An Extensible User Interface for Lean 4

Wojciech Nawrocki
Carnegie Mellon University, Pittsburgh, PA, USA

Edward W. Ayers
Carnegie Mellon University, Pittsburgh, PA, USA

Gabriel Ebner
Microsoft Research, Redmond, WA, USA

Abstract

Contemporary proof assistants rely on complex automation and process libraries with millions of lines of code. At these scales, understanding the emergent interactions between components can be a serious challenge. One way of managing complexity, long established in informal practice, is through varying *external representations*. For instance, algebraic notation facilitates term-based reasoning whereas geometric diagrams invoke spatial intuition. Objects viewed one way become much simpler than when viewed differently. In contrast, modern general-purpose ITP systems usually only support limited, textual representations. Treating this as a problem of human-computer interaction, we aim to demonstrate that *presentations* – UI elements that store references to the objects they are displaying – are a fruitful way of thinking about ITP interface design. They allow us to make headway on two fronts – introspection of prover internals and support for diagrammatic reasoning. To this end we have built an extensible user interface for the Lean 4 prover with an associated `ProofWidgets 4` library of presentation-based UI components. We demonstrate the system with several examples including type information popups, structured traces, contextual suggestions, a display for algebraic reasoning, and visualizations of red-black trees. Our interface is already part of the core Lean distribution.

2012 ACM Subject Classification Human-centered computing → Visualization systems and tools; Software and its engineering → Functional languages

Keywords and phrases user interfaces, human-computer interaction, Lean

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.24

Supplementary Material *Software (User interface)*: `https://github.com/leanprover/vscode-lean4/tree/v0.0.102`, archived at `swh:1:rev:232b31446d71a697ef66cc3f9cdd671e52631317`
Software (ProofWidgets 4): `https://github.com/EdAyers/ProofWidgets4/tree/itp23` archived at `swh:1:dir:2c87d19df4c75dccfab1949cf370d3ca92a37be0`

Funding *Wojciech Nawrocki*: Hoskinson Center for Formal Mathematics.
Edward W. Ayers: Hoskinson Center for Formal Mathematics.

Acknowledgements 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 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 the paper. Tomáš Skřivan, Joachim Breitner, and Sina Hazratpour for trying our systems and suggesting improvements. Chris Lovett and Mariana Alanis for working on `vscode-lean4`. The Lean Zulip community for technical help and ideas.

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 [49]. Restricting the external representations available in ITP systems to only be text is thus a restriction on the way we think [45]. For instance, diagrammatic representations group related information together in ways that sequences of words simply cannot [29]. Since mathematicians and computer scientists rely on graphical calculi and processes such as diagram chases [20], 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 [17, 16], 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.

1.1 Contributions

We report on the design and implementation of a user interface (UI) for the Lean 4 theorem prover [19], of an associated `ProofWidgets 4` library of UI components[1], 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 3.1 the Penrose [47] 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 2.1); and goal states can be interacted with in order to make progress on proofs (Section 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 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.

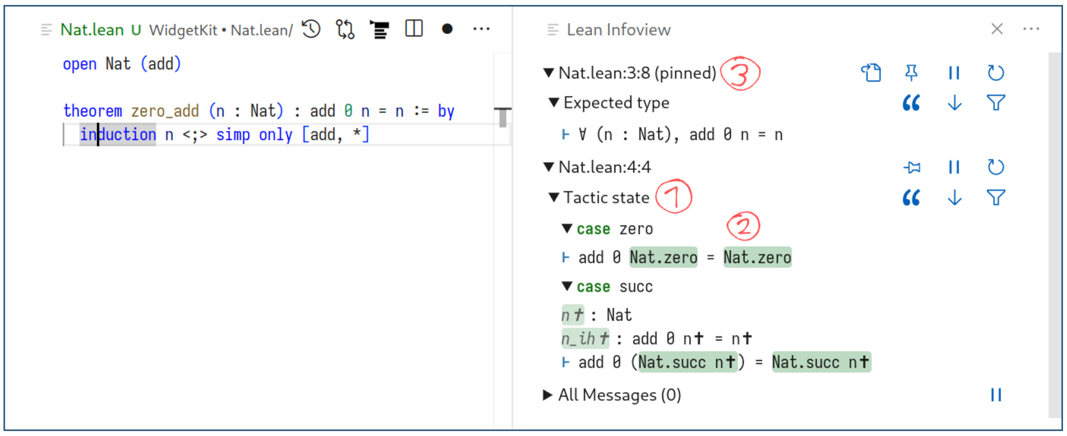
While interfaces supporting subsets of the above have been developed, our system appears to be the first to support all of them in a cohesive way. We give a detailed comparison to other systems in Section 5. The UI is part of the core Lean distribution and has been deployed widely to hundreds of active users, whereas the `ProofWidgets` package can be imported for additional functionality. The UI has been integrated in the VS Code extension `vscode-lean4`[2] as well as in the `Lean 4 Web`[3] online editor.

[1] `https://github.com/EdAyers/ProofWidgets4/tree/itp23`
[2] `https://github.com/leanprover/vscode-lean4/`
[3] `https://lean.math.hhu.de/`

Outline. In Section 2, we introduce the user interface and its interactive features. In Section 3, we demonstrate how to extend the interface by means of several examples. In Section 4, internals of the system and aspects of implementation are discussed. We cover related work in Section 5 and conclude in Section 6.

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 [4]. 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 Figure 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 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 [46], 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.

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 Figure 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).

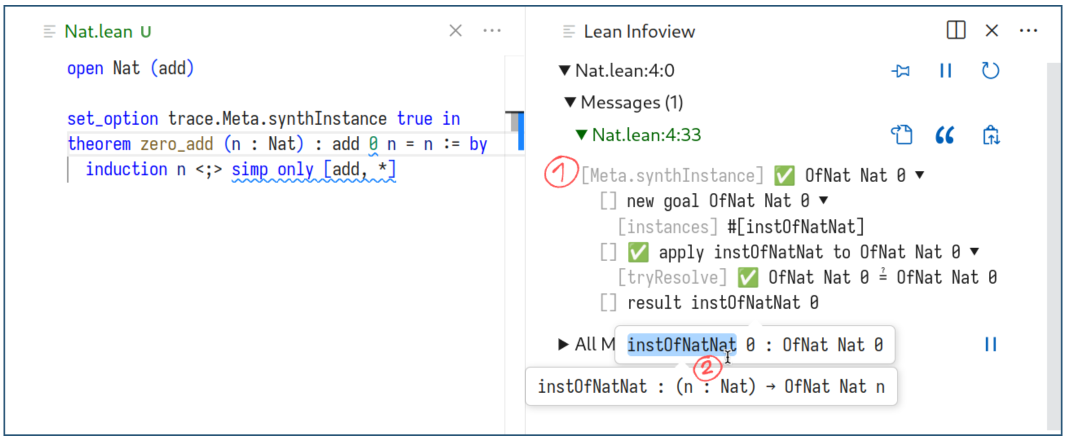
One advantage of the waterfall approach is that the checkpoint can be used as an additional cursor which displays the goal state in one part of the file while we go on to work on another part. We generalize this by allowing one or more text locations to be *pinned* in the infoview. Information about 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.

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 Figure 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 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?



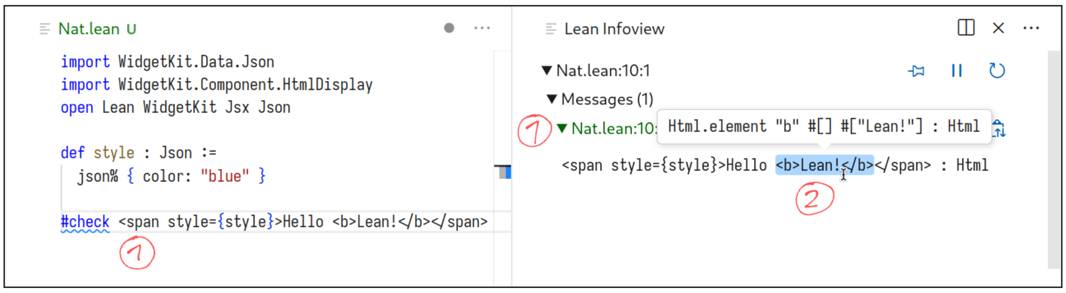
```
open Nat (add)

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 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.

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 [43] 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 needed to pretty-print it all. In Figure 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 Figure 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>
```

```
Html.element "b" #[] #["Lean!"] : Html
<span style={style}>Hello <b>Lean!</b></span> : Html
```

Figure 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.

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 Figure 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 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.

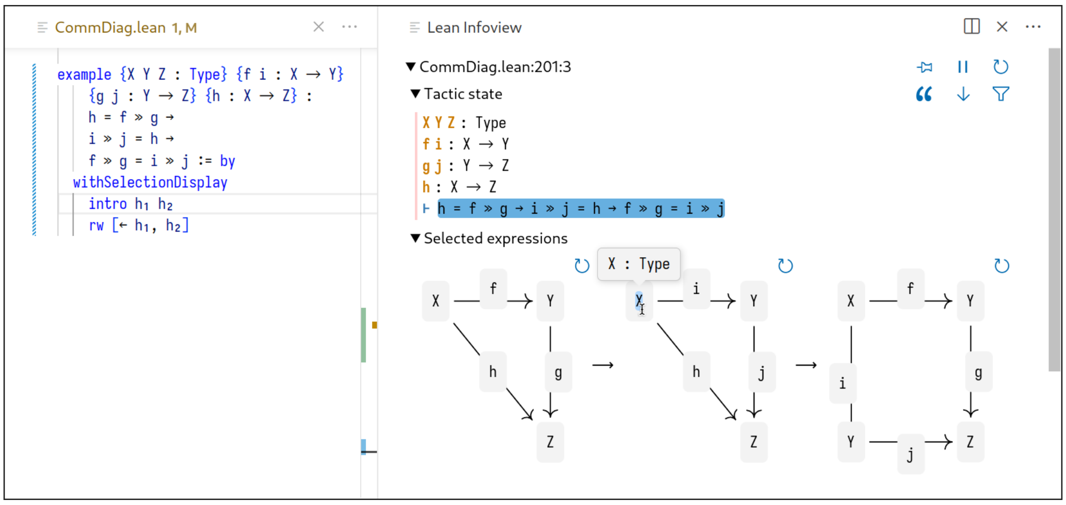
3.1 Diagrams for algebra

In Figure 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 [47], 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 Figure 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 3.3.

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[4]:

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

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



```
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]
```

```
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
```

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[dr, "h"'] & Y \arrow[d, "g"] \\ & Z \end{tikzcd} \longrightarrow \begin{tikzcd} X \arrow[r, "i"] \arrow[dr, "h"'] & Y \arrow[d, "j"] \\ & Z \end{tikzcd} \longrightarrow \begin{tikzcd} X \arrow[r, "f"] \arrow[d, "i"'] & Y \arrow[d, "g"] \\ Y \arrow[r, "j"'] & Z \end{tikzcd}$$

Figure 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.

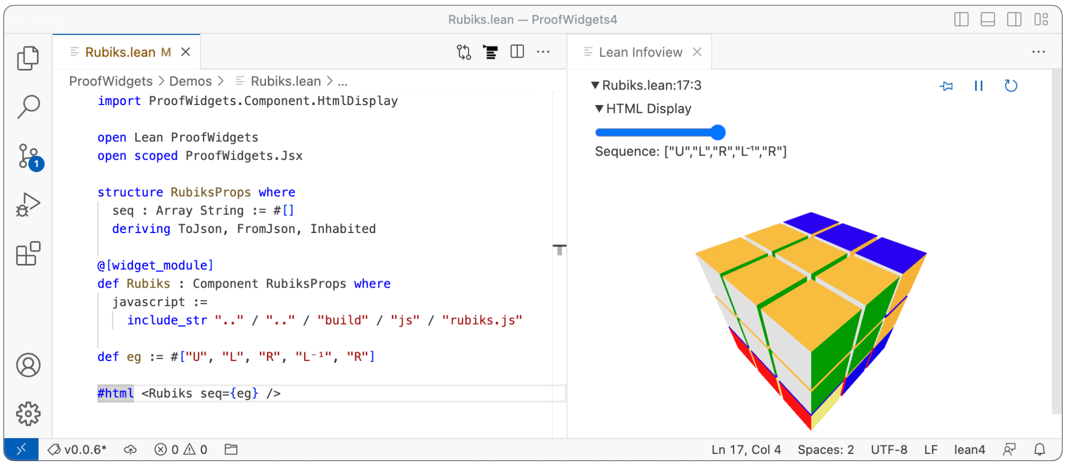
```
@[widget_module]
def PenroseDiagram : Component PenroseDiagramProps where
  javascript := ... -- Details omitted
```

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.

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 4).

More complex visualizations are enabled by building on further JavaScript libraries as in Figure 5. For example, a component integrating a plotting library could be a starting point for plotting functions in a formally verified way [34]. 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 Figure 7.

In the second step, we write a Lean metaprogram to display the user widget. There are many ways to do this in general. Since Figure 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



```
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} />
```

```
Sequence: ["U","L","R","L⁻¹","R"]
```

Figure 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.

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
```

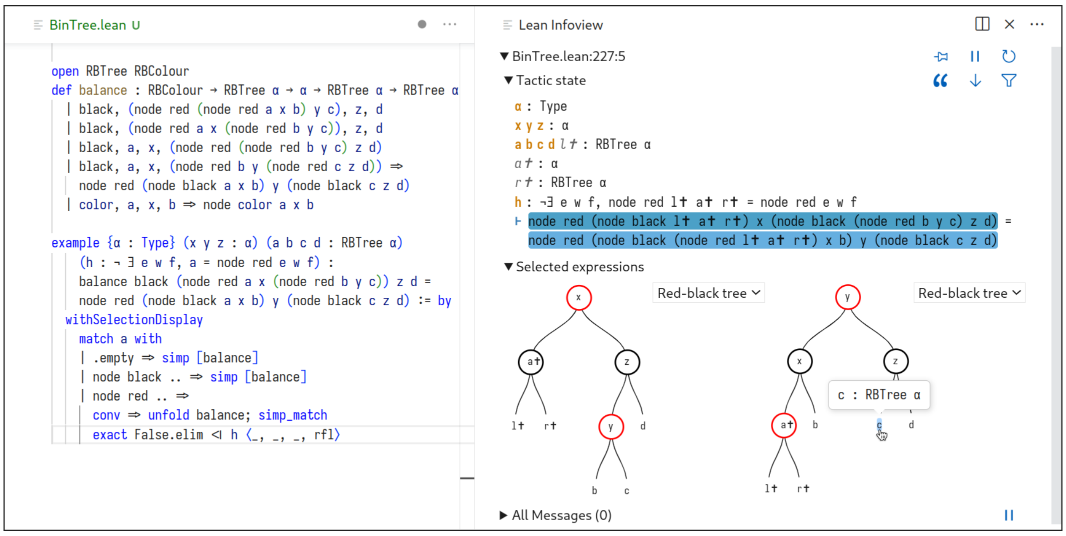
```
    -- Instantiate a PenroseDiagram using a JSX-like EDSL.
    <PenroseDiagram
      embeds={embeds}
      -- Penrose specification of general commutative diagrams.
      dsl={include_str "commutative.dsl"}
      sty={include_str "commutativeOpt.sty"}
      -- The particular diagram we are given.
      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
```

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 Figure 4 just the target type was selected. The `withSelectionDisplay` combinator, which we use there and in Figure 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).

Selecting more than one subexpression can be helpful in comparing differences between these subexpressions, to figure out what remains to be proven. In Figure 6, we copied a balancing function for red-black trees verbatim from Okasaki [37]. 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 Figure 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`[5] 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.

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



```
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
```

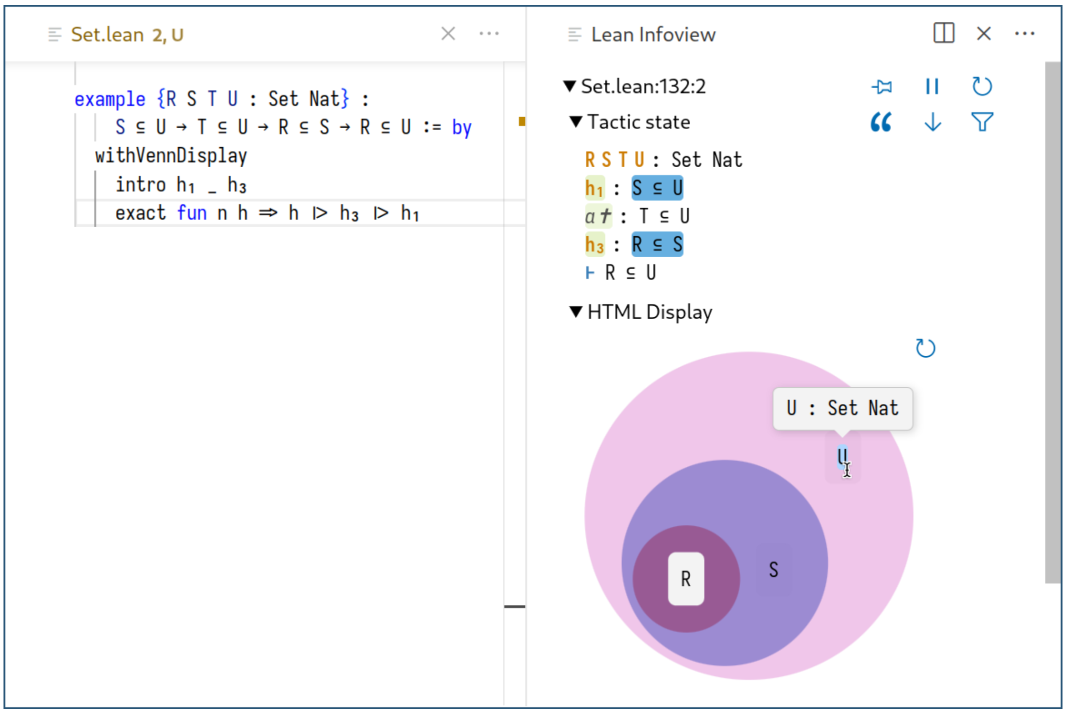
Figure 6 A balancing function for red-black trees is implemented in `balance`. Two terms appearing in the course of a proof about it are selected in the goal and illustrated as trees.

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 Figure 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.

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* [10, 11] 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 [38] 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 [12], or a system based on recent advances in deep learning [28], 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.*



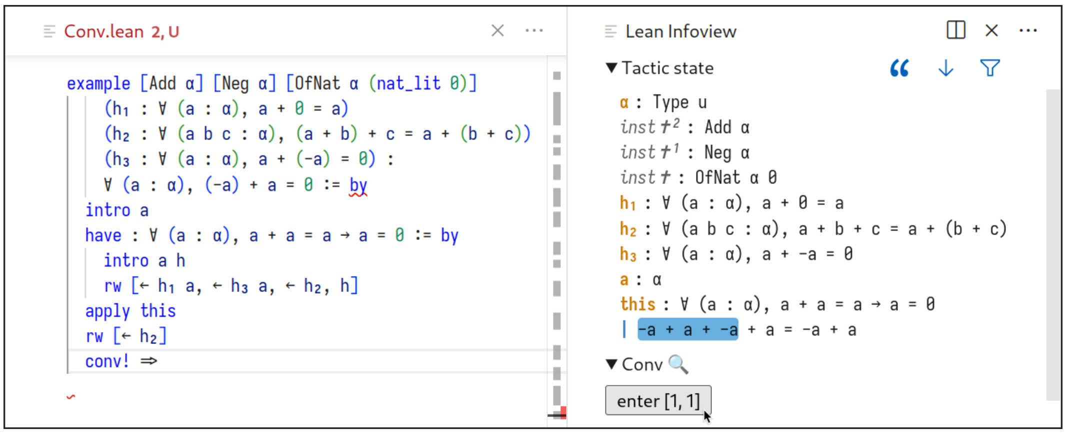
```
example {R S T U : Set Nat} :
    S ⊆ U → T ⊆ U → R ⊆ S → R ⊆ U := by
  withVennDisplay
    intro h₁ _ h₃
    exact fun n h ⇒ h ▷ h₃ ▷ h₁
```

```
▼Set.lean:132:2
 ▼Tactic state
  R S T U : Set Nat
  h₁ : S ⊆ U
  a✝ : T ⊆ U
  h₃ : R ⊆ S
  ⊢ R ⊆ U
 ▼HTML Display
```

Figure 7 A subset of hypotheses relevant to the proof is selected. The set relationships are visualized in one Venn diagram.

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 Figure 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.

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 [40], manipulations of string diagrams [26], or more specific variants in category theory such as ZX-diagrams [18], Globular proofs [6], and `homotopy.io` [41] 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.



```
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 8 A subterm `-a + a + -a` of the goal in a proof about groups [21] 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.

4 Implementation

A complete setup consists of three components. Figure 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[6]). 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.

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

[6] `https://microsoft.github.io/language-server-protocol/`

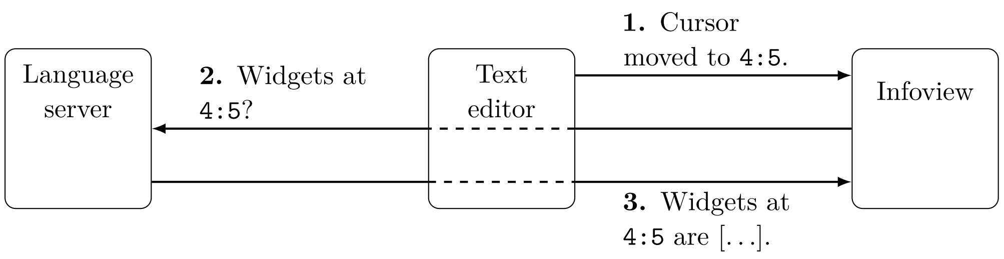


Figure 9 In order to determine which user widgets to show in the infoview, a sequence of messages is exchanged every time the text cursor moves. First, the editor informs the infoview about the new cursor position (here line `4`, column `5`). Then, the infoview queries the language server for the user 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.

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[7], a simple protocol for remote procedure calls which 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 and then infer the type of a subexpression of that (as do the two popups in Figure 2). Since the relevant data is not serialized, client-side code needs a way of referencing objects stored in the server's memory. This is achieved by allowing JSON-RPC payloads to contain opaque references to server-side objects. A value of any type may be referenced opaquely by being marked with the `WithRpcRef` type-level function. To ensure type correctness, runtime type information is stored 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 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 [5].

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

[7] `https://www.jsonrpc.org/`

languages. Using a `FinalizationRegistry`[8] 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 [13]. Upon not seeing any `keepalives` 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[9]. 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 accessible 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 Related work

Our work descends directly from graphical tooling for Lean 3 (the previous version of Lean), notably `ProofWidgets 3` [5] 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 [35] 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` [33, 16, 27] 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

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

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 [8] and its successor Pcoq [2] 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 [9], a display for statements in plane geometry. Nevertheless Pcoq does not appear to support general user-extensibility. The GeoProof [36] 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 [42] focuses on visualizing proof trees and steps, but not the mathematical objects appearing therein such as the diagrams of Figure 4. The recent Actema project [22] aims to extend the interactions available in proof by pointing to drag-and-drop interfaces. KeY [1] and KeYmaera X [24] provide interfaces specific to software verification and purpose-built logics. The *Incredible Proof Machine* [14] 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 [48]. 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` [3] and Clide [32]. 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` [39] supports proof *archival* in Coq and in Lean (via `LeanInk` [15]) 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 [4] used to support expression presentations, but only for the LEGO prover [31]. Feasibility of real-time asynchronous processing was demonstrated in Isabelle/PIDE [46]. 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 $L\Omega UI$ [44], HyperProof [7] and XBarnacle [30] projects.

Finally, we are generally inspired by Engelbart's (to-date not realized!) vision of human intelligence augmented through computer interfaces [23], and the systems of yore which followed it including Smalltalk [25].

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 Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. *Deductive Software Verification - The KeY Book*. Lecture Notes in Computer Science. Springer, 2016. `doi:10.1007/978-3-319-49812-6`.

2 Ahmed Amerkad, Yves Bertot, Loïc Pottier, and Laurence Rideau. Mathematics and Proof Presentation in Pcoq. Technical Report RR-4313, INRIA, November 2001. URL: `https://hal.inria.fr/inria-00072274`.

3 Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. jscoq: Towards hybrid theorem proving interfaces. In Serge Autexier and Pedro Quaresma, editors, *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers, UITP 2016, Coimbra, Portugal, 2nd July 2016*, volume 239 of *EPTCS*, pages 15–27, 2016. `doi:10.4204/EPTCS.239.2`.

4 David Aspinall. Proof general: A generic tool for proof development. In Susanne Graf and Michael I. Schwartzbach, editors, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–43. Springer, Springer, 2000. `doi:10.1007/3-540-46419-0_3`.

5 Edward W. Ayers, Mateja Jamnik, and William T. Gowers. A graphical user interface framework for formal verification. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 4:1–4:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ITP.2021.4`.

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

7 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`.

8 Yves Bertot. The ctcoq system: Design and architecture. *Formal Aspects Comput.*, 11(3):225–243, 1999. `doi:10.1007/s001650050049`.

9 Yves Bertot, Frédérique Guilhot, and Loic Pottier. Visualizing geometrical statements with geoview. In David Aspinall and Christoph Lüth, editors, *Proceedings of the User Interfaces for Theorem Provers Workshop, UITP@TPHOLs 2003, Rome, Italy, September 8, 2003*, volume 103 of *Electronic Notes in Theoretical Computer Science*, pages 49–65. Elsevier, 2003. `doi:10.1016/j.entcs.2004.09.013`.

10 Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160. Springer, 1994. `doi:10.1007/3-540-57887-0_94`.

11 Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *J. Symb. Comput.*, 25(2):161–194, 1998. doi:10.1006/jsco.1997.0171.

12 Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. *J. Autom. Reason.*, 51(1):109–128, 2013. doi:10.1007/s10817-013-9278-5.

13 R. Braden. Requirements for internet hosts - communication layers. RFC 1122, RFC Editor, October 1989. URL: https://www.rfc-editor.org/rfc/rfc1122.txt.

14 Joachim Breitner. Visual theorem proving with the incredible proof machine. In Jasmin Christian Blanchette and Stephan Merz, editors, *International Conference on Interactive Theorem Proving*, pages 123–139. Springer, 2016. doi:10.1007/978-3-319-43144-4_8.

15 Niklas Bülow. Proof visualization for the lean 4 theorem prover, April 2022.

16 David Christiansen, David Darais, and Weixi Ma. The final pretty printer, 2016. URL: https://web.archive.org/web/20230219222209/https://davidchristiansen.dk/drafts/final-pretty-printer-draft.pdf.

17 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.

18 Bob Coecke and Ross Duncan. Interacting quantum observables. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *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*, volume 5126 of *Lecture Notes in Computer Science*, pages 298–310. Springer, 2008. doi:10.1007/978-3-540-70583-3_25.

19 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.

20 Silvia de Toffoli. Chasing the diagram–the use of visualizations in algebraic reasoning. *Review of Symbolic Logic*, 10(1):158–186, 2017. doi:10.1017/s1755020316000277.

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

22 Pablo Donato, Pierre-Yves Strub, and Benjamin Werner. A drag-and-drop proof tactic. In Andrei Popescu and Steve Zdancewic, editors, *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 197–209. ACM, 2022. doi:10.1145/3497775.3503692.

23 Douglas C. Engelbart. Augmenting human intellect: A conceptual framework. Technical report, Stanford Research Institute, October 1962. URL: https://web.archive.org/web/20230220110343/https://dougengelbart.org/pubs/augment-3906.html.

24 Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In Amy P. Felty and Aart Middeldorp, editors, *CADE*, volume 9195 of *LNCS*, pages 527–538. Springer, 2015. doi:10.1007/978-3-319-21401-6_36.

25 Adele Goldberg. *Smalltalk-80 - the interactive programming environment.* Addison-Wesley, 1984.

26 André Joyal and Ross Street. The geometry of tensor calculus, i. *Advances in mathematics*, 88(1):55–112, 1991.

27 Joomy Korkut and David Thrane Christiansen. Extensible type-directed editing. In Richard A. Eisenberg and Niki Vazou, editors, *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2018, St. Louis, MO, USA, September 27, 2018*, pages 38–50. ACM, 2018. doi:10.1145/3240719.3241791.

28 Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. Hypertree proof search for neural theorem proving. *CoRR*, abs/2205.11491, 2022. `doi:10.48550/arXiv.2205.11491`.

29 Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65–100, 1987. `doi:10.1111/j.1551-6708.1987.tb00863.x`.

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

31 Zhaohui Luo and Robert Pollack. Lego proof development system: User's manual. Technical report, LFCS, Edinburgh University, 1992. URL: `https://www.lfcs.inf.ed.ac.uk/reports/92/ECS-LFCS-92-211/`.

32 Christoph Lüth and Martin Ring. A web interface for isabelle: The next generation. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka, and Wolfgang Windsteiger, editors, *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*, volume 7961 of *Lecture Notes in Computer Science*, pages 326–329. Springer, 2013. `doi:10.1007/978-3-642-39320-4_22`.

33 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`.

34 Guillaume Melquiond. Plotting in a formally verified way. In José Proença and Andrei Paskevich, editors, *Proceedings of the 6th Workshop on Formal Integrated Development Environment, F-IDE@NFM 2021, Held online, 24-25th May 2021*, volume 338 of *EPTCS*, pages 39–45, 2021. `doi:10.4204/EPTCS.338.6`.

35 Robert B Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 267–277, 1968.

36 Julien Narboux. A graphical user interface for formal proofs in geometry. *J. Autom. Reason.*, 39(2):161–180, 2007. `doi:10.1007/s10817-007-9071-4`.

37 Chris Okasaki. Red-black trees in a functional setting. *J. Funct. Program.*, 9(4):471–477, 1999. `doi:10.1017/s0956796899003494`.

38 Lawrence C. Paulson. Thoughts on user interfaces for theorem provers, December 2022. URL: `https://web.archive.org/web/20230219221749/https://lawrencecpaulson.github.io/2022/12/14/User_interfaces.html`.

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

40 Kurt Reidemeister. *Knot theory*. BCS Associates, 1983.

41 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. IEEE Press, 2021.

42 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`.

43 Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. Tabled Typeclass Resolution. *CoRR*, 2020. `arXiv:2001.04301v2`.

44 Jörg Siekmann, Stephan Hess, Christoph Benzmüller, Lassaad Cheikhrouhou, Armin Fiedler, Helmut Horacek, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, Martin Pollet, and Volker Sorge. Loui: Lovely omega user interface. *Formal Aspects of Computing*, 11(3):326–342, 1999. `doi:10.1007/s001650050053`.

45 Aaron Stockdill, Daniel Raggi, Mateja Jamnik, Grecia Garcia Garcia, and Peter C.-H. Cheng. Considerations in representation selection for problem solving: A review. In Amrita Basu, Gem Stapleton, Sven Linker, Catherine Legg, Emmanuel Manalo, and Petrucio Viana, editors, *Diagrammatic Representation and Inference*, pages 35–51, Cham, 2021. Springer International Publishing.

46 Makarius Wenzel. Asynchronous user interaction and tool integration in isabelle/pide. In Gerwin Klein and Ruben Gamboa, editors, *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*, volume 8558 of *Lecture Notes in Computer Science*, pages 515–530. Springer, 2014. `doi:10.1007/978-3-319-08970-6_33`.

47 Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. Penrose: from mathematical notation to beautiful diagrams. *ACM Trans. Graph.*, 39(4):144, 2020. `doi:10.1145/3386569.3392375`.

48 Bohua Zhan, Zhenyan Ji, Wenfan Zhou, Chaozhu Xiang, Jie Hou, and Wenhui Sun. Design of point-and-click user interfaces for proof assistants. In Yamine Aït Ameur and Shengchao Qin, editors, *Formal Methods and Software Engineering - 21st International Conference on Formal Engineering Methods, ICFEM 2019, Shenzhen, China, November 5-9, 2019, Proceedings*, volume 11852 of *Lecture Notes in Computer Science*, pages 86–103. Springer, 2019. `doi:10.1007/978-3-030-32409-4_6`.

49 Jiaje Zhang and Donald A. Norman. Representations in distributed cognitive tasks. *Cognitive Science*, 18(1):87–122, 1994. `doi:10.1016/0364-0213(94)90021-3`.