User interfaces for computer-assisted mathematics

Wojciech Nawrocki

Thesis Committee:
Jeremy Avigad (Chair)
Marijn J. H. Heule

A dissertation submitted in fulfillment of the requirements for the degree of Master of Science in Logic, Computation and Methodology.

Department of Philosophy
Carnegie Mellon University
Pittsburgh, PA 15213
USA

Email: wjnawrocki@cmu.edu

October 4, 2023

Contents

1 Introduction

The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called *procedural epistemology*—the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects.

Harold Abelson and Gerald J. Sussman
with Julie Sussman
Structure and Interpretation of Computer Programs

This thesis is about rich, interactive computing environments. More specifically, it is about user interfaces (UIs) for making mathematical constructions, reasoning about them, and dynamically interacting with their representations. More specifically still, it contributes a set of UI features that we (see section 1.1 for who "we" refers to) designed and implemented for Lean 4 [71], an interactive theorem prover (ITP) and programming language, in hope of inching the language's tooling just a tiny bit closer towards providing such a rich environment.

An ITP broadly construed is a software system that works together with a human user (or, increasingly often, an AI agent) to construct mathematical objects. The name "theorem prover" suggests that the objects being built are generally proofs of theorems. While this is often true, "theorem prover" is also an overly specific misnomer – essentially every theorem prover out there allows first introducing *objects* about which we are to prove something. Objects may include numbers, lists, and other kinds of inductive constructions, as well as functions, quotients, and compound structures built out of those. After proving some facts about our objects, we are then able to use those facts in constructing further objects. It is a matter of taste whether theorems and proofs, rather than the objects about which theorems are proven, deserve their privileged position in the name. More accurately, then, an ITP is an interactive "construction yard" for mathematics; but I will continue to use the standard terminology.

The motivation for making this yard more alive and dynamic comes from the playful, experimental spirit of computing and mathematics exemplified in the procedural epistemology of SICP [1]. We want to interact in some way with the entities that we consider. For the simplest of objects this is achievable physically and in a hands-on manner, for example by moving the pebbles on an abacus to do basic arithmetic, or by manipulating a physically realized polyhedron. As abstractions grow in complexity, the concrete approach breaks down and we begin to study symbolic representations. Nevertheless the desire to manipulate and view them from multiple perspectives remains, and we find that we can do this through *computation*.

The most basic form of computation is *evaluation*, roughly the capacity to execute code written in our language of expressions. This includes all kinds of purely algorithmic calculations from the simple task of evaluating $1 + 1$ into 2, to more complex operations such as approximating real numbers by calculating rational sequences to finite precision. Beyond evaluation we have *normalization* which serves to deterministically simplify expressions (possibly involving free variables) into canonical forms, and can subsume evaluation. But the notion of computation I have in mind here is broader than just evaluation or normalization. The issue is that evaluation requires access to executable code realizing the operations of interest. For instance, evaluating $1+1$ needs a program implementing addition. This becomes an obstacle in the presence of non-constructive axioms[1], where structure-driven extraction of executable programs corresponding to proofs in the Brouwer-Heyting-Kolmogorov sense may be unavailable. Nevertheless, even in the presence of the axiom of choice we are able to "compute" facts about objects. For instance, in homotopy theory one may speak of computing homotopy groups even though in general this may require great ingenuity, and an algorithm for doing so is not known [87]. The fundamental difference is that in this case, instead of taking program steps until the process terminates and a canonical form is produced, we are rather applying rewrite rules taken from a potentially non-terminating set. Rewriting has the effect of replacing equals for equals in a compound expression until it looks like what we want.

Whatever flavour they might have, computational processes ultimately allow us to extract useful information from our objects of interest, and to transform them into forms that are in some respect more illuminating than the original. In 1962, Engelbart dared to imagine the workspace of a computer-augmented architect [40]. We can similarly imagine the computer mathematician of the future. Just as the architect uses CAD software to request various *projections* of the model of a building—a view of the outside, a plan of the first floor, schematics of the electrical installation—the mathematician wishes at various points to search for known algebraic invariants of a named topological space, to evaluate a computable function on concrete data, or to display a diagrammatic representation of a theorem statement.

In some ways this future is now. With fast symbolic computation and access to internet data, projections can in many cases be computed quite quickly. The internet provides access to a great variety of computational tooling, databases of invariants such as the LMFDB [55] are available, and powerful automation for ITP systems has been developed. In other ways we are not quite there. Some interfaces can be clumsy and using them not worth the effort. A major motivation of our work is to provide direct, intuitive access to computational mathematics functionality from within an ITP.

Overview. This thesis is about user interfaces for ITP systems. Theorem provers vary in many parameters. One parameter is their approach to *proof refinement*, that is the process of incrementally constructing proofs by filling in successive detail. In section 2, I explain what this means and present some of the options chosen by various systems. This is important to understand because the choice of incremental construction strategy strongly influences the baseline mode of interaction and the user interface. Approaches to ITP interfaces informed in part by their chosen style of refinement are then surveyed in section 3. In section 4, we will discuss the approach to *metaprogramming* taken in Lean 4. Finally in section 5, we introduce the new user interface and `ProofWidgets` support library that we developed for Lean 4. Ultimately, we will see that prover UI design involves thinking about an exciting mix of issues in mathematical logic, type theory, software engineering, and human-computer interaction.

1.1 Attribution and acknowledgements

Sections 1 to 4 have not appeared elsewhere. Section 5 is a lightly edited copy of "An extensible user interface for Lean 4" [75] written jointly with Edward W. Ayers and Gabriel Ebner.

[1]My use of "construction" is intended more generally than how it is used in constructive mathematics.

I wish to thank the Lean team at MSR and KIT: Leonardo de Moura and Sebastian Ullrich for extensive discussions, code review, and improvements to the system, Daniel Selsam for suggesting traces and developing the analyzing delaborator, Daniel Fabian for input on RPC design. The Penrose team: Wode Ni and Sam Estep for considerable help and implementing several features which made our use possible. Jeremy Avigad and Patrick Massot for suggestions, advice, and feedback on a draft of section 5. Tomáš Skřivan, Joachim Breitner, and Sina Hazratpour for trying our systems and suggesting improvements. Chris Lovett and Mariana Alanis for working on `vscode-lean4`. Fernando Larrain and Mario Carneiro for discussions on the implementation of proof assistants. Nikhil Swamy for corrections on the interface of F*. The Lean Zulip community for technical help and ideas. The workers of Taste of India at the Carnegie Mellon campus in Pittsburgh, PA, USA for preparing numerous nutritious meals. Melinda for sticking around. Last but not least, Jeremy Avigad and Marijn Heule for their encouragement, invaluable advice and comments on this thesis.

Lean 4, its user interface, and the programming environment around them are ongoing open-source efforts with many contributors of whom I am only one. Pull requests containing my more relevant contributions specifically are linked below. Any other feature should be considered the work of others.

- leanprover/lean4 #161: Language Server Protocol server MVP
- leanprover/vscode-lean4 #30: LSP-based infoview
- leanprover/lean4 #568: Infoview RPC
- leanprover/lean4 #596: Interactive goals & diagnostics
- leanprover/vscode-lean4 #37: Info-on-hover widgets
- leanprover/vscode-lean4 #151: Move infoview to JavaScript modules
- leanprover/lean4 #1225: RFC: User widgets
- leanprover/vscode-lean4 #193: refactor: preparing for user-widgets and context suggestions
- leanprover/vscode-lean4 #275: Multi-expression selection
- leanprover-community/mathlib4 #363: feat: add commutative diagram widget

2 How formal objects are constructed

Birds fly high in the air and survey broad vistas of mathematics out to the far horizon. They delight in concepts that unify our thinking and bring together diverse problems from different parts of the landscape. Frogs live in the mud below and see only the flowers that grow nearby. They delight in the details of particular objects, and they solve problems one at a time.

Freeman Dyson
Birds and Frogs

It can be surprisingly difficult to formulate precisely what the point of mathematics is. As examined by Thurston in his classic essay "On Proof and Progress in Mathematics" [98], it has the curious recursive quality of serving primarily to advance human understanding of mathematics. In its "informal" variation, that is mathematics conveyed in natural language via sound waves and on paper, blackboard, or PDF, the information conveyed serves to communicate ideas. This may include defining new objects, giving examples, making conjectures, stating and proving theorems. All these are of comparable importance; theorems are often the goal, but the choice of definitions can make or break a theory, and interesting examples can motivate the formation of new fields.

Mathematical communication, traditionally following the definition-theorem-proof schema, can at first appear quite rigid when juxtaposed with, say, a free-flowing poem. In reality, within these general bounds a great deal of variation is allowed. The apparent rigidity is an illusion stemming from the use of an artificial subset of natural language ("Let X be a widget. We will show that X has a property. By Theorem 1 it suffices to consider a subwidget $Y \hookrightarrow X$ such that.."). In reality, there is no prescribed script that every proof must follow. It can proceed in a *forward* fashion, first appearing to tell an unrelated story by making and proving a number of assertions, and eventually demonstrating that the result follows clearly from the assertions. Alternatively it can work *backwards* from the statement to be proven, decomposing it into smaller pieces until nothing remains. Forwards and backwards reasoning can be intermixed in all sorts of combinations. The level of detail and rigour is similarly up for negotiation, depending on factors such as the expected knowledge and skills of the audience, the novelty of the proof method, and the writer's mood on a given day. The work expected of a reader wanting to understand a piece of mathematics ranges from only having to follow small and digestible steps, to having to reprove an entire statement left as an exercise.

Attempts to crystallize this variety into a precise, *formal* definition of proofs[2], which could then itself be mathematically studied, began towards the end of the 19th century. The work of many logicians eventually led to the establishment of foundational formal systems including logics, set theories, and type theories. Besides enabling meta-mathematics—the mathematical study of mathematical tools and processes—formal representations have the useful property of being easier to process algorithmically than natural language, making the mechanization of mathematics possible.

[2]I will sometimes use just the word "proof" to refer to formal proofs; this is meant as a shorthand rather than a philosophical commitment.

Proof trees. A complete survey of all of these representations would take us too far afield. It suffices to observe that essentially all (this is to say, all the ones that I am aware of) of them view proofs as trees, or more generally as directed acyclic graphs. These structures make explicit the implicit dependencies between assertions ("uniqueness of widget Z now follows from the fact that it has property (2)") present in textual proofs.

Recall that first-order logic comes equipped with a language of *propositions* or *formulas* that includes conjunctions $\varphi \wedge \psi$, disjunctions $\varphi \vee \psi$, universal statements $\forall x.\ \varphi(x)$, etc. On top of this, we can introduce a *provability judgment* $\Theta \vdash \varphi$ which we read as "the formula φ is provable from the set of assumptions Θ". When Θ is empty, we write $\vdash \varphi$, asserting that "the formula φ follows from no assumptions", i.e., "φ is a tautology". We derive instances of the provability judgment, i.e., show that particular formulas are provable given particular assumptions, by applying *inference rules.* As analyzed in the case of intuitionistic logic by Martin-Löf [61], the collection of inference rules we choose to use can be seen as defining the meanings of the propositional connectives. For example, consider the introduction rule $(\wedge I)$ for conjunctions presented in natural deduction style.

$$\frac{\Theta \vdash \varphi \qquad \Theta \vdash \psi}{\Theta \vdash \varphi \wedge \psi}\ (\wedge I)$$

This rule can be read in two ways. Going downwards, it means that having already asserted φ and ψ, we can assert $\varphi \wedge \psi$. Going upwards, it says that to prove $\varphi \wedge \psi$ it suffices to prove φ and to prove ψ. Note that the downwards/upwards readings are not the same as informal forwards/backwards reasoning, as we will see soon. Regardless of which way we read it, a proof using this rule (with no antecedents) can be viewed as a tree like the one below.

$$\begin{tikzcd}
\bigtriangledown & & \bigtriangledown \\
\vdash \varphi \text{ by } (\cdots) \arrow[dr, dash] & & \vdash \psi \text{ by } (\cdots) \arrow[dl, dash] \\
& \vdash \varphi \wedge \psi \text{ by } (\wedge I) &
\end{tikzcd}$$

The root node of such a tree is the theorem being proved, whereas other nodes are judgments derived in the course of the proof. Nodes are annotated with their mode of justification, i.e., the rule of inference 'by' which they follow. Edges point to subproofs of premises required by the inference rule. We may abbreviate subtrees as cones $\triangledown$, writing 'by $(\cdots)$' for some unspecified justification.

In some implementation contexts we may be concerned about representation efficiency. At least two tools are available for reducing the size of proofs in situations in which a single proof can be used more than once. One, we may introduce an intermediate lemma ψ via the *cut* rule.[3]

$$\frac{\Lambda \vdash \psi \qquad \Theta, \psi \vdash \varphi}{\Theta, \Lambda \vdash \varphi}\ (\mathit{Cut}\ \psi)$$

[3]Though the cut rule is more relevant in sequent calculus presentations of logic, it is also derivable in structural natural deduction from the rules for implication [9].

When introducing a cut by ψ, we only need to store its (presumably long and complicated) proof once, in the derivation of $\Lambda \vdash \psi$. We may then reuse it arbitrarily many times in the derivation of $\Theta, \psi \vdash \varphi$ using the hypothesis rule $\Theta, \psi \vdash \psi$. For example, in the left tree below a cut by φ is introduced. Alternatively, if our representation of proofs is being implemented as a concrete data structure and the programming language in use supports storing multiple references (or *pointers*) to a single object, we may reuse a subproof using this language mechanism. In this case trees effectively become graphs like the one on the right below. These techniques can exponentially reduce proof size; but we are not concerned with complexity here so let's only speak of trees.

$$\begin{tikzcd}[column sep=small]
\bigtriangledown \arrow[d, no head] & \varphi \vdash \varphi \text{ by } (\mathit{Hyp}) \arrow[dr, no head] & & \varphi \vdash \varphi \text{ by } (\mathit{Hyp}) \arrow[dl, no head] \\
\vdash \varphi \text{ by } (\cdots) \arrow[dr, no head] & & \varphi \vdash \varphi \land \varphi \text{ by } (\land I) \arrow[dl, no head] & \\
 & \vdash \varphi \land \varphi \text{ by } (\mathit{Cut}\ \varphi) & &
\end{tikzcd}$$

$$\begin{tikzcd}
\vdash \varphi \text{ by } (\cdots) \arrow[d, no head, bend right] \arrow[d, no head, bend left] \\
\vdash \varphi \land \varphi \text{ by } (\land I)
\end{tikzcd}$$

Crucially, in computer-assisted mathematics proof trees do not serve as static, immutable entities which we can admire only as wholes. Their role is primarily to aid the incremental construction of proofs. There are two natural orders in which we could do this: *leaves-to-root* or *root-to-leaves*. In root-to-leaves order, we start off with an incomplete tree (nevermind how such an object might be formally defined; here, illustrations suffice) containing just one node for the assertion that we wish to prove, with a missing justification (?). We then proceed to build up parts of the tree, applying inferences *upwards*, until every assertion has been verified. A node with its justification missing is called a *goal*. The sequence of trees below read left-to-right illustrates one way we could build up a proof in this order. Goals are highlighted in blue boxes.

$$\boxed{\varphi \vdash \varphi \lor \psi \text{ by } (?)}$$

$$\begin{tikzcd}
\boxed{\varphi \vdash \varphi \text{ by } (?)} \arrow[d, no head] \\
\varphi \vdash \varphi \lor \psi \text{ by } (\lor I_L\ \psi)
\end{tikzcd}$$

$$\begin{tikzcd}
\varphi \vdash \varphi \text{ by } (\mathit{Hyp}) \arrow[d, no head] \\
\varphi \vdash \varphi \lor \psi \text{ by } (\lor I_L\ \psi)
\end{tikzcd}$$

In the above example we started from a root goal and used the left-introduction rule for disjunctions ($\lor I_L$) to create a child goal (a *subgoal*). We then justified it as a hypothesis (*Hyp*). Alternatively, in leaves-to-root order, we could have applied inferences downwards by first asserting the leaf, and then concluding the theorem, as below. In this mode we are working with a forest of proof trees and gluing them together.

$$\varphi \vdash \varphi \text{ by } (\mathit{Hyp})$$

$$\begin{tikzcd}
\varphi \vdash \varphi \text{ by } (\mathit{Hyp}) \arrow[d, no head] \\
\varphi \vdash \varphi \lor \psi \text{ by } (\lor I_L\ \psi)
\end{tikzcd}$$

In principle, with arbitrary mixing of upwards and downwards reasoning there are combinatorially many ways to assemble any finite proof tree. Nevertheless, there is no loss of generality in constructing proofs in root-to-leaves order. This style, sometimes called *proof refinement*, is preferred by most modern theorem proving software. Two observations might justify why this is the case. One is that upwards construction can be mechanical and driven by the shape of the goal, rendering it amenable to proof automation. For example, there is really only one way to prove a conjunction, and that's to prove both conjuncts. Contrast this with downwards constructions which require 'inventing' the leaves that we start from. The second observation has to do with *goal states*. In general, at any given time, there can be multiple unjustified leaves in the tree, i.e., multiple goals. This frontier viewed as a list is called the goal state. The goal state below contains two goals.

$$\begin{tikzcd}
 & \varphi, \psi, \theta \vdash \psi \text{ by } (\mathit{Hyp}) \arrow[dr, no head] & & \boxed{\varphi, \psi, \theta \vdash \theta \text{ by } (?)} \arrow[dl, no head] \\
\boxed{\varphi, \psi, \theta \vdash \varphi \text{ by } (?)} \arrow[dr, no head] & & \varphi, \psi, \theta \vdash \psi \wedge \theta \text{ by } (\wedge I) \arrow[dl, no head] & \\
 & \varphi, \psi, \theta \vdash \varphi \wedge (\psi \wedge \theta) \text{ by } (\wedge I) & &
\end{tikzcd}$$

The concept of goals and goal states turns out to be tremendeously useful in computer-assisted proof. In proof assistants based on *tactic frameworks* (section 3.2), the goal state is all the user ever sees. This works so well that end users of such systems are often not aware of a proof tree being constructed under the hood; their perspective is like that of a 2-dimensional Flatlander standing on top of the tree, seeing only the frontier, with the rest obscured. In other systems such as those based on *holes* (section 3.3), the whole tree may be visible but it is always possible and commonly useful to zoom in on one of the goals, temporarily forgetting the rest of the tree.

One potential point of confusion is how the forward steps of paper mathematics come into play during root-to-leaves construction. They are not downwards inferences. Rather, they are applications of those rules which 'invent' a new hypothesis, in the sense of having a premise that is not a subformula of the goal sequent: for instance, the cut rule or implication elimination. To a first approximation, these are the creative steps. An intuitionistic proof of the double-negated law of the excluded middle benefits from a cut.

$$\begin{tikzcd}
\bigtriangledown & & \bigtriangledown \\
\neg(\varphi \vee \neg\varphi) \vdash \neg(\varphi \vee \neg\varphi) \to \neg\varphi \text{ by } (\cdots) \arrow[dr, no head] & & \neg(\varphi \vee \neg\varphi), \neg(\varphi \vee \neg\varphi) \to \neg\varphi \vdash \bot \text{ by } (\cdots) \arrow[dl, no head] \\
 & \neg(\varphi \vee \neg\varphi) \vdash \bot \text{ by } (\mathit{Cut}\ \neg(\varphi \vee \neg\varphi) \to \neg\varphi) \arrow[d, no head] & \\
 & \vdash \neg\neg(\varphi \vee \neg\varphi) \text{ by } (\to I) &
\end{tikzcd}$$

2.1 Type theory

So far I wrote only of propositions and their proofs. This might appear at odds with an earlier assertion that constructions of objects are just as important. However, during the 20th century, pioneering computer scientists and mathematicians made a number of observations bringing forward structural relationships between proofs and effective constructions. This tradition includes, among numerous others, the work of de Bruijn, Curry, Feys, and Howard on connections between logical and computational calculi; of Lambek on categorical models of these calculi; of Brouwer, Heyting, and Kolmogorov on interpretations of proofs in intuitionistic mathematics as functions or processes; and of Kleene on realizability.

Informed by these ideas, the most coherent modern account of what it means to construct mathematical objects has been developed in a multitude of flavours of *type theory*. In a type theory, the central object of study is the classification of constructions into types.[4] A type theory includes a language of constructions: basic ones such as integers and functions, but also proofs themselves, as we will see; and a language of types: the type of integers, function types, and proposition-types that classify proofs. Basic examples of classifications are that the integer 2 has type $\mathbb{Z}$, or that the mapping $x \mapsto e^x$ has type $\mathbb{R} \to \mathbb{R}$. Type theories range from the *simple*, capable of proving judgments like the above, to the *dependent*, in which fairly complicated types and type families become available.

Just like the provability judgment $\Theta \vdash \varphi$ of logic we considered above is hypothetical in that φ is proved under the assumptions in Θ, the judgments of any type theory are hypothetical over *contexts*. A context, often denoted by Γ, is a list $x_1 : T_1, x_2 : T_2, \dots, x_n : T_n$ of variables with specified types. Contexts allow us to suppose to existence of constructions with the specified types but otherwise unknown structure. The basic typing judgment $\Gamma \vdash t : T$ says that "if for every $x_i : T_i \in \Gamma$, x_i stands for an arbitrary construction of type T_i, then t is a construction of type T". In general, both t and T may use the variables in Γ. Our two examples are rendered into this language as $\diamond \vdash 2 : \mathbb{Z}$ and $\diamond \vdash x \mapsto e^x : \mathbb{R} \to \mathbb{R}$ with $\diamond$ denoting an empty context.

Proofs are an important special case of mathematical constructions. In first-order logic, derivability of $\Theta \vdash \varphi$ means that φ is provable from Θ, but the judgment itself says nothing about *how* φ was proven. This information is instead recorded in the proof tree: we used 'by (*Rule*)' annotations for it. The loss of information is appropriate when we think of propositions as merely candidates for a Boolean truth value. On the other hand, it can be obstructive when we wish to analyze the structure of proofs, for example to recover their computational content. In type theory, additional information is stored in proof-constructions. Following the dictum of propositions-as-types, we express proof-constructions by embedding a language of propositions into the language of types. To prove a proposition, we construct an inhabitant of the proposition-type. The incremental process of building a proof tree for a formula φ under assumptions Γ then becomes the process of constructing h such that $\Gamma \vdash h : \varphi$. For example, we may begin refining a proof that for any natural number n, there is a prime greater than n, as follows (read the sequence of trees *bottom-to-top*).

[4]This is not the only possible perspective on type theory, but it is mine.

$$\begin{tikzcd}
 & & \boxed{n : \mathbb{N} \vdash ?_4 : ?_2 \text{ prime by } (?)} \arrow[d, dash] & \boxed{n : \mathbb{N} \vdash ?_5 : n < ?_2 \text{ by } (?)} \arrow[dl, dash] \\
\boxed{n : \mathbb{N} \vdash ?_2 : \mathbb{N} \text{ by } (?)} \arrow[dr, dash] & & n : \mathbb{N} \vdash (?_4, ?_5) : ?_2 \text{ prime} \wedge n < ?_2 \text{ by } (\wedge I) \arrow[dl, dash] & \\
 & n : \mathbb{N} \vdash (?_2, (?_4, ?_5)) : \Sigma(p : \mathbb{N}).\, p \text{ prime} \wedge n < p \text{ by } (\Sigma I) \arrow[d, dash] & & \\
 & \vdash \lambda n.\, (?_2, (?_4, ?_5)) : \forall (n : \mathbb{N}).\, \Sigma(p : \mathbb{N}).\, p \text{ prime} \wedge n < p \text{ by } (\forall I) & &
\end{tikzcd}$$

$$\begin{tikzcd}
\boxed{n : \mathbb{N} \vdash ?_2 : \mathbb{N} \text{ by } (?)} \arrow[dr, dash] & & \boxed{n : \mathbb{N} \vdash ?_3 : ?_2 \text{ prime} \wedge n < ?_2 \text{ by } (?)} \arrow[dl, dash] \\
 & n : \mathbb{N} \vdash (?_2, ?_3) : \Sigma(p : \mathbb{N}).\, p \text{ prime} \wedge n < p \text{ by } (\Sigma I) \arrow[d, dash] & \\
 & \vdash \lambda n.\, (?_2, ?_3) : \forall (n : \mathbb{N}).\, \Sigma(p : \mathbb{N}).\, p \text{ prime} \wedge n < p \text{ by } (\forall I) &
\end{tikzcd}$$

$$\begin{tikzcd}
\boxed{n : \mathbb{N} \vdash ?_1 : \Sigma(p : \mathbb{N}).\, p \text{ prime} \wedge n < p \text{ by } (?)} \arrow[d, dash] \\
\vdash \lambda n.\, ?_1 : \forall (n : \mathbb{N}).\, \Sigma(p : \mathbb{N}).\, p \text{ prime} \wedge n < p \text{ by } (\forall I)
\end{tikzcd}$$

$$\boxed{\vdash ?_0 : \forall (n : \mathbb{N}).\, \Sigma(p : \mathbb{N}).\, p \text{ prime} \wedge n < p \text{ by } (?)}$$

In the above sequence we begin by introducing a new variable into the context with $(\forall I)$ to prove the universal statement. Next, we inhabit the existential quantifier (Σ is a proof-relevant version of $\exists$) using (ΣI). This generates two subgoals: one for the witness $?_2$, and one for the proof $?_3$ that this witness is prime and greater than n. Lastly, we decompose the conjuction into two subgoals with $(\wedge I)$, and leave the tree above unfinished. While the initial few steps were mechanical, finishing the proof requires some insight. Multiple witnesses could work: for instance setting $?_2 := \min \{ p \mid p \text{ prime} \wedge p \text{ divides } n! + 1 \}$, we would end up with the complete construction

$$\lambda n.\, (\min \{ p \mid p \text{ prime} \wedge p \text{ divides } n! + 1 \}, (h_1, h_2))$$

assuming that h_1, h_2 are some constructions such that

$$n : \mathbb{N} \vdash h_1 : \min \{ p \mid p \text{ prime} \wedge p \text{ divides } n! + 1 \} \text{ prime}$$

and

$$n : \mathbb{N} \vdash h_2 : n < \min \{ p \mid p \text{ prime} \wedge p \text{ divides } n! + 1 \}$$

Observe that as we build up the proof, information about used inferences flows down into the root construction (highlighted in strawberry). Specifically, every 'by (*Rule*)' step gets recorded in a piece of syntax. Universal introduction corresponds to function introduction, whereas existential and conjunction introduction accumulate pairs. The judgment records not just *that* the proposition was proven, but also *how*.

There are two observations to be made. First, because information about used inferences is stored both in the root and the 'by (*Rule*)' annotations, there is a degree of redundancy. It begs the question whether the whole tree could be reconstructed mechanically from just the syntax of the root judgment. Second, if the proof is sufficiently constructive (e.g. p prime is defined as the Boolean output of a primality checker, min as an algorithm that works on finite subsets of $\mathbb{N}$, divides as a computable modulo operation, etc), it ought to have an algorithm associated to it. Given n, the algorithm should return in finite time a prime greater than n. In the example, it appears that something like such an algorithm is recorded in the root term; but its semantics have not been clarified.[5] Making these observations precise is the point of departure for two opposing schools of thought: *computational* and *intensional* type theory.

Computational type theory. The computational approach was championed by Martin-Löf [82], as well as by Constable and his team in the PRL [15] family of systems. Intuitively, a computational type theory is defined on top of a programming language and is in some sense *about* that language. More precisely, in order to specify a computational theory we begin with a λ-calculus, usually untyped, as well as an operational semantics for it that specifies how programs compute. Because programs of this language will end up implementing the algorithmic content of our proofs, they are sometimes called *realizers* after Kleene. We call fully evaluated programs *canonical*, whereas programs that can undergo further evaluation are called non-canonical. The former class might include 37, $\lambda x.x$, or `Nat` (types are themselves programs), whereas the latter might contain $(\lambda x.x)$ 37 (which reduces to 37) or $\texttt{let}\ (x, y) = (29, 2)\ \texttt{in}\ x + y$ (which reduces to 31). The remainder of a computational theory consists of derived notions defined in reference to canonical terms and evaluation. First, we specify which canonical programs count as canonical types, written $\vdash T$ type for T a canonical program. For instance, $\vdash \texttt{Nat}$ type is a sensible axiom. Second, which canonical programs count as canonical terms inhabiting a given canonical type, written $\vdash t \in T$[6] for t a canonical program and T a canonical type (i.e., a canonical program T such that $\vdash T$ type holds). For example $\vdash 37 \in \texttt{Nat}$. Finally, the typing relation is extended to non-canonical programs by saying that a program t has type T, also written $\vdash t \in T$, when T evaluates to a canonical program T', t evaluates to a canonical program t', $\vdash T'$ type and $\vdash t' \in T'$. An immediate consequence of the above definition is that typing is not decidable: to determine whether a program has some type, we need to run it first. The derivation tree of a proven judgment $\vdash t \in T$ cannot be recovered mechanically from just the syntax of the judgment, answering our earlier question in the negative. Since the map from proof tree to realizer is lossy, in order to prove something we must do a bit more work than just writing down the realizer. Consequently, the user experience of working in most implementations of computational type theory is that of refining derivation trees similarly to what we did for first-order logic and in the proof of the infinitude of primes above. In actual implementations this is usually supported by a tactic framework and can be referred to as *tactic-based* proving. When our goal has type T, we apply refinement rules in order to construct a realizer t inhabiting T, i.e., one for which $\vdash t \in T$ holds. The realizer is generally the *output* of this process, although we can also write one down first and then prove that it has the correct type.

[5] While it is sometimes said that 'proofs are programs', this is only accurate either with a very broad meaning of 'program' in mind, or if we restrict to constructive type theories. It depends on the specifics of each type theory whether terms are executable or not, and one may represent non-computable axioms equally well in this framework.

[6] As opposed to the intensional notation $\vdash t : T$.

Intensional type theory. In contrast, in the intensional approach [60] the order of conceptual priority is different. Rather than specifying a language of programs first and then defining typing judgments in terms of its operational behaviour, we specify all of these at once in a mutually inductive way. For instance, the rule of conjunction introduction (which we used in the example proof of the infinitude of primes) might be postulated as

$$\frac{\Gamma \vdash t : \varphi \qquad \Gamma \vdash u : \psi}{\Gamma \vdash (t, u) : \varphi \wedge \psi} \; (\wedge I)$$

This is essentially the same as conjunction introduction in first-order logic, except that every typing judgment is now accompanied by a *proof term* witnessing how it was proven. The difference with the computational approach is that the above rule would be *provable* (or *admissible*) there instead of being postulated axiomatically. The benefit of axiomatizing the collection of acceptable inference rules is that in doing so we are able to finely control their algorithmic properties. In particular, we can design the system in such a way that the typing judgment becomes decidable: given Γ, t and T, a *typechecking* algorithm can decide whether $\Gamma \vdash t : T$ holds. In order to achieve this, intensional proof terms generally contain more information than the realizers of computational type theory. For instance, the *equality elimination* rule must be annotated with the proof of equality that was eliminated (left) rather than be acting silently (right):[7]

$$\frac{\Gamma \vdash h : T = U \qquad \Gamma \vdash t : T}{\Gamma \vdash h \blacktriangleright t : U} \; (=E) \qquad\qquad \frac{\Gamma \vdash h : T = U \qquad \Gamma \vdash t : T}{\Gamma \vdash t : U} \; (=E')$$

As in computational type theory, we can view the proof term as the output of a proof refinement process. However, decidability of the typing judgment affords us a new mode of working, referred to variously as "programming in the type theory" or *term-mode proving*. In this mode, we simply write down the term t (treating it as an input instead of an output) and ask the system to check $\Gamma \vdash t : T$. To support incremental constructions, the terms we write down are permitted to contain *holes* such as the $?_1, ?_2, \ldots$ we saw earlier. Modern implementations of intensional theories support arbitrary interleavings of tactic-mode and term-mode proof. Constructions may also be annotated with even more information so that their types can be deduced (and only Γ,t remain as inputs to the typechecker), a feature known as *type inference*. Most of the large, modern theorem proving systems based on type theory follow the intensional approach due to its desirable algorithmic features.

An intensional proof term may contain subterms that are not computationally relevant, i.e., do not influence evaluation. For instance, if the only provable equalities are between programs that exhibit identical computational behaviour, then $h \blacktriangleright t$ ought to always evaluate to the same value as just t. These *irrelevant* subterms are removed during *erasure*, a procedure that extracts the pure computational content realizing a proof term. Erasure can be seen as one connection between intensional proof terms and computational realizers.

[7] *Extensional type theory* is sometimes used to mean an axiomatically presented system (i.e., one whose judgments, like in intensional type theory, are not defined in terms of an operational semantics) in which an *equality reflection* rule similar to the one on the right is present.

Tactics vs proof terms. There are many more increasingly subtle syntactic and semantic differences between different flavours of type theory. But as far as user interfaces are concerned, the two styles of proof refinement distinguished here make for a reasonable classification. In tactic-mode, favoured by traditional logics and computational type theories, we incrementally construct a derivation tree for a provability or typing judgment in root-to-leaves order. We will see that this can be achieved by working in a meta-level language (such as described in section 4), or by using tactic frameworks (section 3.2) that output a type inhabitant under the hood. In term-mode, enabled in intensional type theories, we write down a candidate inhabitant for a type and the system checks its correctness for us. To enable incrementality, we rely on incomplete terms including holes (section 3.3).

3 Interfaces for constructing formal objects

It may be possible (unfortunately) to fall into Lisp in this event.

John Harrison
The HOL Light System Reference

The simplest graphical user interface one can have is not to have one at all. A *proof checker* is a piece of software that consumes a machine-readable formal proof (such as a proof term or another representation of a proof tree) and checks its correctness. It outputs `YES` or `NO`, generally accompanied by additional diagnostic information such as which parts of the candidate proof are wrong in the case of a `NO`. Some of the earliest checkers include de Bruijn's AUTOMATH [26] and Trybulec's Mizar [100].

A proof checker can be transformed into a tool for interactive proof refinement by simply running it in a loop: write down a partial proof, save the file, run the checker, inspect its output for errors, correct the errors, repeat. This process is facilitated by sufficiently informative error messages produced by the checker: they ought to say not just where the error is, but also show the relevant goal when a subproof is missing, explain why a given reasoning step is incorrect, etc. This implementation strategy provides a rudimentary development environment, but it has limitations: we may want to retrieve additional information about our formal proof developments, and have the interface provide that quickly; we may want the interface to support renamings and other refactorings commonly used in software engineering; finally, we may want to view mathematical objects in a graphical rather than a text-based form. For reasons such as these, over time more specialized user interfaces geared towards interactive and automated proof have been developed. They exist as standalone programs, editor plugins, and websites.

In this section we will cover four styles of prover interface: programming in a meta-language (section 3.1), tactic scripts (section 3.2), term-mode proving in type theory (section 3.3), and systems with external verification conditions (section 3.4). A broader survey of graphical theorem proving interfaces is given by Ayers [10, section 5.1].

3.1 Meta-proving with abstract types

When we implement a proof checker, we unavoidably do so in some programming language. This suggests a possible user interface: just use that same programming language to prove things! While there are many conceivable ways of achieving this, an influential design has been established in the Edinburgh LCF system [45]. It continues to live on in HOL4 [92] and HOL Light [47].

In this design, we expose proof checking as a library in our chosen programming language. To distinguish the language we program the proving environment *in* from the logical language we write the environment *for*, we call the former the *meta language* and the latter the *object language*. In the case of HOL Light, the object language is a higher-order logic whereas the meta language is OCaml. In our meta language, we define a type `term` of object language terms. Pretty-printing and parsing facilities are generally installed so that `terms` can be written in reasonably natural form rather than as abstract syntax trees. For instance, in HOL Light, we can use the OCaml read-eval-print-loop (REPL) to write down the number 57 and the statement '57 is prime'. Propositions are terms in higher-order logic, so these are both terms. Terms written in backticks are passed through the object language parser.

```
# `57`;;
val it : term = `57`
# `prime 57`;;
val it : term = `prime 57`
```

Next we define a meta type `thm` of theorems, theorems being provable judgments about the object language. For instance, in HOL Light a primality checker can show that 57 is composite. The checker hands us back a value of type `thm`:

```
# PRIME_CONV `prime 57`;;
proving that 57 is composite
val it : thm = |- prime 57 <=> F
```

Since we have full access to the meta programming environment, a priori there is nothing preventing us from creating values of type `thm` that are not actual theorems. Yet for `thm` to be worthy of its name, we want the property that any value of type `thm` has a valid proof. For this reason, when designing the Edinburgh LCF, Milner and collaborators endowed their meta language (appropriately named ML) with a module system and facilities for *type abstraction.* Abstraction allows an ML module K to specify that `thm` is *some* type, but one whose definition may not be observed in any other module, and whose values may only be constructed using functions exposed by K. Thus if each of these functions is sound, then the whole system must be sound: we must only audit the implementation of K for correctness as no other module could possibly derive a theorem without going through K. The module K is called the *kernel*, forming the trusted core of a theorem proving system.

The functions exposed by K will generally be primitive inference rules of the object logic. For instance, to support equality reasoning, the HOL Light kernel includes a function `K.TRANS : thm -> thm -> thm` that implements transitivity of equality. We can use it to prove that $7 + 7 = 11 + 3$ in a roundabout way:

```
# let thm1 = NUM_REDUCE_CONV `7+7`;;
val thm1 : thm = |- 7 + 7 = 14
# let thm2 = NUM_REDUCE_CONV `11+3`;;
val thm2 : thm = |- 11 + 3 = 14
# TRANS;;
val it : thm -> thm -> thm = <fun>
# SYM;;
val it : thm -> thm = <fun>
# TRANS thm1 (SYM thm2);;
val it : thm = |- 7 + 7 = 11 + 3
```

Proof checker vs kernel. Recall that a proof checker consumes a complete, static record of a proof and checks it for correctness. A system capable of producing proof objects that can be verified by an "easy" algorithm (e.g. one that only accepts basic, "self-evident" proof steps) is sometimes said to satisfy the *de Bruijn criterion* [13]. De Brujin designs facilitiate the development of independent checkers that do not need to trust any component of a sophisticated theorem proving system. In contrast, an LCF-style kernel provides a set of functions for *dynamically* checking proofs, or equivalently for dynamically constructing theorems. The internal representation of thm only has to store the minimal amount of information necessary to carry out these checks. Indeed, many HOL family provers represent thm as just the theorem statement in order to save memory: the proof tree is never stored. In this model, the proof tree can be seen as the execution trace of an algorithm that successfully constructs a thm by interacting with the kernel. Nevertheless, when discussing other theorem proving systems built around a core proof checker it is common to conflate terminology and refer to that checker as the kernel: we use “LCF-style kernel” to disambiguate.

Programming tactics. To minimize the amount of trusted code, kernel functionality is intentionally primitive. LCF-style kernels and minimal proof checkers will generally not support proof refinement directly. For instance, the only rules exposed by the HOL Light kernel are the forward inferences. The function TRANS we saw above is the top-to-bottom reading of

$$\frac{\Gamma \vdash t_1 = t_2 \qquad \Delta \vdash t_2 = t_3}{\Gamma, \Delta \vdash t_1 = t_3}\ (\texttt{TRANS})$$

To program other proof strategies we take advantage of the fully-featured meta language that our object logic is embedded in. Over time, the theorem proving community converged on one roughly consistent approach to the implementation of proof strategies: *tactics*. Generally speaking, a tactic is a meta program supporting root-to-leaves inference in the object logic. A tactic reduces one or more goals into zero or more smaller subgoals (as discussed in section 2). When no more subgoals remain, the proof is finished. Some tactics support basic proof refinement by implementing the upwards versions of primitive inference rules. Others provide more powerful proof search algorithms such as decision procedures for decidable theories, potentially producing many inference steps in one invocation.

In HOL-style systems, tactics live in the *subgoal package*. A simple subgoal package can be implemented as follows. First, we represent each goal as a tuple of a list of antecedents and a succedent: `goal = term list * term`. Then, a tactic is a program `tactic = goal -> goalstate` which reduces a given goal into a new `goalstate = goal list * justification`, that is a list of subgoals together with a `justification = thm list -> thm`. The purpose of the justification is to assemble a proof of the original goal given proofs of the subgoals. This is necessary because, recall, the kernel only supports downwards proof checking. I do not know whether a subgoal package as simple as this ever existed in the wild. In practice, a number of additional features are needed to support realistic proofs: tactic combinators/tacticals, metavariables, unification, and so on. A modern, flexible definition of a tactic was contributed independently by Asperti and coauthors [7] to Matita, as well as by Spiwack [94] to Coq. Among other features, their frameworks support tactics operating on multiple goals at once. A similar design is used in Lean 4 (section 4).

Proving with tactics in HOL Light amounts to stating a goal using g and applying tactics using e. Below, we prove an obvious proposition:[8]

```
# g `p ==> q ==> p /\ q`;;
Warning: Free variables in goal: p, q
val it : goalstack = 1 subgoal (1 total)
```

[8]The example is just for illustration. In practice, it would be proven in one step by an automated proof search tactic.

```
`p ==> q ==> p /\ q`

# e(REPEAT DISCH_TAC);;
val it : goalstack = 1 subgoal (1 total)

  0 [`p`]
  1 [`q`]

`p /\ q`

# e(CONJ_TAC THEN (FIRST_ASSUM ACCEPT_TAC));;
val it : goalstack = No subgoals
```

Note that g, e, et al. are ordinary OCaml functions, but the short names hint at their intended use as interactive commands. Thus the OCaml REPL forms a sort of proto-UI for interactive theorem proving. With clever applications of metaprogramming a surprising variety of styles is achievable. For instance, Harrison [46] demonstrates that HOL systems can be extended with support for declarative proof in the style of Mizar ("Note that P is true, so we can deduce Q. The conclusion follows from facts established so far."), as opposed to the more procedural style of HOL ("Strip the quantifier, split into two subgoals, close both using ITAUT_TAC."). In this way, a metaprogrammable system is entirely open and uniformly extensible.

Nevertheless, working in a bare-bones environment such as this can get awkward. A problem that we address in section 5 is that the interface is not referential. More immediately, the REPL provides limited functionality for working with proof scripts. For instance, to record the above interactive session into a single declaration, we have to manually copy-paste our sequence of tactics:

```
# let THM1 = prove(`p ==> q ==> p /\ q`,
                   (REPEAT DISCH_TAC) THEN CONJ_TAC THEN (FIRST_ASSUM ACCEPT_TAC));;
val THM1 : thm = |- p ==> q ==> p /\ q
```

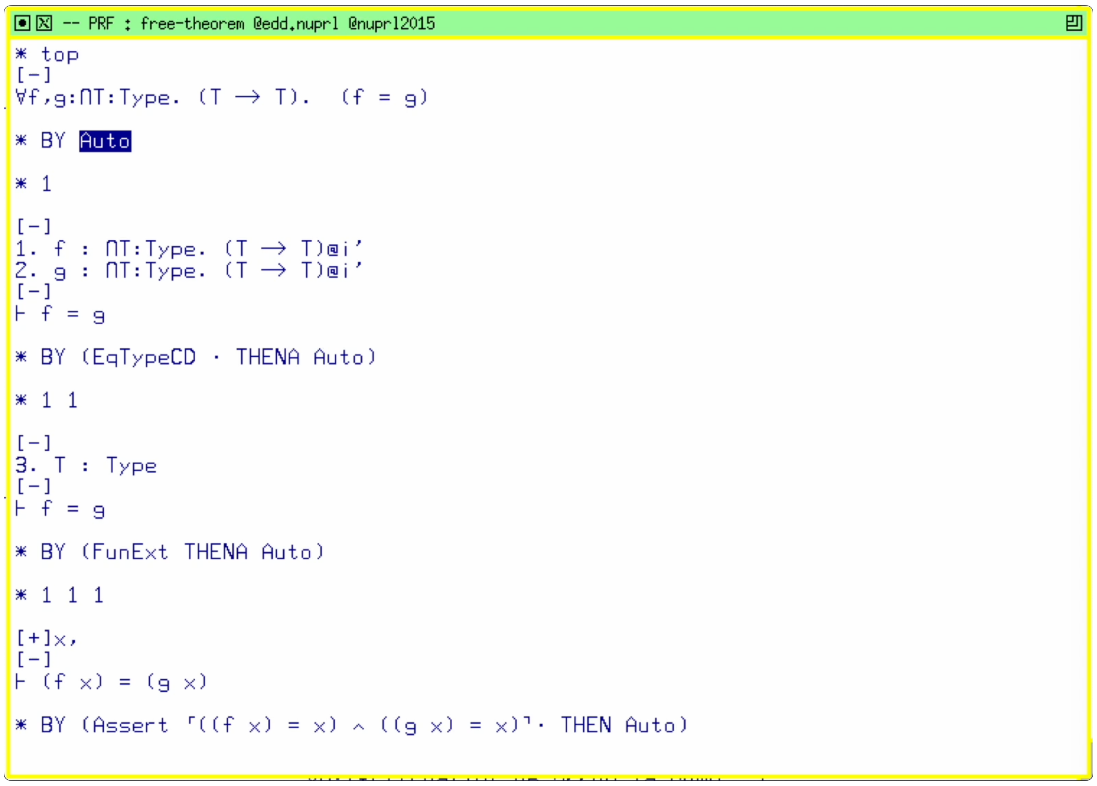
Furthermore, after a proof has been recorded as an invocation of prove, there is no longer an easy way to view the intermediate goal states, e.g. that after REPEAT DISCH_TAC we are to prove $p, q \vdash p \wedge q$. Even after one gains proficiency writing HOL proofs, they remain difficult to read. For these reasons and more, interfaces dedicated to writing, editing, and reading structured tactic scripts have been invented.

3.2 Tactic scripts

In logic, a goal consists of a succedent that is supposed to follow from a list of antecedents. In type theory, it is an *expected type* in a *local context*. In either case, the capability to see the goal state at every step (that is, wedged inbetween successive tactic invocations) is hugely beneficial, if not outright necessary, to understanding formal tactic-based proofs. In this section, we look at a few user interfaces that display goal states and more, pointing out salient and interesting features.

3.2.1 Interleaved goal states

In section 3.1, we saw that a metalanguage REPL provides a simple means of writing tactic proofs. Within an interactive session, the tactic commands are visually interleaved with goal states printed by g and e. The proof display of NuPRL [3], an assistant for computational type theory, takes this visual form as its starting point. NuPRL's display as seen in fig. 3.1 is, however, interactive: any tactic step can be inspected, copied, or edited in which case the remainder of the proof from that step onwards is replayed.



```
-- PRF : free-theorem @edd.nuprl @nuprl2015
* top
[-]
∀f,g:⋂T:Type. (T → T).  (f = g)

* BY Auto

* 1

[-]
1. f : ⋂T:Type. (T → T)@i'
2. g : ⋂T:Type. (T → T)@i'
[-]
⊢ f = g

* BY (EqTypeCD · THENA Auto)

* 1 1

[-]
3. T : Type
[-]
⊢ f = g

* BY (FunExt THENA Auto)

* 1 1 1

[+]x,
[-]
⊢ (f x) = (g x)

* BY (Assert ⌜((f x) = x) ∧ ((g x) = x)⌝· THEN Auto)
```

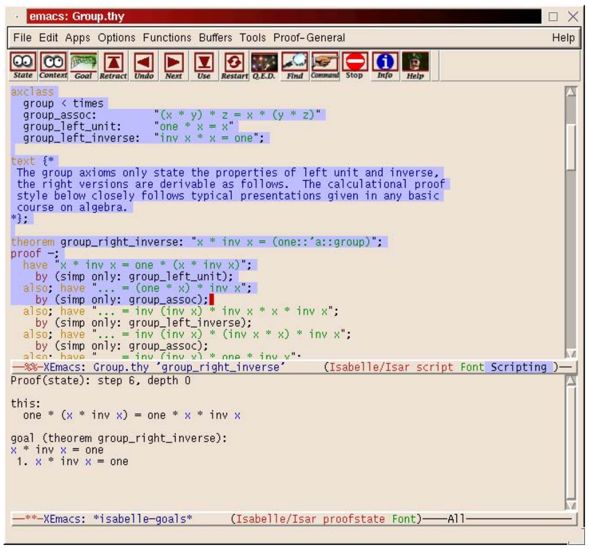
Figure 3.1: The proof of an internal parametricity theorem is displayed in the NuPRL user interface. The first tactic step, using an automated proof search tactic Auto, is selected.

The NuPRL environment provides a number of now standard features: the definition of any symbol can be viewed in a new window, and proof subtrees can be temporarily collapsed; as well as some features that are not commonly seen: a definiendum can be *exploded* by replacing it with its definiens inline, and term editing is structural rather than character-based; finally some features that are unique to NuPRL: to support collaborative proof development, proofs-in-progress can be sent to other users of the system, and are synchronized in real-time as the distributed collaborators make changes. Given NuPRL's development timeline, one may conjecture that the tradition of rich Smalltalk UIs [44] had an influence on the interface.

3.2.2 Two-pane proof editors

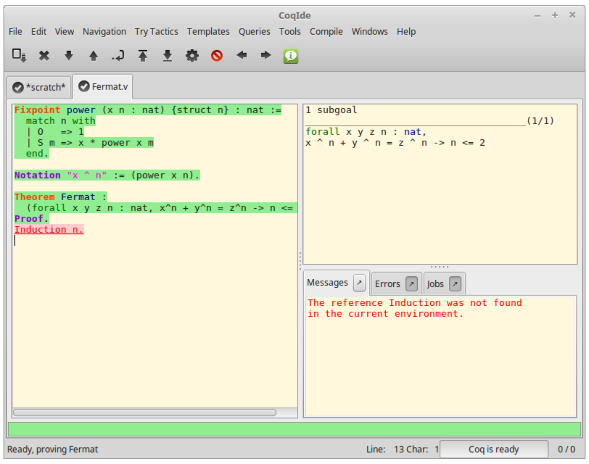
Rather than showing the goal states inline, we may want to keep the proof script together in the text editor buffer, showing the goals in another panel instead. Such a two-pane view of the world was popularized by ProofGeneral [8] (shown in fig. 3.2a) and CoqIDE (fig. 3.2b). Today it is the most common style of ITP interface, used also in Lean 4.

Checkpointed vs continuous interaction. Unlike with interleaved goal states, in the two-pane model not all intermediate goals are visible at all times. Thus we must determine which goals to show when. There are generally two ways to do this.


emacs: Group.thy
File Edit Apps Options Functions Buffers Tools Proof-General Help
State Context Goal Retract Undo Next Use Restart Q.E.D. Find Command Stop Info Help

```
axclass
  group < times
  group_assoc:         "(x * y) * z = x * (y * z)"
  group_left_unit:     "one * x = x"
  group_left_inverse:  "inv x * x = one";

text {*
 The group axioms only state the properties of left unit and inverse,
 the right versions are derivable as follows.  The calculational proof
 style below closely follows typical presentations given in any basic
 course on algebra.
*};

theorem group_right_inverse: "x * inv x = (one::'a::group)";
proof -;
  have "x * inv x = one * (x * inv x)";
    by (simp only: group_left_unit);
  also; have "... = (one * x) * inv x";
    by (simp only: group_assoc);
  also; have "... = inv (inv x) * inv x * x * inv x";
    by (simp only: group_left_inverse);
  also; have "... = inv (inv x) * (inv x * x) * inv x";
    by (simp only: group_assoc);
  also; have "... = inv (inv x) * one * inv x";
--%%-XEmacs: Group.thy 'group_right_inverse'      (Isabelle/Isar script Font Scripting )--
Proof(state): step 6, depth 0

this:
  one * (x * inv x) = one * x * inv x

goal (theorem group_right_inverse):
x * inv x = one
 1. x * inv x = one

--**-XEmacs: *isabelle-goals*      (Isabelle/Isar proofstate Font)----All----------
```

(a) The ProofGeneral interface operating on an Isabelle/Isar proof script in Emacs. In the editor buffer, the prefix of the proof that has been checked is indicated in blue. The goal state at the end of the checked region is shown in a panel below. Screenshot via Aspinall [8].


CoqIde
File Edit View Navigation Try Tactics Templates Queries Tools Compile Windows Help
scratch
Fermat.v
Messages
Errors
Jobs

```
Fixpoint power (x n : nat) {struct n} : nat :=
  match n with
  | 0   => 1
  | S m => x * power x m
  end.

Notation "x ^ n" := (power x n).

Theorem Fermat :
  (forall x y z n : nat, x^n + y^n = z^n -> n <=
Proof.
Induction n.
```

```
1 subgoal
______________________________________(1/1)
forall x y z n : nat,
x ^ n + y ^ n = z ^ n -> n <= 2
```

```
The reference Induction was not found
in the current environment.
```

```
Ready, proving Fermat          Line:  13 Char:  1   Coq is ready   0 / 0
```

(b) A CoqIde window. A proof by induction is in progress in the proof script editor. The current subgoal is shown on the right. Screenshot via the Coq reference manual [97].

Figure 3.2

The traditional model (sometimes referred to as a *waterfall* or, tongue-in-cheek, an *iron curtain*) keeps track of a prefix of the proof script that has been checked so far. The checked part is indicated visually, e.g. in blue in fig. 3.2a, and terminates in a *checkpoint*. In this model, the goal display shows the goal at the checkpoint. The checkpoint is advanced and receded manually by the user. Advancing it corresponds to executing the next step in a tactic proof, or more generally the next command in a buffer. Another way to think about the process is as stepping through a program in a debugger.[9] As of 2022, waterfall interaction remained the most popular in the Coq ecosystem [97], where users prefer ProofGeneral and CoqIDE [4]. It is also used in other systems, for example the Emacs `fstar-mode`[10] for F* [96].

One advantage of checkpointing is its good support for caching. The checked region is "locked" in that any edit made below the checkpoint cannot invalidate it. Thus we never have to re-run a tactic or command that has already been checked. This can be beneficial when working with slow, heavyweight tactics. Additionally, since moving the checkpoint is an intentional action, users have a clear idea of how long it may take and when to do it.

On the other hand, editing anything *above* the checkpoint recedes it back to that position. It then has to be manually advanced again. This adds friction to investigations of how a change at one point affects the result of a command appearing later down in the buffer, and to ordinary proof development for users who prefer to see the results of their changes instantaneously.

The *continuous* model of interaction takes a different approach. Here there is no checkpoint: instead, the goal state at the current text cursor position is always displayed, updating in real-

[9]This is accurate in a quite literal sense since tactics are metaprograms operating on the goal state. On this view, the goal display is a domain-specific display of the memory state.

[10]https://github.com/FStarLang/fstar-mode.el

time as soon as the cursor moves or an edit is made. This variant is closer to how programmers interact with modern development environments. Continuous checking is provided in the Isabelle/PIDE [103] environment, by the VS Code extension for PVS [64], as well as by all editor extensions for Lean. A new system with continuous checking directly inspired by Lean's design is Metamath Zero [28]. It was also implemented in Coqoon [41] for Coq, although Coqoon does not seem to have caught on. The recent Coq LSP[11] project provides both continuous and checkpointed interaction for Coq.

Robustness of tactics. Recall (chapter 2, intensional type theory) that in some formal systems it is possible to work either in tactic-mode or in term-mode. An advantage of tactic scripts over proof terms is that because they express a high-level proof strategy rather than a specific proof directly, they tend to be more robust to changes in the library as well as easier to adapt to prove similar theorems. In many cases the exact same script can be reused between different proofs. In HOL Light we can declare a simple `INDUCT_ADD_TAC` to prove multiple facts about addition by induction.

```
# let INDUCT_ADD_TAC = INDUCT_TAC THEN (ASM_REWRITE_TAC[ADD]);;
val INDUCT_ADD_TAC : tactic = <fun>
# let thm1 = prove(`!n. n + 0 = n`, INDUCT_ADD_TAC);;
val thm1 : thm = |- !n. n + 0 = n
# let thm2 = prove(`!n m. n + (SUC m) = SUC (n + m)`, INDUCT_ADD_TAC);;
val thm2 : thm = |- !n m. n + SUC m = SUC (n + m)
```

On the other hand, when using tactics our proof script is a metaprogram rather than an object-level proof tree. This metaprogram needs to be re-executed every time the formal development is checked. Re-execution can be problematic when slow, heavyweight automation is used. In practice the issue can be solved by providing two versions of slow tactics: a complete one that executes proof search from scratch, as well as a faster one that requires a hint containing sufficient information to reconstruct a proof quickly. For example, the `polyrith`[12] tactic invokes a full Gröbner basis search, but can be hinted with just a linear combination of polynomial equalities.

Even having solved the above issue, we still find ourselves in contexts wherein we do want to write proof terms directly in order to finely control the shape of definitions, as well as the algorithmic content of proofs when "programming in the type theory". When proofs are relevant, term-mode can be a better fit.

3.3 Hole-driven development

The modern approach to term-based proof refinement was first seen in ALF [59] and Epigram [66]. It continues to live on in Agda [76] and Idris [24], where it is the predominant style of programming (or proving: both systems take the BHK interpretation seriously) today. In this style, we operate on incomplete programs containing placeholders known as *typed holes*. In more programming-oriented communities, the approach is known as *type-driven development*.

The idea is to first state the type of the program or proof we want to construct, and then let that type guide us in writing the program. We covered an example proof of the infinitude of primes using holes $?_i$ in section 2, type theory. Here let us carry out a construction of the `map`

[11]https://github.com/ejgallego/coq-lsp
[12]https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/Tactic/Polyrith.lean

function for lists in Agda 2.6.3. We begin by stating the desired type and writing a placeholder for the body.

```
map : {A B : Set} → (A → B) → List A → List B
map f as = ?
```

Upon invoking `load` to start an interactive session, the editor extension replaces the placeholder by a typed hole:

```
map : {A B : Set} → (A → B) → List A → List B
map f as = {!   0!}
```

The intended mode of interacting with holes is via *editor actions* and commands. These are pieces of functionality available in the editor UI that either tell us something about the current partial construction, or directly modify it in a desired way when invoked. In intensional type theories, editor actions act analogously to tactics in the sense that both approaches provide metaprograms that construct proof terms. The major difference is in what is recorded in the source code: tactic-mode proofs save only the tactic script which *will construct the term when executed*, specifying *how* but not *what*, whereas here the constructed proof term is stored directly and once.

We can inspect the hole's type and local context (i.e., the goal) using the `goal-and-context` command:

```
List B
as : List A
f : A → B
B : Set   (not in scope)
A : Set   (not in scope)
```

One possible way to make progress is to invoke the `auto` command which carries out automated term search. It works, but does not quite give us the desired term:

```
map : {A B : Set} → (A → B) → List A → List B
map f as = []
```

Thus let's backtrack and use `case as` instead to split on the head of `as`. This produces two typed holes:

```
map : {A B : Set} → (A → B) → List A → List B
map f [] = {!   0!}
map f (x ∷ as) = {!   1!}
```

In this state, running `auto` finishes the first construction correctly. For the second one it gives another plausible but incorrect construction, namely `map f as`. When this happens, we need to write down a candidate term manually. Suppose we wrote an unusual one:

```
map : {A B : Set} → (A → B) → List A → List B
map f [] = []
map f (x ∷ as) = {! 11 + 3 0!}
```

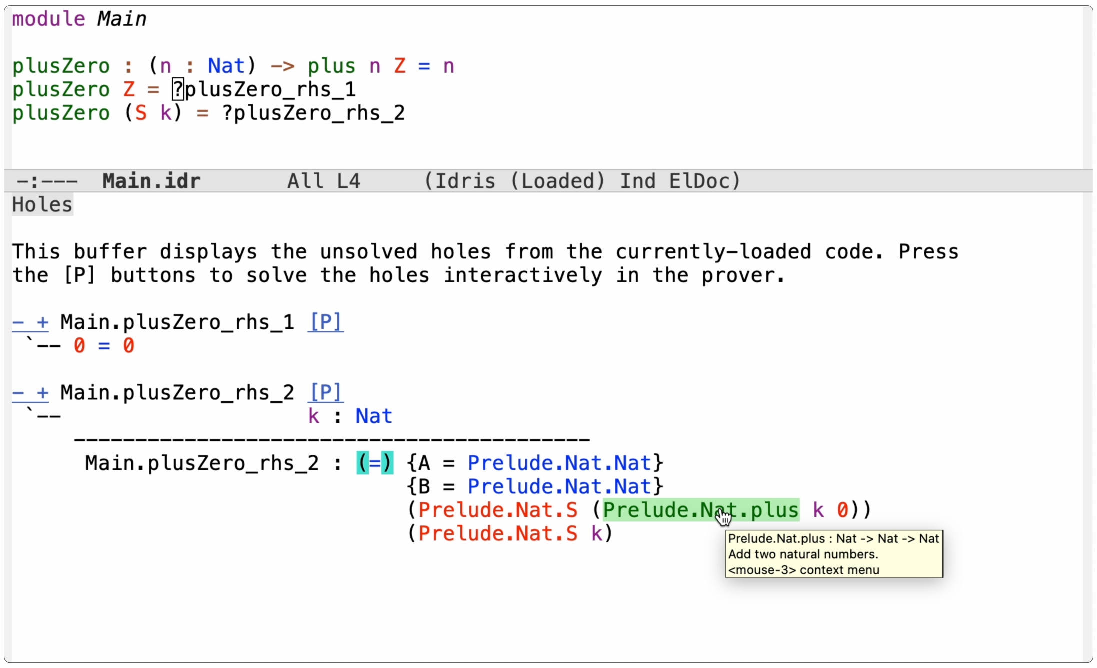
Now a few commands are available for inspecting the candidate $11+3$. We may run `infer-type` to learn its type: this returns $\mathbb{N}$; run `normalize` to compute its normal form: this gives 14; finally, we can run `give` to attempt to solve the goal using the term: this obviously fails. Having learned that this is not the right approach, we `give` the more sensible candidate ? ∷ ?. The system accepts it and renders two new holes:

```
map : {A B : Set} → (A → B) → List A → List B
map f [] = []
map f (x ∷ as) = {!   0!} ∷ {!   1!}
```

At this point the holes are sufficiently type-constrained that `auto` correctly fills in both:

```
map : {A B : Set} → (A → B) → List A → List B
map f [] = []
map f (x ∷ as) = f x ∷ map f as
```

With sufficiently powerful editor support, hole-driven development can be made quite efficient. Some of the most sophisticated interactive environments for general-purpose functional programming based on editor actions have been developed for Idris. `Idris-mode` [68] for Emacs provides the commands we saw in Agda above, as well further features demonstrated in fig. 3.3.



```
module Main

plusZero : (n : Nat) -> plus n Z = n
plusZero Z = ?plusZero_rhs_1
plusZero (S k) = ?plusZero_rhs_2

-:---  Main.idr        All L4      (Idris (Loaded) Ind ElDoc)
Holes

This buffer displays the unsolved holes from the currently-loaded code. Press
the [P] buttons to solve the holes interactively in the prover.

- + Main.plusZero_rhs_1 [P]
 `-- 0 = 0

- + Main.plusZero_rhs_2 [P]
 `--                    k : Nat
     ------------------------------------------
      Main.plusZero_rhs_2 : (=) {A = Prelude.Nat.Nat}
                                {B = Prelude.Nat.Nat}
                                (Prelude.Nat.S (Prelude.Nat.plus k 0))
                                (Prelude.Nat.S k)
```

Figure 3.3: The Emacs UI of `idris-mode` shows a list of currently pending holes. The list is interactive. Hovering over the subterm `Prelude.Nat.plus` shows its type and documentation. A context menu can be opened to normalize the subterm, jump to its definition, show implicit arguments inline as in the case of (=), or find the dependencies and reverse dependencies of a constant.

Holes and metavariables. To implement a system with support for typed holes, we have to clarify what sort of thing they are and endow them with a concrete representation. Differences of representation between different theorem proving systems have a concrete impact on the experience of using these systems.

Observe that a variable corresponds to an entity existing hypothetically in the object language: the sentence $x = 0$ only makes sense ("typechecks") supposing the existence of some $x : \mathbb{N}$.

To indicate that $x = 0$ is a well-formed object-level proposition under that assumption, we make the contextual judgment $x : \mathbb{N} \vdash x = 0 \text{ prop}$.

A hole is similar to a variable, but corresponds to an entity existing hypothetically at the *meta* level. It is difficult to make this apparent when working directly in the object language as we did above in Agda. Let us return to HOL Light where the meta/object boundary is clearer because object-level terms are values of the meta type `term`. Consider the following OCaml function which takes a single term as the argument and substitutes it for `h0` in `h0 + 3`:

```
# let hole_plus_3 (x : term) = subst [x,`(h0:num)`] `h0 + 3`;;
val hole_plus_3 : term -> term = <fun>
```

We evaluate the function on some examples. When the input is of the correct type, we get back a well-formed term.

```
# hole_plus_3 `"eleven"`;;
Exception: Failure "vsubst: Bad substitution list".
# hole_plus_3 `11`;;
val it : term = `11 + 3`
```

As can hopefully be glossed from the above, one possible representation of a hole is as a closure in the meta language. Here the entity existing hypothetically at the meta level is an object-level `x : term`. Since `x` is a variable in the body of the closure, in this representation we refer to holes as *metavariables*.

Unfortunately, representing metavariables this way is unwieldy in practice: an arbitrary closure cannot be inspected, so such an implicit representation makes it complicated or impossible to transform incomplete terms. Instead, we can internalize metavariables by extending the object language with support for holes.

While formal details are out of scope here, one way to make holes in the object language precise using the notion of a metavariable context was worked out by Nanevski, Pfenning, and Pientka [73]. In dual-context style, we can write judgments that look something like

$$?_1 :: (x : \mathbb{N} \vdash \mathbb{N}); n : \mathbb{N} \vdash \lambda(x : \mathbb{N}).\ ?_1 + n : \mathbb{N} \to \mathbb{N}$$

which we read as

Supposing that $?_1$ is some construction of type $\mathbb{N}$ well-formed in context $x : \mathbb{N}$,
and supposing that n is some construction of type $\mathbb{N}$,
$\lambda(x : \mathbb{N}).\ ?_1 + n$ is a construction of type $\mathbb{N} \to \mathbb{N}$.

and could expand to

Supposing that
(supposing that x is some construction of type $\mathbb{N}$, $?_1$ is a construction of type $\mathbb{N}$),
and supposing that n is some construction of type $\mathbb{N}$,
$\lambda(x : \mathbb{N}).\ ?_1 + n$ is a construction of type $\mathbb{N} \to \mathbb{N}$.

illustrating that in this picture, metavariables amount to internalized hypothetical judgments: the term we substitute into a hole can itself depend on object-level variables. Thus every hole comes with an associated local context in which its instantiations are well-formed.

In other approaches, holes become part of the object language indistinct from any other sort of term and are endowed with typing rules as well as operational semantics. Hazel is an editor environment purpose-built for handling incomplete programs. Its core calculus Hazelnut [78] assigns static semantics to holes and editor actions operating on them, while followup work [79] provides operational semantics for incomplete programs. In Hazel, modifying parts of a program provides a live, continuous view of the output.

An important choice in the representation of metavariables is whether they are allowed to depend on each other. In dependent type theory, providing this flexibility allows for goals whose types contain metavariables corresponding to solutions of other goals. The goal $n : \mathbb{N} \vdash ?_3 : ?_2 \text{ prime} \land n < ?_2$ we saw in (section 2, type theory) is an example.

Had we not had this flexibility, the only way to carry out an upwards refinement of a goal of type $\Sigma(p : \mathbb{N}).\ p \text{ prime} \land n < p$ would have been to immediately give the witness. Indeed, an attempt to create two holes in this way fails in Idris 1.3.2.

```
zero : (n : Nat ** n = 0)
zero = (?a ** ?b)

-- When checking deferred type of Main.b:
-- No such variable Main.a
```

3.4 External verification conditions

A final interaction model to cover is seen in tools wherein proofs and some lemma statements *do not matter*. This arises in cases where we want to know that a property holds but do not wish to be burdened with the gruesome details of how it was established. This is commonly the case in software verification: at the end of the day, if our code is functionally correct, memory-safe, or has another desirable property, we don't care how exactly this follows from foundational axioms.[13]

Software verification tools consequently aim to outsource as much of the verification burden as possible to underlying automated theorem provers (ATPs). While processing a formal development, such tools generate *verification conditions* (VCs): statements that we need proven but whose proofs may involve lots of uninteresting detail. Common VCs include termination of recursive functions, subgoals generated by the typechecker, or that a piece of code meets its postcondition assuming the precondition. The VCs are then sent off to the ATP, and some indication of whether they were successfully discharged is shown in the user interface. UIs for verification systems differ on how much information is shown.

In Dafny [54], the recently improved interface [65] attempts to communicate the minimal amount of information necessary to complete a verification. Source locations relevant to failed VCs are displayed prominently to aid the user in fixing their assertions, whereas successful VCs, while possible to inspect if needed, are hidden by default. Dafny uses Z3 [72] as its underlying ATP but provides no other way to prove assertions. The lack of an escape hatch means in particular that it can be difficult or impossible to establish facts that are beyond the reach of automation.

To remediate this issue, other systems have taken a hybrid approach providing a mixture of automated and manual, tactic-based proof. F* [96] and PVS [80] are two systems based on type theory with support for *predicate subtyping* [88] in the case of PVS, and *refinement types* in the case of F*. The rule accepted by PVS is

$$\frac{\Gamma \vdash a : T \qquad \Gamma \vDash b[a/x]}{\Gamma \vdash a : \{x : T \mid b\}}$$

where $x : T \vdash b : \mathit{bool}$ is a predicate and $\{x : T \mid b\}$ is the subtype of those elements of T for which b holds. The premise $\Gamma \vDash b[a/x]$ refers to the truth value of an arbitrary proposition which cannot be algorithmically checked. Since F* postulates a similar law for refinements, the typing judgments of both systems are undecidable. While building up a candidate typing derivation, their typecheckers collect instances of this premise in order to offload verifying them as VCs to underlying automation. PVS generally relies on builtin decision procedures, whereas F* sends its VCs to Z3. In F*, VC proofs are not recorded, but hints that aid Z3 in concluding unsatisfiability more efficiently in the future can be stored in `.hints` files. In PVS a serialized form of the proofs is recorded in a `.prf` file stored separately from the `.pvs` source code written by the user.

Despite theoretical similarities, the experience of using the two systems is different. The Meta-F* [62] metaprogramming framework allows users to write LCF-style tactics that combine primitive backwards inferences in order to either completely close a propositional goal, or

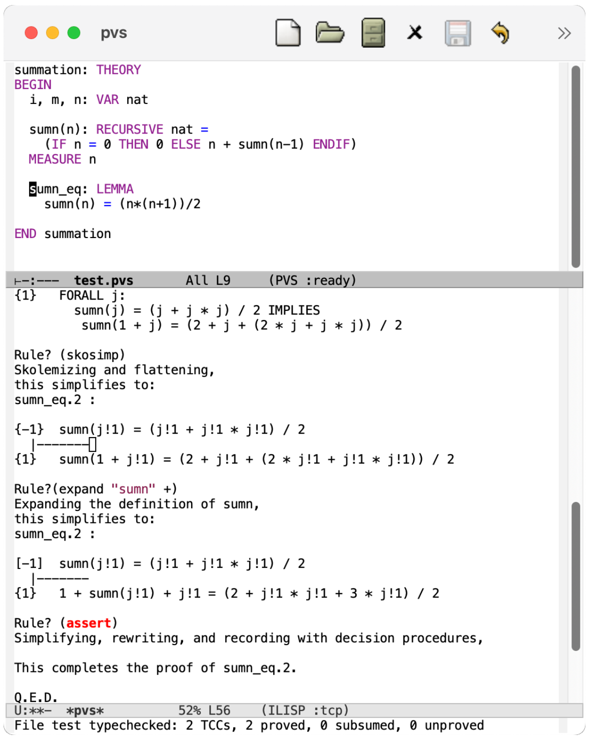
[13] A similar, common attitude shows up in formalised mathematics when it comes to "obvious" statements that nevertheless have to be justified to a theorem prover.

to reduce it to a list of subgoals that are within reach of Z3 (or can be processed further by other tactics). However, there is no interface dedicated to interactively refining the proof of a single VC goal by applying tactics as interactive commands. Instead, it is possible to mark subformulas φ of b (or of another formula that arises somewhere within a program verification development) with a `by_tactic` τ φ annotation that instructs the typechecker to run the tactic τ when φ appears in a positive position[14] within a VC before that VC is shipped off to Z3. Since VCs undergo somewhat unpredictable transformations, it is difficult to tell a priori what goals our tactic τ will end up attempting to solve. In F*, the solution is to instrument the tactic with `dump`, an instruction that displays the goal state the tactic is operating on. Once the tactic is executed, the invocation of `dump` results in the goal (or goals) being shown in the UI. One can also view the sequence of all goal states by setting `--tactic_trace_d 1` on the command line. By consulting the impact of their changes on the shown goals, one may then incrementally develop τ until it satisfactorily manipulates a particular VC. In the Emacs interface shown in fig. 3.4b this proceeds in the checkpointed interaction model (see section 3.2.2), whereas a new VSCode extension[15] provides continuous interaction.

PVS provides a more traditional interactive proof UI. In PVS a distinction is made between VCs and “more interesting” theorems. The VCs are referred to as *type-correctness conditions* (TCCs) because they arise at instances of the aforementioned subtyping rule. The system attempts to discharge TCCs using purpose-built automation. Whenever a TCC fails to be solved immediately, a new interactive proof buffer with that TCC as the goal is shown. The user enters tactics such as `induct` or `grind` that have the usual effect of updating a persistently-displayed goal state. When all goals have been successfully proven, the buffer can be closed. Interactive proof is also the usual way in which manually entered `LEMMA`s, presumably more complex than TCCs, are established. The Emacs interface for PVS shown in fig. 3.4a reflects this design. A dedicated panel (with interleaved goal states) is provided for interactive proof, whereas the number of checked TCCs is noted much more subtly in the status bar. The recent ProofMate interface [63] for PVS provides a more sophisticated proof development environment in VS-Code.

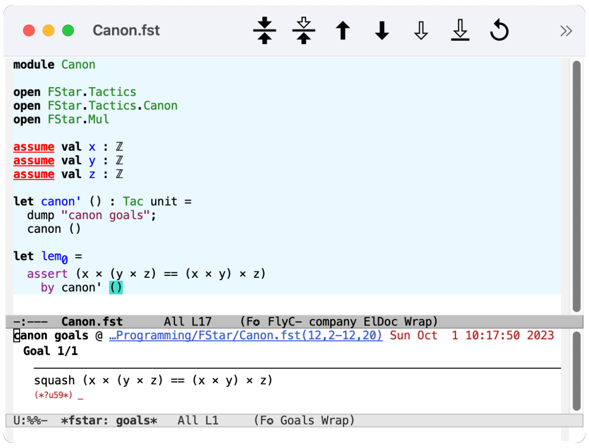
[14]Meaning roughly to the right of an implication where it will need to be proven by Z3, as opposed to formulas appearing on the left which are taken as hypotheses and not proven.

[15]`https://github.com/FStarLang/fstar-vscode-assistant`



```
summation: THEORY
BEGIN
  i, m, n: VAR nat

  sumn(n): RECURSIVE nat =
    (IF n = 0 THEN 0 ELSE n + sumn(n-1) ENDIF)
  MEASURE n

  sumn_eq: LEMMA
    sumn(n) = (n*(n+1))/2

END summation

⊢-:---  test.pvs        All L9      (PVS :ready)
{1}   FORALL j:
        sumn(j) = (j + j * j) / 2 IMPLIES
         sumn(1 + j) = (2 + j + (2 * j + j * j)) / 2

Rule? (skosimp)
Skolemizing and flattening,
this simplifies to:
sumn_eq.2 :

{-1}  sumn(j!1) = (j!1 + j!1 * j!1) / 2
  |-------
{1}   sumn(1 + j!1) = (2 + j!1 + (2 * j!1 + j!1 * j!1)) / 2

Rule?(expand "sumn" +)
Expanding the definition of sumn,
this simplifies to:
sumn_eq.2 :

[-1]  sumn(j!1) = (j!1 + j!1 * j!1) / 2
  |-------
{1}   1 + sumn(j!1) + j!1 = (2 + j!1 * j!1 + 3 * j!1) / 2

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of sumn_eq.2.

Q.E.D.
U:**-  *pvs*            52% L56     (ILISP :tcp)
File test typechecked: 2 TCCs, 2 proved, 0 subsumed, 0 unproved
```

(a) A summation lemma is verified using PVS in Emacs. The topmost buffer contains the theory source code. Below, an interactive proof buffer is used to establish the lemma. The status bar at the bottom displays TCC status.



```
module Canon

open FStar.Tactics
open FStar.Tactics.Canon
open FStar.Mul

assume val x : ℤ
assume val y : ℤ
assume val z : ℤ

let canon' () : Tac unit =
  dump "canon goals";
  canon ()

let lem₀ =
  assert (x × (y × z) == (x × y) × z)
    by canon' ()

-:---  Canon.fst      All L17     (F✪ FlyC- company ElDoc Wrap)
canon goals @ …Programming/FStar/Canon.fst(12,2-12,20) Sun Oct  1 10:17:50 2023
 Goal 1/1

   ─────────────────────────────────────────────────
   squash (x × (y × z) == (x × y) × z)
   (*?u59*) _
U:%%-  *fstar: goals*   All L1      (F✪ Goals Wrap)
```

(b) The F* mode in Emacs is used to establish a simple arithmetic fact. In the source buffer, a tactic canon' that displays the goal state and then runs a normalization procedure is defined and invoked on the lemma. In the goal state buffer below, a goal that the tactic was invoked on is shown.

Figure 3.4

4 Metaprogramming in Lean 4

We want to enable users to work in the simplest model appropriate for their respective use case while always keeping open the option to switch to a lower, more expressive level.

Leonardo de Moura and Sebastian Ullrich
The Lean 4 Theorem Prover and Programming Language

In most of the systems we discussed so far, whether we prove using tactics or "program in the type theory", the meta language is clearly distinct from the object language. In the Edinburgh LCF, the meta language was ML. In HOL Light, one writes tactics in OCaml that operate on proofs in higher-order logic. In Isabelle, tactics are written in Standard ML. Though in Agda one writes object-language terms and propositions directly, the system that checks and compiles these is written in Haskell. The situation in Coq is fuzzier: the system itself is written in OCaml but a number of extensions exist, including MetaCoq [93] which allows writing tactics directly in the object language Gallina.

Idris 2 [23], Lean 4 and F* belong to a new generation of *bootstrapped* dependently-typed languages. In these systems, the implementation (meta) language is also the object logic.[16] Their frontends (elaboration, typechecking, compilation, etc) are programmed entirely or mostly in the meta-object language. I will refer to the unified formal system as just "the language" unless disambiguation is necessary to reduce confusion.

In this section we will cover some notable features of Lean 4 including metaprogramming based on *elaborator reflection*, as well as outline how Lean's frontend operates. The language has been described in detail by Ullrich [102].

4.1 Commands extend the environment

Lean is a dependently-typed, purely functional programming language. The top-level unit of organization corresponding to a single `.lean` file is conventionally called a *module*.[17] Modules are linear sequences of *commands*. Examples of commands include inductive type definitions `inductive` and function definitions `def`.

```
/-- Our very own version of the natural numbers, following the one in the standard library. -/
inductive Nat' where
  | zero : Nat'
  | succ : Nat' → Nat'

/-- The Sudan function was the first known example of a total and computable function
that is not primitive recursive. -/
def sud : Nat → Nat → Nat → Nat
  | 0,   x, y   => x+y
```

[16]An early system with this property was ACL2 [50] whose object logic and meta language were based on Common Lisp.

[17]Note that Lean does not currently have facilities for abstracting over module implementations.

```
| _+1, x, 0   => x
| n+1, x, y+1 =>
  let z := sud (n+1) x y
  sud n z (z+y+1)
-- Lean functions are presumed to be total unless marked otherwise. We give the frontend
-- a hint regarding how it should prove termination.
termination_by sud n _ y => (n, y)
```

For the present discussion it is best to understand Lean modules from the perspective of what the language implementation *does* when processing them rather than in the usual way: as what the contents of a module might *be* (what they might denote). We aim for a procedural rather than a declarative understanding. The above code snippet will be explained from this perspective.

The piece of software (itself written in Lean, via bootstrapping) that processes Lean modules is known as the *frontend*. When invoked on a module, the frontend goes through it in top-to-bottom order, processing the commands one by one. The frontend maintains a number of internal data structures. An important one is the *environment*: a list of all the inductive types, structures and functions introduced so far. It also contains additional metadata such as documentation strings. The environment is viewed as append-only, growing monotonically while the frontend is processing a module.

In our example, the first command encountered is the inductive definition `Nat'`. To process it, the frontend parses the source string, *elaborates* the resulting syntax tree[18] into an inductive type of the underlying type theory, sends the type to be checked by the kernel[19], and finally appends it to the environment if the kernel check was succesful. The sequence of events is summarized in fig. 4.1. If the environment beforehand was e, after the command is processed the environment becomes an extended one $e[\texttt{Nat'} \mapsto \mu T.\,(\texttt{zero} : T) + (\texttt{succ} : T \to T)]$. The second command introducing the function `sud` is processed similarly except in that a function constant, rather than an inductive type, is added to the environment.

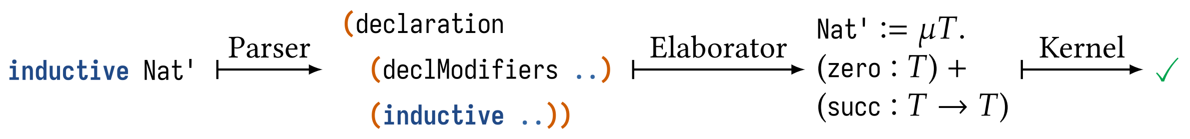


Figure 4.1: The definition of an inductive type `Nat'` is processed by successive stages of the Lean frontend. The parser ingests a string and produces a syntax tree. Syntax is then elaborated into a definition in the core type theory. The definition is checked by the kernel.

The process of elaboration is of particular interest. To see what elaboration is and why it is needed, let us name a few layers of the Lean 4 system explicitly. The logical foundation, "Lean's type theory", is a dependent type theory based on the Calculus of Inductive Constructions due to Coquand and Paulin [36]. Let's call it `LeanTT`. The precise rules and set-theoretic semantics of `LeanTT` have been described by Carneiro [29].

Then, there is the surface-level (also known as *vernacular*) language of Lean. Let us call it `LeanLang`. `LeanLang` is what makes up the source code of a Lean module. A priori, `LeanLang` is

[18]The syntax tree is not fully abstract: information about whitespace is retained, for instance.

[19]This component is a typechecker rather than an LCF-style kernel (see section 3.1, proof checker vs kernel), but it is conventional in the Lean community to call it a kernel.

entirely separate from LeanTT. LeanTT is fairly spartan. It misses many features familiar from LeanLang such as implicit arguments, typeclasses, tactics, even pattern-matching. This makes LeanTT easier to model mathematically and the kernel smaller, reducing the trusted code base whose correctness guarantees correctness of the entire system, and rendering it easier to implement independent typecheckers for LeanTT. However, it also means that the frontend needs to fill in a substantial amount of missing detail when going from LeanLang to LeanTT: it needs to *elaborate* on the input. Thus, the elaborator forms the connection between LeanLang and LeanTT. Elaborators are incredibly complex. For an idea of what they do, I defer to Bauer's excellent explanation.[20]

Distinguishing the two layers resolves a terminological ambiguity: it is common to call a term t of LeanLang a "term of the type theory". What one is really referring to is the LeanTT term that t elaborates to.

Syntactic extensions. The linear, top-to-bottom fashion in which modules are processed means that no command can depend on the results of ones lexically below it. Mutual definitions, such as two functions that call each other recursively, must consequently be wrapped in an explicit `mutual .. end` block in order to be processed successfully. On the other hand, the fact that the entire parse-elaborate-typecheck sequence is run at every command means that each stage of the sequence, including parsing, can depend on modifications to the environment introduced by preceding commands. This design was chosen intentionally by Ullrich and de Moura [101] in order to support syntactic extensions. The `syntax` command declares an extension to the grammar of LeanLang and makes it immediately usable. Under the hood, the extension is just another kind of thing that gets appended to the environment.[21]

```
-- Introduce new syntax for the term `sud`.
syntax "sud(" term,* ")" : term
macro_rules
  | `(sud($n, $x, $y)) => `(sud $n $x $y)

-- Use it immediately.
def twentySeven : Nat :=
  sud(1, 2, 3)
```

Extensible syntax is powerful. It enables defining embedded domain-specific languages (EDSLs) such as React-inspired JSX[22] or a shallow embedding of C[23] and using them right away, even in the same file. On the other hand, it complicates the theoretical description. Since extensions modify the language, there isn't actually a single LeanLang. Instead, a function exists mapping any environment e to its LeanLang(e) determined by the extensions registered in e. Pathological LeanLangs can arise from the capacity to override pieces of basic core syntax. For instance, we can transform anonymous functions into just their bodies.

```
macro_rules
  | `(fun $_ => $t) => `($t)
```

[20] https://mathoverflow.net/a/376973/496340

[21] The actual frontend operates on an object of type Lean.Environment. This object stores a strict subset of what I am referring to as the 'environment' here; in particular, it does not store syntax extensions.

[22] https://github.com/leanprover/std4/pull/267

[23] https://github.com/tydeu/lean4-alloy

```
theorem wat : (fun _ => 27) = 27 :=
  rfl
```

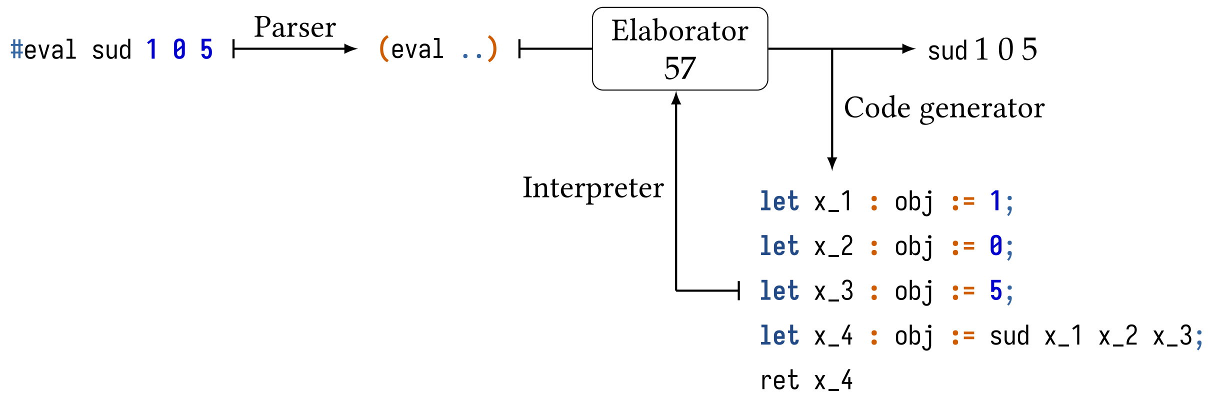
In contrast, LeanTT stays fixed. There is no way to modify kernel behaviour, providing a consistent logical baseline. This design is essentially that of meta-provers such as HOL Light: unlimited expressivity at the cost of having to maintain some discipline.

4.2 Code generation and metaprogramming

The following command that evaluates sud can appear in the same module as the definition of sud. It causes the frontend to helpfully print out the number 57.

```
#eval sud 1 0 5
```

In this section we further complicate the story, just enough to explain why and how `#eval` works, and then introduce Lean metaprogramming. The complication is in adding a third layer: besides the vernacular LeanLangs and the type theory LeanTT, there is an *intermediate representation* LeanIR of executable code.[24] The system includes a *code generator* capable of compiling any executable term of LeanLang to LeanIR. Code so produced can be immediately, dynamically executed in the *interpreter*. This is what happens while processing `#eval` commands. Code for the input term is generated and interpreted. The interpreter's outputs are returned to the elaborator and handled there in order to display them to the user (or show an error in case of interpretation failure). The IR is discarded afterwards. The sequence of events is displayed in fig. 4.2.



```
let x_1 : obj := 1;
let x_2 : obj := 0;
let x_3 : obj := 5;
let x_4 : obj := sud x_1 x_2 x_3;
ret x_4
```

Figure 4.2: The data flow throughout the execution of an `#eval` command. The vernacular syntax `#eval sud 1 0 5` is parsed and elaborated. Besides a term sud 1 0 5 of LeanTT, code in LeanIR is generated for the input term and interpreted. Results of interpretation are fed back to the elaborator.

In the case of `#eval`, the IR is ephemeral: generated, interpreted, and then discarded. Other frontend commands store the results of code generation persistently. Similarly to syntax extensions, code produced in this manner is appended to the environment. For instance, introducing a function via `def` has this effect. The earlier processing of `def sud` caused an executable representation of the Sudan function to be stored by the frontend. The saved LeanIR was used afterwards to evaluate it on concrete inputs, and could be used to compile the Lean

[24] In fact there are multiple IRs, but there is no need to complicate further still.

module to a native binary. There are also commands that persist `LeanTT` terms but are not code-generating, notably `theorem` declarations.

Elaborator reflection. We have seen that algorithms operating on object language entities are programmed — metaprogrammed — in the meta language of a given theorem prover. In a system whose meta and object languages are identified, the meaning of 'metaprogramming' is a little more specific, now referring to the capacity of the language to manipulate its *own* programs. This is sometimes referred to as *reflective* metaprogramming.

In Lean, metaprogramming is enabled by two features: the ability to evaluate Lean code while the frontend is processing a module, as well the fact that the frontend itself is implemented in Lean and can be imported as a library. A basic definition that supports metaprogramming is `Lean.Syntax`, the type of syntax trees. This type is extensible and can express terms of `LeanLang`(e) for *any* e. `Lean.Syntax` is imported by default. We may use it for example to manually write down the syntax of `2 + 2`.

```
open Lean in
def twoPlusTwo : Lean.Syntax :=
  mkNode `«term_+_»
    #[mkNode `num #[mkAtom "2"],
      mkAtom "+",
      mkNode `num #[mkAtom "2"]]
```

Of course writing parsed syntax by hand rapidly ceases being fun. To avoid it, the language comes equipped with a *quotation* (more precisely *quasiquotation*) operator that takes a piece of code and produces the corresponding syntax tree as a value. Quotations are written `` `(..) `` and expand to a value of type `Lean.Syntax` in an appropriate monad. The monad is necessary because parsing accesses the current environment in order to locate registered syntax extensions. To use quotations, we need to import a module from the Lean frontend.

```
import Lean.Meta.Basic

#eval (`(fun x => x) : Lean.MetaM Lean.Syntax)
```

The above code operates essentially identically to fig. 4.2. The principal difference is that while it is being executed in the interpreter, our monadic program `` `(fun x => x) `` invokes frontend components in order to process the quotation. Since the program was initially invoked *by* the frontend, the interaction becomes bidirectional and mutually recursive. In particular, the program being interpreted may modify the frontend state, for example by registering new entities within the environment. This enables support for yet another point of extension: the command `elab` extends `LeanLang` by a new command that, when encountered by the frontend, will be handled by invoking the metaprogram that we gave as input to `elab`. The snippet below exemplifies this.

```
import Lean.Elab

-- Register a new command.
elab "#greet " n:ident : command => do
  Lean.logInfo m!"Hello {n}!"

-- Use it. This prints "Hello world!".
```

```
#greet world
```

The general approach of allowing arbitrary subterms to invoke the elaborator in order to support metaprogramming was pioneered by Christiansen and Brady in Idris [32, 33, 31], as well as by Ebner and collaborators in Lean 3 [39]. It is known as *elaborator reflection*. More broadly, support for source code quotation and runtime evaluation of metaprograms was pioneered in LISP [67]. Modern languages with similar features[25] include Racket [42], Clojure [48], and Julia[26] [20].

To conclude the section, it's worth noting that the metaprogramming framework satisifies an equational law we might expect to hold: that quoting followed by evaluation is the identity transformation, at least if we exclude pathologically self-referential metaprograms that refer to their own `line:column` positions or concrete syntax. The following metaprogram which quotes and then elaborates a command has exactly the same effect (again up to any concrete syntactic metadata being stored) as just that command.

```
import Lean.Elab

-- Prints "So meta!".
#eval do
  let s ← `(command| #eval "So meta!")
  Lean.Elab.Command.elabCommand s
```

The above program can be read as "Parse the command `#eval "So meta!"` into its syntax tree and call that s. Process s in the frontend using `elabCommand`, the function that normally processes commands.". This discussion is why I initially advocated for the procedural perspective over the declarative, viewing Lean for a moment as closer to an effectful language manipulating a persistent environment of pure functions than to a purely functional language. It appears more awkward to accommodate observations made here under the latter view.

4.3 Interactive theorem proving

The practice of theorem proving in Lean 4 is ultimately not too dissimilar to other ITP systems.

The system is foundationally an intensional type theory and supports its two styles of proof refinement. It is established practice in the community to write definitions in term-mode, but prove most theorems in tactic-mode. Utilizing syntactic extensions, custom syntax mirroring textbook mathematical notation is commonly registered.

No domain-specific language is mandated for authoring tactics: they are ordinary metaprograms that manipulate goal states in the `Lean.Elab.Tactic.TacticM` monad, aligning the tactic framework in spirit with its early predecessors in the LCF and HOL family.

There does exist a sublanguage for *using* tactics, i.e., for writing tactic scripts. This language achieves a semi-declarative style of proof script along the lines of Mizar [100] and Isabelle/Isar [104]. Let us briefly illustrate some of its features. For the sake of example, define a `Nat'` that does not clash with the builtin type of natural numbers, as well as addition and introduce an infix notation for it.

[25] Such languages are sometimes described as *homoiconic* or supporting "code as data". However as van Schelven [89] points out, these terms are quite ambiguous and may be better avoided.

[26] `https://docs.julialang.org/en/v1/manual/metaprogramming/`

```
inductive Nat' where
  | zero : Nat'
  | succ : Nat' → Nat'
open Nat'

def add : Nat' → Nat' → Nat'
  | n, zero => n
  | n, succ m => succ (add n m)

infixl:65 " + " => add
```

Next, we prove a basic fact using tactics.

```
@[simp]
theorem succ_add (n m : Nat') : (succ n) + m = succ (n + m) := by
  induction m <;> simp [add, *]
```

The script starts off using `induction` on `m`. This produces two subgoals: the base and inductive cases. Then we use `s <;> t`, a tactic combinator which reads "apply the script `t` to all goals produced by `s`". We solve both goals using the simplifier `simp [add, *]`. The simplifier is an automated proof search tactic based on rewriting. It tries to rewrite by all lemmas tagged with the `@[simp]` annotation. Tagging `succ_add` means that it too may be applied in subsequent proofs.

The proof strategy looks like it will work more generally. We can abstract its *syntax* as a macro and use it to establish other facts.

```
macro "nat_induction" n:ident : tactic =>
  `(tactic| induction $n <;> simp [add, *])

@[simp]
theorem zero_add (n : Nat') : zero + n = n := by
  nat_induction n

theorem add_comm (n m : Nat') : n + m = m + n := by
  nat_induction n
```

Evaluation of tactic scripts is triggered in the frontend upon encountering the `by` keywords. It follows the metaprogram interpretation strategy of fig. 4.2 analogously to `#eval`.

In the next chapter we introduce, and innovate on, the Lean 4 user interface.

5 An extensible user interface for Lean 4

5.1 Introduction

Interactive theorem proving (ITP) distinguishes itself from other approaches to formal methods by structuring proof construction as a feedback loop between a human and a machine. Whether by filling typed holes in a partial term (Agda, Idris) or by issuing meta-level instructions in a tactic-based framework (HOL, Isabelle, Coq), users tend to develop proofs incrementally. At each step, the system displays the *goals* which remain to be proven and the user responds with a further refinement of their proof until there are no more goals left. This loop can be viewed as a dialogue between the user and the ITP system. Yet compared to human-to-human communication, modes of human-computer interaction available in today's general-purpose theorem provers are limited in *form* and in *referentiality*.

They are limited in **form** by being exclusively text-based. Text serves its purpose well: it is simple to process, supported in every system configuration, and universally understandable. Nevertheless, textual representations are only one way of displaying formal processes, statements, and their proofs. Cognitive science researchers have long suspected that *external representations* of concepts and objects outside the mind (for example a drawing on a piece of paper, or the physical disks in a Tower of Hanoi puzzle), complementing *internal* representations within the mind, are not merely an aid but rather an integral component of cognition [107]. Restricting the external representations available in ITP systems to only be text is thus a restriction on the way we think [95]. For instance, diagrammatic representations group related information together in ways that sequences of words simply cannot [53]. Since mathematicians and computer scientists rely on graphical calculi and processes such as diagram chases [99], computer mathematics should naturally support graphical representations.

Interactions are furthermore limited in **referentiality** in that we cannot refer to the objects that a displayed representation signifies by interacting with it directly. This is because the representations do not "remember" what they are representations *of*. Suppose for example that Alice and Bob are collaborating on a proof, using natural language and a blackboard. Suppose Bob attempts to commute x past y in the ring R but Alice notes that this cannot be done because R is not known to be commutative and one may not assume that $x \cdot y = y \cdot x$. At this point, Bob may respond by *referring* directly to R or to the term $x \cdot y$ and asking Alice for further facts about these objects in order to understand the issue and make progress on the proof. This illustrates that in dialogue, it is natural to request actions on an object under consideration by referring to it; dialogue is referential.

But replace Alice with an ITP system and suppose the corresponding message from Alice to Bob is that an instance of the `CommRing` typeclass couldn't be synthesized for the type `R`. To obtain detail on why this failed, the best Bob can generally do is copy-paste the offending type into a separate command, either to re-run the failing operation with more verbose output settings, or to print some extra information about it. Such interruptions are a source of friction which obstructs reasoning about the mathematical objects in question. Copy-pasting is only necessary because the displayed typeclass synthesis error is inert text which has "forgotten" details of the failure. The ITP feedback loop is thus not so much a dialogue as it is a sequence of disjoint request-response pairs. Had the system stored an association between the displayed error and input data involved in the failure instead, Bob would be able to inspect this data by interacting with the error message directly.

Failure of referentiality extends beyond the proof refinement loop, generally limiting the amount of information carried by messages originating in all components including parsing, type inference, proof search, decision procedures, and so on. Since in contemporary proof assistants these components assemble into deep and interconnected stacks, understanding the behaviour of any single component (not to mention emergent phenomena arising from multiple components in combination) can be a serious challenge.

We will show that simply keeping better track of references can improve the state of things. Following Ciccarelli [34, 30], we call reference-preserving UI elements **presentations**. A presentation is a visual or textual display D of an object X with a link back from D to X. Thanks to the link, the *presented* object X can be acted upon in various ways by interacting with D. In our example, Bob could interact with a presentation of the typeclass inference error (by clicking on it or using another input device) in order to obtain more information about `R` or `CommRing`, to jump to their definitions, or to carry out other operations on them. Failure of referentiality can be restated as noting that some UI element is not a presentation.

5.1.1 Contributions

We report on the design and implementation of a user interface (UI) for the Lean 4 theorem prover [71], of an associated `ProofWidgets 4` library of UI components[27], as well as of supporting features in the metaprogramming framework and in the prover itself. Our system aims to enable more natural and efficient interactions with the prover by combining the following features:

- **Displays of arbitrary form.** We build on HTML5 and the web platform as the underlying technology to make visualization easier. Packages from the rich JavaScript ecosystem may be imported and used in the UI. For instance, in section 5.3.1 the Penrose [105] library is used to visualize mathematical objects.
- **Referential presentations.** UI components keep track of, and may act on, the objects they signify. For example, expressions displayed in the UI can be hovered over to see their types and explicit forms (section 5.2.1); and goal states can be interacted with in order to make progress on proofs (section 5.3.3).
- **User-extensibility with reusable components.** The interface can be modified and extended by users, in Lean itself and in JavaScript. Builtin and user-defined components may be composed in arbitrary ways.
- **Live, interactive displays.** UI components can be used immediately, in the same Lean file they are defined in, with changes reflected in the UI in real-time.
- **On-demand computation**. Our presentations are *reactive* in that they compute lazily, in reaction to requests from the user. We can explore large objects such as computation traces (section 5.2.1) by displaying only the relevant parts without processing the rest.
- **General-purpose design.** Like Lean itself, the UI and `ProofWidgets` are not tailored for any specific domain. They enable a variety of applications besides logical reasoning such as plotting, 3D visualisation, and interactive simulations.

While interfaces supporting subsets of the above have been developed, our system appears to be the first to support all of them in a cohesive way. We give a detailed comparison to other systems in section 5.5. The UI is part of the core Lean distribution and has been deployed

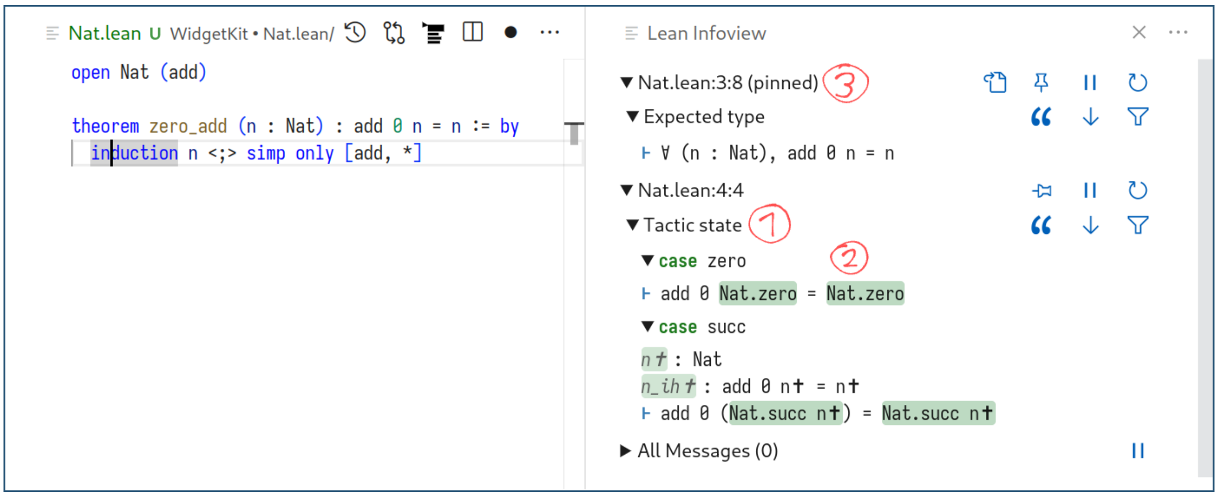
[27] `https://github.com/EdAyers/ProofWidgets4/tree/itp23`

widely to hundreds of active users, whereas the ProofWidgets package can be imported for additional functionality. The UI has been integrated in the VS Code extension vscode-lean4[28] as well as in the Lean 4 Web[29] online editor.

Outline. In section 5.2, we introduce the user interface and its interactive features. In section 5.3, we demonstrate how to extend the interface by means of several examples. In section 5.4, internals of the system and aspects of implementation are discussed. We cover related work in section 5.5 and conclude in section 5.6.

5.2 The user interface

The layout of the Lean 4 user interface does not diverge from the two-pane view of the world popularized by ProofGeneral [8]. In this layout, the first pane in the prover UI is a text editor with the proof script, whereas the second **infoview** pane displays additional information. This includes the current goal state, errors, and messages for the open buffer. All the UI components and extensions which we will discuss are displayed within the infoview. An example infoview state is shown in fig. 5.1.



```
open Nat (add)

theorem zero_add (n : Nat) : add 0 n = n := by
  induction n <;> simp only [add, *]
```

```
▼ Nat.lean:3:8 (pinned)
  ▼ Expected type
    ⊢ ∀ (n : Nat), add 0 n = n
▼ Nat.lean:4:4
  ▼ Tactic state
    ▼ case zero
    ⊢ add 0 Nat.zero = Nat.zero
    ▼ case succ
    n✝ : Nat
    n_ih✝ : add 0 n✝ = n✝
    ⊢ add 0 (Nat.succ n✝) = Nat.succ n✝
▶ All Messages (0)
```

Figure 5.1: The Lean infoview embedded in vscode-lean4. Two tactic-mode goals (Tactic state) at the text cursor are shown (1). Differences in the goals' types and local contexts with respect to the previous state are highlighted (2). A second location containing a term goal (Expected type) is pinned (3).

While the layout is as in ProofGeneral, we do not follow its *waterfall* style of proof script management. In the waterfall style, there is a *checkpoint* to separate the part of the document which had been checked by the prover from that which had not. The checkpoint is advanced manually as an intentional action by the user. It recedes when changes are made to the checked part. Instead, similarly to Isabelle/PIDE [103], Lean adopts a 'stateless' approach that checks the entire buffer in real-time. Under the hood, the system keeps track of immutable snapshots of past and present versions of the document, with new snapshots generated whenever the user edits the script. Contents of the infoview are determined by the latest snapshot and the current text cursor position.

[28]https://github.com/leanprover/vscode-lean4/
[29]https://lean.math.hhu.de/

When the cursor is inside a tactic-mode proof, the goal state at that position is displayed. In tactic proofs, differences between subsequent goal states are highlighted in green or red depending on whether a subexpression was just added or is about to be removed, respectively. This can be useful to see at a glance how a step has impacted, or will impact, the proof state. For instance when proving `∀ (n : Nat), 0 + n = n` by induction on `n`, in the base case `n` becomes `Nat.zero` and this change is highlighted as in fig. 5.1. The diff is computed using a heuristic algorithm operating on kernel-level expression trees. Furthermore when the cursor is over a typed hole (or a finished term), the *term goal* is also displayed. The term goal is the expected (or actual) type and local context of the typed hole (or term). One advantage of the waterfall approach is that the checkpoint can be used as an additional cursor which displays the goal state in one part of the file while we go on to work on another part. We generalize this by allowing one or more text locations to be *pinned* in the infoview. Information about pinned locations is displayed alongside information about the text cursor location. Pinned displays update in real-time which is especially useful to see how changes at one point in the file affect a proof state or evaluation further down.

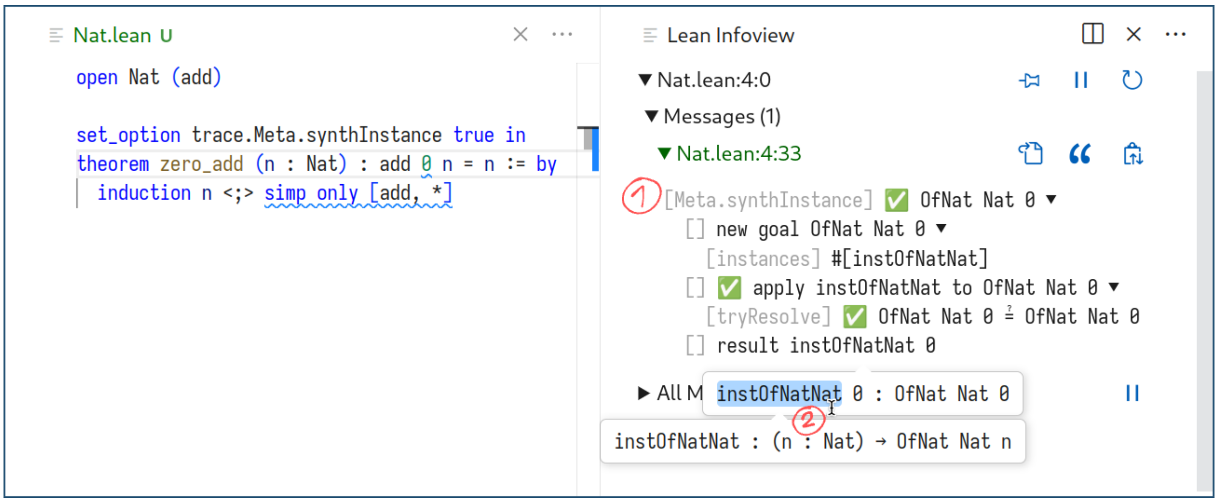
5.2.1 Expression and trace presentations

The infoview's design aims to support pervasive interactivity by displaying most objects as presentations. For instance, every displayed expression, and each of its subexpressions, stores a reference to the type-theoretic term it corresponds to. This can be used to learn additional facts about an expression appearing anywhere in the infoview (in a goal state or an error message or a custom component) by clicking on it or hovering over it as in fig. 5.2. Users can learn expressions' types, see the values inferred at implicit arguments, and jump to symbols' definitions. In this way presentations **increase information locality** by making it retrievable alongside a display of the relevant object. No extra data is computed eagerly; pretty-printing the type of every subexpression, for example, would not be cheap in any sizable goal state. Instead, the link from presentation to underlying object is a memory reference which enables the UI to fetch information from the language server lazily when the user requests it (see section 5.4).

One way to frame the addition of presentations is as a kind of refinement process. We imagine starting from a non-referential user interface appearing in a particular scenario. We then ask:

- Which objects are signified by which parts of the UI?
- Given that UI D signifies object X, which actions applicable to X could we carry out using D?

Guided by the answers, we can enrich interfaces for programming and proving with new interaction points. Consider messages produced by the prover: in Lean, *structured traces* are a feature of the metaprogramming API which collates messages produced during program execution into a tree-shaped record, with edges corresponding to user-defined execution boundaries. For example, the backtracking Prolog-like typeclass search procedure [90] of Lean 4 can be traced, with branches representing attempted and abandoned instances. Many search-based tactics produce traces. Traces of expensive procedures can have thousands of nodes, making them unreadable and slow to pretty-print if displayed in full. Similarly to inferring expression types in the UI, we solve this problem by expanding and pretty-printing subtraces lazily, in reaction to user requests. This means we can explore branches through large trace trees limited only by the memory needed to store the trace data rather than the CPU time



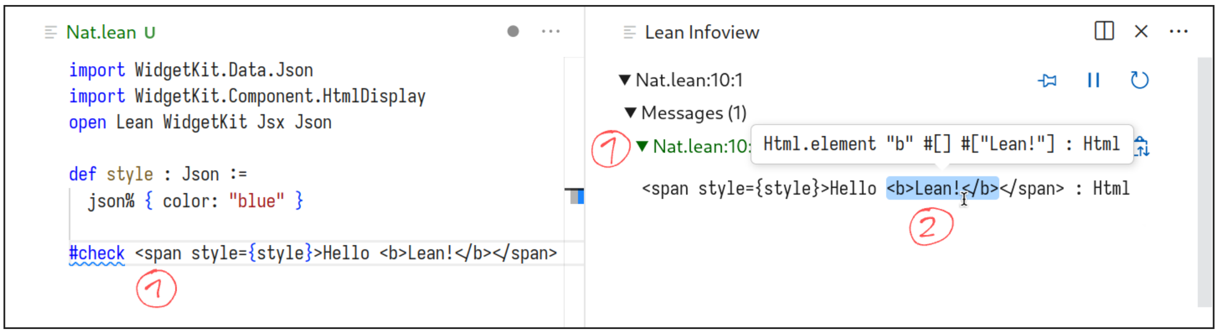
```
open Nat (add)

set_option trace.Meta.synthInstance true in
theorem zero_add (n : Nat) : add 0 n = n := by
  induction n <;> simp only [add, *]
```

```
▼ Nat.lean:4:0
 ▼ Messages (1)
  ▼ Nat.lean:4:33
   [Meta.synthInstance] ✅ OfNat Nat 0 ▾
     [] new goal OfNat Nat 0 ▾
       [instances] #[instOfNatNat]
     [] ✅ apply instOfNatNat to OfNat Nat 0 ▾
       [tryResolve] ✅ OfNat Nat 0 ≟ OfNat Nat 0
     [] result instOfNatNat 0
▶ All M  instOfNatNat 0 : OfNat Nat 0
 instOfNatNat : (n : Nat) → OfNat Nat n
```

Figure 5.2: The numeral notation `0 : Nat` is resolved via typeclass search. A structured trace (1) of the search is explored. A presentation of a pretty-printed typeclass instance is clicked on to display its type (2). Subexpressions within an expression can be selected following its tree structure.

needed to pretty-print it all. In fig. 5.2, an example trace of typeclass instance search is shown. Presentations compose so that the structured trace may contain interactive expressions and other interactive components. In the future we hope to also provide a method of filtering and searching through the trace tree.

Presentations interact well with other language features including syntax extensions. In fig. 5.3, an embedded domain-specific language (EDSL) is used to write down an HTML tree. The tree has an underlying expression of type `Html` which is presented in the infoview using the same EDSL.



```
import WidgetKit.Data.Json
import WidgetKit.Component.HtmlDisplay
open Lean WidgetKit Jsx Json

def style : Json :=
  json% { color: "blue" }

#check <span style={style}>Hello <b>Lean!</b></span>
```

```
▼ Nat.lean:10:1
 ▼ Messages (1)
  ▼ Nat.lean:10:  Html.element "b" #[] #["Lean!"] : Html
   <span style={style}>Hello <b>Lean!</b></span> : Html
```

Figure 5.3: A JSX-like syntax for writing HTML trees inline is used to write down a term of type `Html` in the editor. The `#check` command is used to inspect it in the infoview (1). The type `Html` has an associated pretty-printer which emits the same custom syntax. The pretty-printer sub-output `<b>Lean!</b>` is a presentation of the subterm `Html.element "b" #[] #["Lean!"]` which can be inspected by hovering over it with the mouse (2).

Since presentations are the default, producing them requires no extra effort from the tactic writer. For example, the following snippet defines and then uses a custom command with interactive output. It does this by first using `elab`, a meta-level command that defines new commands with a given syntax, in this case `#check_nat t` where `t` can be any term. The new command is immediately available for use and is invoked with `37` as input.

```
import Lean.Elab.Command
open Lean Elab Command

elab cmd:"#check_nat " t:term : command => liftTermElabM do
  let e : Expr ← Term.elabTerm t (mkConst ``Nat)
  -- The string-like literal m!".." directly embeds expressions {..}.
  logInfoAt cmd m!"{e} has type {mkConst ``Nat}"

#check_nat 37
```

The implementation of `#check_nat` parses and typechecks the term, expecting its type to be `Nat`, and then emits a message. It does this using `logInfoAt` which associates a message with a syntactic span, in this case the span of the `#check_nat` keyword. Just like standard errors and warnings associated with a syntactic range, the message is displayed in the infoview whenever the text cursor is on this span. Since the message directly stores kernel-level expressions (of type `Expr`), they are automatically displayed as interactive presentations.

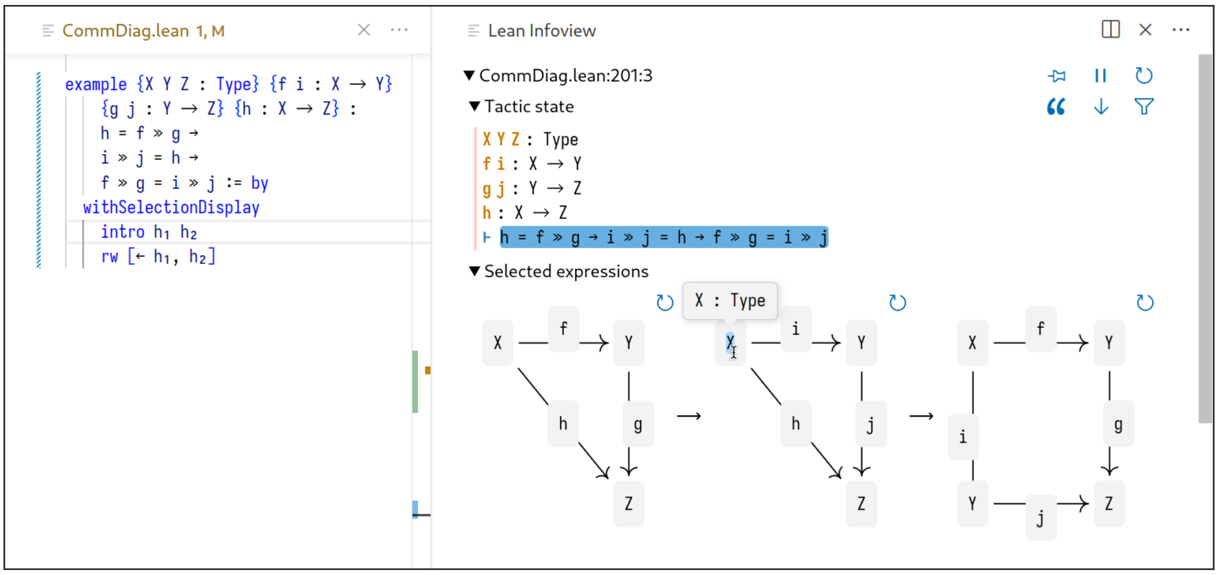
5.3 `ProofWidgets 4`: programmable, referential interfaces

While the builtin presentations of expressions, goals and messages provide a common interface for all uses, the design's main strength is its extensibility and composability. Users can build domain-specific interfaces dubbed **user widgets**. A user widget is a ReactJS UI component capable of invoking Lean metaprograms and editing the proof script. User widgets can implement new presentations and new ways of interacting with the prover. User widgets are usually displayed by related tactics or commands – for example the HTML display in fig. 5.5 is stored by the `#html` command. Storing a widget is analogous to how messages are emitted with `logInfoAt`: informally, instead of stating "there is an error or warning at this syntactic span", we state "there is a user widget at this syntactic span". Both the user interface and the associated tactic code can be developed in tandem alongside each other, allowing for quick development cycles.

In this section we will consider user widgets that extend the goal display in various ways. Here referentiality – the idea that displays should store references to objects they signify – is also core to our approach. Recall that the object displayed by an expression presentation (section 5.2.1) is an expression together with its local context (approximately corresponding to a judgment $\Gamma \vdash t : T$ of the type theory). Executing with access to that allows us to, for example, infer its type and display it to the user. Similarly, widgets extending the goal display can reference the current goal state.

5.3.1 Diagrams for algebra

In fig. 5.4 the goal is an implication between statements in the language of category theory. We choose to display it as commutative diagrams connected by implication arrows. Here our support for importing JavaScript libraries shines – while it may seem like a trivial engineering choice, the ability to build on the immense NPM software ecosystem dramatically cuts down development time. One such library, Penrose [105], expresses general mathematical diagramming as an optimization problem. The user writes a specification describing which shapes the diagram should include (in `dsl` and `sub` files) as well as which constraints on their layout will make the diagram sound and beautiful (in a `sty` file). An energy minimization solver then runs and an SVG image is generated. The `ProofWidgets` component wrapping Penrose is composable in that it may include further components (in fig. 5.4 labels on objects and morphisms are interactive expression components) and dually may become part of a larger display. We hope it will prove useful to working algebraists. While the display demonstrated here does not act on the goal, proof methods such as diagram chases could also be implemented with `ProofWidgets`. We expand on this in section 5.3.3.


CommDiag.lean 1, M

```
example {X Y Z : Type} {f i : X → Y}
    {g j : Y → Z} {h : X → Z} :
    h = f ≫ g →
    i ≫ j = h →
    f ≫ g = i ≫ j := by
  withSelectionDisplay
    intro h₁ h₂
    rw [← h₁, h₂]
```

Lean Infoview

▼ CommDiag.lean:201:3

▼ Tactic state

```
X Y Z : Type
f i : X → Y
g j : Y → Z
h : X → Z
⊢ h = f ≫ g → i ≫ j = h → f ≫ g = i ≫ j
```

▼ Selected expressions

Figure 5.4: A target type in the language of category theory is selected. The statement is displayed as a sequence of commutative diagrams connected by implication arrows.

From the user's perspective, implementing a display such as this one proceeds in two steps. First, we wrap Penrose into a reusable `ProofWidgets` component. The Lean definition of the `PenroseDiagram` component is as follows[30]:

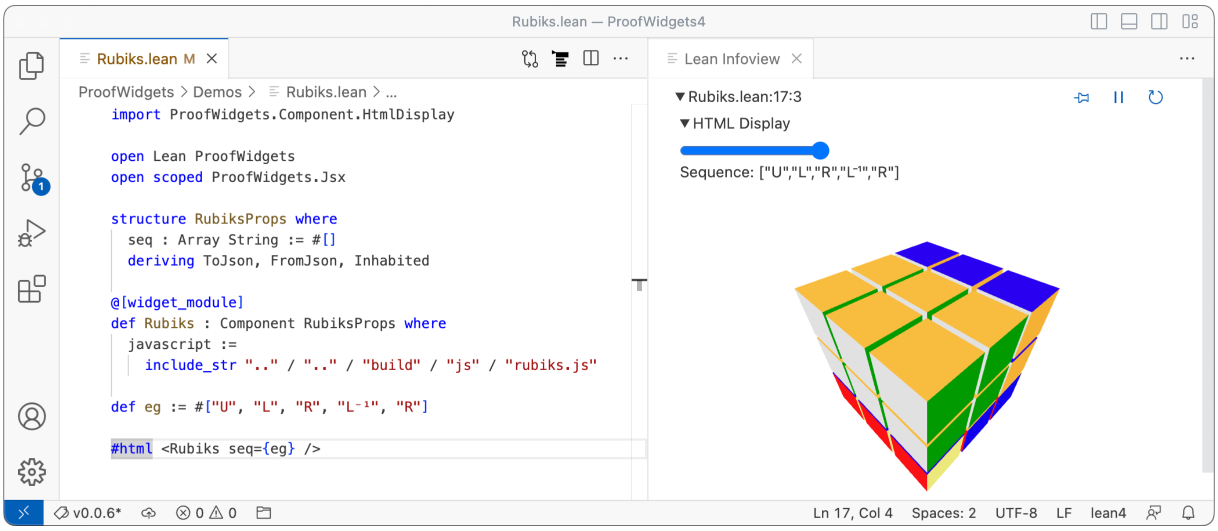
```
structure PenroseDiagramProps where
  embeds : Array (String × EncodableHtml)
  dsl    : String
  sty    : String
  sub    : String
  deriving RpcEncoding

@[widget_module]
def PenroseDiagram : Component PenroseDiagramProps where
  javascript := ... -- Details omitted
```

Values of type `Component Props` serve to encapsulate JavaScript user widget implementations as Lean definitions. The index type `Props` specifies a Lean encoding of the type of data expected by the component. In this case `Props = PenroseDiagramProps` contains fields describing a specific diagram (`dsl` / `sty` / `sub`) as well as other widgets to nest within it (`embeds`). To give another example, one variant of the interactive expression component has type `Component ExprWithCtx` where `ExprWithCtx` is an expression together with its local context.

[30]Details are highly likely to change as the library evolves.

The field `javascript` contains a JavaScript implementation of the component. To a first approximation, it could be viewed as having dynamic type `Props → HTML`. It may be written inline but it is preferrable to point at a file on disk. In the latter case one may use tooling we have developed to integrate building TypeScript files into the build of a Lean package using the Lake (Lean Make) build system. Communication with the infoview is set up using the `@[widget_module]` attribute and the `deriving RpcEncoding` annotation. `@[widget_module]` saves the JavaScript code in a global storage from which it can be retrieved for execution in the infoview, whereas `deriving RpcEncoding` generates code to serialize and deserialize values of a type, in this case `PenroseDiagramProps`. This is necessary to support distributed computation (see section 5.4).

More complex visualizations are enabled by building on further JavaScript libraries as in fig. 5.5. For example, a component integrating a plotting library could be a starting point for plotting functions in a formally verified way [69]. Finally, we note that this first step of wrapping JavaScript functionality in a `Component` can be skipped when the necessary UI component already exists. Thus it is desirable to write reusable components. For instance, `PenroseDiagram` is not specific to algebra but supports general constraint-based diagramming; we use it again in fig. 5.7.



Rubiks.lean — ProofWidgets4
Rubiks.lean M
Lean Infoview
ProofWidgets > Demos > Rubiks.lean > ...

```
import ProofWidgets.Component.HtmlDisplay

open Lean ProofWidgets
open scoped ProofWidgets.Jsx

structure RubiksProps where
  seq : Array String := #[]
  deriving ToJson, FromJson, Inhabited

@[widget_module]
def Rubiks : Component RubiksProps where
  javascript :=
    include_str ".." / ".." / "build" / "js" / "rubiks.js"

def eg := #["U", "L", "R", "L⁻¹", "R"]

#html <Rubiks seq={eg} />
```

```
▼Rubiks.lean:17:3
 ▼HTML Display
 Sequence: ["U","L","R","L⁻¹","R"]
```

Figure 5.5: The `Rubiks` component loads the `three.js` library in order to create a 3D visualization of a Rubik's cube. An HTML tree `<Rubiks seq={eg} />` containing an instance of this component is passed to the `#html` command. This command can be used to render HTML trees in the infoview with a user widget (HTML Display). The sequence of rotations `eg` is determined by the Lean script.

In the second step, we write a Lean metaprogram to display the user widget. There are many ways to do this in general. Since fig. 5.4 uses an *Expr presenter*, we will describe this approach. Like most provers, Lean features an *elaborator* which translates surface-level (*vernacular*) syntax into fully explicit terms of the underlying type theory by filling in implicit arguments, finding typeclass instances, resolving ambiguous notation, inserting coercions, and so on. Lean 4 also contains a *delaborator* which essentially does the inverse – it attempts to make an explicit term human-readable by heuristically removing detail while ensuring that the elaborator can still process the resulting vernacular. Eliding detail, the delaborator has type `Expr → MetaM Term` where `Expr` is the type of kernel terms, `Term` the type of abstract syntax trees corresponding to vernacular terms, and `MetaM` an appropriate monad. By composing with a pretty-printer for syntax trees we get the full pretty-printer of type `Expr → MetaM String`.

An **`Expr` presenter** is a `ProofWidgets` metaprogram which can be viewed as one generalization of the above process. Rather than producing strings, we output HTML trees which may include user widgets. As the name suggests, it is aimed at producing presentations of mathematical objects. The set of `Expr` presenters is user-extensible. We dispatch to the appropriate one based on characteristics of the given `Expr` such as using a known constant at the top level. This echoes the general design philosophy of Lean 4 as a tower of abstractions: some uses of `ProofWidgets` are expressed mostly simply by writing an `Expr` presenter, and for those that are not it is possible to drop to a lower level of abstraction.

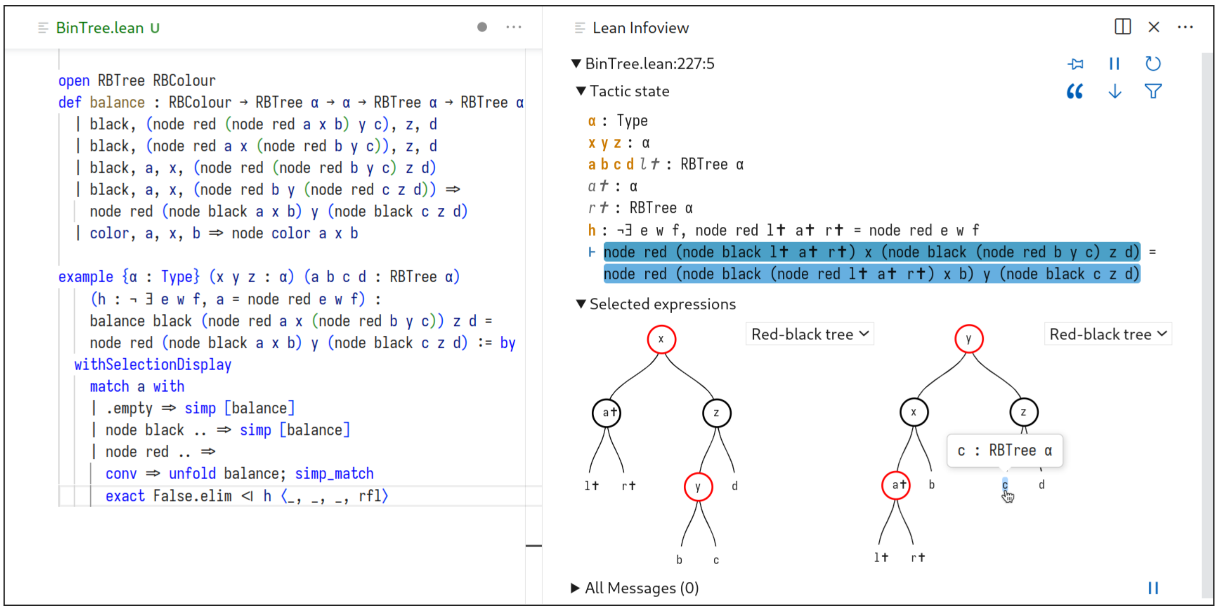
To use this framework in our example, we wrote a Penrose specification for general commutative diagrams, as well as an `Expr` presenter that translates equalities of morphisms in a category into diagram descriptions which use that specification. A representative code fragment follows.

```
/-- Expressions to display as labels in a diagram. -/
abbrev ExprEmbeds := Array (String × Expr)

open scoped Jsx in
def mkCommDiag (sub : String) (embeds : ExprEmbeds) : MetaM EncodableHtml := do
  -- Pretty-print kernel terms into interactive labels for the diagram.
  let embeds ← embeds.mapM fun (s, h) =>
    return (s, EncodableHtml.ofHtml
      <InteractiveCode fmt={← Widget.ppExprTagged h} />)
  return EncodableHtml.ofHtml
    -- Instantiate a PenroseDiagram using a JSX-like EDSL.
    <PenroseDiagram
      embeds={embeds}
      -- Penrose specification of general commutative diagrams.
      dsl={include_str "commutative.dsl"}
      sty={include_str "commutativeOpt.sty"}
      -- The particular diagram we are given.
      sub={sub} />

... -- Definitions of commSquareM? and commTriangleM? elided

/-- Present an expression as a commutative diagram. -/
@[expr_presenter]
def commutativeDiagramPresenter : ExprPresenter where
  userName := "Commutative diagram"
  present type := do
    -- Attempt to deconstruct `type` into a commutative square or triangle
    -- and use `mkCommDiag` if successful.
    if let some d ← commSquareM? type then
      return some d
    if let some d ← commTriangleM? type then
```

```
      return some d
    return none
```

5.3.2 Selection contexts

On a blackboard, we can underline and point to expressions and objects in order to highlight the relevant parts of a formula or depiction when explaining an argument. Analogously, a **selection context** is a subset of (subexpressions of) goals, hypotheses, and (subexpressions of) hypothesis types appearing in a goal state. The user specifies it by shift-clicking on the respective elements in the infoview. The current selection context is passed as input to user widgets that pertain to the goal. In fig. 5.4 just the target type was selected. The `withSelectionDisplay` combinator, which we use there and in fig. 5.6, is a tactic combinator that associates a general-purpose widget with the range of the entire nested tactic script, and then runs the script unchanged. The widget displays each selected expression using registered `Expr` presenters (if multiple presenters apply, a choice can be made in the UI).



```
open RBTree RBColour
def balance : RBColour → RBTree α → α → RBTree α → RBTree α
  | black, (node red (node red a x b) y c), z, d
  | black, (node red a x (node red b y c)), z, d
  | black, a, x, (node red (node red b y c) z d)
  | black, a, x, (node red b y (node red c z d)) ⇒
    node red (node black a x b) y (node black c z d)
  | color, a, x, b ⇒ node color a x b

example {α : Type} (x y z : α) (a b c d : RBTree α)
    (h : ¬ ∃ e w f, a = node red e w f) :
    balance black (node red a x (node red b y c)) z d =
    node red (node black a x b) y (node black c z d) := by
  withSelectionDisplay
    match a with
    | .empty ⇒ simp [balance]
    | node black .. ⇒ simp [balance]
    | node red .. ⇒
      conv ⇒ unfold balance; simp_match
      exact False.elim <| h ⟨_, _, _, rfl⟩
```

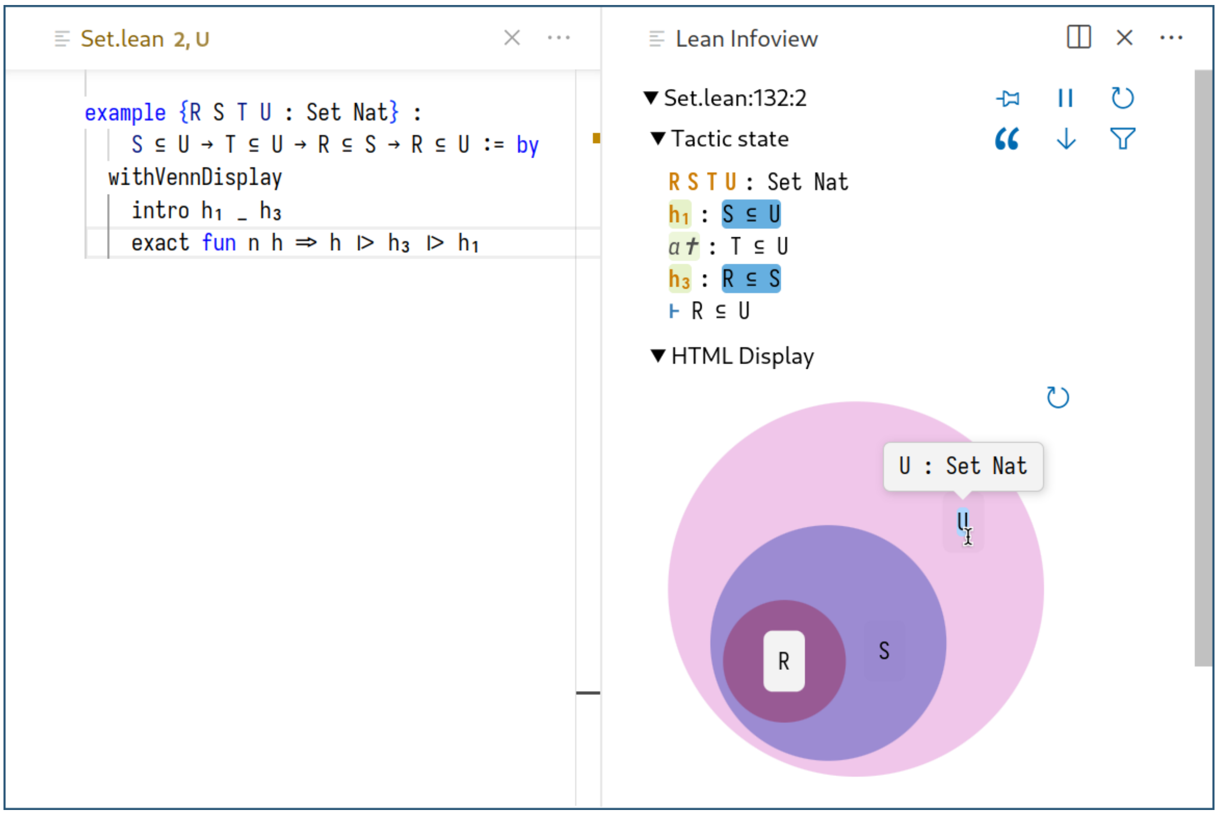
```
▼ BinTree.lean:227:5
▼ Tactic state
  α : Type
  x y z : α
  a b c d l† : RBTree α
  a† : α
  r† : RBTree α
  h : ¬∃ e w f, node red l† a† r† = node red e w f
  ⊢ node red (node black l† a† r†) x (node black (node red b y c) z d) =
    node red (node black (node red l† a† r†) x b) y (node black c z d)
▼ Selected expressions
  c : RBTree α
```

Figure 5.6: A balancing function for red-black trees is implemented in `balance`. Two terms appearing in the course of a proof about it are selected in the goal and illustrated as trees.

Selecting more than one subexpression can be helpful in comparing differences between these subexpressions, to figure out what remains to be proven. In fig. 5.6, we copied a balancing function for red-black trees verbatim from Okasaki [77]. As it turns out, due to overlapping patterns in the definition of `balance`, the reduction law one might expect does not hold in all cases. It does hold when `balance` is called after inserting one node into a well-formed red-black tree because in that case, the invariants ensure that no more than one red-red edge exists. In fig. 5.6, we can see at a glance from their visual representations that the two selected trees cannot be equal, so an invariant must have been violated. In this way diagrams appearing live during proof development serve as cognitive aids. The visualization of general red-black trees uses the `react-d3-tree`[31] library to do most of the heavy lifting and took less than an hour to prototype. Afterwards, figures from Okasaki's paper are reproduced by the system with no further effort.

[31]`https://github.com/bkrem/react-d3-tree`

Finally, rather than using `withSelectionDisplay` which treats elements of the selection context as independent, users may choose to visualize the selection context as one entity. This is useful when the global information contained therein can be coherently diagrammed. In fig. 5.7, two subset relations are relevant to the proof whereas a third one is not. We use `PenroseDiagram` together with Penrose's builtin support for Venn diagrams to display the two relations which imply the conclusion. The combinator `withVennDisplay` used here works similarly to `withSelectionDisplay` except in that, rather than emitting the general-purpose selection display widget, it produces an instance of a Venn diagram specifically.



```
example {R S T U : Set Nat} :
  S ⊆ U → T ⊆ U → R ⊆ S → R ⊆ U := by
withVennDisplay
  intro h₁ _ h₃
  exact fun n h => h |> h₃ |> h₁
```

```
R S T U : Set Nat
h₁ : S ⊆ U
a✝ : T ⊆ U
h₃ : R ⊆ S
⊢ R ⊆ U
```

Figure 5.7: A subset of hypotheses relevant to the proof is selected. The set relationships are visualized in one Venn diagram.

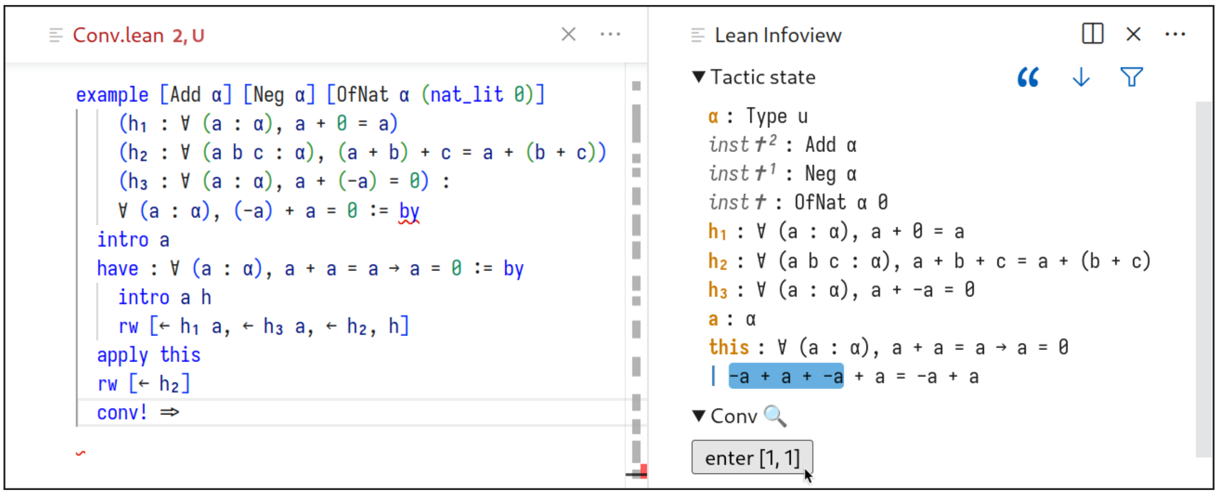
5.3.3 Contextual suggestions and graphical calculi

Beyond providing static displays of goal states guided by the selection context, user widgets may invoke Lean metaprograms, access proof states, and edit the proof script. Since metaprograms can also display user widgets, the link between widgets and metaprogramming is bidirectional. It is possible to make progress on proofs through the UI.

One application of this functionality could be *proof by pointing* [18, 19] which, to a first approximation, demands that the UI should allow guiding proof synthesis by pointing (with a mouse, for example) at the term to use, decompose, or otherwise manipulate in the next proof step. Since the selection context already contains terms which the user pointed out, a proof by pointing widget would only need to respond to clicks by inserting appropriate tactics into the proof script. On the other hand Paulson argues [81] that certain specific variations of this idea, such as guiding term rewriting by hand, are better served by powerful automation. Ultimately some combination of both appears most likely to be useful. For example, a piece of

Sledgehammer-like automation [21], or a system based on recent advances in deep learning [52], could suggest proof steps that make progress on the proof in a manner related to the current selection context. `ProofWidgets` avoids committing to any single approach by remaining agnostic about which actions or graphical proof methods are available, instead leaving the choice to users and their particular applications. What we hope to achieve is to make the implementation of *any* such method as frictionless as possible by providing a library of basic components. We envision it being used for *contextual suggestions* and *graphical calculi*.

Contextual suggestions are provided by *suggestion providers*. These are metaprograms which, given a goal state and selection context, return a list of relevant or potentially useful tactics that the user may then pick from. For example, proof by pointing implementations could be viewed as suggestion providers which suggest tactics to carry out the desired goal transformation. Like the set of `Expr` presenters, the set of suggestion providers is user-extensible rather than fixed. In fig. 5.8 we demonstrate how a user widget presenting a suggestion can operate. In Lean, the `conv` tactic mode allows "zooming in" on a subexpression of the target or a hypothesis type in order to apply local transformations. In the figure, a suggestion provider returns a `conv` tactic which would put the selected subexpression in focus. The tactic is then displayed in the infoview and may be inserted by clicking the button.



```
example [Add α] [Neg α] [OfNat α (nat_lit 0)]
    (h₁ : ∀ (a : α), a + 0 = a)
    (h₂ : ∀ (a b c : α), (a + b) + c = a + (b + c))
    (h₃ : ∀ (a : α), a + (-a) = 0) :
    ∀ (a : α), (-a) + a = 0 := by
  intro a
  have : ∀ (a : α), a + a = a → a = 0 := by
    intro a h
    rw [← h₁ a, ← h₃ a, ← h₂, h]
  apply this
  rw [← h₂]
  conv! ⇒
```

```
α : Type u
inst✝² : Add α
inst✝¹ : Neg α
inst✝ : OfNat α 0
h₁ : ∀ (a : α), a + 0 = a
h₂ : ∀ (a b c : α), a + b + c = a + (b + c)
h₃ : ∀ (a : α), a + -a = 0
a : α
this : ∀ (a : α), a + a = a → a = 0
⊢ -a + a + -a + a = -a + a
```

Figure 5.8: A subterm `-a + a + -a` of the goal in a proof about groups [37] is selected. The `conv` user widget by Robin Böhne and Jakob von Raumer displays a button suggesting a tactic which would zoom in on the selected subterm. Clicking the button inserts the tactic into the proof script.

As we observed in section 1, diagrams serve as cognitive aids in a variety of mathematical pursuits. **Graphical calculi** are distinguished from general depictions by being *active*, meaning that manipulations of the depiction correspond to steps in a proof; *sound*, meaning that valid manipulations are valid proof steps; and ideally *complete*, meaning that every proof in a chosen class can be expressed graphically. Examples include the Reidemeister moves on knot diagrams [84], manipulations of string diagrams [49], or more specific variants in category theory such as ZX-diagrams [35], Globular proofs [12], and `homotopy.io` [85] proofs. A formalization of any of these graphical languages could be accompanied by a `ProofWidgets` component which translates manipulations of a graphical proof state in the infoview into tactic steps in the Lean proof script.

5.4 Implementation

A complete setup consists of three components. Figure 5.9 outlines an example interaction between them.

The language server is written in Lean. It communicates with the editor and with the infoview via the Language Server Protocol (LSP[32]). Through the LSP it provides standard code intelligence facilities – go-to-definition, type hovers, autocompletion, etc. Proof states and related objects such as terms of the type theory are stored in the server.

The infoview is written in TypeScript. It is a self-contained web application displayed by the editor. Client-side JavaScript code from user widgets executes here.

The text editor is chosen by the user. Besides storing the proof script, the editor connects to the server, manages the infoview, and mediates between them. To support both, the editor must be capable of communicating via LSP and displaying web content. For example, the Visual Studio Code extension `vscode-lean4` embeds the infoview in a webview pane which it has control over.

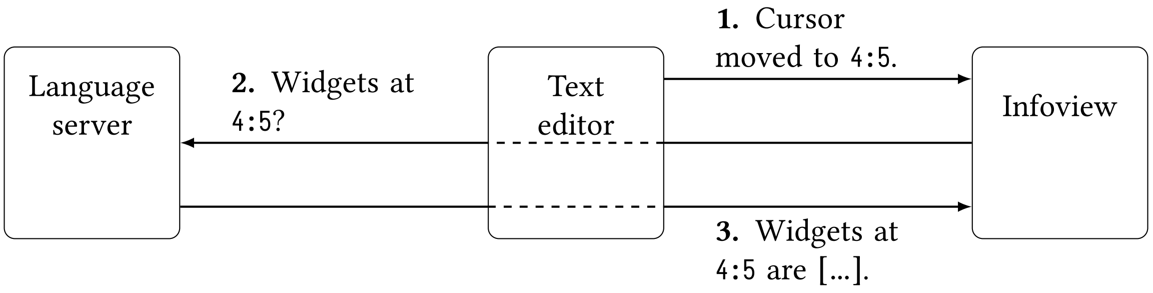


Figure 5.9: In order to determine which user widgets to show in the infoview, a sequence of messages is exchanged every time the text cursor moves. First, the editor informs the infoview about the new cursor position (here line 4, column 5). Then, the infoview queries the language server for the user widgets that should be shown at that position. Finally, the server replies with a list of user widgets. Its contents are then displayed in the infoview. The editor acts as a proxy for infoview–server communication, indicated by dashed lines.

Remote procedure calls. The infoview and the server are independent programs which may not even execute on the same computer. Indeed, this happens when Lean is used over SSH or in a cloud-based service such as Gitpod. In these cases, the editor and infoview execute on the user's local machine whereas the server is remote. Certain objects stored in the server's memory should not be serialized and sent to the infoview over the network due to their size – for instance, the environment (which stores known theorems, definitions, metadata, etc) can weigh several gigabytes in sufficiently large proof developments. Consequently, metaprograms which operate on heavy objects must execute in the server.

Nevertheless, user widgets need to run such metaprograms, for example to try a tactic or infer an expression's type: both of these need access to the environment. Therefore widgets must be able to invoke methods on the server. To enable this, we designed a foreign function interface for Lean with support for remote calls from JavaScript. The interface is effectively an extension to LSP. The LSP is based on JSON-RPC[33], a simple protocol for remote procedure calls which

[32] `https://microsoft.github.io/language-server-protocol/`
[33] `https://www.jsonrpc.org/`

encodes argument and output data as JSON. For example, to request a symbol's definition, the editor invokes the `textDocument/definition` method by sending a JSON record of the file, line, and column where an instance of the symbol occurs. The return value sent back by the server is the definition's location. To support arbitrary other functionality, we made the registry of procedures that can be invoked on the server via JSON-RPC user-extensible. To mark a Lean procedure as remotely callable, one annotates it with `@[server_rpc_method]`. A procedure so marked must be of the type `A → RequestM B` where `RequestM` is a monad with access to server state and `A,B` are JSON-serializable types. (De)serialization routines are autogenerated by annotating a type definition with `deriving ToJson, FromJson` or `deriving RpcEncoding`.

Remote references. When making multiple remote calls, widgets need to pass data between the metaprograms they invoke, for example to compute an expression's explicit form ef and then infer the type of a subexpression of ef (as do the two popups in fig. 5.2). Passing data is non-trivial because widgets execute on the client whereas metaprograms run on the server. Since the relevant data is not serialized, client-side code needs a way of referring to objects stored in the server's memory. This is achieved by allowing JSON-RPC payloads to contain references to server-side objects. Given a Lean type `α`, we write `WithRpcRef α` for the type of references to values of type `α`. Our implementation has special support for `WithRpcRef` markers which are serialized as UID strings. To ensure type correctness, i.e., prevent a reference of one type from being converted to a reference of another type, runtime type information is stored in the server and checked on any remote reference access.

Remote references are the backbone of our implementation of presentations. One use is found in expression presentations. Recall from section 5.3.1 that Lean features a *delaborator*, a system for converting kernel-level expressions back into syntax trees. To implement expression presentations, the delaborator has been extended with the ability to tag syntax subtrees with references to subexpressions of the original expression. These references are encoded using `WithRpcRef`. The exact tagging strategy has been described by Ayers and coauthors [11].

Allowing the client to refer to server-side objects presents us with a classic memory management problem – when is it safe for the server to delete objects for which remote references have been created? Conveniently, both Lean and JavaScript are garbage-collected languages. Using a `FinalizationRegistry`[34] we can instruct the JavaScript garbage collector to send a memory release instruction to the server when it collects the corresponding client-side reference. This is cooperative and may fail in case of client-side errors: a client which does not release server-side memory could cause it to leak. While we can't prevent this in general using only server-side mechanisms, we require the client to regularly send `keepalive` messages inspired by the Transmission Control Protocol [22]. Upon not seeing any `keepalive`s for sufficiently long, the server frees all remote references. This eliminates a class of disconnection- and hang-related memory leaks.

Dynamic code delivery. User widgets are required to be self-contained JavaScript modules[35]. This is considered a low-level target – users may employ any libraries and toolchains they need (for example TypeScript), as long as the eventual compilation or transpilation output matches the required format. Modules are registered in Lean using the `@[widget_module]` annotation. Upon being so annotated, modules are stored in a content-addressed cache ac-

[34] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/FinalizationRegistry
[35] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

cessible to the server. In order to display a particular user widget, the infoview fetches its source module from the cache using a remote procedure call, and then dynamically loads this source. User widgets execute in a runtime environment including the `@leanprover/infoview` library which they may import. This library exposes builtin functionality of the infoview (it can be used to display expression, structured trace, or goal presentations) as well as methods of communicating with the editor (these can be used for instance to edit the proof script or place another Lean file in focus) and services for making remote procedure calls from user widgets.

5.5 Related work

Our work descends directly from graphical tooling for Lean 3 (the previous version of Lean), notably `ProofWidgets 3` [11] and the previous infoview. `ProofWidgets 4` is a complete redesign and reengineering. Compared to `ProofWidgets 4`, the previous version was not able to incorporate JavaScript libraries which we make heavy use of; used purely server-side rendering which resulted in disruptive latency approaching seconds [70] in distributed settings where code editor and prover reside on different machines (e.g. cloud-based services such as Gitpod); and could not handle asynchronous events which are necessary to invoke long-running computations from the UI. Compared to the previous version we have lost (and hope to regain) the ability to program UI event handlers directly in Lean rather than in JavaScript. We expect this to become feasible when a JavaScript or WebAssembly backend is developed for Lean. The previous infoview did not support goal diffs, structured traces, interactive messages, or selection contexts.

The work of Mehnert, Christiansen, Korkut, and coauthors on `idris-mode` [68, 30, 51] encouraged us to dream of richer programming environments, and suggested presentations as a useful concept. `idris-mode` is primarily limited by the practical difficulty of embedding web-based and non-textual interfaces in Emacs with which it is tightly integrated.

User interfaces for theorem provers can be broadly categorized along two axes. Along one, they can either be built for a specific domain and use case only, or they can be *tool-making tools* designed for extensibility. Along another axis, they can either be integrated with a special-purpose formal system such as a synthetic axiomatization of geometry, or they can support a general foundation. Our interface is designed from the ground up to fall on the right of both axes, that is to support general interface extensions in a general-purpose theorem prover. Existing work tends to place towards the left of at least one of the axes, with many tools excelling at providing fixed sets of UI functionality.

CtCoq [16] and its successor Pcoq [5] were early systems which focused on displaying formulas and proofs in natural language with mathematical notation, on structured editing of proofs, and on proof by pointing. Some extensions to Pcoq have been developed by its authors, notably GeoView [17], a display for statements in plane geometry. Nevertheless Pcoq does not appear to support general user-extensibility. The GeoProof [74] project improved on GeoView by supporting proof construction, rather than just viewing, in the geometric display. However, GeoProof was developed as a standalone application that did not use Pcoq. Robert's PeaCoq [86] focuses on visualizing proof trees and steps, but not the mathematical objects appearing therein such as the diagrams of fig. 5.4. The recent Actema project [38] aims to extend the interactions available in proof by pointing to drag-and-drop interfaces. KeY [2] and KeYmaera X [43] provide interfaces specific to software verification and purpose-built logics. The *Incredible Proof Machine* [25] is a browser-based diagrammatic prover. We hope that our framework enables the creation of similar purpose-specific tooling for the Lean proof assistant.

A recent interface which *does* aim at general-purpose proving and domain-specific extensions is that of HolPy [106]. Compared to `ProofWidgets 4`, at this moment HolPy stresses proof by pointing and LaTeX display but not general visualization of objects or computations.

Another class of interfaces and tools are web-based ones including `jsCoq` [6] and Clide [58]. The comparison here is subtler – while `jsCoq` in particular allows building websites intermixing Coq snippets and UI components, it doesn't seem to provide a way for these components to invoke the metaprogramming API and directly manipulate proof state. It may be that the potential for powerful extensions is there, but was simply never realized in practice. The recent `Alectryon` [83] supports proof *archival* in Coq and in Lean (via `LeanInk` [27]) by storing recorded proof states alongside beautified proof scripts. In contrast, our system serves proof *development* by providing a live display with a variety of graphical representations. We would, however, like to store a static form of these representations in `LeanInk` outputs in the future.

Other systems intersect with our featureset in various ways. ProofGeneral [8] used to support expression presentations, but only for the LEGO prover [57]. Feasibility of real-time asynchronous processing was demonstrated in Isabelle/PIDE [103]. Both PeaCoq and Coq itself contain similar goal diffing capabilities to ours. Multi-representation GUIs for proof assistants were pioneered in the 1990s by the *LΩUI* [91], HyperProof [14] and XBarnacle [56] projects.

Finally, we are generally inspired by Engelbart's (to-date not realized!) vision of human intelligence augmented through computer interfaces [40], and the systems of yore which followed it including Smalltalk [44].

5.6 Conclusion

We designed and implemented an extensible user interface for the Lean 4 theorem prover together with `ProofWidgets 4`, a supporting library of metaprograms and UI components. The interface is based on presentations: UI elements that store references to the objects they are displaying. Presentations enable detailed introspection of tactics and systems comprising the prover. Extending the interface with `ProofWidgets 4` empowers users to work with a variety of interactive, graphical representations. Building on the JavaScript ecosystem enables quick prototyping. The framework's domain of applicability includes exploring computation traces, symbolic visualization and exploration of mathematical objects and data structures, custom interfaces for tactics and tactic modes, data visualization, function plotting, and interactive simulations. Supporting not only expert users, it could be used in education to build interactive textbooks and tutorials. We demonstrated example user widgets diagramming mathematical data and suggesting possible proof steps from within the UI.

In tune with the overall design philosophy of Lean 4, every layer of the visual stack can be extended. We provide a *tool-making tool* which enables the creation of rich environments for program and proof in science and mathematics.

References

[1] Harold Abelson and Gerald J. Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985. ISBN: 0-262-51036-7.

[2] Wolfgang Ahrendt et al. *Deductive Software Verification - The KeY Book.* Lecture Notes in Computer Science. Springer, 2016. ISBN: 978-3-319-49811-9. DOI: 10.1007/978-3-319-49812-6. URL: https://doi.org/10.1007/978-3-319-49812-6.

[3] Stuart F. Allen. *Nuprl Editor and Interface.* English. Cornell University. Sept. 23, 2003. URL: https://web.archive.org/web/20230927022418/https://www.nuprl.org/KB/show.php?ID=29/.

[4] Ana de Almeida Borges et al. "Lessons for Interactive Theorem Proving Researchers from a Survey of Coq Users". In: *14th International Conference on Interactive Theorem Proving (ITP 2023).* Ed. by Adam Naumowicz and René Thiemann. Vol. 268. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 12:1–12:18. ISBN: 978-3-95977-284-6. DOI: 10.4230/LIPIcs.ITP.2023.12. URL: https://drops.dagstuhl.de/opus/volltexte/2023/18387.

[5] Ahmed Amerkad et al. *Mathematics and Proof Presentation in Pcoq.* Tech. rep. RR-4313. INRIA, Nov. 2001. URL: https://hal.inria.fr/inria-00072274.

[6] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. "jsCoq: Towards Hybrid Theorem Proving Interfaces". In: *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers, UITP 2016, Coimbra, Portugal, 2nd July 2016.* Ed. by Serge Autexier and Pedro Quaresma. Vol. 239. EPTCS. 2016, pp. 15–27. DOI: 10.4204/EPTCS.239.2. URL: https://doi.org/10.4204/EPTCS.239.2.

[7] Andrea Asperti et al. *A new type for tactics.* 2009. URL: https://hdl.handle.net/11585/101036.

[8] David Aspinall. "Proof General: A generic tool for proof development". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Susanne Graf and Michael I. Schwartzbach. Vol. 1785. Lecture Notes in Computer Science. Springer. Springer, 2000, pp. 38–43. DOI: 10.1007/3-540-46419-0_3. URL: https://link.springer.com/content/pdf/10.1007/3-540-46419-0_3.pdf.

[9] Jeremy Avigad. *Mathematical Logic and Computation.* Cambridge University Press, 2022. DOI: 10.1017/9781108778756.

[10] Edward W. Ayers. "A Tool for Producing Verified, Explainable Proofs". PhD thesis. University of Cambridge, 2021. DOI: 10.17863/CAM.81869. URL: https://www.repository.cam.ac.uk/handle/1810/334452.

[11] Edward W. Ayers, Mateja Jamnik, and William T. Gowers. "A Graphical User Interface Framework for Formal Verification". In: *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference).* Ed. by Liron Cohen and Cezary Kaliszyk. Vol. 193. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 4:1–4:16. DOI: 10.4230/LIPIcs.ITP.2021.4. URL: https://doi.org/10.4230/LIPIcs.ITP.2021.4.

[12] Krzysztof Bar, Aleks Kissinger, and Jamie Vicary. "Globular: an online proof assistant for higher-dimensional rewriting". In: *Leibniz International Proceedings in Informatics.* Vol. 52. ncatlab.org/nlab/show/Globular. 2016, 34:1–34:11.

[13] Henk Barendregt and Herman Geuvers. "Proof-Assistants Using Dependent Type Systems". In: *Handbook of Automated Reasoning (in 2 volumes).* Ed. by John Alan Robinson and Andrei Voronkov. Elsevier and MIT Press, 2001, pp. 1149–1238. DOI: 10.1016/b978-044450813-3/50020-5. URL: https://doi.org/10.1016/b978-044450813-3/50020-5.

[14] Jon Barwise and John Etchemendy. “Hyperproof: Logical reasoning with diagrams”. In: *Working Notes of the AAAI Spring Symposium on Reasoning with Diagrammatic Representations*. 1992. URL: `https://www.aaai.org/Papers/Symposia/Spring/1992/SS-92-02/SS92-02-016.pdf`.

[15] Joseph L. Bates and Robert L. Constable. “Proofs as Programs”. In: *ACM Trans. Program. Lang. Syst.* 7.1 (1985), pp. 113–136. DOI: `10.1145/2363.2528`. URL: `https://doi.org/10.1145/2363.2528`.

[16] Yves Bertot. “The CtCoq System: Design and Architecture”. In: *Formal Aspects Comput.* 11.3 (1999), pp. 225–243. DOI: `10.1007/s001650050049`. URL: `https://doi.org/10.1007/s001650050049`.

[17] Yves Bertot, Frédérique Guilhot, and Loic Pottier. “Visualizing Geometrical Statements with GeoView”. In: *Proceedings of the User Interfaces for Theorem Provers Workshop, UITP@TPHOLs 2003, Rome, Italy, September 8, 2003*. Ed. by David Aspinall and Christoph Lüth. Vol. 103. Electronic Notes in Theoretical Computer Science. Elsevier, 2003, pp. 49–65. DOI: `10.1016/j.entcs.2004.09.013`. URL: `https://doi.org/10.1016/j.entcs.2004.09.013`.

[18] Yves Bertot, Gilles Kahn, and Laurent Théry. “Proof by Pointing”. In: *Theoretical Aspects of Computer Software, International Conference TACS ’94, Sendai, Japan, April 19-22, 1994, Proceedings*. Ed. by Masami Hagiya and John C. Mitchell. Vol. 789. Lecture Notes in Computer Science. Springer, 1994, pp. 141–160. ISBN: 3-540-57887-0. DOI: `10.1007/3-540-57887-0\_94`. URL: `https://doi.org/10.1007/3-540-57887-0%5C_94`.

[19] Yves Bertot and Laurent Théry. “A Generic Approach to Building User Interfaces for Theorem Provers”. In: *J. Symb. Comput.* 25.2 (1998), pp. 161–194. DOI: `10.1006/jsco.1997.0171`. URL: `https://doi.org/10.1006/jsco.1997.0171`.

[20] Jeff Bezanson et al. “Julia: A Fast Dynamic Language for Technical Computing”. In: *CoRR* abs/1209.5145 (2012). arXiv: `1209.5145`. URL: `http://arxiv.org/abs/1209.5145`.

[21] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. “Extending Sledgehammer with SMT Solvers”. In: *J. Autom. Reason.* 51.1 (2013), pp. 109–128. DOI: `10.1007/s10817-013-9278-5`. URL: `https://doi.org/10.1007/s10817-013-9278-5`.

[22] R. Braden. *Requirements for Internet Hosts - Communication Layers*. RFC 1122. RFC Editor, Oct. 1989. URL: `https://www.rfc-editor.org/rfc/rfc1122.txt`.

[23] Edwin C. Brady. “Idris 2: Quantitative Type Theory in Practice”. In: *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*. Ed. by Anders Møller and Manu Sridharan. Vol. 194. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 9:1–9:26. DOI: `10.4230/LIPIcs.ECOOP.2021.9`. URL: `https://doi.org/10.4230/LIPIcs.ECOOP.2021.9`.

[24] Edwin C. Brady. “Idris, a general-purpose dependently typed programming language: Design and implementation”. In: *J. Funct. Program.* 23.5 (2013), pp. 552–593. DOI: `10.1017/S095679681300018X`. URL: `https://doi.org/10.1017/S095679681300018X`.

[25] Joachim Breitner. “Visual theorem proving with the Incredible Proof Machine”. In: *International Conference on Interactive Theorem Proving*. Ed. by Jasmin Christian Blanchette and Stephan Merz. Springer, 2016, pp. 123–139. DOI: `10.1007/978-3-319-43144-4_8`. URL: `https://doi.org/10.1007/978-3-319-43144-4_8`.

[26] Nicolaas Govert de Bruijn. “AUTOMATH, a Language for Mathematics”. In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by Jörg H. Siekmann and Graham Wrightson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 159–200. ISBN: 978-3-642-81955-1. DOI: `10.1007/978-3-642-81955-1_11`. URL: `https://doi.org/10.1007/978-3-642-81955-1_11`.

[27] Niklas Bülow. *Proof Visualization for the Lean 4 Theorem Prover.* Apr. 2022.

[28] Mario Carneiro. "Metamath Zero: Designing a Theorem Prover Prover". In: *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings.* Ed. by Christoph Benzmüller and Bruce R. Miller. Vol. 12236. Lecture Notes in Computer Science. Springer, 2020, pp. 71–88. DOI: 10.1007/978-3-030-53518-6_5. URL: https://doi.org/10.1007/978-3-030-53518-6%5C_5.

[29] Mario Carneiro. *The Type Theory of Lean.* 2019. URL: https://github.com/digama0/lean-type-theory.

[30] David Christiansen, David Darais, and Weixi Ma. *The Final Pretty Printer.* 2016. URL: https://davidchristiansen.dk/drafts/final-pretty-printer-draft.pdf.

[31] David R. Christiansen and Edwin C. Brady. "Elaborator reflection: extending Idris in Idris". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016.* Ed. by Jacques Garrigue, Gabriele Keller, and Eijiro Sumii. ACM, 2016, pp. 284–297. ISBN: 978-1-4503-4219-3. DOI: 10.1145/2951913.2951932. URL: https://doi.org/10.1145/2951913.2951932.

[32] David Raymond Christiansen. "Dependent type providers". In: *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming, WGP 2013, Boston, Massachusetts, USA, September 28, 2013.* Ed. by Jacques Carette and Jeremiah Willcock. ACM, 2013, pp. 25–34. ISBN: 978-1-4503-2389-5. DOI: 10.1145/2502488.2502495. URL: https://doi.org/10.1145/2502488.2502495.

[33] David Raymond Christiansen. "Type-Directed Elaboration of Quasiquotations: A High-Level Syntax for Low-Level Reflection". In: *Proceedings of the 26th 2014 International Symposium on Implementation and Application of Functional Languages, IFL '14, Boston, MA, USA, October 1-3, 2014.* Ed. by Sam Tobin-Hochstadt. ACM, 2014, 1:1–1:9. ISBN: 978-1-4503-3284-2. DOI: 10.1145/2746325.2746326. URL: https://doi.org/10.1145/2746325.2746326.

[34] Eugene Charles Ciccarelli. "Presentation based user interfaces". PhD thesis. Massachusetts Institute of Technology, Cambridge, MA, USA, 1984. URL: https://hdl.handle.net/1721.1/15346.

[35] Bob Coecke and Ross Duncan. "Interacting Quantum Observables". In: *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations.* Ed. by Luca Aceto et al. Vol. 5126. Lecture Notes in Computer Science. Springer, 2008, pp. 298–310. DOI: 10.1007/978-3-540-70583-3_25. URL: https://doi.org/10.1007/978-3-540-70583-3%5C_25.

[36] Thierry Coquand and Christine Paulin. "Inductively defined types". In: *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings.* Ed. by Per Martin-Löf and Grigori Mints. Vol. 417. Lecture Notes in Computer Science. Springer, 1988, pp. 50–66. ISBN: 3-540-52335-9. DOI: 10.1007/3-540-52335-9_47. URL: https://doi.org/10.1007/3-540-52335-9%5C_47.

[37] R.A. Dean. *Elements of Abstract Algebra.* Wiley international edition. Wiley, 1966. ISBN: 9780471204510. URL: https://books.google.com/books?id=kmulxmBgkxoC.

[38] Pablo Donato, Pierre-Yves Strub, and Benjamin Werner. "A drag-and-drop proof tactic". In: *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022.* Ed. by Andrei Popescu and Steve Zdancewic. ACM, 2022, pp. 197–209. DOI: 10.1145/3497775.3503692. URL: https://doi.org/10.1145/3497775.3503692.

[39] Gabriel Ebner et al. “A metaprogramming framework for formal verification”. In: *Proc. ACM Program. Lang.* 1.ICFP (2017), 34:1–34:29. DOI: 10.1145/3110278. URL: https://doi.org/10.1145/3110278.

[40] Douglas C. Engelbart. *Augmenting Human Intellect: A Conceptual Framework.* Tech. rep. Oct. 1962. URL: https://web.archive.org/save/https://www.dougengelbart.org/pubs/augment-3906.html.

[41] Alexander John Faithfull et al. “Coqoon - An IDE for Interactive Proof Development in Coq”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings.* Ed. by Marsha Chechik and Jean-François Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 316–331. ISBN: 978-3-662-49673-2. DOI: 10.1007/978-3-662-49674-9_18. URL: https://doi.org/10.1007/978-3-662-49674-9%5C_18.

[42] Matthias Felleisen et al. “The Racket Manifesto”. In: *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA.* Ed. by Thomas Ball et al. Vol. 32. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 113–128. DOI: 10.4230/LIPIcs.SNAPL.2015.113. URL: https://doi.org/10.4230/LIPIcs.SNAPL.2015.113.

[43] Nathan Fulton et al. “KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems”. In: *CADE.* Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. LNCS. Springer, 2015, pp. 527–538. DOI: 10.1007/978-3-319-21401-6_36.

[44] Adele Goldberg. *Smalltalk-80 - the interactive programming environment.* Addison-Wesley, 1984. ISBN: 978-0-201-11372-3.

[45] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF.* Vol. 78. Lecture Notes in Computer Science. Springer, 1979. ISBN: 3-540-09724-4. DOI: 10.1007/3-540-09724-4. URL: https://doi.org/10.1007/3-540-09724-4.

[46] John Harrison. “A Mizar Mode for HOL”. In: *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs’96.* Ed. by Joakim von Wright, Jim Grundy, and John Harrison. Vol. 1125. Lecture Notes in Computer Science. Turku, Finland: Springer-Verlag, Aug. 26, 1996, pp. 203–220.

[47] John Harrison. “HOL Light: An Overview”. In: *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009.* Ed. by Stefan Berghofer et al. Vol. 5674. Lecture Notes in Computer Science. Munich, Germany: Springer-Verlag, 2009, pp. 60–66.

[48] Rich Hickey. “The Clojure programming language”. In: *Proceedings of the 2008 Symposium on Dynamic Languages, DLS 2008, July 8, 2008, Paphos, Cyprus.* Ed. by Johan Brichau. ACM, 2008, p. 1. DOI: 10.1145/1408681.1408682. URL: https://doi.org/10.1145/1408681.1408682.

[49] André Joyal and Ross Street. “The geometry of tensor calculus, I”. In: *Advances in Mathematics* 88.1 (1991), pp. 55–112. ISSN: 0001-8708. DOI: https://doi.org/10.1016/0001-8708(91)90003-P. URL: https://www.sciencedirect.com/science/article/pii/000187089190003P.

[50] Matt Kaufmann and J Strother Moore. “An Industrial Strength Theorem Prover for a Logic Based on Common Lisp”. In: *IEEE Trans. Software Eng.* 23.4 (1997), pp. 203–213. DOI: 10.1109/32.588534. URL: https://doi.org/10.1109/32.588534.

[51] Joomy Korkut and David Thrane Christiansen. “Extensible type-directed editing”. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Devel-*

opment, TyDe@ICFP 2018, St. Louis, MO, USA, September 27, 2018. Ed. by Richard A. Eisenberg and Niki Vazou. ACM, 2018, pp. 38–50. DOI: 10.1145/3240719.3241791. URL: https://doi.org/10.1145/3240719.3241791.

[52] Guillaume Lample et al. "HyperTree Proof Search for Neural Theorem Proving". In: *CoRR* abs/2205.11491 (2022). DOI: 10.48550/arXiv.2205.11491. arXiv: 2205.11491. URL: https://doi.org/10.48550/arXiv.2205.11491.

[53] Jill H. Larkin and Herbert A. Simon. "Why a Diagram is (Sometimes) Worth Ten Thousand Words". In: *Cognitive Science* 11.1 (1987), pp. 65–100. DOI: https://doi.org/10.1111/j.1551-6708.1987.tb00863.x.

[54] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20. URL: https://doi.org/10.1007/978-3-642-17511-4%5C_20.

[55] The LMFDB Collaboration. *The L-functions and modular forms database*. https://www.lmfdb.org. [Online; accessed 30 April 2023]. 2023.

[56] Helen Lowe and David Duncan. "XBarnacle: Making Theorem Provers More Accessible". In: Lecture Notes in Computer Science 1249 (1997). Ed. by William McCune, pp. 404–407. DOI: 10.1007/3-540-63104-6_39. URL: https://doi.org/10.1007/3-540-63104-6_39.

[57] Z. Luo and R. Pollack. "LEGO Proof Development System: User's Manua". In: *LFCS Report Series*. LFCS, Edinburgh University, 1992.

[58] Christoph Lüth and Martin Ring. "A Web Interface for Isabelle: The Next Generation". In: *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*. Ed. by Jacques Carette et al. Vol. 7961. Lecture Notes in Computer Science. Springer, 2013, pp. 326–329. DOI: 10.1007/978-3-642-39320-4_22. URL: https://doi.org/10.1007/978-3-642-39320-4%5C_22.

[59] Lena Magnusson and Bengt Nordström. "The ALF Proof Editor and Its Proof Engine". In: *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*. Ed. by Henk Barendregt and Tobias Nipkow. Vol. 806. Lecture Notes in Computer Science. Springer, 1993, pp. 213–237. ISBN: 3-540-58085-9. DOI: 10.1007/3-540-58085-9_78. URL: https://doi.org/10.1007/3-540-58085-9%5C_78.

[60] Per Martin-Löf. "An Intuitionistic Theory of Types: Predicative Part". In: *Logic Colloquium '73*. Ed. by H.E. Rose and J.C. Shepherdson. Vol. 80. Studies in Logic and the Foundations of Mathematics. Elsevier, 1975, pp. 73–118. DOI: https://doi.org/10.1016/S0049-237X(08)71945-1. URL: https://www.sciencedirect.com/science/article/pii/S0049237X08719451.

[61] Per Martin-Löf. "On the Meanings of the Logical Constants and the Justifications of the Logical Laws". In: *Nordic Journal of Philosophical Logic* 1.1 (1996), pp. 11–60.

[62] Guido Martínez et al. "Meta-F*: Metaprogramming and Tactics in an Effectful Program Verifier". In: *CoRR* abs/1803.06547 (2018). arXiv: 1803.06547. URL: http://arxiv.org/abs/1803.06547.

[63] Paolo Masci and Aaron Dutle. "Proof Mate: An Interactive Proof Helper for PVS (Tool Paper)". In: *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*. Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund,

and Ivan Perez. Vol. 13260. Lecture Notes in Computer Science. Springer, 2022, pp. 809–815. ISBN: 978-3-031-06772-3. DOI: 10.1007/978-3-031-06773-0_44. URL: https://doi.org/10.1007/978-3-031-06773-0%5C_44.

[64] Paolo Masci and César A. Muñoz. "An Integrated Development Environment for the Prototype Verification System". In: *Electronic Proceedings in Theoretical Computer Science* 310.nil (2019), pp. 35–49. DOI: 10.4204/eptcs.310.5. URL: http://dx.doi.org/10.4204/EPTCS.310.5.

[65] Mikael Mayer. *Making Verification Compelling: Visual Verification Feedback for Dafny.* 2023. URL: https://web.archive.org/web/20230419201732/https://dafny.org/blog/2023/04/19/making-verification-compelling-visual-verification-feedback-for-dafny (visited on 04/19/2023).

[66] Conor McBride. "Dependently typed functional programs and their proofs". PhD thesis. University of Edinburgh, UK, 2000. URL: https://hdl.handle.net/1842/374.

[67] John McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". In: *Commun. ACM* 3.4 (1960), pp. 184–195. DOI: 10.1145/367177.367199. URL: https://doi.org/10.1145/367177.367199.

[68] Hannes Mehnert and David Christiansen. *Tool Demonstration: An IDE for Programming and Proving in Idris.* 2014. URL: https://davidchristiansen.dk/pubs/dtp2014-idris-mode.pdf.

[69] Guillaume Melquiond. "Plotting in a Formally Verified Way". In: *Proceedings of the 6th Workshop on Formal Integrated Development Environment, F-IDE@NFM 2021, Held online, 24-25th May 2021.* Ed. by José Proença and Andrei Paskevich. Vol. 338. EPTCS. 2021, pp. 39–45. DOI: 10.4204/EPTCS.338.6. URL: https://doi.org/10.4204/EPTCS.338.6.

[70] Robert B Miller. "Response time in man-computer conversational transactions". In: *Proceedings of the December 9-11, 1968, fall joint computer conference, part I.* 1968, pp. 267–277.

[71] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings.* Ed. by André Platzer and Geoff Sutcliffe. Vol. 12699. Lecture Notes in Computer Science. Springer, 2021, pp. 625–635. DOI: 10.1007/978-3-030-79876-5_37. URL: https://doi.org/10.1007/978-3-030-79876-5%5C_37.

[72] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24. URL: https://doi.org/10.1007/978-3-540-78800-3%5C_24.

[73] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. "Contextual modal type theory". In: *ACM Trans. Comput. Log.* 9.3 (2008), 23:1–23:49. DOI: 10.1145/1352582.1352591. URL: https://doi.org/10.1145/1352582.1352591.

[74] Julien Narboux. "A Graphical User Interface for Formal Proofs in Geometry". In: *J. Autom. Reason.* 39.2 (2007), pp. 161–180. DOI: 10.1007/s10817-007-9071-4. URL: https://doi.org/10.1007/s10817-007-9071-4.

[75] Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner. "An Extensible User Interface for Lean 4". In: *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland.* Ed. by Adam Naumowicz and René

Thiemann. Vol. 268. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 24:1–24:20. DOI: 10.4230/LIPIcs.ITP.2023.24. URL: https://doi.org/10.4230/LIPIcs.ITP.2023.24.

[76] Ulf Norell. "Dependently Typed Programming in Agda". In: *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures.* 2008, pp. 230–266. DOI: 10.1007/978-3-642-04652-0_5.

[77] Chris Okasaki. "Red-Black Trees in a Functional Setting". In: *J. Funct. Program.* 9.4 (1999), pp. 471–477. DOI: 10.1017/s0956796899003494. URL: https://doi.org/10.1017/s0956796899003494.

[78] Cyrus Omar et al. "Hazelnut: a bidirectionally typed structure editor calculus". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 86–99. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009900. URL: https://doi.org/10.1145/3009837.3009900.

[79] Cyrus Omar et al. "Live functional programming with typed holes". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 14:1–14:32. DOI: 10.1145/3290327. URL: https://doi.org/10.1145/3290327.

[80] Sam Owre, John M. Rushby, and Natarajan Shankar. "PVS: A Prototype Verification System". In: *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings.* Ed. by Deepak Kapur. Vol. 607. Lecture Notes in Computer Science. Springer, 1992, pp. 748–752. DOI: 10.1007/3-540-55602-8_217. URL: https://doi.org/10.1007/3-540-55602-8%5C_217.

[81] Lawrence C. Paulson. *Thoughts on user interfaces for theorem provers.* Dec. 2022. URL: https://web.archive.org/web/20230219221749/https://lawrencecpaulson.github.io/2022/12/14/User_interfaces.html.

[82] Per Martin-Löf. "Constructive Mathematics and Computer Programming". In: *Logic, Methodology and Philosophy of Science VI.* Ed. by L. Jonathan Cohen et al. Vol. 104. Studies in Logic and the Foundations of Mathematics. Elsevier, 1982, pp. 153–175. DOI: https://doi.org/10.1016/S0049-237X(09)70189-2. URL: https://www.sciencedirect.com/science/article/pii/S0049237X09701892.

[83] Clément Pit-Claudel. "Untangling mechanized proofs". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020.* Ed. by Ralf Lämmel, Laurence Tratt, and Juan de Lara. ACM, 2020, pp. 155–174. ISBN: 978-1-4503-8176-5. DOI: 10.1145/3426425.3426940. URL: https://doi.org/10.1145/3426425.3426940.

[84] K. Reidemeister. *Knotentheorie.* Ergebnisse der Mathematik und ihrer Grenzgebiete v. 1, no. 1. Springer, 1932. URL: https://books.google.com/books?id=jrMGQwAACAAJ.

[85] David Reutter and Jamie Vicary. "High-Level Methods for Homotopy Construction in Associative n-Categories". In: *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science.* LICS '19. Vancouver, Canada: IEEE Press, 2021.

[86] Valentin Robert. "Front-end tooling for building and maintaining dependently-typed functional programs". PhD thesis. University of California, San Diego, USA, 2018. URL: http://www.escholarship.org/uc/item/9q3490fh.

[87] Julio Rubio and Francis Sergeraert. "Constructive algebraic topology". In: *Bulletin des Sciences Mathematiques* 126.5 (2002), pp. 389–412.

[88] John M. Rushby, Sam Owre, and Natarajan Shankar. "Subtypes for Specifications: Predicate Subtyping in PVS". In: *IEEE Trans. Software Eng.* 24.9 (1998), pp. 709–720. DOI: 10.1109/32.713327. URL: https://doi.org/10.1109/32.713327.

[89] Klaas van Schelven. *Don't say "Homoiconic"*. https://www.expressionsofchange.org/dont-say-homoiconic/. 2018.

[90] Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. *Tabled Typeclass Resolution*. 2020. arXiv: 2001.04301v2 [cs.PL].

[91] Jörg Siekmann et al. "LOUI: Lovely OMEGA user interface". In: *Formal Aspects of Computing* 11.3 (1999). Ed. by James Woodcock, pp. 326–342. DOI: 10.1007/s001650050053. URL: https://doi.org/10.1007/s001650050053.

[92] Konrad Slind and Michael Norrish. "A Brief Overview of HOL4". In: *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*. Ed. by Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar. Vol. 5170. Lecture Notes in Computer Science. Springer, 2008, pp. 28–32. DOI: 10.1007/978-3-540-71067-7_6. URL: https://doi.org/10.1007/978-3-540-71067-7%5C_6.

[93] Matthieu Sozeau et al. "The MetaCoq Project". In: *J. Autom. Reason.* 64.5 (2020), pp. 947–999. DOI: 10.1007/s10817-019-09540-0. URL: https://doi.org/10.1007/s10817-019-09540-0.

[94] Arnaud Spiwack. "Verified Computing in Homological Algebra. (Calculs vérifiés en algèbre homologique)". PhD thesis. École Polytechnique, Palaiseau, France, 2011. URL: https://tel.archives-ouvertes.fr/pastel-00605836.

[95] Aaron Stockdill et al. "Considerations in Representation Selection for Problem Solving: A Review". In: *Diagrammatic Representation and Inference*. Ed. by Amrita Basu et al. Cham: Springer International Publishing, 2021, pp. 35–51. ISBN: 978-3-030-86062-2.

[96] Nikhil Swamy et al. "Dependent types and multi-monadic effects in F". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 256–270. DOI: 10.1145/2837614.2837655. URL: https://doi.org/10.1145/2837614.2837655.

[97] The Coq Development Team. *The Coq Proof Assistant*. Version 8.17. June 2023. DOI: 10.5281/zenodo.8161141. URL: https://doi.org/10.5281/zenodo.8161141.

[98] William P. Thurston. "On proof and progress in mathematics". In: (1994). DOI: 10.48550/ARXIV.MATH/9404236. URL: https://arxiv.org/abs/math/9404236.

[99] Silvia de Toffoli. "Chasing the Diagram–the Use of Visualizations in Algebraic Reasoning". In: *Review of Symbolic Logic* 10.1 (2017), pp. 158–186. DOI: 10.1017/s1755020316000277.

[100] Andrzej Trybulec and Howard A. Blair. "Computer Assisted Reasoning with MIZAR". In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence. Los Angeles, CA, USA, August 1985*. Ed. by Aravind K. Joshi. Morgan Kaufmann, 1985, pp. 26–28. URL: http://ijcai.org/Proceedings/85-1/Papers/006.pdf.

[101] Sebastian Ullrich and Leonardo de Moura. "Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages". In: *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Vol. 12167. Lecture Notes in Computer Science. Springer, 2020, pp. 167–182. ISBN: 978-3-030-51053-4. DOI: 10.1007/978-3-030-51054-1_10. URL: https://doi.org/10.1007/978-3-030-51054-1%5C_10.

[102] Sebastian Andreas Ullrich. "An Extensible Theorem Proving Frontend". PhD thesis. Karlsruher Institut für Technologie (KIT), 2023. 243 pp. DOI: 10.5445/IR/1000161074.

[103] Makarius Wenzel. “Asynchronous User Interaction and Tool Integration in Isabelle/PIDE”. In: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings.* Ed. by Gerwin Klein and Ruben Gamboa. Vol. 8558. Lecture Notes in Computer Science. Springer, 2014, pp. 515–530. DOI: `10.1007/978-3-319-08970-6\_33`. URL: `https://doi.org/10.1007/978-3-319-08970-6%5C_33`.

[104] Markus Wenzel. “Isabelle, Isar - a versatile environment for human readable formal proof documents”. PhD thesis. Technical University Munich, Germany, 2002. URL: `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.pdf`.

[105] Katherine Ye et al. “Penrose: from mathematical notation to beautiful diagrams”. In: *ACM Trans. Graph.* 39.4 (2020), p. 144. DOI: `10.1145/3386569.3392375`. URL: `https://doi.org/10.1145/3386569.3392375`.

[106] Bohua Zhan et al. “Design of Point-and-Click User Interfaces for Proof Assistants”. In: *Formal Methods and Software Engineering - 21st International Conference on Formal Engineering Methods, ICFEM 2019, Shenzhen, China, November 5-9, 2019, Proceedings.* Ed. by Yamine Aït Ameur and Shengchao Qin. Vol. 11852. Lecture Notes in Computer Science. Springer, 2019, pp. 86–103. DOI: `10.1007/978-3-030-32409-4\_6`. URL: `https://doi.org/10.1007/978-3-030-32409-4%5C_6`.

[107] Jiaje Zhang and Donald A. Norman. “Representations in distributed cognitive tasks”. In: *Cognitive Science* 18.1 (1994), pp. 87–122. ISSN: 0364-0213. DOI: `https://doi.org/10.1016/0364-0213(94)90021-3`. URL: `https://www.sciencedirect.com/science/article/pii/0364021394900213`.