A Graphical User Interface Framework for Formal Verification

Edward W. Ayers
DPMMS, University of Cambridge, UK

Mateja Jamnik
Department of Computer Science and Technology, University of Cambridge, UK

W. T. Gowers
DPMMS, University of Cambridge, UK

Abstract

We present the "ProofWidgets" framework for implementing general user interfaces (UIs) within an interactive theorem prover. The framework uses web technology and functional reactive programming, as well as metaprogramming features of advanced interactive theorem proving (ITP) systems to allow users to create arbitrary interactive UIs for representing the goal state. Users of the framework can create GUIs declaratively within the ITP's metaprogramming language, without having to develop in multiple languages and without coordinated changes across multiple projects, which improves development time for new designs of UI. The ProofWidgets framework also allows UIs to make use of the full context of the theorem prover and the specialised libraries that ITPs offer, such as methods for dealing with expressions and tactics. The framework includes an extensible structured pretty-printing engine that enables advanced interaction with expressions such as interactive term rewriting. We exemplify the framework with an implementation for the leanprover-community fork of Lean 3. The framework is already in use by hundreds of contributors to the Lean mathematical library.

2012 ACM Subject Classification Software and its engineering → Software usability

Keywords and phrases User Interfaces, ITP

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.4

Supplementary Material The supplementary material presented with this paper is incorporated into the leanprover-community GitHub repositories.
Software (Server): `https://github.com/leanprover-community/lean/tree/master/library/init/meta/widget`; archived at `swh:1:dir:65d6fe171a7697793be204922aba83f2a94f5d20`
Software (Client): `https://github.com/leanprover/vscode-lean/blob/master/infoview/widget.tsx`; archived at `swh:1:cnt:e713dbc30927867464effc1e51fd1230cd961cbd`

Funding *Edward W. Ayers*: EPSRC 1804138.

Acknowledgements Enormous thanks to Gabriel Ebner and Brian Gen-ge Chen for reviewing the PRs and polishing the implementation. Also thanks for using the framework to create their own widgets, contributing and suggesting improvements: Daniel Fabian; Robert Y. Lewis; Markus Himmel; Minchao Wu; Kendall Frey; Patrick Massot.

1 Introduction

Modern ITP systems such as Isabelle, Coq and Lean use advanced language servers and protocols to interface with text editors to produce a feature-rich proving experience. These systems have helpful features such as syntax highlighting and code completion suggestions as would be found in normal programming language tooling. They additionally include prover-specific features such as displaying the goal state and providing interactive suggestions

© Edward W. Ayers, Mateja Jamnik, and W. T. Gowers;
licensed under Creative Commons License CC-BY 4.0

of tactics to apply in proof construction. ITP offers some additional GUI[1] challenges above what one might find in developing an editor extension for a standard programming language, because the process of proving is inherently interactive: the user is constantly observing the goal state of the prover and using this information to inform their next command in the construction of a proof.

During research into new ways of interacting within Lean 3 theorem prover, we became frustrated with the development workflow for prototyping GUIs for ITP. Each time the interface design changes, one needs to coordinate changes across three different codebases; the Lean core, the VSCode[2] editor extension and our project directory. It became clear that any approach to creating GUIs in which the editor code needs to be aware of the datatypes used within the ITP metalogic is doomed to require many coordinated changes across multiple codebases. This inspired our alternative approach; we designed a full-fledged GUI framework in the metalogic of the ITP itself. This approach has the advantage of tightening the development loop and has more general use outside of our particular project, and across different ITP systems in general.

In this paper we present the ProofWidgets framework, which enables implementation of UIs with a wide variety of features, for example:

- Interactive term inspection: the ability to inspect the tree structure of expressions by hovering the mouse over different parts of a pretty printed expression.
- Discoverable tactics: interactive suggestions of available tactics for a given goal.
- Discoverable term rewriting: the ability to inspect the equational rewrites at a particular position in the expression tree.
- Custom visualisations of structures like matrices, plots and graphs as well as rendering LaTeX mathematical typesetting.
- The ability to implement language features such as go-to-definition for pretty-printed expressions (as opposed to just text that appears in the editor).

Some of these features have been implemented to an extent within other provers (see Section 5). However, ProofWidgets provides a unified, underlying framework for implementing user interfaces in general, which can be used to implement these features in a customisable, extensible and portable way.

The contributions presented in this paper are:

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

This paper is structured as follows. In Section 2 we provide an overview of some background topics to contextualise the work. Section 3 details the design goals and specification of the ProofWidgets framework. Section 4 presents the ProofWidgets implementation for Lean 3. Section 5 discusses related approaches. Section 6 is the conclusion and contains some potential ideas for future work and extensions to the framework.

[1] Graphical User Interface
[2] Visual Studio Code `code.visualstudio.com`

2 Background

2.1 Web-apps

Web-apps are ubiquitous in modern software. By a web-app, we mean any software that uses modern browser technology to implement a graphical application. Web-apps are attractive targets for development because they are platform independent and can be delivered from a server on the internet using a browser or be packaged as an entirely local app using a packaging framework such as electron. Many modern desktop and mobile applications such as VSCode are thinly veiled browser windows.

To summarise the anatomy of a web-app: the structure of a web-page is dictated by a tree structure called the Document Object Model (DOM). The DOM is an abstract representation of the tree structure of an XML or HTML document with support for event handling as might occur as a result of user interaction. The word *fragment* is used to denote a valid subtree structure that is not the entire document. So for example, an "HTML fragment" is used to denote a snippet of HTML that could be embedded within an HTML document. With the help of a CSS style sheet, the web browser paints this DOM to the screen in a way that can be viewed and interacted with by a user. Through the use of JavaScript and event handlers, a webpage may manipulate its own DOM in response to events to produce interactive web-applications. Modern browsers support W3C standards for many advanced features: video playback, support for touch and ink input methods, drag and drop, animation, 3D rendering and many more. HTML also has a set of widely supported accessibility features called ARIA which can be used to ensure that apps are accessible to all. The power of web-apps to create portable, fully interactive user interfaces has clear applications for ITP and indeed many have already been created (see Section 5 for a review).

2.2 Code editors and client-server protocols

Some modern code editors such as Atom and VSCode are built using web technology. In order to support tooling features such as go-to-definition and hover information, these editors act as the client in a client/server relationship with an independently running *language server*. As the user modifies the code in the client editor, the client communicates with the server: notifying it when the document changes and sending requests for specific information based on the user's interactions. As noted in the introduction, in ITP this communication is more elaborate than in a normal programming language.

The most important thing to note here is that changing the communication protocol between the client and the server is generally hard, because the developer has to update the protocol in both the server and the client. There may even be multiple clients. This makes it difficult to quickly iterate on new UI designs. A way of solving this protocol problem is to offer a much tighter integration by combining the codebases for the editor and the ITP. This is the approach taken by Isabelle/PIDE/jEdit [21] and has its own trade-offs as discussed further in Section 5.

2.3 Functional GUI frameworks

Most meta-level programming languages for ITPs are functional programming languages.[3] However GUIs are inherently mutable objects that need to react to user interaction. Reactive programming [1] enables the control of the inherently mutating GUI within a pure functional

[3] ML and Scala for Isabelle, OCaml for Coq, Lean for Lean.

programming interface.[4] The ideas of reactive programming have achieved a wide level of adoption in web-app development thanks to the React JavaScript library and the Elm programming language [6].

The programming model used by these reactive frameworks is to model a user interface as a pure *view* function from a data source (e.g., a shopping list) to a DOM tree and an *update* function for converting user input events to a new version of the data (e.g., adding an item to the shopping list). Once the update function is applied and the data has been updated, the system reevaluates the view function on the new data and mutates the DOM to update. Although this may sound inefficient - recomputing the entire tree each time - there is an optimisation available: if the view function contains nested view functions, one can memoise these functions and avoid updating the parts of the DOM that have not changed.

A performance bottleneck in web-apps is the layout and painting stages of the browser rendering pipeline; the process by which the abstract DOM is converted to pixels on a screen.[5] One way to reduce the processing time spent on layout and painting is to minimise the number of changes made to the DOM. Both Elm and React achieve this through use of a "Virtual DOM" (VDOM). This is where a shadow tree isomorphic to the DOM is kept. When the data updates, the view function creates a new VDOM tree. This tree is then *diffed* with the previous VDOM tree to produce a minimal set of changes to the real DOM. In React, this diffing algorithm is called *reconciliation*.[6]

2.4 Lean

Lean [8] is an interactive theorem prover whose underlying logic is a dependent type theory called the calculus of inductive constructions. Lean verifies the correctness of proofs using its kernel, which type-checks proof terms (terms whose type is a proposition). Most users of Lean prove theorems using its tactic language. This language amounts to a sequence of invocations of the `tactic` monad, which enables one to write proof scripts in a similar style to that popularised by the LCF provers [12]. Most notably for our purposes, between each tactic invocation, Lean stores the *goal state* at that point, which amounts to a list of contexts and types that need to be inhabited to complete the proof. This goal state is pretty printed and sent for viewing in the client editor as plaintext, with some additional formatting (i.e., syntax highlighting) applied in the client. Lean also allows users to write custom tactic languages.

3 Framework architecture

The ProofWidgets framework has the following design goals:

- Programmers write GUIs using the metaprogramming framework of the ITP.
- Programmers are given an API that can produce arbitrary DOM fragments, including inline CSS styles.
- No cross-compilation to JavaScript or WebAssembly: the GUI-generating code must run in the same environment as the tactic system. This ensures that the user interaction handlers have full access to the tactic execution context, including the full database of definitions

[4] A similar paradigm is that of *functional reactive programming* (FRP) first invented by Elliot [9]. This is distinguished from general reactive programming by the explicit modelling of time.

[5] See this chromium documentation entry for more information on critical paths in browser rendering. `https://www.chromium.org/developers/the-rendering-critical-path`

[6] `https://reactjs.org/docs/reconciliation.html`

and lemmas, as well as all of the metaprogramming library. In a cross-compilation based approach (implementation difficulty notwithstanding), the UI programmer would have to choose which parts of this context to export to the client.

- To support interactively discoverable tactics, the system needs to be able to command the client text editor to modify its sourcetext.
- The pretty printer must be extended to allow for "interactive expressions": expressions whose tree structure may be explored interactively.
- Programmers should be able to create visualisations of their data.
- It should be convenient for programmers to be able to style their GUIs in a consistent manner.
- The GUI programming model should include some way of managing local UI state, for example, whether or not a tooltip is open.
- The GUI should be presented in the same output panel that the plaintext goal state was presented in.
- The framework should be backwards compatible with the plaintext goal state system. Users should be able to opt out of the GUI if they do not like it or want to use a non web-app editor such as Emacs.

These goal specifications led us to design ProofWidgets to use a declarative VDOM-based architecture similar to that used in the Elm programming language [6] and the React JavaScript library. By using the same programming model, we can leverage the familiarity with commonly used React and Elm. In the following subsections we will detail the design of ProofWidgets, starting with the UI programming model (Section 3.1) and the client/server protocol (Section 3.2).

3.1 UI programming model

New user interfaces are created using the `Html` and `Component` types. A user may define an HTML fragment by constructing a member of the inductive datatype `Html`, which is either an element (e.g., `<div></div>`), a string or an object called a component to be discussed shortly.

These fragments can have *event handlers* attached to them. For example, a button could have an event attribute `onclick` (as used in Listing 1) which accepts a handler $h : (\texttt{Unit} \to \alpha)$ sending the unit type to a member of some type α. When this interface is rendered in the client and the button is clicked, the server is notified and causes the node to "emit" the element $h() : \alpha$. The value of $h()$ is then propagated towards the root of the `Html` tree until it reaches a component.

A component is an inductive datatype taking two type parameters: π (the props type) and α (the action type).[7] It represents a stateful object in the user interface tree where the state $s : \sigma$ can change as a result of a user interaction event. By "stateful" we mean an object which holds some mutating state for the lifetime of the user interface. Through the use of components, it is possible to describe the behaviour of this state without having to leave the immutable world of a pure functional programming language. Three functions determine the behaviour of the component:

- $\texttt{init} : \pi \to \sigma$ initialises the state.
- $\texttt{view} : \pi \to \sigma \to \texttt{Html}\ \alpha$ maps the state to a VDOM tree.

[7] This is designed to be familiar to those who use React components `https://reactjs.org/docs/components-and-props.html`.

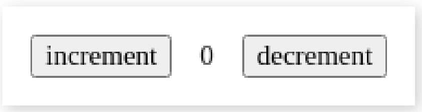


Figure 1 The output rendering of `counter` created in Listing 1.

- `update` : $\pi \to \alpha \to \sigma \to \sigma \times$ `Option` β is run when a user event is triggered in the child HTML tree returned by `view`. The emitted value $a : \alpha$ is used to produce a tuple $\sigma \times$ `Option` β consisting of a new state $s : \sigma$ and optionally, a new event $b : \beta$ to emit. If the new event is provided, it will propagate further towards the root of the VDOM tree and be handled by the next component in the sequence.

For example, a simple counter component (see Listing 1 and Figure 1) has an integer s for a state, and updating the state is done through clicking on the "increment" and "decrement" buttons which will emit 1 and -1 when clicked. The values a are used to update the state to $a + s$. Creating stateful components in this way has a variety of practical uses when building user interfaces for inspecting and manipulating the goal state. We will see in Section 4.1 that a state is used to represent which expression the user has clicked. Indeed, an entire tactic state can be stored as the state of the component. Then the update function runs various tactics to update the tactic state and output the new result.

Listing 1 Pseudocode listing showing a simple counter app showcasing statefulness. The output is shown in Figure 1.

```
counter : Component Unit Empty
counter := with_state
  ( init = (p ↦ 0)
  , view = (p ↦ i ↦
    <div>
      <button onclick={() ↦ 1}>"increment"</button>
      i
      <button onclick={() ↦ −1}>"decrement"</button>
    </div>)
  , update = (p ↦ a ↦ s ↦ (a + s, none))
  )
```

3.2 Client/server protocol

Once the programmer has built an interface using the API introduced in Section 3.1, it needs to be rendered and delivered to the browser output window. ProofWidgets extends the architecture discussed in Section 2.2 with an additional protocol for controlling the life-cycle of a user interface rendered in the client editor (Figure 2). When a sourcefile for the prover is opened (in Figure 2, `myfile.lean`), the server begins parsing, elaborating and verifying this sourcefile as usual. The server incrementally annotates the sourcetext as it is processed and these annotations are stored in memory. The annotations include tracing diagnostics messages as well as thunks[8] of the goal states at various points in a proof. When the user

[8] A thunk is a lazily evaluated expression.

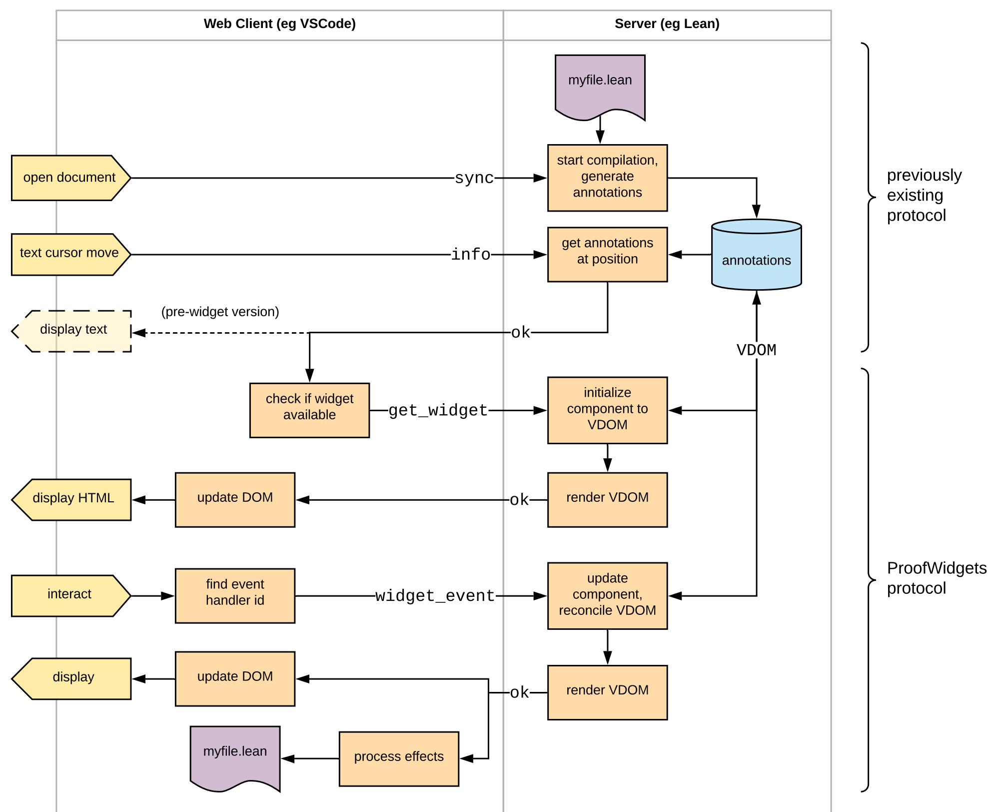


Figure 2 The architecture of the ProofWidgets client/server communication protocol. Arrows that span the dividing lines between the client and server components are API requests and responses. The contribution of this paper is present in the section marked "ProofWidgets protocol". The arrows crossing the boundary between the client and server applications are sent in the form of JSON messages. Rightward arrows are **requests** and leftward arrows are **responses**.

clicks on a particular piece of sourcecode in the editor ("text cursor move" in Figure 2), the client makes an `info` request for this position to the server, which responds with an `ok` response containing the logs at that point.

The ProofWidgets protocol extends the `info` messages to allow the prover to similarly annotate various points in the document with VDOM trees as introduced in Section 2.3. These annotating components have the type `Component TacticState Empty` where `TacticState` is the current state of the prover and `Empty` is the uninhabited type. A default component for rendering goals of proof scripts is provided, but users may override this with their own components. The VDOM trees are derived from this component, where the VDOM has the same tree structure as the `Html` datatype (i.e., a tree of elements, strings and components), but the components in the VDOM tree also contain the current state and the current child subtree of the component. This serves the purpose of storing a model of the current state of the user interface. These VDOMs can be *rendered* to HTML fragments that are sent to the client editor and presented in the editor's output window.

There are two ways to create a VDOM tree from a component: from scratch using **initialisation** or by updating an existing VDOM tree using **reconciliation**.

Initialisation is used to create a fresh VDOM tree. To initialise a component, the system first calls `init` to produce a new state s. s is fed to the `view` method to create an `Html` tree t. Any child components in t are recursively initialised.

The inputs to reconciliation are an existing VDOM tree v and a new `Html` tree t. t is created when the `view` function is called on a parent component. The goal of reconciliation is to create a new VDOM tree matching the structure of t, but with the component states from v transferred over. The tree diffing algorithm that determines whether a state should be transferred is similar to the React reconciliation algorithm[9] and so we will omit a discussion of the details here. The main point is that when a user interface changes, the states of the components are preserved to give the illusion of a mutating user interface.

For interaction, the HTML fragment returned from the server may also contain event handlers. Rather than being calls to JavaScript methods as in a normal web-app, the client editor intercepts these events and forwards them to the server using a `widget_event` request. The server then **updates** the component according to the event to produce a new `Html` tree that is reconciled with the current VDOM tree. The ProofWidgets framework then responds with the new HTML fragment derived from the new VDOM tree. In order to ensure that the correct event handler is fired, the client receives a unique identifier for each handler that is present on the VDOM and returns this identifier upon receiving a user interaction. So, in effect, the ITP server performs the role of an event handler: processing a user interaction and then updating the view rendered to the screen accordingly. In addition to updating the view, the response to a `widget_event` request may also contain **effects**. These are commands to the editor, for example revealing a certain position in the file or inserting text at the cursor position. Effects are used to implement features such as go-to definition and modifying the contents of sourcefiles in light of a suggested modification to advance the proof state. If a second user interaction event occurs while the first is being handled, the server will queue these events.

The architecture design presented above is a different approach to how existing tools handle the user interface. It offers a much smaller programming API consisting of `Component` and `Html` and a client/server protocol that supports the operation of arbitrary user interfaces controlled by the ITP server. Existing tools (Section 5) instead give fixed APIs for interaction with the ITP, or support rendering of custom HTML without or with limited interactivity.

To implement ProofWidgets for an ITP system, it is necessary to implement the three subsystems that have been summarised in this section: a programming API for components; the client editor code (i.e., the VSCode extension) that receives responses from the server and inserts HTML fragments to the editors output window; and the server code to initialise, reconcile and render these components.

4 Implementation and applications

In this section, we present the Lean implementation of ProofWidgets and discuss a set of example widgets for interacting with proof objects.

The VSCode extension for Lean[10] has an output pane called the “infoview” (the right-hand pane in Figure 3) which is configured to use the ProofWidgets protocol. This infoview window runs as a sandboxed web-browser instance and uses React to manage updating the

[9] `https://reactjs.org/docs/reconciliation.html`
[10] `https://github.com/leanprover/vscode-lean`

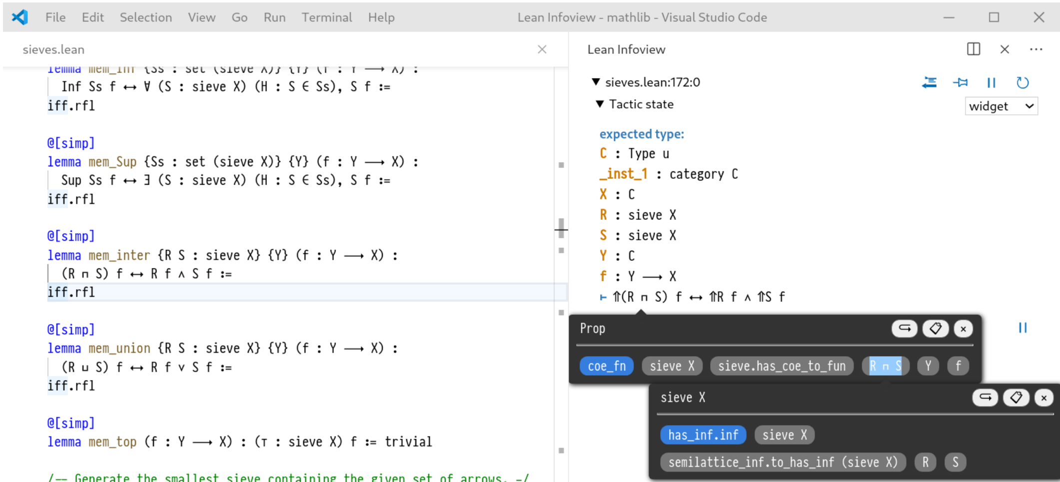


Figure 3 Screenshot showing the interactive expression view in action within the Lean theorem prover. The left-hand pane is the Lean source document and the right-hand pane is the infoview showing the context and expected type at the editor's cursor. There are two nested tooltips; one giving information about an expression in the infoview and the other on an expression within the first tooltip. The information that the tooltip provides is customisable, currently showing the type of the expression and a list of implicit arguments for the given expression.

DOM based on the HTML fragments received from the server. Lean ProofWidgets can also work over remote proving sessions (where the client editor is running on a different machine to the ITP server, possibly in another country).

4.1 Interactive Expressions

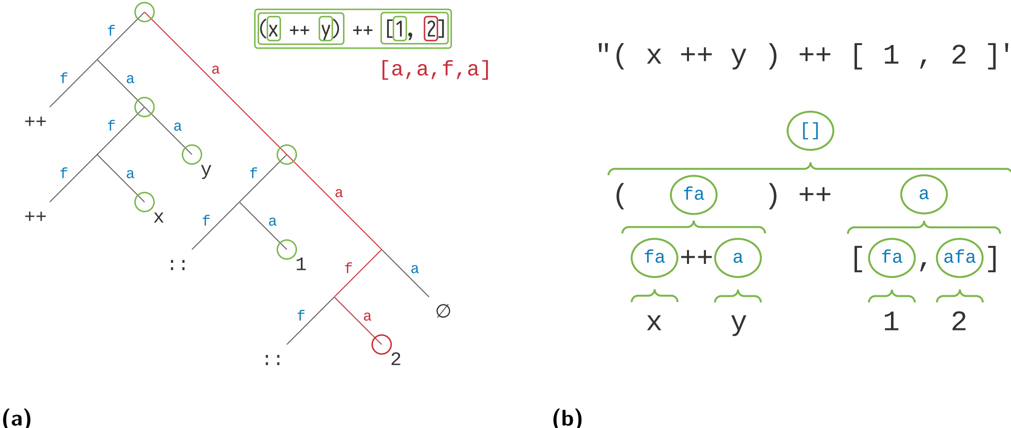
By "interactive expressions" we mean augmenting a printed expression string with a mapping structure such that the software can determine the correspondence between substrings and subexpressions. This mapping can then be used to create interactive term inspectors and tactics. As will be discussed in Section 5, pretty printing with this additional information is not a novel feature, however the way in which we have designed it makes these features available for producing proofs. Richly decorating expressions returned from the prover was first described by Bertot and Laurent as 'proof by pointing' [4].

An example of the interactive expression module in action is given in Figure 3: as one hovers over the various expressions and subexpressions in the infoview, one gets semantically correct highlighting for the expressions, and when you click on a subexpression, a tooltip appears containing type information. This tooltip is itself a component and so can also be interacted with, including opening a nested tooltip.

A number of other features are demonstrated in Figure 3:

- Hovering over subterms highlights the appropriate parts of the pretty printed string.
- The buttons in the top right of the tooltip activate effects including a "go to definition" button and a "copy type to clipboard" button.
- Expressions within the tooltip can also be explored in a nested tooltip. This is possible thanks to the state tracking system detailed in Section 3.2.

Note that the Lean editor already has features for displaying type information for the source document with the help of hover info. However, this tooltip mechanism is only textual (not interactive) and only works on expressions that have been written in the source document. Prior to ProofWidgets there was no way to inspect expressions as they appeared in the infoview.



(a) **(b)**

Figure 4 An expression tree for `(x ++ y) ++ [1,2]` is shown in Figure 4a. Each `f` or `a` above the lines is an expression coordinate. The red `[a,a,f,a]` example of an expression address, corresponding to the red line on the tree. Each green circle in the tree will pretty-print to a string independent of the expression tree above it. Figure 4b shows a `eformat` tree produced by pretty-printing the expression `(x ++ y) ++ [1,2]`. The green circles are `eformat.tag` constructors and the blue address text within is the relative address of the `tag` in relation to the `tag` above it. So that means that the full expression address for a subterm can be recovered by concatenating the `Address`es above it in the tree, for example the 2 subexpression is at `[] ++ [a] ++ [a,f,a] = [a,a,f,a]`.

All of the code which dictates the appearance and behaviour of the infoview widget is written in Lean and reloads dynamically when its code is changed. This means that users can produce their own custom tooltips and improve upon the infoview experience within their own Lean project.

In terms of performance, the generality of the ProofWidgets architecture means that it is possible for the user of the UI to execute long-running calculations or to render a large VDOM. This would result in creating a sluggish UI. However, for realistic use cases, such as producing a goal state widget as shown in Figure 3, the system should be responsive. Using the Chromium developer tools, the round trip time from a mouse pointer movement to updating the pixels on the screen was measured to be less than 80ms on an Intel i7 laptop from 2012 for a goal state widget with over 1000 VDOM nodes. This means that the system can easily handle events that need to be responsive without requiring more advanced hardware requirements than would be needed for ITP without ProofWidgets.

To support interactive expressions, we modified the Lean pretty-printer. Prior to our modifications, the pretty-printer would take an expression and a context for the expression and produce a member of the `format` type. This is implemented as a symbolic expression or "sexpr" *a la* LISP [17]. Our modification causes the pretty-printer to instead produce an instance of `eformat`. `eformat` is the same as `format` except that certain points in the sexpr tree are tagged with two pieces of information: the subexpression that produced the substring and an *expression address* indicating where the subexpression lies in the parent expression. The expression address is a list of *expression coordinates* used to reference subterms of an expression. By *expression coordinate*, we mean an enum that indexes the recursive arguments in the constructors for an expression. This is visualised in Figure 4a and Figure 4b. Listing 2 gives a simplified picture of the datatypes used to define expression coordinates and `eformat`.

The `eformat` tags act as a reversed source-map between the resulting sexpr and the original expression. This tagging also works beneath specialised syntax such as lists `[1,2,3]` and set comprehensions. This tagged string is used to create ProofWidgets that allow users

Listing 2 Simplified datatypes to demonstrate expression coordinates and `eformat`. Here the simplified `expr` datatype has four constructors for creating variables, function application, lambda abstraction and constants. Next, define coordinates `coord` on `expr`. Each constructor of `coord` corresponds to a different recursive argument in a constructor for `expr`. An `eformat` is a string with structural information present.

```
inductive expr
| var   : string → expr
| app   : expr → expr → expr
| lam   : string → expr → expr
| const : string → expr

inductive coord
| f | a | lam_body

def address := list coord

inductive eformat
| tag : expr → address → eformat → eformat
| append : eformat → eformat → eformat
| of_string : string → eformat
```

to interactively inspect various parts of the expression in the infoview. In the case of a subexpression being below a binder (e.g., in the body of a lambda expression) the pretty printer instantiates the de-Bruijn [7] variable with a dummy local variable, so the given subexpression doesn't contain free de-Bruijn variables and still typechecks without having to know the binders above the subexpression.

To render an interactive expression, we define a stateful component:[11]

$$p : \texttt{component}\ (\texttt{tactic_state} \times \texttt{expr})\ \texttt{empty}$$

The `tactic_state` object includes some contextual information such as metavariable context that are needed to print the expression correctly. The state of p includes an optional `address` of the expression. When the user hovers over a particular point in the printed `eformat`, the expression address corresponding to that part of the string is calculated using the `tags` and this address is set as the state of the component. This address is then used to colour in the portion of the string that is currently hovered over by the user's mouse cursor which gives the semantic-aware highlighting effect.

4.2 Use within the Lean community

Our Lean implementation of the ProofWidgets framework has been adopted in to the leanprover-community fork of Lean 3 and mathlib, Lean's library of formalised mathematics [16]. The library is highly active and has hundreds of contributors using the widgets system[12] to render goal states.

[11] The sourcecode for this component within mathlib, the Lean mathematical library, can be found at `https://github.com/leanprover-community/mathlib/blob/master/src/tactic/interactive_expr.lean`.
[12] See `https://leanprover-community.github.io/mathlib_stats.html` for statistics on mathlib contributions.

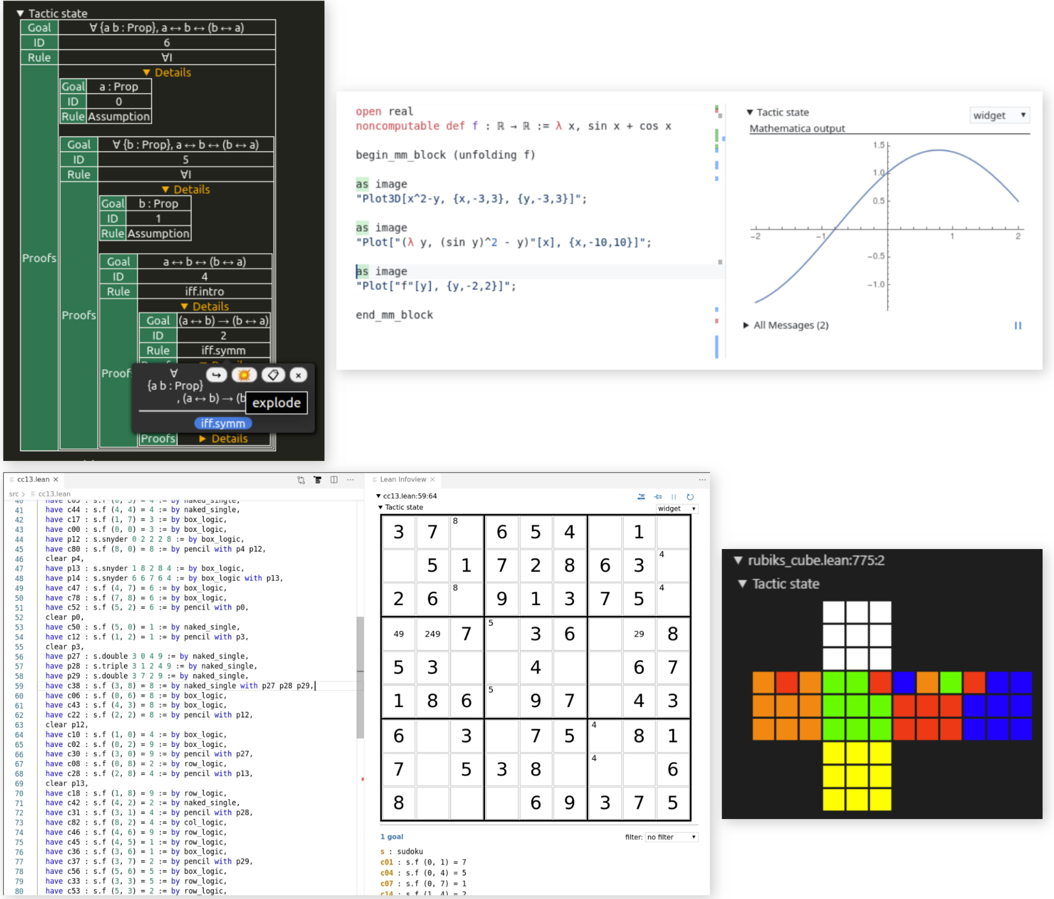


Figure 5 Community-made projects using ProofWidgets. In order, these are: `explode` proof viewer for inspecting proof terms by Minchao Wu; *Mathematica bridge* for viewing plots using Mathematica by Robert Y Lewis and Minchao Wu; *Sudoku solver and visualiser* by Markus Himmel; *Rubik's cube formalisation* by Kendall Frey.

The implementation in Lean has already been picked up by the community to make a wide variety of graphical interface solutions which can be viewed in Figure 5. Of particular note is the Mathematica bridge by Lewis and Wu [13], which connects Lean to Wolfram Mathematica and uses our framework to show Lean functions plotted by Mathematica.

ProofWidgets are also in use within mathlib to rapidly prototype new user interface features as they are requested. As a concrete example: in the development of the Lean VSCode extension, it was requested that it should be possible to filter some of the variables in the goal state to declutter the output window (see Figure 6). This was achieved by reparsing the textual goal state emitted by the Lean server component and removing the filtered items using regular expressions. Using ProofWidgets, it was required to add some specific code for the VSCode client – supporting such a feature in other editors would require rewriting the filtering code. Additionally, if the Lean server changes how the goal state is formatted, the filtering code would need to be rewritten. Even if an API which allows more semantic access to the expression structure is used, such as SerAPI [10], there is still the problem that the filtering code has to be written multiple times for each supported editor. Using ProofWidgets, the filtering code can be written once *in Lean itself* and it then works in

Figure 6 Example widget for filtering goal states, an example of being able to implement new UI features within Lean.

any editor that supports the widgets API (at the time of writing VSCode and a prototype version of the web editor). Furthermore, Lean users are free to make any custom tweaks to the UI without needing to make any changes to the editor code.

5 Related work

We now list some of the other tools and systems for creating user interfaces within the context of interactive theorem proving and how they relate to our work.

Isabelle's *Prover IDE* (PIDE) was developed by Wenzel in Scala and is based on the jEdit text editor [21]. It richly annotates Isabelle documents and proof states to provide inline documentation, interactive and customisable commands, and automatic insertion of text, among other features. Isabelle's development environment allows users to code their own UI in Scala, which performs a similar role to ProofWidgets. PIDE broadly shares the design goals of ProofWidgets, but approaches the problem differently.

An advantage of the ProofWidgets approach compared to PIDE's is that the API between the editor and the prover can be smaller since, in ProofWidgets, the appearance and behaviour is entirely dictated by the server. In contrast, the implementation of PIDE is tightly coupled to the bundled jEdit editor, which has some advantages over ProofWidgets in that it gives more control to the developer to create new UIs. The downside of PIDE's approach here is that one must maintain this editor and so supporting any other editor with feature-parity becomes difficult. ProofWidgets also makes use of modern web technology which is ubiquitously supported. In contrast, PIDE uses a Java UI library called Swing. Creating custom UIs in PIDE requires coding in both Scala and StandardML, and the result does not easily generalise to the VSCode Isabelle extension.

There have been some recent efforts to support VSCode as a client editor for Isabelle files [22]. A web-based client for Isabelle, called *Clide* [15] was developed, although ultimately it provides only a subset of the functionality of the jEdit version.

SerAPI [10] is a library for machine-machine interaction with the Coq theorem prover. This supports some web-based IDE projects such as *jsCoq* [11] and, recently, Alectyron [18]. Alectyron enables users to embed web-based representations of data. SerAPI contrasts to ProofWidgets in that it expects another program to be responsible for displaying graphical elements such as goal states and visualisations; in the ProofWidgets architecture all of the UI appearance and behaviour code is also written in Lean, and the web-app client can render general UIs emitted by the system.

There are some older GUI-centric theorem provers that should be mentioned: *LΩUI* [19], *HyperProof* [3] and *XBarnacle* [14]. These tools were all highly innovative for including graphical and multimodal representations of proofs, however the code for these has succumbed

to bit rot, to the extent that they can only be viewed through the screenshots that were included with the papers. Source code for Ωmega and CLAM (which $L\Omega UI$ and XBarnacle use respectively) can be found in the Theorem Prover Museum.[13]

Vicary's *Globular* [2] and Breitner's *Incredible Proof Machine* [5] also inspired our work. These tools are natively web-based and offer a visual representation of the proof state for users to manipulate. A lot of the motivation behind ProofWidgets was to bring some of this visual pixiedust to a more general, heavyweight proof assistants.

6 Conclusion and future work

We designed the ProofWidgets framework: a general client/server protocol, a functional UI programming API and a system for tagging pretty-printed expressions. The framework is implemented in Lean to allow for rapid development of new modalities of interaction with provers, all within the metalanguage of Lean. This enables a faster development cycle for improving the UI of Lean which has a direct impact on the users of ITP. We hope that these benefits can be brought to other interactive theorem provers in the future.

6.1 Future work

In terms of performance, in order to produce responsive interfaces that use long-running tactics (e.g., searching a library or running a solver) it will be necessary to provide a mechanism for allowing concurrency. At the moment, if a long-running operation is needed to produce output, this will block the rendering process and the UI will become unresponsive for the length of the operation. Currently, Lean has a `task` type which represents a 'promise' to the result of a long-running operation, which could be utilised to solve this problem. This could be cleanly incorporated in ProofWidgets by providing an additional hook `with_task` (see Listing 3):

Listing 3 Adding concurrency to components.

```
component.with_task
  (get_task : π → Task τ)
  : (Component ((Option τ × π) α) → (Component π α)
```

Here, `get_task` returns a long-running task object and the props for the inner component transition from none to some $t : \tau$ upon the completion of the task. Cancelling a task is implemented simply by causing a rerender.

The next implementation project is to support Lean 4. Lean 4 has a bootstrapped compiler, so the reconciling code can be written in Lean 4 itself without having to modify the core codebase as was necessary for Lean 3. Lean 4 has an overhauled, extensible parser system [20] which could be used to create an HTML-like domain-specific language directly within Lean.

[13] `https://theoremprover-museum.github.io/`

References

1 Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, 2013. doi:10.1145/2501654.2501666.

2 Krzysztof Bar, Aleks Kissinger, and Jamie Vicary. Globular: an online proof assistant for higher-dimensional rewriting. *Log. Methods Comput. Sci.*, 14(1), 2018. doi:10.23638/LMCS-14(1:8)2018.

3 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.

4 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.

5 Joachim Breitner. Visual theorem proving with the incredible proof machine. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 123–139. Springer, 2016. doi:10.1007/978-3-319-43144-4_8.

6 Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for guis. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 411–422. ACM, 2013. doi:10.1145/2491956.2462161.

7 Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. North-Holland, 1972. URL: http://alexandria.tue.nl/repository/freearticles/597619.pdf.

8 Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP):34:1–34:29, 2017. doi:10.1145/3110278.

9 Conal Elliott and Paul Hudak. Functional reactive animation. In Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman, editors, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, pages 263–273. ACM, 1997. doi:10.1145/258948.258973.

10 Emilio Jesús Gallego Arias. Serapi: Machine-friendly, data-centric serialization for coq. Technical report, MINES ParisTech, 2016. URL: https://core.ac.uk/download/pdf/51221893.pdf.

11 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*, volume 239 of *Electronic Proceedings in Theoretical Computer Science*, pages 15–27. Open Publishing Association, 2017. doi:10.4204/EPTCS.239.2.

12 Mike Gordon. From LCF to HOL: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 169–186. The MIT Press, 2000.

13 Robert Y. Lewis. An extensible ad hoc interface between lean and mathematica. In Catherine Dubois and Bruno Woltzenlogel Paleo, editors, *Proceedings of the Fifth Workshop on Proof eXchange for Theorem Proving, PxTP 2017, Brasília, Brazil, 23-24 September 2017*, volume 262 of *EPTCS*, pages 23–37, 2017. doi:10.4204/EPTCS.262.4.

14 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.

15 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`.

16 The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3372885.3373824`.

17 John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960. `doi:10.1145/367177.367199`.

18 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`.

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

20 Sebastian Ullrich and Leonardo de Moura. Beyond notations: Hygienic macro expansion for theorem proving languages. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2020. `doi:10.1007/978-3-030-51054-1_10`.

21 Makarius Wenzel. Isabelle/jedit - A prover IDE within the PIDE framework. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, volume 7362 of *Lecture Notes in Computer Science*, pages 468–471. Springer, 2012. `doi:10.1007/978-3-642-31374-5_38`.

22 Makarius Wenzel. Isabelle/pide after 10 years of development. In *UITP workshop: User Interfaces for Theorem Provers.*, 2018. URL: `https://sketis.net/wp-content/uploads/2018/08/isabellepide-uitp2018.pdf`.