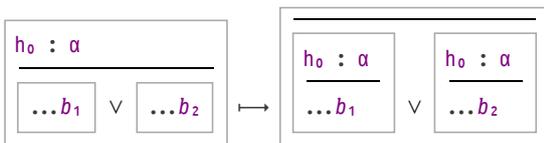
Definition A.39 (hoisting θ boxes): Before defining how to make a tactic act on a `Box` zipper we need to define an additional operation, called θ -**hoisting**. This is where an θ -box is lifted above its parent box. This definition extends to θ -hoisting path entries.

(A.40). Definition of θ -lifting.

```

 $\theta$ -lift
: Box           $\rightarrow$  Box
| ( $\mathcal{G}$   $h$  ( $\theta$   $b_1$   $b_2$ ))  $\mapsto$   $\theta$  ( $\mathcal{G}$   $h$   $b_1$ ) ( $\mathcal{G}$   $h$   $b_2$ )
| ( $\mathcal{G}$   $m$  ( $\theta$   $b_1$   $b_2$ ))  $\mapsto$   $\theta$  ( $\mathcal{G}$   $m$   $b_1$ ) ( $\mathcal{G}$   $m$   $b_2$ )
| ( $\mathcal{A}$   $b_0$   $x$  ( $\theta$   $b_1$   $b_2$ ))  $\mapsto$   $\theta$  ( $\mathcal{A}$   $b_0$   $x$   $b_1$ ) ( $\mathcal{A}$   $b_0$   $x$   $b_2$ )
| ( $\mathcal{A}$  ( $\theta$   $b_1$   $b_2$ )  $x$   $b_0$ )  $\mapsto$   $\theta$  ( $\mathcal{A}$   $b_1$   $x$   $b_0$ ) ( $\mathcal{A}$   $b_2$   $x$   $b_0$ )
| ( $\mathcal{U}$   $x$   $v$  ( $\theta$   $b_1$   $b_2$ ))  $\mapsto$   $\theta$  ( $\mathcal{U}$   $x$   $v$   $b_1$ ) ( $\mathcal{U}$   $x$   $v$   $b_2$ )
| -  $\mapsto$  none

```

(A.41). Diagrammatic example of θ -lift acting on a \mathcal{G} -box.(A.42). Example of θ -lift acting on an \mathcal{G} -box.

Lemma A.43: θ -hoisting is a sound box-tactic.

Proof: by compatibility (Lemma 3.15) it suffices to show that the results are equal, but then this is a corollary of the WLOG proof from the compatibility lemma (3.17).

The motivation behind the hoisting is that θ boxes are a form of backtracking state in a similar spirit to a logic monad [KSFS05]. However by including the branching θ box on the tree, it is possible to structurally share any context that is shared among the backtracking branches. Hoisting an θ box has the effect of causing the backtracking branches to structurally share less of the context with each other. In the most extreme case, we can repeatedly apply θ -hoisting to move all of the branches to the top of the box structure, at which point we arrive at something isomorphic to what would arise as a result of using the logic monad form of backtracking.

[KSFS05] **Kiselyov, Oleg; Shan, Chung-chieh; Friedman, Daniel P; et al.** *Backtracking, interleaving, and terminating monad transformers: (functional pearl)* (2005) ACM SIGPLAN Notices

Given $z : \text{Zipper}$, define $z.\text{goal}$ to be first metavariable declared on $z.\text{path}$. Hence given a $z : \text{Zipper}$ with the cursor having shape \mathcal{G} , we can extract a metavariable context $M = \text{mctx } z$ and a special goal metavariable $z.\text{goal}$, from these, create $ts : \text{TacticState}$ ⁸⁶.

⁸⁶ See Section 2.4.4.

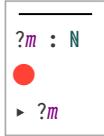
We can now run a tactic `t : Tactic` on `ts`. If successful, this will return a new tactic state `ts'`. `ts'` will have an extension of M ⁸⁷, call this $M + \Delta$ where Δ is a list of declarations and assignments successively applied to M . The task here is to create a new `z' : Zipper` which includes the new goals and assignments in Δ .

⁸⁷ See (A.9).

The crux of the task here is to use Δ to construct a new `p' : Path` such that unzipping `<p', (M + Δ) z.cursor> : Zipper` will result in a sound box-tactic. In general, this will mean placing new metavariable declarations from Δ as new `g` entries in the path and deleting `g` entries which are assigned in Δ . Additionally, it may be necessary to hoist θ boxes and re-order existing `g` entries. This has to be done carefully or else we can introduce ill-formed boxes.

For example, take the zipper shown in (A.44) with metavariable context M containing one metavariable `?m : N`. There is a tactic `apply List.length` that will act on the goal `?m` by declaring a pair of new metavariables `?α : sort` and `?l : List ?α` and assigning `?m` with `List.length ?l`. After performing this tactic, the new metavariable context $M + \Delta$ is (A.45).

(A.44). Initial zipper.

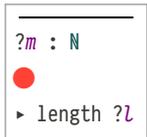


(A.45). The metavariable context after performing the `apply List.length` tactic at (A.44).

$$M + \Delta = \{ \langle ?m := \text{length } ?l, \Gamma \rangle, \langle ?\alpha, \text{Type}, \Gamma \rangle, \langle ?l, \text{List } ?\alpha, \Gamma \rangle \}$$

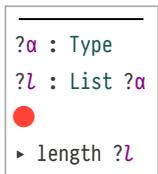
But now if our `z.path` is `[g (?m : N)]`, unzipping `<[g (?m : N)], (M + Δ) z.cursor>` (A.46) will not result in a valid `b : Box` (i.e. one that we can derive a judgement $\vdash b : \beta$) because the `result` will depend on metavariables `?α` and `?l` which do not have corresponding `g`-binders.

(A.46). Result of applying $M + \Delta$ from (A.45) to (A.44). This is not a well-formed `Box` because the metavariable `?l` is not bound.



We need to make sure that the new metavariables `?α` and `?l` are abstracted and added to the `Path` as shown in (A.47), so when we unzip we end up with a well-formed `Box`.

(A.47). Correct update of the zipper to reflect $M + \Delta$.



We can find additional complications if the tactic declares new metavariables in a context E other than the current local context Γ . This may happen as a result of calling `intros` or performing an induction step. In these cases, some additional work must be done to ensure that the newly declared metavariable is placed at the correct point in the `Box` such that the context produced by the path above it is the same as E . This is tackled in the next section through the definition of a function called `update`.

The procedure for correctly adjusting the path to produce valid boxes as exemplified by (A.47) is as follows. Define a function

`update : MvarContextExtension → Path → Option Path`. Here, `MvarContextExtension` (see (A.9)) is either a metavariable assignment or a metavariable declaration for M .

Definition A.48 (update): `update Δ p` is defined such that

`(update Δ p).mctx = extend Δ p.mctx` where `extend` is defined in (A.9). To do this, `update` will either insert a new `g` path entry or delete a `g` according to whether Δ is a declaration

or an assignment. Then it will reorder the λ declarations such that it respects the dependency ordering \triangleright .

In general, this is not always possible since there may be a pair of declarations $d_1, d_2 : \text{MvarDecl}$ such that $d_1 \triangleright d_2$ but $d_2.\text{context} \leq d_1.\text{context}$, and so there is not necessarily an ordering of the declarations which is topological on both \leq and \triangleright . This case can be handled in theory through redeclaring metavariables and using delayed abstractions as I discuss in Appendix A.3.1, however I do not analyse or implement this case here because it arises rarely in practice: most of the newly declared metavariables will not be in a different context or in a subcontext of $p.\text{ctx}$. Furthermore most of the tactics which do cause complex declarations to appear such as `intro` have an equivalent box-tactic.

```
update : MvarContextExtension → Zipper → Option Zipper
update (declare ?m α Γ) ⟨p, b⟩ :=
  assert p.mctx; Γ ⊢ α : sort
  if (Γ ≤ p.ctx):
    obtain E such that [..Γ, ..E] = p.ctx
    obtain [..p₀, ..p₁] = p such that
      p₀.ctx = Γ and p₀ ⊢ α : sort
    p ← [..p₀, λ ?m:α, ..p₁]
    reorder p
    return ⟨p, b⟩
  else if (Γ > p.ctx):
    obtain E such that [..p.ctx, x₀, ..., xₙ] = Γ
    p ← [..p, λ₂ (λ x₀ $ ... $ xₙ $ λ ?m) y]
    reassign ?m in b with (?m x₀ ... xₙ)
    reorder p
    return ⟨p, b⟩
  else (Γ ≰ p.ctx):
    — that is, Γ and p.ctx are incomparable
    fail — this case is not supported

update (assign ?m v) ⟨p, b⟩ :=
  assert (p.mctx.assign ?m v) is valid
  delete (λ ?m) from p
  assign v to ?m in p
  reorder p
  return p
```

(A.49). Pseudocode definition of `update`. See the remarks after this code block for more information.

Some remarks on (A.49):

- `reorder p` performs a reordering on the λ binders in p to respect \triangleright and contexts. As noted earlier, in certain circumstances this may not be possible, in which case `fail`.
- To account for θ -boxes: before performing the above `reordering`, `lift`⁸⁸ all θ items in p so that the resulting $..p_1$ does not contain any θ -binders.
- In the case that $\Delta = \text{declare } ?m \alpha \Gamma$ and $\Gamma \leq p.\text{ctx}$, I am making the assumption that α doesn't depend on any metavariables whose context is outside Γ . This can occur if the offending metavariable is wrapped in a delayed abstraction. I discuss this caveat in Appendix A.3.1.
- The case where $\Gamma > p.\text{ctx}$ works by reassigning the newly declared metavariable $?m$ with its skolemised version and wrapping the declaration in a series of λ boxes. A circumstance where this can occur is if the `intros` tactic was used. I do not analyse the soundness of this case in further detail because it is expected that in these cases the `intro move` on `Boxes` (3.25) should be used instead.

⁸⁸ Analogously to (A.40).

An example of performing the `assign` case of (A.49) is given in (A.50):

```

┌───────────┐
│ ?t₁ : N    │
│ ?t₂ : P ?t₁ │
│ ?t₃ : N    │
└───────────┘
```

(A.50).

Suppose that a tactic assigned $?t_1$ with $?t_3 + 4$. Then without any reordering the box would look like (A.51):

```

?t2 : P (?t3 + 4)
?t3 : N

```

However this is not a valid box because the \mathcal{G} -binder for $?t_2$ depends on a variable that is not in scope. Fortunately as discussed in Lemma A.20 a total dependency ordering of metavariables in the same context always exists, and so in `update` we can use the 'goal-swap' (A.38) path-reordering box-tactic to rearrange the goals to obey this. This is performed in the `reorder` step in `update` (A.49).

The way that `update` is defined means that

- $(\text{update } (\text{declare } ?m \ \alpha \ \Gamma) \ z).\text{mctx} = \text{extend } (\text{declare } ?m \ \alpha \ \Gamma) \ z.\text{mctx}$ for a valid declaration with $\Gamma \leq p.\text{ctx}$ (up to reordering of the declarations in $z.\text{mctx}$). As mentioned above, the other case $\Gamma > p.\text{ctx}$ is not considered here. Let $\langle p', b' \rangle = \text{update } (\text{declare } ?m \ \alpha \ \Gamma) \ \langle p, b \rangle$ and $p \vdash b : \alpha$. Then $p' \vdash b' : \alpha$ too because $p'.\text{ctx} = p.\text{ctx}$ and $p'.\text{mctx} = \text{extend } \Delta \ p.\text{mctx}$. And we saw in Appendix A.1.2 that adding a declaration to a metavariable context preserves type judgements.
- Similarly, we have $(\text{update } (\text{assign } ?m \ v) \ z).\text{mctx} = \text{extend } (\text{assign } ?m \ v) \ z.\text{mctx}$ for a valid assignment. Now let $\langle p', b' \rangle = \text{update } (\text{assign } ?m \ v) \ \langle p, b \rangle$ and $p \vdash b : \alpha$. Then similarly $p' \vdash \{\{?m \mapsto v\}\} b : \{\{?m \mapsto v\}\} \alpha$.

We can now put together the components. Now let $zz : \text{Zipper} \rightarrow \text{Zipper}$ be some function that navigates to a certain point in a `Box` and let `tac` be a tactic, we can define the 'escape hatch' tactic procedure, depicted below in Figure A.52.

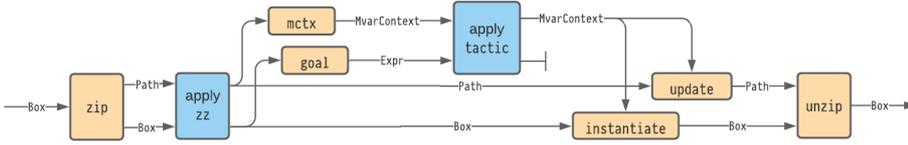


Figure A.52. Data-flow of a `Box` having a tactic applied at a certain point. The `Box` is first zipped to produce a `Path`, cursor `Box` and metavariable context. A monad is run on this state to produce a `Zipper` at the required location at which point a tactic is run with the goal being the goal at the cursor. The resulting metavariable context is then used to instantiate the `Zipper` and unzipped to produce a new `Box`.

⁸⁹ Lean 3 calls it a delayed abstraction macro.

⁹⁰ An abstraction is when free variables are replaced with bound variables.

A.3.1. A variant of \mathcal{A} supporting delayed abstraction

A delayed abstraction $e[x]$ is a special type of expression⁸⁹ constructed with a local variable x and an expression e which may depend on x . A delayed abstraction represents an abstraction of x on e ⁹⁰. This is used when e depends on a metavariable $?m$ whose declaration context contains x . In this case, performing the abstraction immediately would be premature because the metavariable $?m$ might need to depend on x .

There are a few possible variants and extensions of the design of \mathcal{A} boxes which I considered. The main limitation of \mathcal{A} boxes as detailed above is that the structure of b_1 is inscrutable when zipped on b_2 . So in the above example we could not infer the structure of the function through reasoning on b_2 . In order to model this, it is necessary to invoke delayed abstractions.

To get this to work, when zipping to b_2 , one needs to first zip to the final $R \ r$ in b_1 , and then instantiate b_2 with a *delayed abstraction* of r . Then, when performing $\text{up } \mathcal{A}_2$, if a metavariable under a delayed abstraction has been assigned you need to unzip back through the entirety of b_1 and add any new goals to the path. So a variant `PathItem.A1` would change to `Path → Expr → List Name → Expr`.

Appendix B

ProofWidgets tutorial

This appendix introduces you to the design of the widgets framework through a series of simple examples.

The code here is written in Lean 3. If you wish to try out the code examples yourself, I recommend that you install Lean 3 and [Microsoft Visual Studio Code](#) -- henceforth referred to as VSCode. Any installation guide I place here runs the risk of becoming out of date, and so I refer the reader to the [leanprover-community installation instructions](#).

Making a widget is as simple as opening a Lean file in VSCode and adding the following snippet:

```
open widget (B.1).  
#html "hello world!"
```

Clicking on `#html` will reveal the widget in the infoview panel. The infoview is the output window for Lean, containing the goal state and messages from the Lean server. If you can't see it you can open it by clicking the display goal button . If you can't see this button then you may need to install the Lean extension for VSCode, consult the [community installation instructions](#) for more information.

However the above example is scarcely more impressive than `#print "hello world!"`, so let's start with:

```
#html h "div" [className "purple f3 pa3"] ["hello world!"] (B.2).
```

This effectively generates the following HTML:

```
<div class="purple f3 pa3">hello world!</div> (B.3).
```

which is then rendered in the infoview:

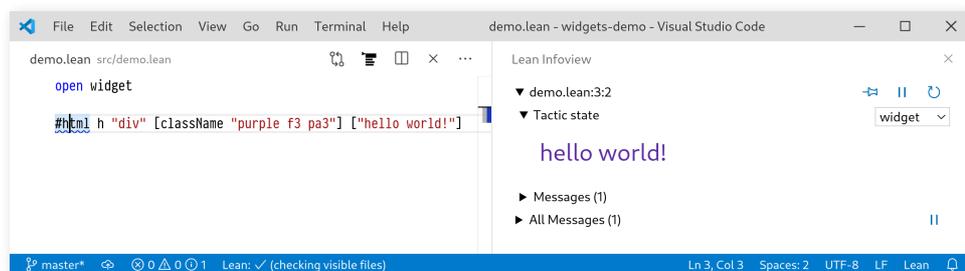


Figure B.4. A hello world widget.

`h` is a shorthand alias to create a new HTML element. The mysterious `"purple f3 pa3"` string is a set of CSS class identifiers. The infoview client comes with a stylesheet called [Tachyons](#) that watches for these identifiers and uses them to change the appearance of the `div`. The codes can appear cryptically terse but one gets used to them; `"f3"` means 'use font size 3' and `"pa3"` means 'add some padding'.

We can view the type signature of `h`:

```
#check @h (B.5).  
-- h : Π {α : Type}, string → list (attr α) → list (html α) → html α
```

The type parameter `α` is called the *action type*. It is the type of the data that will be returned when the user interacts with the interface in some way. To see this, consider this example with a button;

`h "button" [on_click (λ _, "I got clicked!")] ["click me!"]` (B.6).

`on_click : (unit → α) → α` is called an **event handler** and allows the widget writer to specify an action to be emitted when the given button is clicked. However in order to have this action do something, we need to connect it to a `component`:

```
meta def my_button : component unit empty :=
  component.ignore_action
  $ component.with_effects (λ ⟨⟩ x,
    [widget.effect.insert_text x]
  )
  $ component.pure (λ ⟨⟩,
    h "button" [ on_click (λ ⟨⟩, "/- I got clicked! -/")
      , className "ma3"
    ] ["click me!"]
  )
)

#html html.of_component () my_button
```

(B.7).

Clicking the resulting button causes the text `"/- I got clicked! -/"` to be inserted on a new line above the cursor.

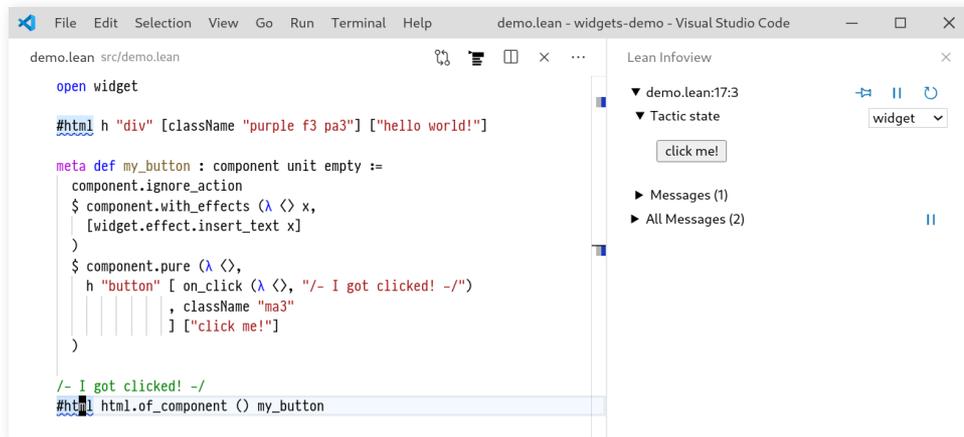


Figure B.8. Window state after clicking the button.

So how does this work? `component`s are responsible for handling all of the non-pure aspects of widgets. That is, statefulness and side-effects. I will discuss state later, in the above example we can see an example of a side-effect. Let us write $\pi \Rightarrow \alpha$ to mean `component π α`.

The first part of the above definition uses `component.pure`:

`component.pure : (Props → list (html Action)) → (Props ⇒ Action)` (B.9).

This creates a 'pure' component that takes a **view function** mapping `Props` (in our example `Props = unit`) to a list of html elements, in our case a single button with action type `string`. The next component in the list tells us what to do in the event of an action

```
component.with_effects :
  (Props → Action → list effect)
  → (Props ⇒ Action) → (Props ⇒ Action)
```

(B.10).

Here, `effect` is a built-in datatype used to represent the side effects that widgets offer. `effect.alert : string → effect` indicates that the client app should display an alert message to the user. Other effects include copying a string to the clipboard or inserting some text in the current Lean file. The function passed to `with_effects` is called whenever the user causes an action to occur and the effects in the returned list are executed by the client in the order they appear.

Finally, `component.ignore_action : (Props ⇒ Action) → (Props ⇒ empty)` throws away all actions that might be emitted by the inner component.

B.0.1. Statefulness

The final piece of the puzzle is statefulness, most apps require some concept of local state. For example:

- the position in a scrollable panel
- the open/closed state of a collapsible panel

These are all parts of the state which matter for the UI, rather than data-centric states. There might also be some state that should be reflected in the data, such as the state of a document. This is not what a stateful component is for.

A simple counter component is given below.

```
open widget (B.11).

variables { $\pi$   $\alpha$  : Type}

inductive counter_action
| increment
| decrement

open counter_action

meta def counter_init :  $\pi \rightarrow \mathbb{Z}$ 
| _ := 0

meta def counter_props_changed :  $\pi \rightarrow \pi \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$ 
| _ _ i := i

meta def counter_update :  $\pi \rightarrow \mathbb{Z} \rightarrow$  counter_action  $\rightarrow \mathbb{Z} \times$  option  $\alpha$ 
| _ i increment := (i + 1, none)
| _ i decrement := (i - 1, none)

meta def counter_view : ( $\mathbb{Z} \times \pi$ )  $\rightarrow$  list (html counter_action)
| (i, _) :=
  h "div" [] [
    h "button" [on_click ( $\lambda$  <, increment) ] ["+"],
    html.of_string $ to_string $ i,
    h "button" [on_click ( $\lambda$  <, decrement)] ["-"]
  ]

meta def simple_counter : component  $\pi$   $\alpha$  :=
component.with_state
  counter_action
   $\mathbb{Z}$ 
  counter_init
  counter_props_changed
  counter_update
$ component.pure counter_view

#html simple_counter
```

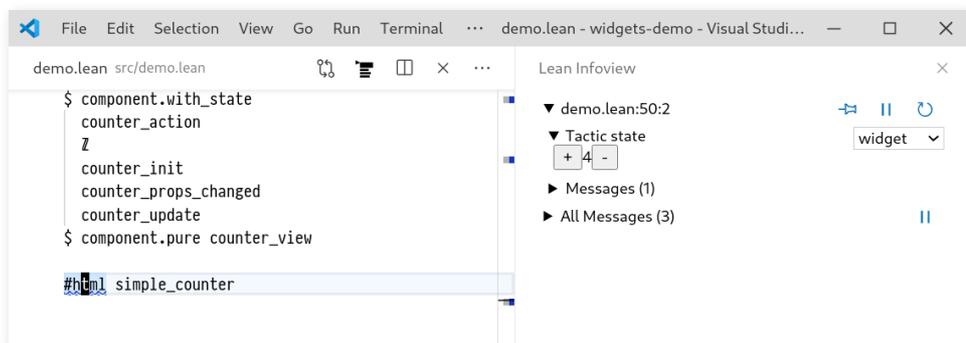


Figure B.12. The counter widget.

Let's unpack what this is doing. The main things to focus on are the arguments to `with_state`:

```
component.with_state (B.13).
  (InnerAction State : Type)
  (init : Props  $\rightarrow$  State)
  (props_changed : Props  $\rightarrow$  Props  $\rightarrow$  State  $\rightarrow$  State)
  (update : Props  $\rightarrow$  State  $\rightarrow$  InnerAction  $\rightarrow$  State  $\times$  option Action)
  : ((State  $\times$  Props)  $\Leftrightarrow$  InnerAction)  $\rightarrow$  (Props  $\Leftrightarrow$  Action)
```

The essential idea is to maintain a local state `s` that wraps the inner component `c : (State × Props) ⇒ InnerAction`. The other arguments dictate how this state should be initialised and mutate over the lifecycle of the component.

- `init p` provides the initial value of `s` given props `p`.
- `props_changed p0 p1 s0` gives the system a chance to update the state if the props of the component change from `p0` to `p1`.
- `update p s a` updates the state in the event that the inner component `c` emits an action `a`. It may optionally return its own action that should be emitted by the outer component.

So in the case of the counter above, the `Props` type `π` is ignored. The `State` type is an integer and the `InnerAction` is `counter_action`, whose values are either 'increment' or 'decrement'. `init` produces the initial counter value `0`. `props_changed` just keeps the state the same. `counter_update` increments or decrements the state depending on the value of the given `counter_action`.

Note that the state is local to an individual counter, so for example we can write

```
#html h "div" [] [
  html.of_component () simple_counter,
  html.of_component () simple_counter
]
```

(B.14).

and the ProofWidgets framework will automatically track an independent state for the two different counters, even when more components are added or taken away. The ProofWidgets system does this by assigning each `component` an identity and then holding a persistent, mutating state for each of these components.

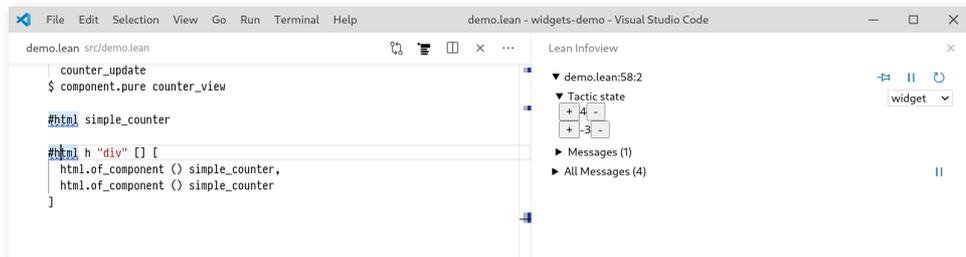


Figure B.15. Multiple counters with independent state.

Even when there is no need for state, components serve an important role of avoiding recomputing unchanged parts of the widget tree. This is done with `component.with_should_update`.

```
component.with_should_update
  (comp : Props → Props → bool)
  : (Props ⇒ Action) → (Props ⇒ Action)
```

(B.16).

Every time the component is recomputed, as for example in a view method, the given `comp` function is used to compare the previous value for `Props` with the new value. If it returns false, then the widgets rendering system does not re-render the inner component and instead uses the cached value.

Appendix C

The rendering algorithm of ProofWidgets

In this appendix I will detail the algorithm used to create interactive, browser-renderable HTML from a user's implementation of a `Component`. The design of the algorithm is largely a simplified variant of the 'tree reconciliation' algorithm used in web-based FRP frameworks such as React and Elm. The React reconciliation algorithm has been generalised many times⁹¹ and the algorithm has been documented on the React website⁹². I hope that this appendix will prove useful to people who wish to implement ProofWidgets for themselves.

The pseudocode language described in Section 2.2 will be used throughout to make the relevant parts as salient as possible. I'm going to first walk through a simple todo-list example to make the setting clear and concrete.

C.1. Motivating ProofWidgets with a todo list app

A first approximation to a user interface is as a way of rendering information. So suppose that we have some datatype `M` that we care about, for example `M` could be the list of items in a todo list. Then we could write a function `view : M → UI` to express the process of creating an image on the screen. The most general formulation of `UI` would be an image `ℝ2 → colour` and indeed some FRP frameworks explore this [Ello1], however we will content ourselves with an abstract tree of HTML⁹³[I will call this abstract tree the DOM in accordance with Section 5.2 to be converted to pixels on a screen by a web-browser.

```
Attr A ::=
| val (key : String) (value : String)
| click (handler : Unit → A)

Html A ::=
| element (tag : String) (attrs : List (Attr A)) (children : List (Html A))
| of_string (value : String)
```

So here the html `<div id="x">hello</div>` would be represented as `element "div" [val "id" "x"] [of_string "hello"] : Html A`. However, let's keep using the XML style `<div/>` notation.

As before the `A` type is the *action* type and represents the type of objects that the tree can emit when the user interacts with it. Now we can create a button as

```
element "button" [click (() ↦ 3)] ["click me!"] : Html N
```

 or `<button click={() ↦ 3}>click me!</button>`. This emits the number `3` whenever the button is clicked.

As before, let's introduce `Component`s. A `Component : Type → Type → Type` allows us to encapsulate state at a particular point in the tree.

⁹¹ An example is the react-three-fiber project which lets one build declarative scenegraphs for a 3D scene. <https://docs.pmnd.rs/react-three-fiber>

⁹² <https://reactjs.org/docs/reconciliation.html>

[Ello1] Elliott, Conal *Functional Image Synthesis* (2001) Proceedings of Bridges

93

(C.1).

(C.2).

```

of_component : Component P A → P → Html A

pure : (P → Html A) → Component P A

with_state
  (init : P → S)
  (props_changed : P → P → S → S)
  (update : P → S → B → S × Option A)
  (c : Component (S × P) B)
  : Component P A

map_action (A → B) : Html A → Html B := ... -- reader exercise

```

The arguments `P` and `A` are called the **prop type** and the **action type**. The actions of a component perform the same role as with `Html A`, that is, they are the type of object that is returned when the user interacts with the UI in some way. The props are effectively the input arguments to the component. The reason for including `P` as a type argument (instead of writing something representable like `P → Component A`) is that it allows the component to update the state in the case of the props changing. That is, it allows the component to model the input `p : P` as a stream of values to pull from rather than as a single value. An example use of this is adding a custom check that the component needs to update; a component showing the contact information for a single student doesn't need to re-render when the entire database updates, only when the entry for that student changes.

So for example we can make a `textbox` using components:

(C.3).

```

namespace textbox

Action ::=
| add
| text_change (s : String)

State := String

init : Unit → String
| () → ""

props_changed : Unit → Unit → State → State
| () → () → s → s

update : Unit → State → Action → (State × Option String)
| () → s → add → ("", some s)
| () → s0 → text_change s1 → (s1, none)

view : (State × Unit) → Html Action
view (s, ()) :=
<div>
  <input type="text" onchange={s1 → text_change s1} value={s}/>
  <button click={() → add}></button>
</div>

comp : Component Unit String :=
  with_state Action State init props_changed update $ pure view

end textbox

textbox : Html String := of_component textbox.comp ()

```

The resulting `textbox` element contains some hidden state, namely the text content of the textbox before the "+" button is pressed. We could then place this within a larger component such as a `todolist`:

(C.4).

```

TodoItem := (label : String) × (done : Bool)
TodoList := List TodoItem

TodoAction ::=
| mark_as_done (index : N)
| add_task (value : String)

initial : TodoList :=
[ ("get groceries", false)
, ("put on instagram", false)
]

update : TodoList → TodoAction → TodoState
| s      ↦ (mark_as_done i) ↦ (s with s.l[i].done ← true)
| s      ↦ (add_task v)    ↦ (s with s.l      ← s.l ++ [(v, false)])

view : TodoList → Html TodoAction
view l :=
<ul>
  {l.mapi $ (i, label, done) ↦
    <li>
      {if done then "[x]" else "[ ]"}
      {label}
      {if not done then
        <button click={() ↦ mark_as_done i}>
          mark done
        </button>
        else []}
    </li>
  }
<hr/>
<li> {textbox} </li>
</ul>

```

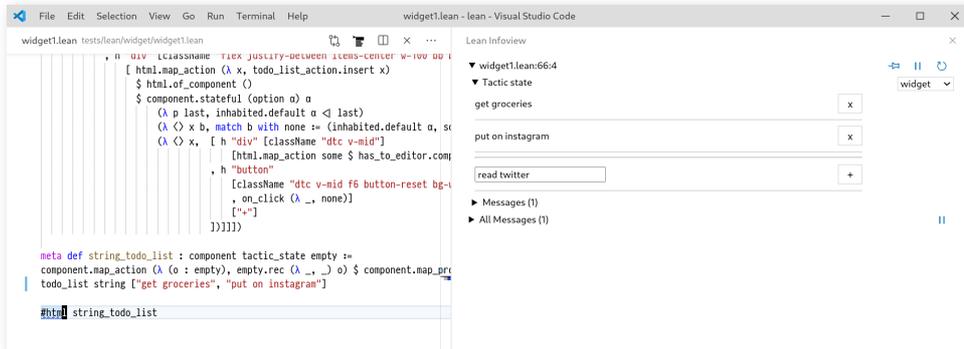


Figure C.5. A todo list component implemented in Lean 3 using an inner `textbox` component, demonstrating state encapsulation. As we type "read twitter" in the textbox, the state of the textbox component is updating but the external component's state does not change.

C.2. Abstracting the Html datatype

So now we have a framework for building apps. However now before explaining how to create a model for a widget, I want to generalise this a little further. Because really the algorithm for maintaining the view model is more general than with `Html` and instead is more about trees where events on the trees can cause changes. This work differs from React, Elm and similar functional UI frameworks in that it generalises the algorithm to an arbitrary tree structure.

So let's generalise `Html A` to instead be some arbitrary inductive datatype. To do this, note that we can write

(C.6).

```

HtmlBase A X ::=
| element : String → List (Attr A) → List X → HtmlBase A X
| of_string : String → HtmlBase A X

```

And now `Html A` is the fixpoint of `HtmlBase A`. That is, `Html A = HtmlBase A (Html A)`.

In this section we will abstract `HtmlBase A` to an arbitrary 'base functor'

`Q (E : Type) (X : Type) : Type`. The `E` type parameter represents the points where event handlers should exist on the tree.

Further, choose some type `EventArgs` to represent the data that the UI can send back to an event handler as a result of an interaction with the user. In the todo-list example in the previous section, this was `Unit` for the `click` events and `String` for the textbox change events. For simplicity let's assume that all of the event args are the same. So in the case of `Html` we would choose `Q` such that `Fix (Q (EventArgs → A))` is isomorphic to `Html A`.

In this section I will assume that `Q` is traversable on both arguments and has coordinates `C`⁹⁴.

⁹⁴ Please refer to Section 2.3.2.

Then our input will be an object `UTree`:

(C.7).

```
UTree A :=
| mk (v : Q (EventArgs → A) (UTree A))
| of_component (c : Component P A) (p : P)

Hook P_0 A_0 P_i A_i :=
  (S : Type)
  × (init : P_0 → S × P_i)
  × (reconcile : P_0 → S → S × Option P_i)
  × (update : A_i → S → S × Option P_i × Option A_0)

Component P A ::=
| pure (view : P → UTree A)
| with_hook (h : Hook P A P' A') (c : Component P' A')
```

C.3. Holding a general representation of the UI state.

So our goal here is to make a `VComponent` type that represents the UI state of a given system.

(C.8).

```
ComponentId := N

VTree A
| mk (v : Q (EventArgs → A) (VTree A))
| of_component
  (id : ComponentId)
  (c : Component P A)
  (p : P)
  (vc : VComponent P A)

VComponent P A :=
| pure (view : P → UTree A) (render : VTree A)
| with_hook (h : Hook P A P' A') (s : h.S) (vc : VComponent P' A')
```

So here, the `UTree` is a tree that the programmer creates to specify the behaviour of the widget and the `VTree` is this tree along with the states produced by each `Hook`. To create a `VTree` from a `UTree`, we use `init`:

(C.9).

```
mutable id_count ← 0

init : UTree A → VTree A :=
| mk v → mk (init <$> v)
| of_component c p → of_component (id_count++) c p (init c p)

init : Component P A → P → VComponent P A
| pure f → p → pure f $ f p
| with_hook h c → p →
  let (s, p') := h.init p in with_hook h s $ init c p'
```

Where `id_count++` returns the current value of `id_count` and increments it. A pure functional implementation would avoid this mutability by wrapping everything in a state monad, but I have omitted this in the name of reducing clutter.

C.4. Reconciliation

Once the states are initialised, we need a way of updating a tree. This is the process where we have a `v : VTree` representing the current state of the application and an `u : UTree` representing some new tree that should replace `v`. For example `u` might be a tree created for the todo-list app with a new task in the list. However all of the states on `v` must be 'carried over' to the new tree `u`. In our todo-list example, this is the constraint that the text in the new-todo textbox should not be reset if we interact with some other part of the app.

Sticking with the lingo of ReactJS, let's call this reconciliation, however [ReactJS reconciliation](#) is a more specialised affair. The purpose of reconciliation is to compare the UI tree from before the change with a new UI tree and to match up any components whose state needs to be preserved. Additionally, it is used as an efficiency measure so that components whose props have not changed do not need to be recomputed.

(C.10).

```

Q_reconcile
  (init : Y → X)
  (rec : X → Y → X)
  (old : Q E X)
  (new : Q E Y)
  : Q E X

reconcile
  : VTree A          → UITree A          → VTree A
  | mk v            ↦ mk u                ↦
    mk $ Q_reconcile init reconcile v u

  | of_component i c0 _ vc ↦ of_component c p ↦
    if c ≠ c0 then init (of_component c p)
    else of_component i c p (reconcile vc c p)

  | _                ↦ t                ↦ init t

reconcile
  : VComponent P A    → P → VComponent P A
  | pure f v          ↦ p ↦ pure f $ reconcile v $ f p
  | with_hook h s vc  ↦ p ↦
    match h.reconcile p s with
    | (s', none)     ↦ with_hook h s' vc
    | (s', some p') ↦ with_hook h s' $ reconcile vc p'

```

Here, `Q_reconcile` is a function that must be implemented per `Q`. It should work by comparing `old : Q E X` and `new : Q E Y` and then pairing off children of `x : X`, `y : Y` of `old` and `new` and applying the recursive `rec x y` to reconcile the children. In the case of children `y` that can't be paired with a child from the `old` tree, they should be converted using `init y`.

For example, in the case of `Q` being `HtmlBase`, an algorithm similar to the [one given by ReactJS](#) is used.

There is a little fudge in the above code, namely that `c ≠ c0` is not technically decidable: `view f = view g` would require checking equality on functions `f = g`. So the necessary fudge is to instead perform a reference equality check on the components or to demand that each `Component` is also constructed with some unique identifier for checking. In the actual Lean implementation, the equality check is performed by comparing the hashes of the VM objects for `c` and `c0`.

Once we have built a `VTree` representing the current interaction state of the widget, we need to convert this to pixels on the screen. The first step here is to strip out the components and produce a pure tree representing the UI tree that will be displayed. For the implementation of ProofWidgets in Lean, the final format to be sent to the client is a JSON object representing the HTML DOM that will then be rendered using React and the web browser.

Suppose that the functor `Q _ X` has coordinates `CE`. This means that we can make `get` and `coords` for `VTree A`, with the type `List C × CE`.

```

get : (List C × CE) → VTree X → Option (EventArgs → A)
| ([], e)      ↦ mk v ↦ get e v
| (h :: t, e) ↦ mk v ↦ do
  u : VTree X ← get h v,
  get (t, e) u
| _ ↦ _ ↦ none

coords : VTree X → List (List C × CE)
| (mk v) ↦
  for (e : CE) in coords v:
  yield ([], e)
  for (c : C) in coords v:
  some (x : VTree X) ← get v c
  for (l, e) in coords x:
  yield ([c, ..l], e)

```

We will use this to attach to each event handler in the `VTree` a distinct, serialisable address. This is needed because we will be sending the output tree via JSON to the client program and then if the user clicks on a button or hovers over some text, we will need to be able to discover which part of the interface they used.

```

HandlerId := (adr : (List C × CE)) × (route : List ComponentId)

OutputTree := Q (HandlerId) (OutputTree)

render : List ComponentId → VTree A → OutputTree
| l      ↦ mk v      ↦
  u : Q HandlerId (VTree A) ← traverse_with_coord (a ↦ h ↦ (a, l)) v,
  render l <$> u
| l      ↦ (of_component i c p vc) ↦ render ([..l, i]) vc

render : List ComponentId → VComponent P A → OutputTree
| l ↦ pure v r ↦ render l r
| l ↦ with_hook h s vc ↦ render l vc

```

Now that the output tree is generated, we can send it off to the client for turning in to pixels. Then the system need only wait for the user to interact with it. When the user does interact, the client will find out through the browser's API, and this will in turn be used to track down the `HandlerId` that was assigned for this event. The client will then send a `widget_event` payload back to the ProofWidgets engine consisting of `e : EventArgs` and a handler `i : HandlerId`. The `handle_event` routine takes `e` and `i` and updates the VDOM object by finding the event handler addressed by `i`, running the event handler and then propagating the resulting action back up to the root node of the tree.

```

handle_event (e : EventArgs) (i : HandlerId)
: VComponent P A → (Option A) × (VComponent P A)
| pure f v      ↦ let (a, v) := handle_event e i v in (a, pure v)
| with_hook h s v ↦
  let (b, v) := handle_event e i v in
  match b with
  | none := (none, with_hook h s v)
  | (some b) :=
    match h.update b s with
    | (s, none , a) := (a, with_hook h s v)
    | (s, some p, a) := (a, with_hook h s $ reconcile v p)

handle_event (e : EventArgs)
: HandlerId → VTree A → (Option A) × VTree A
| (l, []) ↦ mk v ↦
  (some h) ← get l v
  handle_action (h e)
| (l, [i, ..r]) ↦ v ↦
  for x in get_component_coords v:
  some (of_component i c p vc) ← get co v
  (oa, vc) ← handle_event e (l, r) vc
  return (oa, set x v $ of_component i c p vc)
| _ ↦ _ ↦ throw "event handler not found"

```

This completes the definition of the core abstract widget loop.

Lemma C.14: The above recursive definitions are all well-founded. And so the reconciliation algorithm will not loop indefinitely.

Proof: By inspecting the structure of the above recursive function definitions. One can verify that at each recursive call, the size of the `UTree`, `VTree`, `Component` and `VComponent` instances being recursed over is strictly decreasing. This means that the algorithm will terminate by a routine application of well-founded recursion. I have confirmed this by implementing a simplified version of the presented algorithm in Lean 3 and making use of Lean's automation on checking the well-foundedness of recursive functions.

C.5. Implementation

The implementation of these algorithms is in C++, and the Lean-facing API is decorated with the `meta` keyword to indicate that it is untrusted by the kernel. The actual API for `ProofWidgets` in the Lean 3 theorem prover needs to be untrusted because:

- It is convenient to allow HTML attributes to take values in untrusted datatypes such as `float`.
- The C++ implementation needs to handle additional component types not mentioned in the previous subsection such as effects.
- Inductive type declarations in Lean do not have good support for having lists of recursive arguments (eg `inductive T | mk : (list T) → T`), but it makes most sense to implement the API with lists to match the XML document model.
- The only consumer of the `ProofWidgets` API is the Lean source, so we don't lose much by not trusting the definitions.

`init`, `reconcile` and `handle_event` terminate by inspection since the each recursive call acts on a strictly smaller subtree.

Suppose that the user triggers two events in rapid succession, it might be the case that the second event is triggered before the UI has updated, and so the `HandlerId` of the second event refers to the old `VTree` which may have been updated. If the latency of the connection between client and server is low, then this is not much of an issue. However it may become apparent when using `ProofWidgets` remotely such as with VSCode's develop-over-SSH feature. Or if `ProofWidget`'s rendering methods rely on a long-running task such as running some proof automation. In these cases the UI may become unresponsive or behave in unexpected ways. There are two approaches to dealing with this:

- Simply throw away all of these queued events. However this generally leads to unresponsive UI, particularly for mouse movement driven events such as hovering over subterms.
- Attempt to call `handle_event` with the out-of-date `HandlerId` anyway. This makes the app more responsive, because multiple events can be queued during `handle_event` instead of dropped. If the second event's `HandlerId` is no longer a valid address for the `VTree`, then it will harmlessly error and be ignored. The difficult case is when the `HandlerId`'s address is *valid* but points to a different handler than the one rendered in the DOM. For example, there could be a pair of buttons, and clicking the first button swaps the order of the buttons. If the user clicks this button rapidly, then the second click will apply to the swapped button instead of the same button twice.

Currently, the Lean 3 implementation uses the first approach in this list, however this can cause unresponsiveness if a long-running tactic is used in the rendering method. As noted in Section 5.9, I intend to overcome this by adding a task-based system to components (5.19) rather than fix the event model.

Appendix D

Material for evaluation study

For the experimental user study, the participants were given a training document and a form to fill in. In this appendix I reproduce the form that they saw with some formatting changes.

D.1. Advertising email

Sent on the 5th October 2020 to the University of Cambridge mathematics mailing list.

Help us learn how mathematicians think and get a £10 amazon voucher!

We are looking for undergraduate and postgraduate mathematics students to participate in a study to help us understand how mathematicians think about proofs.

In the study, we will be asking you to read and rate various proofs of some lemmas from Part IA Group Theory and IB Metric and Topological Spaces, including some proofs which have been formally checked by a computer. You won't need to prove any of the lemmas yourself. The experiment will be on Zoom and take about an hour. As sweetening, you will receive a £10 Amazon voucher for participating!

To find out more, or to be part of our study, email Edward Ayers at e.w.ayers@maths.cam.ac.uk

D.2. Training Document

Lean is a computer program that can be used to create and verify mathematical proofs.

This document is a reference to help understand the proofs given in the experiment.

D.2.1. Expressions

Some of the proofs will use a logical system called *dependent type theory*. In contrast to set theory, this means that every term x has a type T written as $x : T$. For example

```
-- this is a comment (D.1).  
  
4 : ℕ      -- 4 is natural number  
ε : ℝ      -- ε is a real number  
f : X → Y -- f has the type of a function from X to Y
```

Definitions are created with the `def` command:

```
def my_function (x y : ℕ) := x * x + y (D.2).  
  
my_function 4 5 -- returns 21
```

D.2.2. Propositions

Propositions have the type `Prop`, and given $P : \text{Prop}$ and $Q : \text{Prop}$, we can write:

- $P \wedge Q$ for P and Q
- $P \vee Q$ for P or Q
- $P \rightarrow Q$ for P implies Q . Note arrows associate on the right, so $P \rightarrow Q \rightarrow R$ is $P \rightarrow (Q \rightarrow R)$.

- $\forall (x : X) (\epsilon : \mathbb{R}), P$ for all $x : X$ and $\epsilon : \mathbb{R}$ we have P .
- $\exists (x : X), P$: there exists an x such that P

Propositions are themselves types, so if we have $h : P$, we can read this as saying h is a proof of P . Instead of using a name like h , we can also reference propositions using $\langle \rangle$: for example $\langle 4 < 5 \rangle$ to refer to the proof that 4 is less than 5 .

D.2.3. Sets

Given a type X , we can make the type $\text{set } X$ of sets of elements of X . Given $A : \text{set } X$, and $a : X$ we can write $a \in A$ to be the proposition that a belongs to set A . The usual set theoretical operations still apply;

- the empty set $\phi : \text{set } X$
- union $A \cup B$
- intersection $A \cap B$
- set comprehension $\{a : \mathbb{N} \mid a > 4\}$ to mean the set of naturals $a : \mathbb{N}$ such that $a > 4$.

D.2.4. Tactic notation

Tactic notation is used to prove theorems, there are many tactics which will be explained as they come up.

The lines between `begin` and `end` are called *commands* and are used to change the *goal state* of the proof.

(D.3).

```
theorem an_example : (∃ (x : ℕ), x > 3) → (∃ (y : ℕ), y > 4) :=
begin
  -- introduce an assumption
  assume h₁ : (∃ x, x > 3),

  -- obtain x from the assumption
  obtain ⟨x, h₂⟩ : ∃ x, x > 3,
  { from h₁,
  },

  -- remind ourselves of the goal
  show ∃ (y : ℕ), y > 4,

  -- choose y to be x + 1
  use x + 1,

  -- show that x + 1 > 4 using a calculation
  calc x + 1 > 3 + 1 : by apply nat.add_lt_add_right h₂ 1
      ... = 4      : by norm_num
end
```

D.2.5. Reference table for tactics

Table D.4.

Name	description	Example	before	after
<code>assume</code>	Unwraps a forall or implication goal.	<code>assume h : P</code>	$\vdash P \rightarrow Q$	$(h : P) \vdash Q$
Untitled		<code>assume $\epsilon : \mathbb{R}$</code>	$\vdash \forall (\epsilon : \mathbb{R}), \epsilon + 1 > \epsilon$	$(\epsilon : \mathbb{R}) \vdash \epsilon + 1 > \epsilon$
<code>show</code>	State the current goal. Doesn't change the goal state.	<code>show P</code>	$\vdash P$	$\vdash P$
<code>have</code>	Make a new goal for <code>P</code> .	<code>have h : P</code>	$\vdash Q$	$\vdash P, (h:P) \vdash Q$
<code>obtain</code>	Similar to <code>have</code> . Make a new goal for <code>P</code> and eliminate <code>\exists</code> and <code>\wedge</code> .	<code>obtain (x, h1, h2) : $\exists (x : X), P \wedge Q$</code>	$\vdash R$	$\vdash \exists (x : X), P \wedge Q, (x : X) (h1 : P) (h2 : Q) \vdash R$
<code>use</code>	Gives a value for an <code>\exists</code> goal.	<code>use 5</code>	$\vdash \exists (\epsilon : \mathbb{R}), \epsilon > 0$	$\vdash 5 > 0$
<code>apply</code>	Apply the given expression to the goal.	<code>apply <P → Q></code>	$\vdash Q$	$\vdash P$
<code>split</code>	Splits an <code>\wedge</code> goal.	<code>split</code>	$\vdash P \wedge Q$	$\vdash P, \vdash Q$
<code>calc</code>	Used to write out a chain of equalities and inequalities.	<code>calc 4 < 5 : by norm_num ... = x : by assumption</code>	$(h : 5 = x) \vdash 4 < x$	done
<code>let</code>	Defines a new constant.	<code>let x := 5</code>	$\vdash P$	$\vdash P$
<code>rewrite</code>	Rewrites an expression with the given equation.	<code>rewrite <x = y></code>	$\vdash x + y < 4$	$\vdash x + x < 4$

Table D.5. Table D.4 continued.

Name	description	Example	before	after
<code>cases</code>	Splits an 'or' hypothesis in to a pair of cases.	<code>cases h</code>	$(h : P \vee Q) \vdash R$	$P \vdash R, Q \vdash R$
<code>finish</code>	Tries to solve the goal by repeatedly applying assumptions.	<code>finish</code>		
<code>simp</code>	Tries to solve the goal by simplifying expressions	<code>simp</code>		
<code>norm_num</code>	Solves various simple number problems.	<code>norm_num</code>	$\vdash 5 > 0$	done
<code>ring</code>	Tries to normalise and solve equations and inequalities in rings.	<code>ring</code>	$\vdash (x + y)^2 = x^2 + 2 * x * y + y^2$	done
<code>assumption</code>	Find a previous <code>assume</code> or <code>have</code> result that achieves the goal	<code>assumption</code>	$(h : P) \vdash P$	done

D.3. Proof scripts

note: any paragraphs prepended with "**note:**" are not seen by the experiment participants. Additionally, the order in which they were presented the proofs and lemmas was randomised for each participant, with the exception of Lemma 1 which was always shown first.

D.3.1. Lemma 1. The composition of group homomorphisms is a group homomorphism.

The composition of two group homomorphisms is a group homomorphism.

D.3.1.1. Definitions

Here we write the group product on two group elements $x, y : G$ as $x * y$ and write $x^{-1} : G$ as the inverse element of x . The identity is written as $e : G$.

Given a pair of groups G, H , a function $f : G \rightarrow H$ is a **group homomorphism** whenever $f(x * y) = f x * f y$ for all $x, y \in G$.

```
variables {G H I : Type} [group G] [group H] [group I]
variables {f : G → H} {g : H → I}
def is_hom (f : G → H) := ∀ (x y : G), f (x * y) = f x * f y
(g ∘ f) x := g (f (x))
```

(D.6).

D.3.1.2. Proof A

```
theorem hom_composition : is_hom f → is_hom g → is_hom (g ∘ f) :=
begin
  assume hf : is_hom f,
  assume hg : is_hom g,
  assume x y : G,
  calc g (f (x * y)) = g (f x * f y)      : by rewrite hf
    ... = (g (f x)) * (g (f y))          : by rewrite hg,
end
```

(D.7).

D.3.1.3. Proof B

Let G, H and I be groups and let $f : G \rightarrow H$ and $g : H \rightarrow I$ where f and g are group homomorphisms. Let x and y be elements of G . Since f and g are group homomorphisms, we have

$$\begin{aligned} (g \circ f)(x * y) &= g(f(x * y)) \\ &= g(f(x) * f(y)) \\ &= g(f(x)) * g(f(y)) \\ &= (g \circ f)(x) * g(f(y)) \\ &= (g \circ f)(x) * (g \circ f)(y) \end{aligned}$$

(D.8).

We are done.

D.3.1.4. Proof C

note: Adapted from Dexter Chua's notes on IA Group Theory [Chu18].

[Chu18] Chua, Dexter Cambridge Notes (2018) <https://dec41.user.srcf.net/notes>

Let $f : G \rightarrow H$ and $g : H \rightarrow I$ be group homomorphisms and let x and y be elements of G . Then we have

$$\begin{aligned} (g \circ f)(x * y) &= g(f(x * y)) \\ &= g(f(x) * f(y)) \\ &= g(f(x)) * g(f(y)) \\ &= (g \circ f)(x) * (g \circ f)(y) \end{aligned}$$

(D.9).

and we are done.

D.3.2. Lemma 2. The union of two open sets is open.

D.3.2.1. Definitions

Recall that a metric space is a type X equipped with a distance function

$$\text{dist} : X \rightarrow X \rightarrow \mathbb{R}$$

(D.10).

such that the following properties hold:

- $\text{dist } x \ x = 0$ for all $x : X$
- $\text{dist } x \ y = \text{dist } y \ x$ for all $x, y : X$
- $\text{dist } x \ z \leq \text{dist } x \ y + \text{dist } y \ z$ for all $x, y, z : X$

We say a subset $A : \text{set } X$ is **open** when

$$\forall (x : X), (x \in A) \rightarrow \exists (\epsilon : \mathbb{R}), (\epsilon > 0) \wedge \forall (y : X), \text{dist } y \ x < \epsilon \rightarrow y \in A$$

(D.11).

D.3.2.2. Proof A

(D.12).

```

example : is_open A → is_open B → is_open (A ∪ B) :=
begin
  assume h1 : is_open A
  assume h2 : is_open B,
  assume y : X,
  assume h3 : y ∈ A ∪ B,
  cases <y ∈ A ∪ B>,
  { -- in the case that y ∈ A
    obtain <η, η_pos, h4> : ∃ η, (η > 0) ∧ ∀ x, dist x y < η → x ∈ A,
    { apply <is_open A>,
      apply <y ∈ A>,
    },
    show ∃ ε, (ε > 0) ∧ ∀ x, dist x y < ε → x ∈ A ∪ B,
    use [η, η_pos],
    show ∀ x, dist x y < η → x ∈ A ∪ B,
    assume x : X,
    assume h5 : dist x y < η,
    show x ∈ A ∪ B,
    apply set.subset_union_left, -- A ⊆ A ∪ B
    show x ∈ A,
    finish,
  }, { -- in the case that y ∈ B
    obtain <θ, θ_pos, hθ> : ∃ θ, (θ > 0) ∧ ∀ x, dist x y < θ → x ∈ B,
    { apply <is_open B>,
      apply <y ∈ B>,
    },
    use [θ, θ_pos],
    assume x : X,
    assume h5 : dist x y < θ,
    apply set.subset_union_right, -- B ⊆ A ∪ B
    finish,
  }
end

```

D.3.2.3. Proof B

Let X be a metric space and A, B be sets on X . Assume A and B are open. Let $y \in A \cup B$. We must choose $\varepsilon > 0$ such that $\forall (x : X), \text{dist } x y < \varepsilon \rightarrow x \in A \cup B$. Let x be a point in X where $\text{dist } x y < \varepsilon$. We must show that $x \in A \cup B$. Since $y \in A \cup B$, either $y \in A$ or $y \in B$.

In the case $y \in A$: Since $A \subseteq A \cup B$, it suffices to show $x \in A$. Since A is open and $y \in A$, there exists $\eta > 0$ such that $x \in A$ whenever $\text{dist } x y < \eta$. Therefore, setting ε to be η we are done.

In the case $y \in B$: Since $B \subseteq A \cup B$, it suffices to show $x \in B$. Since B is open and $y \in B$, there exists $\theta > 0$ such that $x \in B$ whenever $\text{dist } x y < \theta$. Therefore, setting ε to be θ we are done.

D.3.2.4. Proof C

note: Adapted from *Measure, Topology and Fractal Geometry* by Gerald Edgar [Edg07].

Let $y \in A \cup B$. Then either $y \in A$ or $y \in B$. In the case that $y \in A$, there is some $\varepsilon > 0$ such that for all $x : X$, $\text{dist } x y < \varepsilon$ implies $x \in A$. Similarly for $y \in B$. So $A \cup B$ is an open set.

[Edg07] **Edgar, Gerald** *Measure, topology, and fractal geometry* (2007) **publisher** Springer

D.3.3. Lemma 3. The kernel of a group homomorphism is a normal subgroup.

D.3.3.1. Definitions

Given groups G and H and a function $f : G \rightarrow H$, f is a group homomorphism if we have $f(x * y) = (f x) * (f y)$ for all x, y in G . If f is a group homomorphism then it may also be shown that $f e = e$ (where e is the identity element) and $f(x^{-1}) = (f x)^{-1}$.

Define the **kernel** of f to be the set $\{k : G \mid f k = e\}$. It can be shown that the kernel of f is a subgroup of G .

A subgroup K of G is said to be **normal** when for all $k \in K$ and all $g \in G$, we have $g * k * g^{-1} \in K$.

(D.13).

```

variables {G : Type} [group G]
variables {H : Type} [group H]

def is_hom (f : G → H) :=
  ∀ (x y : G), f (x * y) = f x * f y

variables {f : G → H} [is_hom f]

def is_hom.one : f e = e := -- proof omitted

def is_hom.inv : ∀ (x : G), f (x⁻¹) = (f x)⁻¹ := -- proof omitted

def kernel (f : G → H) [is_hom f] : subgroup G :=
  {k : G | f k = e},
  ... -- proof that the kernel is a subgroup omitted

def is_normal (K : subgroup G) :=
  ∀ (k : G), (k ∈ K) → ∀ (g : G), g * k * g⁻¹ ∈ K

```

D.3.3.2. Proof A

(D.14).

```

theorem kernel_is_normal : is_normal (kernel f) :=
begin
  assume k : G,
  assume h₁ : k ∈ kernel f,
  assume g : G,
  calc f (g * k * g⁻¹) = f (g * k) * f g⁻¹ : by rewrite <is_hom f>
    ... = f g * f k * f g⁻¹ : by rewrite <is_hom f>
    ... = f g * e * f g⁻¹ : by rewrite <k ∈ kernel f>
    ... = f g * e * (f g)⁻¹ : by rewrite (is_hom.inv f)
    ... = e : by simplify
end

```

D.3.3.3. Proof B

Let $k \in \text{kernel } f$ and g be an element of G . We must show $f (g * k * g^{-1}) = e$. Since $f k = e$, we have

(D.15).

$$\begin{aligned}
 f (g * k * g^{-1}) &= f (g * k) * f g^{-1} \\
 &= f g * f k * f g^{-1} \\
 &= f g * e * f g^{-1} \\
 &= f g * f g^{-1} \\
 &= f (g * g^{-1}) \\
 &= f e \\
 &= e
 \end{aligned}$$

We are done.

D.3.3.4. Proof C

note: Adapted from [Dexter Chua's notes on IA Group Theory](#)

Given homomorphism $f : H \rightarrow G$, and some $g \in G$, for all $k \in \text{kernel } f$, we have $f (g * k * g^{-1}) = f g * f k * (f g)^{-1} = f g * e * (f g)^{-1} = e$. Therefore $g * k * g^{-1} \in \text{kernel } f$ by definition of the kernel.

D.3.4. Lemma 4. The intersection of two open sets is open.

D.3.4.1. Proof A

`theorem intersection_is_open {A B : set X} : is_open A → is_open B → is_open (A ∩ B) :=`

(D.16).

```
begin
  assume h₁ : is_open A,
  assume h₂ : is_open B,
  assume y : X,
  assume h₃ : y ∈ A ∩ B,
  cases <y ∈ A ∩ B>,
  obtain <η, η_pos, h_η> : ∃ η, (η > 0) ∧ ∀ x, dist x y < η → x ∈ A,
  apply <is_open A>, apply <y ∈ A>,
  obtain <θ, θ_pos, h_θ> : ∃ θ, (θ > 0) ∧ ∀ x, dist x y < θ → x ∈ B,
  apply <is_open B>, apply <y ∈ B>,
  let ε := min η θ,
  have ε_pos : ε > 0,
  apply lt_min <η > 0> <θ > 0>,
  show ∃ (ε : ℝ), ε > 0 ∧ ∀ (x : X), (dist x y < ε) → (x ∈ (A ∩ B)),
  use [ε, <ε > 0>],
  assume x : X,
  assume h₄ : dist x y < ε,
  have : dist x y < η,
  calc dist x y < min η θ : <dist x y < ε>
    ... ≤ η           : min_le_left _ _,
  have : dist x y < θ,
  calc dist x y < min η θ : <dist x y < ε>
    ... ≤ θ           : min_le_right _ _,
  show x ∈ A ∩ B,
  split,
  show x ∈ A,
  apply h_η, apply <dist x y < η>,
  show x ∈ B,
  apply h_θ, apply <dist x y < θ>,
end
```

D.3.4.2. Proof B

Let y be an element of $A \cap B$. Then $y \in A$ and $y \in B$. Therefore, since A is open, there exists $\eta > 0$ such that $x \in A$ whenever $\text{dist } x y < \eta$ and since B is open, there exists $\theta > 0$ such that $x \in B$ whenever $\text{dist } x y < \theta$. We must choose $\varepsilon > 0$ such that $x \in A \cap B$ whenever $\text{dist } x y < \varepsilon$. Suppose $\text{dist } x y < \varepsilon$. Then $\text{dist } x y < \eta$ if $\varepsilon \leq \eta$ and $\text{dist } x y < \theta$ if $\varepsilon \leq \theta$. We are done by setting ε to be $\min \eta \theta$.

D.3.4.3. Proof C

note: Adapted from *Measure, Topology and Fractal Geometry* by Gerald Edgar [Edg07].

[Edg07] **Edgar, Gerald** *Measure, topology, and fractal geometry* (2007) **publisher** Springer

Suppose A and B are both open. Let $y \in A \cap B$. Since A is open, there is $\eta > 0$ with $\text{dist } x y < \eta \rightarrow x \in A$ for all x . Also, since B is open, there is $\theta > 0$ with $\text{dist } x y < \theta \rightarrow x \in B$ for all x . Therefore, if ε is the minimum of η and θ , then we have $x \in A \cap B$ whenever $\text{dist } x y < \varepsilon$ for all x . So $A \cap B$ is an open set.

D.4. Consent form

Each participant signed this form before their session started.

This study is part of my PhD research on creating better proof assistant tools for mathematicians.

I am interested in how people compare proofs of theorems created using a proof assistant and created using natural language.

The experiment consists of 4 rounds. In each round you will be asked to compare and evaluate a set of proofs for the same mathematical lemma. Some of the proofs are written in Lean, software for creating formally verified proofs and others are written in the style of natural language. Before the experiment starts there will be a brief training phase on the syntax of Lean. After the experiment, there will be a debrief phase involving some discussion and a brief questionnaire.

Please note that none of the tasks are a test of you or your mathematical ability; the goal is to understand the properties of proofs that you find understandable and useful.

The experiment will be conducted remotely via Zoom. Please use a desktop OS such as Linux, macOS or Windows with a copy of Zoom installed. Please make sure that you are in a quiet environment suitable for a meeting.

D.4.1. Confidentiality

The following data will be recorded:

- name
- email address
- audio or video recording of the experiment

These will only be used to communicate with you, or better understand your responses and will only be visible to me, the experimenter. If video is recorded, it will be deleted immediately after the experiment and only the audio will be kept. Your name, email and recordings will never be publicly released and will be deleted by 1st February 2021.

Additionally, the following data during the experiment will be textually recorded and anonymised.

- Your answers to the forms and surveys during the experiment.
- Any verbal answers and comments you give during the experiment. Note that you can explicitly request for me to discard any of these if you wish.
- Transcripts of quotes from the audio recording may be publicly released.

These may be released publicly but your anonymity will be protected in any papers, peer review, institutional repositories and presentations that result from this work.

D.4.2. Finding out about the results

If interested, you can email me at edward.ayers@outlook.com in 2021 to hear about the results of the study.

D.4.3. Record of consent

Your signature below indicates that you have understood the information about the experiment and consent to your participation. The participation is voluntary and you may refuse to answer certain questions and withdraw from the study and request your data be deleted at any time with no penalty. This does not waive your legal rights. You should have received a copy of the consent form for your own record. If you have further questions related to this research, please contact me at edward.ayers@outlook.com.

Bibliography

- [ABB+16] **Ahrendt, Wolfgang; Beckert, Bernhard; Bubel, Richard; Hähnle, Reiner; Schmitt, Peter H.; Ulbrich, Mattias** *Deductive Software Verification - The KeY Book* (2016) **publisher** Springer **doi** 10.1007/978-3-319-49812-6 **isbn** 978-3-319-49811-9 <https://doi.org/10.1007/978-3-319-49812-6>
- [ADL10] **Aspinall, David; Denney, Ewen; Lüth, Christoph** *Tactics for hierarchical proof* (2010) Mathematics in Computer Science **volume** 3 **number** 3 **pages** 309--330 **publisher** Springer **doi** 10.1007/s11786-010-0025-6 <https://doi.org/10.1007/s11786-010-0025-6>
- [AGJ19] **Ayers, E. W.; Gowers, W. T.; Jamnik, Mateja** *A human-oriented term rewriting system* (2019) KI 2019: Advances in Artificial Intelligence - 42nd German Conference on AI **volume** 11793 **pages** 76--86 **editors** Benz Müller, Christoph; Stuckenschmidt, Heiner **organization** Springer **publisher** Springer **doi** 10.1007/978-3-030-30179-8_6 <https://www.repository.cam.ac.uk/bitstream/handle/1810/298199/main.pdf?sequence=1>
- [AH97] **Archer, Myla; Heitmeyer, Constance** *Human-style theorem proving using PVS* (1997) International Conference on Theorem Proving in Higher Order Logics **pages** 33--48 **editors** Gunter, Elsa L.; Felty, Amy P. **organization** Springer **doi** 10.1007/BFb0028384 <https://doi.org/10.1007/BFb0028384>
- [AJG21] **Ayers, E. W.; Jamnik, Mateja; Gowers, W. T.** *A graphical user interface framework for formal verification* (2021) Interactive Theorem Proving **volume** 193 **pages** 4:1--4:16 **editors** Cohen, Liron; Kaliszyk, Cezary **publisher** Schloss Dagstuhl - Leibniz-Zentrum für Informatik **doi** 10.4230/LIPIcs.ITP.2021.4 <https://doi.org/10.4230/LIPIcs.ITP.2021.4>
- [ALW07] **Aspinall, David; Lüth, Christoph; Winterstein, Daniel** *A framework for interactive proof* (2007) Towards Mechanized Mathematical Assistants **pages** 161--175 **editors** Kauers, Manuel; Kerber, Manfred; Miner, Robert; *et al.* **publisher** Springer **doi** 10.1007/978-3-540-73086-6_15 https://doi.org/10.1007/978-3-540-73086-6_15
- [AMM18] **Adámek, Jiří; Milius, Stefan; Moss, Lawrence S** *Fixed points of functors* (2018) Journal of Logical and Algebraic Methods in Programming **volume** 95 **pages** 41--81 **doi** 10.1016/j.jlamp.2017.11.003 <https://doi.org/10.1016/j.jlamp.2017.11.003>
- [AZHE10] **Aigner, Martin; Ziegler, Günter M; Hofmann, Karl H; Erdos, Paul** *Proofs from the Book* (2010) **publisher** Springer **isbn** 978-3-662-57264-1 <https://doi.org/10.1007/978-3-662-57265-8>
- [Ano94] **Anonymous** *The QED manifesto* (1994) Automated Deduction--CADE **volume** 12 **pages** 238--251 http://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/wiedijk_2.pdf Unofficially credited as Robert Boyer
- [Asp00] **Aspinall, David** *Proof General: A generic tool for proof development* (2000) International Conference on Tools and Algorithms for the Construction and Analysis of Systems **volume** 1785 **pages** 38--43 **editors** Graf, Susanne; Schwartzbach, Michael I. **organization** Springer **publisher** Springer **doi** 10.1007/3-540-46419-0_3 https://link.springer.com/content/pdf/10.1007/3-540-46419-0_3.pdf
- [BBF+16] **Blanchette, Jasmin Christian; Böhme, Sascha; Fleury, Mathias; Smolka, Steffen Juilf; Steckermeier, Albert** *Semi-intelligible Isar proofs from machine-generated proofs* (2016) Journal of Automated Reasoning **volume** 56 **number** 2 **pages** 155--200 **publisher** Springer **doi** 10.1007/s10817-015-9335-3 <https://doi.org/10.1007/s10817-015-9335-3>
- [BBH105] **Bundy, Alan; Basin, David; Hutter, Dieter; Ireland, Andrew** *Rippling: meta-level guidance for mathematical reasoning* (2005) **volume** 56 **publisher** Cambridge University Press **isbn** 978-0-521-83449-0 <https://books.google.co.uk/books?id=dZzbl-lnjVEC>
- [BCC+13] **Bainomugisha, Engineer; Carreton, Andoni Lombide; Cutsem, Tom van; Mostinckx, Stijn; Meuter, Wolfgang de** *A survey on reactive programming* (2013) ACM Computing Surveys (CSUR) **volume** 45 **number** 4 **pages** 1--34 **editor** Hankin, Chris **publisher** ACM New York, NY, USA **doi** 10.1145/2501654.2501666 <https://doi.org/10.1145/2501654.2501666>
- [BCF+97] **Benz Müller, Christoph; Cheikhrouhou, Lassaad; Fehrer, Detlef; Fiedler, Armin; Huang, Xiaorong; Kerber, Manfred; Kohlhase, Michael; Konrad, Karsten; Meier, Andreas; Melis, Erica; Schaarschmidt, Wolf; Siekmann, Jörg H.; Sorge, Volker** *Omega: Towards a Mathematical Assistant* (1997) Automated Deduction - CADE-14 **volume** 1249 **pages** 252--255 **editor** McCune, William **publisher** Springer **doi** 10.1007/3-540-63104-6_23 https://doi.org/10.1007/3-540-63104-6_23
- [BCJ+06] **Buchberger, Bruno; Crăciun, Adrian; Jebelean, Tudor; Kovács, Laura; Kutsia, Temur; Nakagawa, Koji; Piroi, Florina; Popov, Nikolaj; Robu, Judit; Rosenkranz, Markus; Windsteiger, Wolfgang** *Theorema: Towards computer-aided mathematical theory exploration* (2006) Journal of Applied Logic **volume** 4 **number** 4 **pages** 470--504 **editor** Benz Müller, Christoph **publisher** Elsevier **doi** 10.1016/j.jal.2005.10.006 <https://doi.org/10.1016/j.jal.2005.10.006>

- [BE92] **Barwise, Jon; Etchemendy, John** *Hyperproof: Logical reasoning with diagrams* (1992) Working Notes of the AAAI Spring Symposium on Reasoning with Diagrammatic Representations <https://www.aaai.org/Papers/Symposia/Spring/1992/SS-92-02/SS92-02-016.pdf>
- [BF97] **Blum, Avrim L; Furst, Merrick L** *Fast planning through planning graph analysis* (1997) Artificial intelligence **volume** 90 **number** 1-2 **pages** 281--300 **editor** Bobrow, Daniel G. **publisher** Elsevier **doi** 10.1016/S0004-3702(96)00047-1 [https://doi.org/10.1016/S0004-3702\(96\)00047-1](https://doi.org/10.1016/S0004-3702(96)00047-1)
- [BG01] **Bachmair, Leo; Ganzinger, Harald** *Resolution theorem proving* (2001) Handbook of automated reasoning **pages** 19--99 **editors** Robinson, J. A.; Voronkov, A. **publisher** Elsevier
- [BGM+13] **Bird, Richard; Gibbons, Jeremy; Mehner, Stefan; Voigtländer, Janis; Schrijvers, Tom** *Understanding idiomatic traversals backwards and forwards* (2013) Proceedings of the 2013 ACM SIGPLAN symposium on Haskell **pages** 25--36 <https://lirias.kuleuven.be/retrieve/237812>
- [BJK+16] **Buchberger, Bruno; Jebelean, Tudor; Kutsia, Temur; Maletzky, Alexander; Windsteiger, Wolfgang** *Theorema 2.0: computer-assisted natural-style mathematics* (2016) Journal of Formalized Reasoning **volume** 9 **number** 1 **pages** 149--185 **doi** 10.6092/issn.1972-5787/4568 <https://doi.org/10.6092/issn.1972-5787/4568>
- [BKM95] **Boyer, Robert S; Kaufmann, Matt; Moore, J Strother** *The Boyer-Moore theorem prover and its interactive enhancement* (1995) Computers & Mathematics with Applications **volume** 29 **number** 2 **pages** 27--62 **publisher** Elsevier
- [BM72] **Boyer, R. S.; Moore, J. S.** *The sharing structure in theorem-proving programs* (1972) Machine intelligence **volume** 7 **pages** 101--116 **editors** Meltzer, B.; Michie, D. **publisher** Edinburgh University Press <https://www.cs.ute.xas.edu/~moore/publications/structure-sharing-mi7.pdf>
- [BM73] **Boyer, Robert S.; Moore, J. Strother** *Proving Theorems about LISP Functions* (1973) IJCAI **pages** 486--493 **editor** Nilsson, Nils J. **publisher** William Kaufmann <http://ijcai.org/Proceedings/73/Papers/053.pdf>
- [BM90] **Boyer, Robert S; Moore, J Strother** *A theorem prover for a computational logic* (1990) International Conference on Automated Deduction **pages** 1--15 **organization** Springer <https://www.cs.utexas.edu/users/boyer/ftp/cli-reports/054.pdf>
- [BMR+20] **Brown, Tom B.; Mann, Benjamin; Ryder, Nick; Subbiah, Melanie; Kaplan, Jared; Dhariwal, Prafulla; Neelakantan, Arvind; Shyam, Pranav; Sastry, Girish; Askell, Amanda; Agarwal, Sandhini; Herbert-Voss, Ariel; Krueger, Gretchen; Henighan, Tom; Child, Rewon; Ramesh, Aditya; Ziegler, Daniel M.; Wu, Jeffrey; Winter, Clemens; Hesse, Christopher; Chen, Mark; Sigler, Eric; Litwin, Mateusz; Gray, Scott; Chess, Benjamin; Clark, Jack; Berner, Christopher; McCandlish, Sam; Radford, Alec; Sutskever, Ilya; Amodei, Dario** *Language Models are Few-Shot Learners* (2020) NeurIPS **editors** Larochelle, Hugo; Ranzato, Marc'Aurelio; Hadsell, Raia; *et al.* <https://proceedings.nips.cc/paper/2020/hash/1457c0d6bfbcb4967418bfb8ac142f64a-Abstract.html>
- [BN10] **Böhme, Sascha; Nipkow, Tobias** *Sledgehammer: judgement day* (2010) International Joint Conference on Automated Reasoning **pages** 107--121 **editors** Giesl, Jürgen; Hähnle, Reiner **organization** Springer **doi** 10.1007/978-3-642-14203-1_9 https://doi.org/10.1007/978-3-642-14203-1_9
- [BN98] **Baader, Franz; Nipkow, Tobias** *Term rewriting and all that* (1998) **publisher** Cambridge University Press **isbn** 978-0-521-45520-6
- [BSV+93] **Bundy, Alan; Stevens, Andrew; Van Harmelen, Frank; Ireland, Andrew; Smail, Alan** *Rippling: A heuristic for guiding inductive proofs* (1993) Artificial Intelligence **volume** 62 **number** 2 **pages** 185--253 **publisher** Elsevier **doi** 10.1016/0004-3702(93)90079-Q [https://doi.org/10.1016/0004-3702\(93\)90079-Q](https://doi.org/10.1016/0004-3702(93)90079-Q)
- [BT98] **Bertot, Yves; Théry, Laurent** *A generic approach to building user interfaces for theorem provers* (1998) Journal of Symbolic Computation **volume** 25 **number** 2 **pages** 161--194 **publisher** Elsevier **doi** 10.1006/jsc.1997.0171 <https://doi.org/10.1006/jsc.1997.0171>
- [Bau20] **Bauer, Andrej** *What makes dependent type theory more suitable than set theory for proof assistants?* (2020) <https://mathoverflow.net/q/376973> MathOverflow answer
- [Bil05] **Bille, Philip** *A survey on tree edit distance and related problems* (2005) Theoretical computer science **volume** 337 **number** 1-3 **pages** 217--239 **publisher** Elsevier **doi** 10.1016/j.tcs.2004.12.030 <https://doi.org/10.1016/j.tcs.2004.12.030>
- [Ble81] **Bledsoe, Woodrow W** *Non-resolution theorem proving* (1981) Readings in Artificial Intelligence **pages** 91--108 **editor** Meltzer, Bernard **publisher** Elsevier **doi** 10.1016/0004-3702(77)90012-1 [https://doi.org/10.1016/0004-3702\(77\)90012-1](https://doi.org/10.1016/0004-3702(77)90012-1)
- [Bre16] **Breitner, Joachim** *Visual theorem proving with the Incredible Proof Machine* (2016) International Conference on Interactive Theorem Proving **pages** 123--139 **editors** Blanchette, Jasmin Christian; Merz, Stephan **publisher** Springer **doi** 10.1007/978-3-319-43144-4_8 <https://www.info.uni-karlsruhe.de/uploads/publikationen/breitner16incredible.pdf>

- [Bun02] **Bundy, Alan** *A critique of proof planning* (2002) Computational Logic: Logic Programming and Beyond **pages** 160–177 **editors** Kakas, Antonis C.; Sadri, Fariba **publisher** Springer **doi** 10.1007/3-540-45632-5_7 https://doi.org/10.1007/3-540-45632-5_7
- [Bun11] **Bundy, Alan** *Automated theorem provers: a practical tool for the working mathematician?* (2011) Annals of Mathematics and Artificial Intelligence **volume** 61 **number** 1 **pages** 3–14 **doi** 10.1007/s10472-011-9248-8 <https://doi.org/10.1007/s10472-011-9248-8>
- [Bun88] **Bundy, Alan** *The use of explicit plans to guide inductive proofs* (1988) International conference on automated deduction **volume** 310 **pages** 111–120 **editors** Lusk, Ewing L.; Overbeek, Ross A. **organization** Springer **publisher** Springer **doi** 10.1007/BFb0012826 <https://doi.org/10.1007/BFb0012826>
- [Bun98] **Bundy, Alan** *Proof Planning* (1998) **publisher** University of Edinburgh, Department of Artificial Intelligence <https://books.google.co.uk/books?id=h7hrHAAACAAJ>
- [CC13] **Czaplicki, Evan; Chong, Stephen** *Asynchronous functional reactive programming for GUIs* (2013) ACM SIGPLAN Conference on Programming Language Design and Implementation **pages** 411–422 **editors** Boehm, Hans-Juergen; Flanagan, Cormac **publisher** ACM **doi** 10.1145/2491956.2462161 <https://doi.org/10.1145/2491956.2462161>
- [CFK+09] **Cramer, Marcos; Fisseni, Bernhard; Koepke, Peter; Kühlwein, Daniel; Schröder, Bernhard; Veldman, Jip** *The Naproche Project: Controlled Natural Language Proof Checking of Mathematical Texts* (2009) Controlled Natural Language, Workshop on Controlled Natural Language **volume** 5972 **pages** 170–186 **editor** Fuchs, Norbert E. **publisher** Springer **doi** 10.1007/978-3-642-14418-9_11 https://doi.org/10.1007/978-3-642-14418-9_11
- [CH88] **Coquand, Thierry; Huet, Gérard P.** *The Calculus of Constructions* (1988) Information and Computation **volume** 76 **number** 2/3 **pages** 95–120 **publisher** Elsevier **doi** 10.1016/0890-5401(88)90005-3 [https://doi.org/10.1016/0890-5401\(88\)90005-3](https://doi.org/10.1016/0890-5401(88)90005-3)
- [CL18] **Choi, Dongkyu; Langley, Pat** *Evolution of the ICARUS cognitive architecture* (2018) Cognitive Systems Research **volume** 48 **pages** 25–38 **publisher** Elsevier **doi** 10.1016/j.cogsys.2017.05.005 <https://doi.org/10.1016/j.cogsys.2017.05.005>
- [CMM+17] **Corneli, Joseph; Martin, Ursula; Murray-Rust, Dave; Pease, Alison; Puzio, Raymond; Rino Nesin, Gabriela** *Modelling the way mathematics is actually done* (2017) Proceedings of the 5th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design **pages** 10–19 **editors** Sperber, Michael; Bresson, Jean **organization** ACM **publisher** ACM **doi** 10.1145/3122938.3122942 <https://doi.org/10.1145/3122938.3122942>
- [Car19] **Carneiro, Mario** *Lean's Type Theory* (2019) <https://github.com/digama0/lean-type-theory/releases/download/v1.0/main.pdf>
- [Chu18] **Chua, Dexter** *Cambridge Notes* (2018) <https://dec41.user.srcf.net/notes> GitHub: <https://github.com/dalcde/cam-notes>
- [Com20] **The Mathlib Community** *The Lean Mathematical Library* (2020) Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs **pages** 367–381 **publisher** Association for Computing Machinery **doi** 10.1145/3372885.3373824 **isbn** 9781450370974 <https://doi.org/10.1145/3372885.3373824>
- [Coq] **The Coq Development Team** *The Coq Reference Manual* (2021) <https://coq.inria.fr/distrib/current/refman/#>
- [DH93] **Dalianis, Hercules; Hovy, Eduard** *Aggregation in natural language generation* (1993) European Workshop on Trends in Natural Language Generation **volume** 1036 **pages** 88–105 **editors** Adorni, Giovanni; Zock, Michael **organization** Springer **publisher** Springer **doi** 10.1007/3-540-60800-1_25 https://doi.org/10.1007/3-540-60800-1_25
- [DJP06] **Dennis, Louise A; Jamnik, Mateja; Pollet, Martin** *On the Comparison of Proof Planning Systems: lambdaCLAM, Omega and IsaPlanner* (2006) Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning **volume** 151 **number** 1 **pages** 93–110 **editors** Carette, Jacques; Farmer, William M. **publisher** Elsevier **doi** 10.1016/j.entcs.2005.11.025 https://www.cl.cam.ac.uk/~mj201/publications/comp_pp_final.pdf
- [DKL20] **De Lon, Adrian; Koepke, Peter; Lorenzen, Anton** *Interpreting Mathematical Texts in Naproche-SAD* (2020) Intelligent Computer Mathematics **pages** 284–289 **editors** Benz Müller, Christoph; Miller, Bruce **publisher** Springer International Publishing **doi** 10.1007/978-3-030-53518-6_19 **isbn** 978-3-030-53518-6 https://doi.org/10.1007/978-3-030-53518-6_19
- [Dav09] **Davis, Jared Curran** *A self-verifying theorem prover* (2009) <https://search.proquest.com/openview/96fda9a67e5fa11fb241ebf4984c7368/1?pq-origsite=gscholar&cbl=18750>
- [DeB80] **De Bruijn, Nicolaas Govert** *A survey of the project AUTOMATH* (1980) To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism **pages** 579–606 **editors** Hindley, J.R.; Seldin, J.P. **publisher** Academic Press <https://research.tue.nl/files/2092478/597605.pdf>

- [Dow01] **Dowek, Giles** *Higher-order unification and matching* (2001) Handbook of automated reasoning **volume 2 pages** 1009--1063 **editors** Robinson, Alan; Voronkov, Andrei **publisher** Elsevier
- [EH97] **Elliott, Conal; Hudak, Paul** *Functional reactive animation* (1997) Proceedings of the second ACM SIGPLAN international conference on Functional programming **pages** 263--273 **editors** Peyton Jones, Simon L.; Tofte, Mads; Berman, A. Michael **doi** 10.1145/258948.258973 <https://doi.org/10.1145/258948.258973>
- [EUR+17] **Ebner, Gabriel; Ullrich, Sebastian; Roesch, Jared; Avigad, Jeremy; de Moura, Leonardo** *A metaprogramming framework for formal verification* (2017) Proceedings of the ACM on Programming Languages **volume 1 number** ICFP **pages** 1--29 **editor** Wadler, Philip **publisher** ACM New York, NY, USA **doi** 10.1145/3110278 <https://doi.org/10.1145/3110278>
- [Edg07] **Edgar, Gerald** *Measure, topology, and fractal geometry* (2007) **publisher** Springer
- [Ello1] **Elliott, Conal** *Functional Image Synthesis* (2001) Proceedings of Bridges <http://conal.net/papers/bridges2001/>
- [FGM+07] **Foster, J Nathan; Greenwald, Michael B; Moore, Jonathan T; Pierce, Benjamin C; Schmitt, Alan** *Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem* (2007) ACM Transactions on Programming Languages and Systems (TOPLAS) **volume 29 number 3 pages** 17--es **publisher** ACM New York, NY, USA <https://hal.inria.fr/inria-00484971/file/lenses-toplas-final.pdf>
- [FM87] **Felty, Amy; Miller, Dale** *Proof explanation and revision* (1987) **number** MS-CIS-88-17 **institution** University of Pennsylvania https://repository.upenn.edu/cgi/viewcontent.cgi?article=1660&context=cis_reports
- [Fle19] **Flet-Berliac, Yannis** *The Promise of Hierarchical Reinforcement Learning* (2019) The Gradient <https://thegradient.pub/the-promise-of-hierarchical-reinforcement-learning>
- [GAA+13] **Gonthier, Georges; Asperti, Andrea; Avigad, Jeremy; Bertot, Yves; Cohen, Cyril; Garillot, François; Le Roux, Stéphane; Mahboubi, Assia; O'Connor, Russell; Biha, Sidi Ould; others** *A machine-checked proof of the odd order theorem* (2013) International Conference on Interactive Theorem Proving **pages** 163--179 **organization** Springer <https://hal.inria.fr/docs/00/81/66/99/PDF/main.pdf>
- [GFA09] **Grossman, Tovi; Fitzmaurice, George W.; Attar, Ramtin** *A survey of software learnability: metrics, methodologies and guidelines* (2009) Proceedings of the 27th International Conference on Human Factors in Computing Systems **pages** 649--658 **editors** Olsen, Dan R. Jr.; Arthur, Richard B.; Hinckley, Ken; *et al.* **publisher** ACM **doi** 10.1145/1518701.1518803 <https://doi.org/10.1145/1518701.1518803>
- [GG17] **Ganesalingam, Mohan; Gowers, W. T.** *A fully automatic theorem prover with human-style output* (2017) Journal of Automated Reasoning **volume 58 number 2 pages** 253--291 **doi** 10.1007/s10817-016-9377-1 <https://doi.org/10.1007/s10817-016-9377-1>
- [GK18] **Gatt, Albert; Krahmer, Emiel** *Survey of the state of the art in natural language generation: Core tasks, applications and evaluation* (2018) Journal of Artificial Intelligence Research **volume 61 pages** 65--170 **doi** 10.1613/jair.5477 <https://doi.org/10.1613/jair.5477>
- [GKN15] **Grabowski, Adam; Kornilowicz, Artur; Naumowicz, Adam** *Four decades of Mizar* (2015) Journal of Automated Reasoning **volume 55 number 3 pages** 191--198 **editors** Trybulec, Andrzej; Trybulec Kuperberg, Krystyna **publisher** Springer **doi** 10.1007/s10817-015-9345-1 <https://doi.org/10.1007/s10817-015-9345-1>
- [GM05] **Grégoire, Benjamin; Mahboubi, Assia** *Proving equalities in a commutative ring done right in Coq* (2005) International Conference on Theorem Proving in Higher Order Logics **volume 3603 pages** 98--113 **editors** Hurd, Joe; Melham, Thomas F. **organization** Springer **doi** 10.1007/11541868_7 <http://cs.ru.nl/~freek/courses/tt-2014/read/10.1.1.61.3041.pdf>
- [GPJ17] **Galleo Arias, Emilio Jesús; Pin, Benoît; Jouvelot, Pierre** *jsCoq: Towards Hybrid Theorem Proving Interfaces* (2017) Proceedings of the 12th Workshop on User Interfaces for Theorem Provers **volume 239 pages** 15-27 **editors** Autexier, Serge; Quaresma, Pedro **publisher** Open Publishing Association **doi** 10.4204/EPTCS.239.2 <https://arxiv.org/pdf/1701.07125>
- [Gal16] **Galleo Arias, Emilio Jesús** *SerAPI: Machine-Friendly, Data-Centric Serialization for Coq* (2016) **institution** MINES ParisTech <https://hal-mines-paristech.archives-ouvertes.fr/hal-01384408/file/serapi.pdf>
- [Gan10] **Ganesalingam, Mohan** *The language of mathematics* (2010) **publisher** Springer **doi** 10.1007/978-3-642-37012-0 **isbn** 978-3-642-37011-3 <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.211.9027&rep=rep1&type=pdf>
- [Gon08] **Gonthier, Georges** *Formal proof--the four-color theorem* (2008) Notices of the AMS **volume 55 number 11 pages** 1382--1393 **editor** Magid, Andy
- [Gor00] **Gordon, Mike** *From LCF to HOL: a short history* (2000) Proof, language, and interaction **pages** 169--186 **editors** Plotkin, Gordon D.; Stirling, Colin; Tofte, Mads **doi** 10.1.1.132.8662 **isbn** 9780262161886 <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.132.8662&rep=rep1&type=pdf>

- [Gow10] **Gowers, W. T.** *Rough structure and classification* (2010) Visions in Mathematics **pages** 79--117 **publisher** Springer <https://www.dpmms.cam.ac.uk/~wtg10/gafavisions.ps>
- [Gre19] **Grebing, Sarah Caecilia** *User Interaction in Deductive Interactive Program Verification* (2019) <https://d-nb.info/198309989/34>
- [Gri75] **Grice, Herbert P** *Logic and conversation* (1975) Speech acts **pages** 41--58 **publisher** Brill http://rrt2.neostrada.pl/mioduszewska/course_265_reading%201b.pdf
- [HAB+17] **Hales, Thomas C; Adams, Mark; Bauer, Gertrud; Dang, Dat Tat; Harrison, John; Hoang, Truong Le; Kaliszyk, Cezary; Magron, Victor; McLaughlin, Sean; Nguyen, Thang Tat; Nguyen, Truong Quang; Nipkow, Tobias; Obua, Steven; Pleso, Joseph; Rute, Jason M.; Solovyev, Alexey; Ta, An Hoai Thi; Tran, Trung Nam; Trieu, Diep Thi; Urban, Josef; Vu, Ky Khac; Zumkeller, Roland** *A formal proof of the Kepler conjecture* (2017) Forum of Mathematics, Pi **volume** 5 **organization** Cambridge University Press **doi** doi:10.1017/fmp.2017.1 <https://doi.org/doi:10.1017/fmp.2017.1>
- [HBC99] **Holland-Minkley, Amanda M; Barzilay, Regina; Constable, Robert L** *Verbalization of High-Level Formal Proofs*. (1999) AAAI/LAAI **pages** 277--284 **editors** Hendler, Jim; Subramanian, Devika **publisher** AAAI Press / The MIT Press <http://www.aaai.org/Library/AAAI/1999/aaai99-041.php>
- [HF97] **Huang, Xiaorong; Fiedler, Armin** *Proof Verbalization as an Application of NLG* (1997) International Joint Conference on Artificial Intelligence **pages** 965--972 <http://ijcai.org/Proceedings/97-2/Papers/025.pdf>
- [HHPW96] **Hall, Cordelia V; Hammond, Kevin; Peyton Jones, Simon I; Wadler, Philip L** *Type classes in Haskell* (1996) ACM Transactions on Programming Languages and Systems (TOPLAS) **volume** 18 **number** 2 **pages** 109--138 **publisher** ACM New York, NY, USA **doi** 10.1145/227699.227700 <https://doi.org/10.1145/227699.227700>
- [HM19] **Hoek, Keeley; Morrison, Scott** *lean-rewrite-search* *GitHub repository* (2019) <https://github.com/semorrison/lean-rewrite-search>
- [HRW+21] **Han, Jesse Michael; Rute, Jason; Wu, Yuhuai; Ayers, Edward W; Polu, Stanislas** *Proof Artifact Co-training for Theorem Proving with Language Models* (2021) arXiv preprint arXiv:2102.06203 <https://arxiv.org/pdf/2102.06203>
- [Halo5] **Hales, Thomas C** *A proof of the Kepler conjecture* (2005) Annals of mathematics **pages** 1065--1185 **publisher** Mathematics Department, Princeton University <http://annals.math.princeton.edu/wp-content/uploads/annals-v162-n3-p01.pdf>
- [Halo7] **Hales, Thomas C** *The Jordan curve theorem, formally and informally* (2007) The American Mathematical Monthly **volume** 114 **number** 10 **pages** 882--894 **publisher** Taylor & Francis <https://www.maths.ed.ac.uk/~v1ranick/papers/hales3.pdf>
- [Har09] **Harrison, John** *HOL Light: An Overview*. (2009) TPHOLs **volume** 5674 **pages** 60--66 **editors** Berghofer, Stefan; Nipkow, Tobias; Urban, Christian; *et al.* **organization** Springer **doi** 10.1007/978-3-642-03359-9_4 https://doi.org/10.1007/978-3-642-03359-9_4
- [Hue97] **Huet, Gérard** *Functional Pearl: The Zipper* (1997) Journal of functional programming **volume** 7 **number** 5 **pages** 549--554 **publisher** Cambridge University Press <http://www.st.cs.uni-sb.de/edu/seminare/2005/advanced-fp/docs/huet-zipper.pdf>
- [Hur95] **Hurkens, Antonius J. C.** *A simplification of Girard's paradox* (1995) International Conference on Typed Lambda Calculi and Applications **pages** 266--278 **editors** Dezani-Ciancaglini, Mariangiola; Plotkin, Gordon D. **organization** Springer **doi** 10.1007/BFb0014058 <https://doi.org/10.1007/BFb0014058>
- [IA12] **Inglis, Matthew; Alcock, Lara** *Expert and novice approaches to reading mathematical proofs* (2012) Journal for Research in Mathematics Education **volume** 43 **number** 4 **pages** 358--390 **publisher** National Council of Teachers of Mathematics <https://pdfs.semanticscholar.org/494e/7981ee892d500139708e53901d6260bd83b1.pdf>
- [IJR99] **Ireland, Andrew; Jackson, Michael; Reid, Gordon** *Interactive proof critics* (1999) Formal Aspects of Computing **volume** 11 **number** 3 **pages** 302--325 **publisher** Springer **doi** 10.1007/s001650050052 <https://doi.org/10.1007/s001650050052>
- [Ire92] **Ireland, Andrew** *The use of planning critics in mechanizing inductive proofs* (1992) International Conference on Logic for Programming Artificial Intelligence and Reasoning **pages** 178--189 **editor** Voronkov, Andrei **organization** Springer **doi** 10.1007/BFb0013060 <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.40.3546&rep=rep1&type=pdf>
- [JR12] **Jaskelioff, Mauro; Rypacek, Ondrej** *An Investigation of the Laws of Traversals* (2012) Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia **volume** 76 **pages** 40--49 **editors** Chapman, James; Levy, Paul Blain **doi** 10.4204/EPTCS.76.5 <https://doi.org/10.4204/EPTCS.76.5>
- [Jam01] **Jamnik, Mateja** *Mathematical Reasoning with Diagrams: From Intuition to Automation* (2001) **publisher** CSLI Press **isbn** 9781575863238

- [KB70] **Knuth, Donald E; Bendix, Peter B** *Simple word problems in universal algebras* (1970) Computational Problems in Abstract Algebra **pages** 263-297 **editor** Leech, John **publisher** Pergamon **doi** <https://doi.org/10.1016/B978-0-08-012975-4.50028-X> **isbn** 978-0-08-012975-4 <https://www.cs.tufts.edu/~nr/cs257/archive/don-knuth/knuth-bendix.pdf>
- [KEH+09] **Klein, Gerwin; Elphinstone, Kevin; Heiser, Gernot; Andronick, June; Cock, David; Derrin, Philip; Elkaduwe, Dhammika; Engelhardt, Kai; Kolanski, Rafal; Norrish, Michael; Sewell, Thomas; Tuch, Harvey; Winwood, Simon** *seL4: Formal verification of an OS kernel* (2009) Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles **pages** 207--220 **editors** Matthews, Jeanna Neefe; Anderson, Thomas E. **organization** ACM **doi** 10.1145/1629575.1629596 <https://doi.org/10.1145/1629575.1629596>
- [KKY95] **Kambhampati, Subbarao; Knoblock, Craig A; Yang, Qiang** *Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning* (1995) Artificial Intelligence **volume** 76 **number** 1 **pages** 167--238 **doi** 10.1016/0004-3702(94)00076-D [https://doi.org/10.1016/0004-3702\(94\)00076-D](https://doi.org/10.1016/0004-3702(94)00076-D)
- [KMM13] **Kaufmann, Matt; Manolios, Panagiotis; Moore, J Strother** *Computer-aided reasoning: ACL2 case studies* (2013) **volume** 4 **publisher** Springer
- [KSFS05] **Kiselyov, Oleg; Shan, Chung-chieh; Friedman, Daniel P; Sabry, Amr** *Backtracking, interleaving, and terminating monad transformers: (functional pearl)* (2005) ACM SIGPLAN Notices **volume** 40 **number** 9 **pages** 192--203 **editors** Danvy, Olivier; Pierce, Benjamin C. **publisher** ACM New York, NY, USA **doi** 10.1145/1086365.1086390 <https://doi.org/10.1145/1086365.1086390>
- [Knu86] **Knuth, Donald E.** *The TeXbook* (1986) **publisher** Addison-Wesley **isbn** 0-201-13447-0
- [Kuh14] **Kuhn, Tobias** *A survey and classification of controlled natural languages* (2014) Computational linguistics **volume** 40 **number** 1 **pages** 121--170 **publisher** MIT Press **doi** 10.1162/COLI_a_00168 https://doi.org/10.1162/COLI_a_00168
- [LC20] **Lample, Guillaume; Charton, François** *Deep Learning For Symbolic Mathematics* (2020) ICLR **publisher** OpenReview.net <https://openreview.net/forum?id=S1eZYeHFDS>
- [LCT08] **Langley, Pat; Choi, Dongkyu; Trivedi, Nishant** *Icarus user's manual* (2008) **institution** Institute for the Study of Learning and Expertise <http://www.isle.org/~langley/papers/manual.pdf>
- [LD97] **Lowe, Helen; Duncan, David** *XBarnacle: Making Theorem Provers More Accessible* (1997) 14th International Conference on Automated Deduction **volume** 1249 **pages** 404--407 **editor** McCune, William **publisher** Springer **doi** 10.1007/3-540-63104-6_39 https://doi.org/10.1007/3-540-63104-6_39
- [LP03] **Lämmel, Ralf; Peyton Jones, Simon** *Scrap Your Boilerplate* (2003) Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003, Proceedings **volume** 2895 **pages** 357 **editor** Ogori, Atsushi **publisher** Springer **doi** 10.1007/978-3-540-40018-9_23 https://doi.org/10.1007/978-3-540-40018-9_23
- [LR13] **Lüth, Christoph; Ring, Martin** *A web interface for Isabelle: The next generation* (2013) International Conference on Intelligent Computer Mathematics **pages** 326--329 **organization** Springer <http://www.informatik.uni-bremen.de/~clueth/papers/cicm2013.pdf>
- [LYWP21] **Li, Wenda; Yu, Lei; Wu, Yuhuai; Paulson, Lawrence C.** *IsarStep: a Benchmark for High-level Mathematical Reasoning* (2021) International Conference on Learning Representations <https://openreview.net/forum?id=Pzj6fzU6wkj>
- [Lew17] **Lewis, Robert Y.** *An Extensible Ad Hoc Interface between Lean and Mathematica* (2017) Proceedings of the Fifth Workshop on Proof eXchange for Theorem Proving, PxTP 2017, Brasília, Brazil, 23-24 September 2017 **volume** 262 **pages** 23--37 **editors** Dubois, Catherine; Paleo, Bruno Woltzenlogel **doi** 10.4204/EPTCS.262.4 <https://doi.org/10.4204/EPTCS.262.4>
- [Low97] **Lowe, Helen** *Evaluation of a Semi-Automated Theorem Prover (Part I)* (1997) <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.48.2318&rep=rep1&type=pdf>
- [MAKR15] **de Moura, Leonardo; Avigad, Jeremy; Kong, Soonho; Roux, Cody** *Elaboration in Dependent Type Theory* (2015) CoRR **volume** abs/1505.04324 <http://arxiv.org/abs/1505.04324>
- [MBO8] **de Moura, Leonardo; Björner, Nikolaj** *Z3: An efficient SMT solver* (2008) International conference on Tools and Algorithms for the Construction and Analysis of Systems **pages** 337--340 **editors** Ramakrishnan, C. R.; Rehof, Jakob **organization** Springer **doi** 10.1007/978-3-540-78800-3_24 https://doi.org/10.1007/978-3-540-78800-3_24
- [MH93] **Monk, Andrew; Haber, Jeanne** *Improving your human-computer interface: a practical technique* (1993) **publisher** Prentice Hall **isbn** 9780130100344 <https://books.google.co.uk/books?id=JN9QAAAAAAAJ>
- [MKA+15] **de Moura, Leonardo; Kong, Soonho; Avigad, Jeremy; Van Doorn, Floris; von Raumer, Jakob** *The Lean theorem prover (system description)* (2015) International Conference on Automated Deduction **volume** 9195 **pages** 378--388 **editors** Felty, Amy P.; Middeldorp, Aart **organization** Springer **doi** 10.1007/978-3-319-21401-6_26 https://kilthub.cmu.edu/articles/The_Lean_Theorem_Prover_system_description_/6492815/files/11937416.pdf

- [MP08] **McBride, Conor; Paterson, Ross** *Applicative programming with effects* (2008) J. Funct. Program. **volume** 18 **number** 1 **pages** 1--13 **doi** 10.1017/S0956796807006326 <https://personal.cis.strath.ac.uk/conor.mcbride/IdiomLite.pdf>
- [MS99] **Melis, Erica; Siekmann, Jörg** *Knowledge-based proof planning* (1999) Artificial Intelligence **volume** 115 **number** 1 **pages** 65--105 **editor** Bobrow, Daniel G. **publisher** Elsevier **doi** 10.1016/S0004-3702(99)00076-4 [https://doi.org/10.1016/S0004-3702\(99\)00076-4](https://doi.org/10.1016/S0004-3702(99)00076-4)
- [MUP79] **de Millo, Richard A; Upton, Richard J; Perlis, Alan J** *Social processes and proofs of theorems and programs* (1979) Communications of the ACM **volume** 22 **number** 5 **pages** 271--280 **doi** 10.1145/359104.359106 <https://doi.org/10.1145/359104.359106>
- [Mar84] **Martin-Löf, Per** *Intuitionistic type theory* (1984) **volume** 1 **publisher** Bibliopolis **isbn** 978-88-7088-228-5 <http://people.csail.mit.edu/jgross/personal-website/papers/academic-papers-local/Martin-Lof80.pdf>
- [McBoo] **McBride, Conor** *Dependently typed functional programs and their proofs* (2000) <http://hdl.handle.net/1842/374>
- [McC60] **McCarthy, John** *Recursive functions of symbolic expressions and their computation by machine, Part I* (1960) Communications of the ACM **volume** 3 **number** 4 **pages** 184--195 **publisher** ACM New York, NY, USA **doi** 10.1145/367177.367199 <https://doi.org/10.1145/367177.367199>
- [Mic78] **Michener, Edwina Rissland** *Understanding understanding mathematics* (1978) Cognitive science **volume** 2 **number** 4 **pages** 361--383 **publisher** Wiley Online Library **doi** https://doi.org/10.1207/s15516709cog0204_3 https://online.library.wiley.com/doi/pdf/10.1207/s15516709cog0204_3
- [Mil72] **Milner, Robin** *Logic for computable functions description of a machine implementation* (1972) **institution** Stanford University <https://apps.dtic.mil/dtic/tr/fulltext/u2/785072.pdf>
- [Nev74] **Nevins, Arthur J** *A human oriented logic for automatic theorem-proving* (1974) Journal of the ACM **volume** 21 **number** 4 **pages** 606--621 **publisher** ACM New York, NY, USA **doi** 10.1145/321850.321858 <https://doi.org/10.1145/321850.321858>
- [Noro8] **Norell, Ulf** *Dependently typed programming in Agda* (2008) International school on advanced functional programming **volume** 5832 **pages** 230--266 **editors** Koopman, Pieter W. M.; Plasmeijer, Rinus; Swierstra, S. Doaitse **organization** Springer **doi** 10.1007/978-3-642-04652-0_5 https://doi.org/10.1007/978-3-642-04652-0_5
- [PLB+17] **Pease, Alison; Lawrence, John; Budzynska, Katarzyna; Corneli, Joseph; Reed, Chris** *Lakatos-style collaborative mathematics through dialectical, structured and abstract argumentation* (2017) Artificial Intelligence **volume** 246 **pages** 181--219 **publisher** Elsevier **doi** 10.1016/j.artint.2017.02.006 <https://www.sciencedirect.com/science/article/pii/S004370217300267>
- [PP89] **Pfenning, Frank; Paulin-Mohring, Christine** *Inductively defined types in the Calculus of Constructions* (1989) International Conference on Mathematical Foundations of Programming Semantics **pages** 209--228 **organization** Springer <https://kilthub.cmu.edu/ndownloader/files/12096983>
- [Paso7] **Paskevich, Andrei** *The syntax and semantics of the ForTheL language* (2007) <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.211.8865&rep=rep1&type=pdf> English translation of a portion of Paskevich's PhD thesis
- [Pau89] **Paulson, Lawrence C** *The foundation of a generic theorem prover* (1989) Journal of Automated Reasoning **volume** 5 **number** 3 **pages** 363--397 **doi** 10.1007/BF00248324 <https://doi.org/10.1007/BF00248324>
- [Pau98] **Paulson, Lawrence C** *The inductive approach to verifying cryptographic protocols* (1998) Journal of Computer Security **volume** 6 **number** 1-2 **pages** 85--128 <http://content.iospress.com/articles/journal-of-computer-security/jcs102>
- [Pau99] **Paulson, Lawrence C** *A generic tableau prover and its integration with Isabelle* (1999) Journal of Universal Computer Science **volume** 5 **number** 3 **pages** 73--87 **doi** 10.3217/jucs-005-03-0073 <https://doi.org/10.3217/jucs-005-03-0073>
- [Pit20] **Pit-Claudel, Clément** *Untangling mechanized proofs* (2020) SLE 2020: Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering **pages** 155--174 **doi** 10.1145/3426425.3426940 <https://doi.org/10.1145/3426425.3426940>
- [Poi14] **Poincaré, Henri** *Science and method* (1914) **translator** Halsted, George Bruce **publisher** Amazon (out of copyright) **isbn** 978-1534945906 <https://archive.org/details/sciencemethod00poinuoft> The version at URL is translated by Francis Maitland
- [Pro13] **The Univalent Foundations Program** *Homotopy Type Theory: Univalent Foundations of Mathematics* (2013) **publisher** Institute for Advanced Study <https://homotopytypetheory.org/book/>
- [Pól62] **Pólya, George** *Mathematical Discovery* (1962) **publisher** John Wiley & Sons <https://archive.org/details/GeorgePolyaMathematicalDiscovery>

- [RD00] **Reiter, Ehud; Dale, Robert** *Building natural language generation systems* (2000) **publisher** Cambridge University Press <https://dl.acm.org/doi/book/10.5555/331955>
- [RN10] **Russell, Stuart J.; Norvig, Peter** *Artificial Intelligence - A Modern Approach* (2010) **publisher** Pearson Education **isbn** 978-0-13-207148-2 <http://aima.cs.berkeley.edu/>
- [RSS+20] **Raggi, Daniel; Stapleton, Gem; Stockdill, Aaron; Jamnik, Mateja; Garcia, Grecia Garcia; Cheng, Peter C.-H.** *How to (Re)represent it?* (2020) 32nd IEEE International Conference on Tools with Artificial Intelligence **pages** 1224-1232 **publisher** IEEE **doi** 10.1109/ICTAI50040.2020.00185 <https://doi.org/10.1109/ICTAI50040.2020.00185>
- [RV02] **Riazanov, Alexandre; Voronkov, Andrei** *The design and implementation of VAMPIRE* (2002) AI communications **volume** 15 **number** 2-3 **pages** 91--110 <http://content.iospress.com/articles/ai-communications/aic259>
- [Ran94] **Ranta, Aarne** *Syntactic categories in the language of mathematics* (1994) International Workshop on Types for Proofs and Programs **pages** 162--182 **editors** Dybjer, Peter; Nordström, Bengt; Smith, Jan M. **organization** Springer **doi** 10.1007/3-540-60579-7_9 https://doi.org/10.1007/3-540-60579-7_9
- [Ran95] **Ranta, Aarne** *Context-relative syntactic categories and the formalization of mathematical text* (1995) International Workshop on Types for Proofs and Programs **pages** 231--248 **editors** Berardi, Stefano; Coppo, Mario **organization** Springer **doi** 10.1007/3-540-61780-9_73 <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.7.8663&rep=rep1&type=pdf>
- [Ros53] **Rosser, J. Barkley** *Logic for Mathematicians* (1953) **publisher** McGraw-Hill **isbn** 978-0-486-46898-3
- [SB01] **Snyder, Wayne; Baader, Franz** *Unification theory* (2001) Handbook of automated reasoning **volume** 1 **pages** 447-533 **editors** Robinson, Alan; Voronkov, Andrei **publisher** Elsevier
- [SB18a] **Steen, Alexander; Benz Müller, Christoph** *The higher-order prover Leo-III* (2018) International Joint Conference on Automated Reasoning **volume** 10900 **pages** 108--116 **editors** Galmiche, Didier; Schulz, Stephan; Sebastiani, Roberto **publisher** Springer **doi** 10.1007/978-3-319-94205-6_8 https://doi.org/10.1007/978-3-319-94205-6_8
- [SB18b] **Sutton, Richard S; Barto, Andrew G** *Reinforcement learning: An introduction* (2018) **publisher** MIT press <http://incompleteideas.net/book/the-book-2nd.html>
- [SBRT18] **Stathopoulos, Yiannos; Baker, Simon; Rei, Marek; Teufel, Simone** *Variable Typing: Assigning Meaning to Variables in Mathematical Text* (2018) NAACL-HLT 2018 **pages** 303--312 **editors** Walker, Marilyn A.; Ji, Heng; Stent, Amanda **publisher** Association for Computational Linguistics **doi** 10.18653/v1/n18-1028 <https://doi.org/10.18653/v1/n18-1028>
- [SCO95] **Stenning, Keith; Cox, Richard; Oberlander, Jon** *Contrasting the cognitive effects of graphical and sentential logic teaching: reasoning, representation and individual differences* (1995) Language and Cognitive Processes **volume** 10 **number** 3-4 **pages** 333--354 **publisher** Taylor & Francis <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.3906&ep=rep1&type=pdf>
- [SCV19] **Schulz, Stephan; Cruanes, Simon; Vukmirović, Petar** *Faster, Higher, Stronger: E 2.3* (2019) Proc. of the 27th CADE, Natal, Brasil **number** 11716 **pages** 495--507 **editor** Fontaine, Pascal **publisher** Springer <http://www.lehre.dhbw-stuttgart.de/~ssschulz/bibliography.html#SCV:CADE-2019>
- [SH17] **Sterling, Jonathan; Harper, Robert** *Algebraic Foundations of Proof Refinement* (2017) CoRR **volume** abs/1703.05215 <http://arxiv.org/abs/1703.05215>
- [SHB+99] **Siekmann, Jörg; Hess, Stephan; Benz Müller, Christoph; Cheikhrouhou, Lassaad; Fiedler, Armin; Horacek, Helmut; Kohlhase, Michael; Konrad, Karsten; Meier, Andreas; Melis, Erica; Pollet, Martin; Sorge, Volker** *LOUI: Lovely OMEGA user interface* (1999) Formal Aspects of Computing **volume** 11 **number** 3 **pages** 326-342 **editor** Woodcock, James **publisher** Springer **doi** 10.1007/s001650050053 <https://doi.org/10.1007/s001650050053>
- [SORS01] **Shankar, Natarajan; Owre, Sam; Rushby, John M; Stringer-Calvert, Dave WJ** *PVS prover guide* (2001) Computer Science Laboratory, SRI International, Menlo Park, CA <https://pvs.csl.sri.com/doc/pvs-prover-guide.pdf>
- [SP82] **Smyth, Michael B; Plotkin, Gordon D** *The category-theoretic solution of recursive domain equations* (1982) SIAM Journal on Computing **volume** 11 **number** 4 **pages** 761--783 **publisher** SIAM http://wrap.warwick.ac.uk/46312/1/WRAP_Smyth_cs-rr-014.pdf
- [SRV01] **Sekar, R; Ramakrishnan, I.V.; Voronkov, Andrei** *Term Indexing* (2001) Handbook of automated reasoning **volume** 2 **pages** 1855--1900 **editors** Robinson, Alan; Voronkov, Andrei **publisher** Elsevier
- [ST16] **Stathopoulos, Yiannos A; Teufel, Simone** *Mathematical information retrieval based on type embeddings and query expansion* (2016) COLING 2016 **pages** 2344--2355 **editors** Calzolari, Nicoletta; Matsumoto, Yuji; Prasad, Rashmi **publisher** ACL <https://www.aclweb.org/anthology/C16-1221/>
- [Sac74] **Sacerdoti, Earl D** *Planning in a hierarchy of abstraction spaces* (1974) Artificial intelligence **volume** 5 **number** 2 **pages** 115--135 **publisher** Elsevier **doi** 10.1016/0004-3702(74)90026-5 [https://doi.org/10.1016/0004-3702\(74\)90026-5](https://doi.org/10.1016/0004-3702(74)90026-5)

- [Sch12] **Schreier, Margrit** *Qualitative content analysis in practice* (2012) **publisher** SAGE Publications **isbn** 9781849205931 <https://uk.sagepub.com/en-gb/eur/qualitative-content-analysis-in-practice/book234633>
- [Sie90] **Sierpinska, Anna** *Some remarks on understanding in mathematics* (1990) For the learning of mathematics **volume** 10 **number** 3 **pages** 24--41 **publisher** FLM Publishing Association <https://www.flm-journal.org/Articles/43489F40454C8B2E06F334CC13CCA8.pdf>
- [Sie94] **Sierpinska, Anna** *Understanding in mathematics* (1994) **volume** 2 **publisher** Psychology Press **isbn** 9780750705684
- [Spi11] **Spivack, Arnaud** *Verified computing in homological algebra, a journey exploring the power and limits of dependent type theory* (2011) <https://pastel.archives-ouvertes.fr/pastel-00605836/document>
- [Spi87] **Spinoza, Benedict** *The chief works of Benedict de Spinoza* (1887) **translator** Elwes, R.H.M. **publisher** Chiswick Press <https://books.google.co.uk/books?id=tn109KVEd2UC&ots=WiBRHdjSjY&dq=the%20philosophy%20of%20benedict%20spinoza&lr&pg=PA3#v=onepage&q&f=false>
- [Ste17] **Steele Jr., Guy L.** *It's Time for a New Old Language* (2017) <http://2017.clojure-conj.org/guy-steele/> Invited talk at Clojure/Conj 2017. Slides: <http://groups.csail.mit.edu/mac/users/gjs/6.945/readings/Steele-MIT-April-2017.pdf>
- [Tat77] **Tate, Austin** *Generating project networks* (1977) Proceedings of the 5th International Joint Conference on Artificial Intelligence. **pages** 888--893 **editor** Reddy, Raj **organization** Morgan Kaufmann Publishers Inc. **doi** 10.5555/1622943.1623021 <https://dl.acm.org/doi/abs/10.5555/1622943.1623021>
- [Tho92] **Thomassen, Carsten** *The Jordan-Schönflies theorem and the classification of surfaces* (1992) The American Mathematical Monthly **volume** 99 **number** 2 **pages** 116--130 **publisher** Taylor & Francis <https://www.jstor.org/stable/2324180>
- [UM20] **Ullrich, Sebastian; de Moura, Leonardo** *Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages* (2020) Automated Reasoning **pages** 167--182 **editors** Peltier, Nicolas; Sofronie-Stokkermans, Viorica **publisher** Springer International Publishing **doi** 10.1007/978-3-030-51054-1_10 https://doi.org/10.1007/978-3-030-51054-1_10
- [VKB18] **Vicary, Jamie; Kissinger, Aleks; Bar, Krzysztof** *Globular: an online proof assistant for higher-dimensional rewriting* (2018) Logical Methods in Computer Science **volume** 14 **publisher** Episciences.org <https://core.ac.uk/download/pdf/79162392.pdf> project website: <http://ncatlab.org/nlab/show/Globular>
- [VLP07] **Verchinine, Konstantin; Lyaletski, Alexander; Paskevich, Andrei** *System for Automated Deduction (SAD): a tool for proof verification* (2007) International Conference on Automated Deduction **pages** 398--403 **editor** Pfenning, Frank **organization** Springer **doi** 10.1007/978-3-540-73595-3_29 https://doi.org/10.1007/978-3-540-73595-3_29
- [VLPA08] **Verchinine, Konstantin; Lyaletski, Alexander; Paskevich, Andrei; Anisimov, Anatoly** *On correctness of mathematical texts from a logical and practical point of view* (2008) International Conference on Intelligent Computer Mathematics **pages** 583--598 **editors** Autexier, Serge; Campbell, John A.; Rubio, Julio; *et al.* **organization** Springer **doi** 10.1007/978-3-540-85110-3_47 https://doi.org/10.1007/978-3-540-85110-3_47
- [VSP+17] **Vaswani, Ashish; Shazeer, Noam; Parmar, Niki; Uszkoreit, Jakob; Jones, Llion; Gomez, Aidan N.; Kaiser, Lukasz; Polosukhin, Illia** *Attention is All you Need* (2017) Neural Information Processing Systems **pages** 5998--6008 **editors** Guyon, Isabelle; von Luxburg, Ulrike; Bengio, Samy; *et al.* <https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html>
- [Wado03] **Wadler, Philip** *A prettier printer* (2003) The Fun of Programming, Cornerstones of Computing **pages** 223--243 **editors** Gibbons, Jeremy; de Moor, Oege **publisher** Palgrave MacMillan <http://www.cs.ox.ac.uk/publications/books/fop>
- [Wen12] **Wenzel, Makarius** *Isabelle/jEdit-A Prover IDE within the PIDE Framework*. (2012) Intelligent Computer Mathematics - 11th International Conference **volume** 7362 **pages** 468--471 **editors** Jeuring, Johan; Campbell, John A.; Carette, Jacques; *et al.* **publisher** Springer **doi** 10.1007/978-3-642-31374-5_38 https://doi.org/10.1007/978-3-642-31374-5_38
- [Wen18] **Wenzel, Makarius** *Isabelle/PIDE after 10 years of development* (2018) UITP workshop: User Interfaces for Theorem Provers. <https://sketis.net/wp-content/uploads/2018/08/isabellepide-uitp2018.pdf>
- [Wen99] **Wenzel, Markus** *Isar - A Generic Interpretative Approach to Readable Formal Proof Documents* (1999) Theorem Proving in Higher Order Logics **volume** 1690 **pages** 167--184 **editors** Bertot, Yves; Dowek, Gilles; Hirschowitz, André; *et al.* **publisher** Springer **doi** 10.1007/3-540-48256-3_12 https://doi.org/10.1007/3-540-48256-3_12
- [Wie00] **Wiedijk, Freek** *The de Bruijn Factor* (2000) <http://www.cs.ru.nl/F.Wiedijk/factor/factor.pdf>
- [Wie07] **Wiedijk, Freek** *The QED manifesto revisited* (2007) Studies in Logic, Grammar and Rhetoric **volume** 10 **number** 23 **pages** 121--133 <http://www.cs.ru.nl/~freek/pubs/qed2.pdf>
- [Zha16] **Zhan, Bohua** *AUTO2, a saturation-based heuristic prover for higher-order logic* (2016) International Conference on Interactive Theorem Proving **pages** 441--456 **editors** Blanchette, Jasmin Christian; Merz, Stephan **organization** Springer **doi**

10.1007/978-3-319-43144-4_27 https://doi.org/10.1007/978-3-319-43144-4_27

[deB72] **de Bruijn, Nicolaas Govert** *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem* (1972) *Indagationes Mathematicae (Proceedings)* **volume** 75 **number** 5 **pages** 381--392 **organization** North-Holland <http://alexandria.tue.nl/repository/freearticles/597619.pdf>