ProofViz: An Interactive Visual Proof Explorer

Daniel Melcer[1(✉)] and Stephen Chang[2]

[1] Northeastern University, Boston, MA 02115, USA
melcer.d@northeastern.edu
[2] University of Massachusetts Boston, Boston, MA 02125, USA
stephen.chang@umb.edu

Abstract. We introduce ProofViz, an extension to the Cur proof assistant that enables interactive visualization and exploration of in-progress proofs. The tool displays a representation of the underlying proof tree, information about each node in the tree, and the partially-completed proof term at each node. Users can interact with the proof by executing tactics, changing the focus, or undoing previous actions. We anticipate that ProofViz will be useful both to students new to tactic-based theorem provers, and to advanced users developing new tactics.

Keywords: Proof assistants · IDEs · GUI tools

1 Introduction

The Curry-Howard correspondence [3,8] is a fundamental insight connecting logic and programming. Specifically, a proposition in a logic corresponds to a type in a programming language, and a proof of that proposition is a program inhabiting that type. In such a language, type checking corresponds to proof checking, and in this manner program properties may be directly verified in the language without resorting to external specification languages or tools. This influential insight has been applied to a wide variety of features such as polymorphism, concurrency, and resource consumption, and has inspired the creation of numerous languages and proof assistants such as Coq, Agda, Idris, LF, NuPRL, F*, HOL4, and Lean (Wadler [18] recently surveyed its history in detail). Collectively, these tools are pushing the boundaries of software development and have even been used to verify parts of some mainstream software [5,19].

These proof assistants still have a steep learning curve, however, and none of the options for beginners are ideal. Some introductory books, such as *The Little Typer* [7], teach theorem proving via straightforward construction of the aforementioned "proofs as programs". While this method is direct and does not hide anything from the learner, it also does not scale well beyond small examples. Other popular texts [14] rely entirely on a separate "tactic" language to generate the proofs, despite the fact that such scripts are often "inscrutable" [15] because they hide much of the proof information from users.

V. Zsók and J. Hughes (Eds.): TFP 2021, LNCS 12834, pp. 116–135, 2021.
https://doi.org/10.1007/978-3-030-83978-9_6

We present ProofViz, a new kind of graphical IDE for the Cur [1] proof assistant, that bridges the gap between manual proof construction and tactic-based proofs. We believe this tool will be especially beneficial for new users of tactic-based proof assistants because they can more easily see, and thus understand, the parts of an in-progress proof such as the partial proof term, remaining subgoals, and the location of those subgoals in the proof tree. They can also directly perform actions on the proof such as navigating to proof tree nodes, executing or undoing additional tactics, and saving the actions in order to switch back and forth between our tool and a traditional editor. Our tool can be valuable for advanced users as well, e.g., for tasks such as creating and debugging new tactics, where the ability to see the underlying proof term is crucial. Finally, we conjecture that having an extensible GUI will enable many more helpful actions that are not possible in text-based IDEs, such as widescale refactoring of the proof structure, displaying domain-specific proof information, advanced searching of large proofs, and showing available tactics or hints for possible next steps.

The rest of the paper explains the details, and is organized as follows:

- Section 2 introduces relevant background about the Curry-Howard correspondence and tactic scripts;
- Section 3 presents a larger case study that illustrates how ProofViz smooths the transition to tactic-based proof assistants;
- Section 4 shows that ProofViz can be useful to advanced users as well; specifically it shows, via two case studies, how ProofViz can aid the development of new tactics and the maintenance of existing tactics;
- Section 5 discusses Cur, its tactic system, and ProofViz in more technical detail;
- Section 6 compares the tool to related work; and finally,
- Section 7 evaluates ProofViz, discusses future work and concludes.

2 Background: Tactics vs Proof Terms

According to the Curry-Howard correspondence, a logical proposition corresponds to a type `P`, and the proposition can be proved by constructing a program `p` with type `P`. For example, implication corresponds to the function type, universal quantification corresponds to polymorphism, and logical conjunction corresponds to a product type. Thus, a function with type `(∀ (P Q) (→ (And P Q) (And Q P))` in Fig. 1 (top) proves the commutativity of conjunction. Specifically, if `P` and `Q` are any two types, i.e., propositions, then a proof of their conjunction is a pair data structure that combines a value of type `P` (i.e. a proof of `P`) with a value of type `Q`. If we have such a pair, then to "prove" the commuted proposition `(And Q P)`, we simply need to extract the first and second components of the original proof and combine them in the reverse order.

Such manual proof term construction, however, becomes infeasible as proofs get larger. Thus, many proof assistant programmers use a separate "tactic" language that generates the proof. Figure 1 (bot) shows the same proposition along with a tactic script that proves it, where each line of the script generates one

```
; the following program proves the proposition,
; i.e., has type: (∀ (P Q) (→ (And P Q) (And Q P)))
(λ [P : Type] [Q : Type] [pq : (And P Q)]
    (pair (second pq) (first pq)))


; Alternate proof of the same theorem using a tactic script
(define-theorem (∀ (P Q) (→ (And P Q) (And Q P)))
  (intros P Q pq)
  (destruct pq #:as [p q])
  constructor
  (by-apply q)
  (by-apply p))
```

Fig. 1. (top) A program that proves the commutativity of conjunction; (bot) the same program, generated with a tactic script

piece of the proof term in Fig. 1 (top). Specifically, `(intros P Q pq)` generates the lambda parameters, `(destruct pq #:as [p q])` extracts the components of the pair and names them, `constructor` creates a new pair, and `(by-apply q)` and `(by-apply p)` puts the components into the new pair in reverse order.

Such a tactic script is typically developed in an interactive editor that can execute the script step-by-step and show a snapshot of the in-progress proof at each step. Figure 2 shows a few such snapshots for Fig. 1's proof script. Each snapshot has two parts: the *context* above the dotted line shows known assumptions, and the *goal* below the line shows a part of the proposition that is left to prove. The left snapshot, which shows the state immediately after running the `destruct` tactic, has a context above the line containing propositions `P` and `Q`, as well as proofs of those propositions. Below the line, the snapshot shows that we still must prove the consequent of the implication, i.e., the commuted conjunction. The right snapshot, which shows the proof state immediately after running the `constructor` tactic step, shows (below the line) that we have two subgoals left to prove, the first of which is `q` (the second, not shown yet, is `p`), which can be proven easily by applying the facts that we know above the line.

But here we begin to see a problem, which is that the tactic script by itself does not make much sense to a user who later looks at it, because it hides what is happening: the generation of the proof term. Thus, students who rely too much on tactics might not fundamentally understand how theorem provers work. Instead, they might come away thinking that theorem proving is merely the application of ad-hoc "pattern matching" rules (e.g., a popular textbook [14] often gives advice like "where the goal to be proved is exactly the same as some hypothesis in the context or some previously proved lemma ... use the `apply` tactic"). Such superficial techniques could hinder learning since they may not scale to larger proofs where the patterns are not as obvious.

```
; Proof state after destruct step
P : Type
Q : Type
p : P
q : Q
--------
(And Q P)
```

```
; Proof state after constructor step

P : Type
Q : Type
p : P
q : Q
--------
q          (subgoal 1 of 2)
```

Fig. 2. Intermediate views of the proof state while stepping through the tactic script in Fig. 1: (left) the proof state after running the `destruct` tactic; (right) the proof state after running the `constructor` tactic

Ideally, a novice could use an IDE that more naturally bridges the gap between manual proof construction and tactic scripts. During an undergraduate independent study, the first author was motivated to create and use such a tool, in order to better understand the connection between logic and programs that underpins the majority of modern proof assistants.

3 A Case Study: `add1+=+add1`

This section presents a more complex example that illustrates how ProofViz smooths the learning curve for beginning proof assistant users. Specifically, we show how to prove a basic arithmetic theorem:

```
(∀ (n j) (== (add1 (+ n j)) (+ n (add1 j))))  ; add1+=+add1
```

3.1 Inductive Proofs, Eliminators, and Equality

This seemingly basic theorem requires that students first learn many additional features of the language. First, it uses `Nat`, an inductively defined family. Inductive families [4], as found in languages like Coq, mostly resemble the algebraic datatypes found in functional languages like Haskell or ML. For example, here is the definition of `Nat` from Cur's standard library, which generates the usual data and type constructors (following *The Little Typer*, we use the more descriptive "add1" name for the successor constructor in this example):

```
(define-datatype Nat : Type
  [z : Nat]
  [add1 : (→ Nat Nat)])
```

Inductive families go beyond plain functional datatypes, however, because they allow parameterization over both types and terms, e.g., the type of an *indexed* list parameterizes over both the type of the list element *and* includes an additional `Nat` value that represents the length of the list.

Every inductive family definition also generates an *eliminator* for that type, which generalizes the pattern matching found in functional languages. Following the terminology of Mcbride [11], the eliminator for natural numbers has the form `(elim-Nat n P mz ms)` where `n` is the *target* to eliminate, `P` is the *motive* that computes the return type of the elimination, and the remaining arguments are *methods* corresponding to each case of the data type: the eliminator returns `mz` when `n` is zero and calls `ms` when `n` is a successor. Method `mz` must have type `(P z)`, i.e., the motive applied to zero, while `ms` must have type `(∀ [k : Nat] (→ (P k) (P (add1 k))))`, which mirrors a proof by induction: for any `k`, given a proof of `(P k)`, i.e., the induction hypothesis that results from recursively calling the eliminator with `k`, we must output a term with type `(P (add1 k))`. For example, here is addition, implemented with `elim-Nat`:

```
(define +
  (λ [n : Nat] [m : Nat]
    (elim-Nat
       n  ; target to eliminate
       (λ (n) Nat)  ; motive
       m  ; method for zero case
       (λ [n-1 : Nat] [ih : Nat] (add1 ih)))))  ; method for successor
```

The first addend `n` is the target of elimination and, according to the motive, the result is always a `Nat`. Specifically, when `n` is zero, the result is `m`; otherwise, the result is one plus the result of the recursive call `(+ n-1 m)`.

Finally, since propositions are types, inductive families can be used to define new propositions, where the data constructors are proofs of that proposition. One such proposition is the equality type `==`, which comes with a constructor `(same x)` (sometimes called `refl` or `reflexivity`) that represents a proof of `(== x x)`. In other words, we can only construct a proof of equality between two things that are equal.

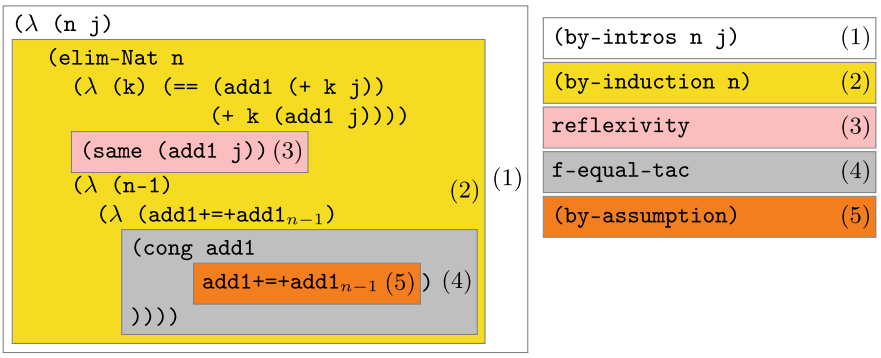
3.2 Matching Tactics with Proof Terms

Figure 3a shows a program proving our `add1+=+add1` theorem. It calls `elim-Nat` with four arguments: a target `n`, a motive function, and two methods corresponding to the zero and successor cases. When `n` is zero, the result is `(same (add1 j))`, which has the equality type we want, i.e., the motive applied to zero:

```
(∀ (j) (== (add1 j) (add1 j)))  ; zero case
```

When `n` is not zero, the result of the `elim-Nat` is the result of applying the second method to two arguments: `n - 1`, and the result of recursively calling the eliminator with `n - 1`, where the latter exactly corresponds to the inductive hypothesis in a proof by induction. Using this, we must construct a proof of:

```
; successor case
(∀ (j) (== (add1 (+ (add1 n-1) j)) (+ (add1 n-1) (add1 j))))
```



(a) Manual proof of `add1+=+add1` [7]

(b) `ntac` proof of `add1+=+add1`

Fig. 3. The correspondence between a manually constructed proof term and an equivalent tactic script. While these two proofs are written in very different styles, every part of the manual term corresponds to a tactic. ProofViz allows users to view which tactic generated each part of a proof term.

which, when simplified, following the definition of the + function above, becomes:

```
; successor case, simplified
(∀ (j) (== (add1 (add1 (+ n-1 j))) (add1 (+ n-1 (add1 j)))))
```

We can see that this proposition is exactly the inductive hypothesis, with an extra `add1` around it. To go from the inductive hypothesis to what we need, we can use `cong`, which is a theorem about a basic property of functions: `(∀(A B) [x : A] [y : A] [f : (→ A B)] (→ (== x y) (== (f x) (f y))))`, i.e., applying the same function to equal values produces equal results.

Figure 3a shows the result of manually constructing a proof term following step-by-step exercises from *The Little Typer*. The same theorem can also be proved via a Cur tactic script, as shown in Fig. 3b. The execution of this sequence constructs a proof term that is remarkably similar to the manually constructed version. A student, however, cannot see this correspondence, nor can they see any of the intermediate proof parts mentioned in this subsection.

3.3 Using ProofViz to Understand Induction Tactics

The two sides of Fig. 3 and their colored components summarize the correspondence between tactics and proof terms that we would like to see. Figure 4 presents ProofViz, which shows this exact correspondence. Briefly, our tool's user interface contains three panes; the tree view, the node information panel, and the interaction panel.

– In the tree view (Fig. 5b), the proof's state is displayed as a tree. A proof tree has a single node that is marked as its "focus", which represents the

current proof subgoal. Subsequently executed tactics will add nodes at this focus point, and the tool includes controls to collapse all nodes of the tree that are unrelated to the focus. We are working to further optimize ProofViz's interface to allow concise viewing of other information subsets. The various node types and colors in the tree view are discussed in Sect. 5.2.

– A primary contribution of ProofViz is that each tactic in the proof script is connected to the part of the proof term that this tactic generates. This information is shown in the node information panel (Fig. 5c) when a single node in the tree view is selected. In addition to the list of variables in the context and the types of these variables, this panel also shows the expected output type of the node. Some node types have node-specific information or actions available. For example, "hole" nodes allow for the proof focus to be set to that node (see Sect. 5.2) and "apply" nodes include a list of expected types for its subtrees, as well as the output of the combined result.
– The rightmost interactions panel (also in Fig. 5c) allows the user to execute additional tactics, and the tool allows undoing and redoing an arbitrary number of these interactions. If an error occurs during tactic evaluation, details are printed to the console and the proof tree is not modified.

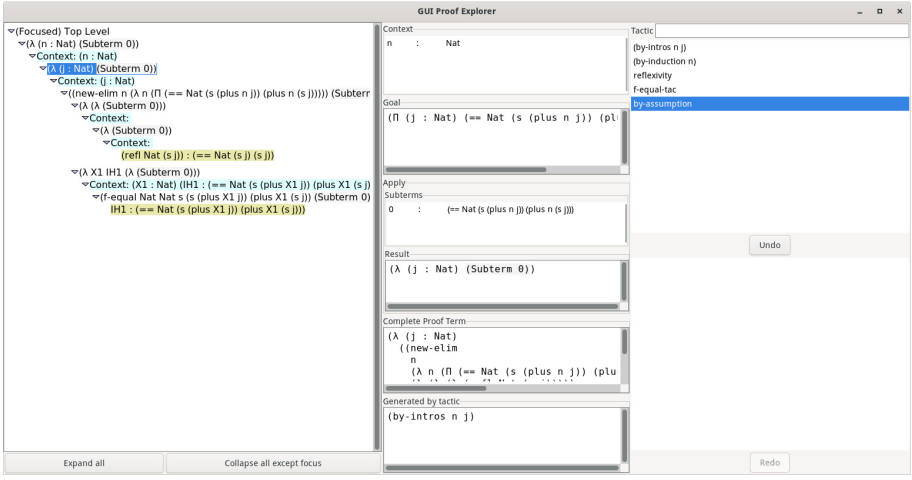
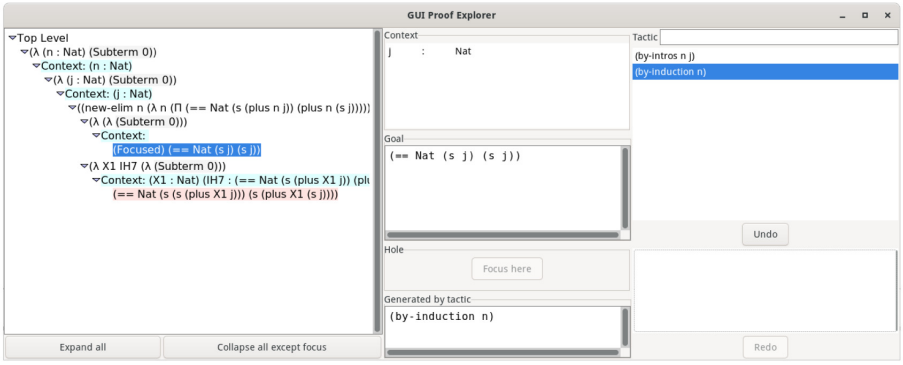
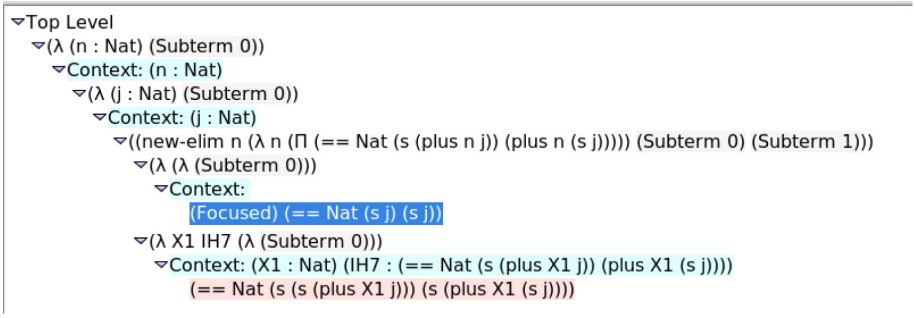


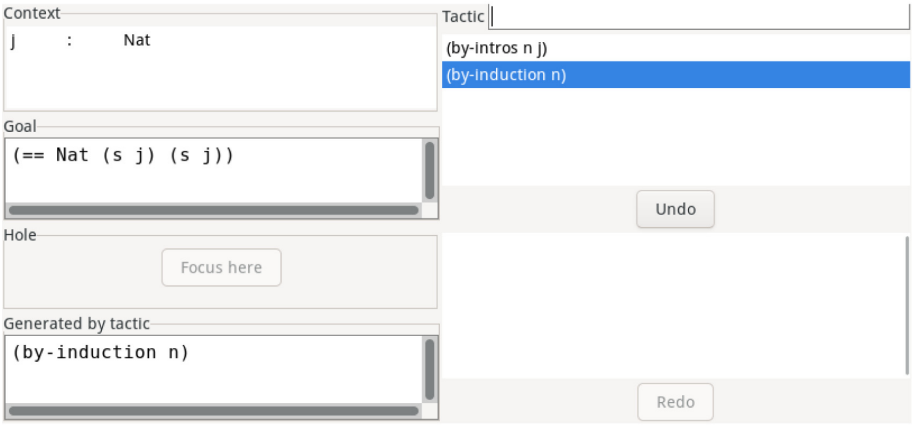
Fig. 4. A view of the interface with a completed proof of `add1+=+add1`. Note: instead of `elim-Nat`, Cur uses a general `new-elim` for inductive datatypes. The `cong` and `same` equality constructors are also have different names, `f-equal` and `refl`, respectively. ProofViz also shows which tactic generated the selected node, in the middle panel; the user can explore different nodes of the proof without changing the focus.



(a) ProofViz, immediately after running `(by-induction n)`. Information about the currently selected node is visible in the middle panel.



(b) The tree view after running `(by-induction n)`. There are two holes generated by this tactic; one for the base case, and one for the inductive case. The inductive case has an inductive hypothesis, `IH7`, available in its context.



(c) The node information and interactions panel, after running `(by-induction n)`. The selected node shows that it was generated by the induction tactic.

Fig. 5. ProofViz displays all nodes of the proof tree, making it clear to the user how each tactic affects the generated proof term.

Overall, our tool enables users, especially students, to gain more insight into their proofs. For example, with only a conventional view of a proof, the `by-intros` tactic appears to just "move" variables from the goal into the context. With the highlighted correspondences shown in Fig. 3, it becomes clear that `by-intros` really "wraps" the rest of the proof into a lambda. Instead of our goal being a function type (i.e., an implication), we've now assumed a proof of the input type (i.e., the antecedent) and now need only to generate a term with the function's return type (i.e., the consequent). The parameters of the lambda are thus in the context as assumptions when generating the body of the function. Similarly, `by-assumption` merely corresponds to finding an assumption in the context whose type "fits" correctly, and then using its name directly. By exploring the tree view and node information panel in ProofViz, a student can directly see these correspondences and build up their intuition.

Also, our `add1+=+add1` example in Fig. 3 involves inductively defined natural numbers, and thus its proof requires induction. Conventionally, a student might be told to just "use the `by-induction` tactic" as a way to deal with such proofs, but they would not necessarily gain insight into what is actually happening, or why this tactic works. With ProofViz, as seen in Fig. 5a, a student can see what `by-induction` actually does—it creates a `new-elim` node (in Cur `new-elim` is a general eliminator that dispatches to the type-specific ones like `elim-Nat`), and sets up the next subgoals which must be proved. Once those subgoals have proofs, ProofViz shows how they will be assembled into the completed proof term, as highlighted in Fig. 5b.

Lastly, this example gives insight into why tactics are indeed useful, because they can help manage the amount of boilerplate that must be written. For example, with a manually constructed term, the induction motive, a necessary but somewhat formulaic part of an inductive proof, must be written by hand. In contrast, the `by-induction` tactic uses the goal type to automatically generate this part of the term. This type of demonstration conveys the effectiveness of tactics for reducing repetitive code, and may reduce a student's skepticism about whether tactics are even useful.

As seen in this example, we believe that ProofViz can mitigate the steep learning curve associated with proof assistants by showing how each tactic affects the generated proof tree. We envision a student could begin by merely interacting with and exploring a pre-written library of such proofs, while a more advanced student could use ProofViz while writing their own proofs to ensure that the generated proof term matches their intuition of what each tactic does.

4 Tactic Development with ProofViz

Though we have shown how ProofViz can be useful for beginners, it is not limited to only such applications. In this section, we show how ProofViz can help advanced users as well, specifically to develop and debug tactics themselves, where it is often critical to be able to see how the tactics manipulate and generate the underlying proof term.

4.1 Tactic Development: `f-equal`

While ProofViz does not replace traditional testing of new tactics, it can assist with debugging in the course of tactic development. Similar to beginners, tactic developers may find it useful to see exactly how each tactic affects the proof tree. More specifically, in Fig. 3b, we used `f-equal-tac` in our proof script to generate an application of *The Little Typer*'s `cong` theorem but, until recently, the tactic did not exist! We had to add the tactic ourselves and fortunately, we had ProofViz available to help us do this more easily.

Figure 6 summarizes our iterative development process. As a first step, instead of an `f-equal-tac` tactic, we started with an equivalent, but much more complicated, call to a `by-apply` tactic that applies an `f-equal` (Cur's name for the `cong` theorem) function; this is shown in Fig. 6a. Then, we created a new tactic that simply does the same thing as the aforementioned `by-apply`, as seen in Fig. 6b, but this was very verbose and cumbersome to use. We then iteratively improved the tactic so that it could infer all the arguments from the expected goal type, eventually obtaining the simple tactic invocation shown in Fig. 6c. While the implementation details of the tactic itself are not important for this paper, what is important is that after each incremental change, we used ProofViz to verify that the resulting tree structure, subterm types, and generated syntax were what we expected, as shown in Fig. 6d.

4.2 Tactic Maintenance: `by-induction`

When developing ProofViz, we also noticed that the `by-induction` tactic was behaving strangely, but only when used with certain other tactics. Using ProofViz and the information it provides, we were able to quickly discover that the tactic was producing subgoals with incorrect types, as seen in Fig. 7a. Specifically, "Subterm 0" in the figure corresponds to the zero case in our `add1+=+add1` proof from Sect. 3 and thus should have type `(== (s j) (s j))` (Cur uses the name `s` instead of `add1` for the successor `Nat` constructor). Similarly, "Subterm 1" should have a type corresponding to the inductive step in the proof. After deploying a fix for this, ProofViz then allowed us to quickly validate that the revised tactic produces correct goal types, as seen in Fig. 7b. Without being able to see the underlying proof information with ProofViz, debugging and fixing this tactic would have been much more difficult.

5 Implementation Details

ProofViz works with Cur, a new proof assistant [1] that operates in the Racket ecosystem [6], with an emphasis on easy extensibility [2]. This capability has been used to extend Cur with features such as experimental type systems, e.g., sized types, and SMT solver integration. Further, these additional components are modular, meaning that they may be added without changes to any existing languages and do not break existing code, yet they are not isolated like third

```
(by-apply f-equal
  #:with Nat Nat s
   (s (plus X1 j))
   (plus X1 (s j)))
```

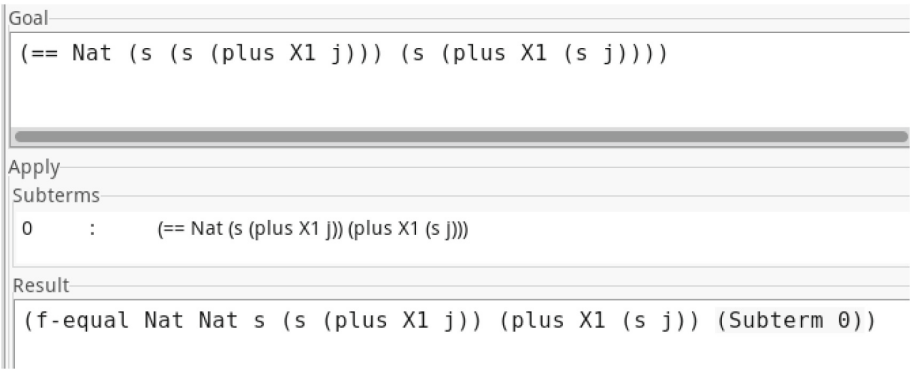
(a) An explicit invocation of `f-equal`.

```
(f-equal-tac
  Nat Nat s
   (s (plus X1 j))
   (plus X1 (s j)))
```

(b) The first version of `f-equal-tac`.

```
f-equal-tac
```

(c) The final invocation of `f-equal-tac`.



(d) The goal, subterms, and generated syntax for the generated nodes from all three versions are nearly identical.

Fig. 6. The progression of `f-equal` forms. Initially, the function needed to be called manually with `by-apply`. The first version of the tactic kept all arguments explicit, but later versions of the tactic inferred all of the arguments from the goal. ProofViz was used to check that each successive version had the correct behavior. Note that the name `f-equal-tac` is used for the tactic to distinguish it from the function.

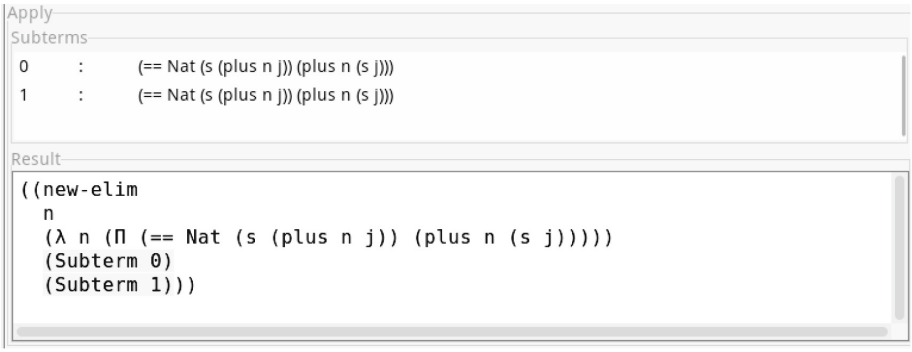
party tools in other systems. This is because the underlying mechanism—Racket macros—enables easy communication with other components in the ecosystem.

ProofViz is implemented as a similar extension, and thus its implementation did not require any changes to the core language. It required only minimal enhancement to `ntac`, the main tactic system used by Cur programmers, to allow tagging proof nodes with arbitrary data (discussed further in Sect. 5.2).

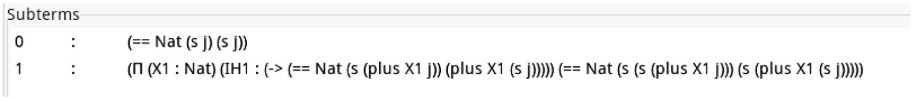
5.1 Using ProofViz

Figure 8a shows a basic proof script. A `#lang cur` on the first line declares the start of a Cur program. The next `require` line imports the `cur/ntac` library, which contains implementations of many basic tactics commonly used in other proof assistants. The rest of the program binds `id` to the term produced by the subsequent `ntac` proof script, which proves the identity function type.

To invoke our GUI tool, shown in Fig. 8b, a programmer can simply import another library, `cur/ntac-visual`, and then invoke the `ntac/visual` proof environment. This environment is implemented as an ordinary Racket macro. When run, the program will launch ProofViz, initially displaying the partial proof



(a) Incorrect intermediate goal types generated by `by-induction`. The term substituted into `(Subterm 0)` must have type `(== Nat (s j) (s j))`; this is the actual goal type of a "hole" node farther down in the subtree.



(b) The corrected internal goal types for `by-induction`. The "Result" sub-window is unchanged by the fix.

Fig. 7. While developing ProofViz, we found that some tactics produced incorrect goal types at internal boundaries. We were able to use the tool to easily validate the fixes.

generated by the listed tactics (Fig. 4 shows a screenshot). Note that our tool is launched by running the program itself. It is independent of any specific IDE; the proof script itself could have been edited with any editor.

5.2 Implementation

Internally, most tactic systems represent an in-progress proof as a tree. Each tactic then transforms this tree, gradually filling in more information until the proof is complete. There are several varieties of tree nodes in Cur.

A "hole" node represents a node on the tree that must be filled by a value of a specific type. The ProofViz tree view displays this expected type, and highlights the node in red. In an interactive tactic-based theorem prover, hole nodes typically correspond to subgoals. For a proof to be considered complete, the proof tree must not have any hole nodes.

Tactics may also generate "apply" nodes, which combine the values from multiple subtrees into one value of an expected type. For example, an induction tactic applied to the natural numbers will generate an "apply" node with two subtrees. Initially, both of these subtrees will be hole nodes; one with a goal type to prove the theorem for the base case, and one with a goal type for the inductive case. The final piece of an "apply" node is a metafunction to combine the subterms into a larger term that proves the theorem in general. For induction,

```
#lang cur
(require cur/ntac)

(define id
  (ntac
    (∀ (A : Type) (a : A) A)
    (intros A a)
    assumption))
```

(a) A basic tactic script in Cur

```
#lang cur
(require cur/ntac
         cur/ntac-visual)
(define id
  (ntac/visual
    (∀ (A : Type) (a : A) A)
    (intros A a)
    assumption))
```

(b) Running our GUI proof explorer

Fig. 8. The addition of ProofViz to an existing proof script. The user must import an extra library, and change the invocation of `ntac` to `ntac/visual`, a macro provided by ProofViz.

this combining function takes as input a proof term that proves the base case and a term that proves the inductive case. Its output is a term that eliminates the inductive datatype value, with the two input subterms placed in their necessary positions. The tree view displays the output of the combining function, even when the apply's subterms contain holes. The implementation of this is further discussed in the **Apply Outputs** subsection below. In the tree view, the places where subterms are substituted into the output of an apply node are highlighted in light gray. An "apply" node can also bind variables that may be referenced in any of its subtrees, by generating a lambda in the output. With induction, for example, inductive cases will include extra variables in their context with the induction hypothesis.

However, an apply node can only provide instructions to the tactic system to assemble pieces of concrete syntax produced by subtrees; these nodes do not provide bookkeeping information about any new names available in the context. "Context" nodes serve this purpose, informing `ntac` that a name is available in a given subtree. These are typically generated as direct descendants of "apply" nodes. The tree view panel shows the names and types of all variables that such context nodes introduce, and highlights such nodes in blue.

An "apply" may also be a leaf node, in which case the combining function takes no arguments and produces a complete term whose type matches the goal. Equivalently, an "exact" node contains a syntax literal to appear in the generated proof term. Exact nodes are highlighted in green.

In summary, apply and exact nodes generate syntax that will become part of the final proof term, while hole and context nodes solely perform bookkeeping functions. There is also a fifth node type for bookkeeping, `ntt-done`, that only appears at the top level of the proof tree. The tree view displays a short summary of each of these nodes' content, allowing the user to see the proof's internal structure at a glance.

Apply Outputs. An "apply" node works by declaring the expected types of a number of subterms. When concrete terms of the correct type are available, the apply node contains a combining function that accepts all of these subterms and outputs a new term; this term is of the apply node's output type. In order to compactly visualize a proof, ProofViz must be able to show the local transformations of each apply node in isolation, without needing to provide a term of the correct type. ProofViz must also show apply nodes in a partially completed proof, where subterms with the expected type may be unavailable.

It is possible to do this because the combining function of an apply node treats each subterm as an opaque value. Thus, to display a string representation of an apply node, the tool creates several placeholder terms that typecheck as the expected type, but show only as `(Subterm n)`, where `n` is the index of this term in the apply node. These placeholders are used as the input to the combining function when generating the text representation of the apply node. Finally, the output of the combining function is converted into a string and displayed.

Navigation. To allow for the "Focus Here" functionality, the tool generates a sequence of navigation instructions from the top of the proof tree. These instructions are read by a new tactic, `navigate`, shown in Fig. 9. This tactic starts by setting the focus at the root of the proof tree and then reads the sequence of instructions, which has three possible cases:

- The proof tree's root will always be a marker node, called `ntt-done` (represented as top-level in the tree view), with a single subtree. Then the `path-down-done` instruction moves the focus to this subtree. This instruction should only appear in the beginning of the navigate sequence.
- The `path-down-context` instruction likewise moves the focus to the subtree of the context node.
- The `path-down-apply` instruction is parameterized with a numeric index. Since an apply node can have multiple subtrees, the index is used to determine which subtree to focus on.

These three navigation instructions are sufficient to jump directly to any location in the proof tree.

```
(navigate (path-down-done) (path-down-apply 0) (path-down-context)
          (path-down-apply 0) (path-down-context) (path-down-apply 1)
          (path-down-apply 0) (path-down-context))
```

Fig. 9. An example navigation tactic generated by ProofViz. The tactic jumps to the root of the proof tree, then descends to a specific node given by the instructions in the navigation tactic.

Scoping. Since Cur's AST values include binding information that is computed from its context in a program, our tool must be slightly careful about scoping. For example, when `display-focus-tree` is used, since `ntac` has already executed all previous tactics in the proof script outside the context of our tool, if any of these tactics introduced variables into the context, they may not be referenced by tactics executed with the tool.

To work around this, if `ntac/visual` is used, ProofViz executes the proof script as if they were entered in the tool's interactions panel, so all bindings have the proper context. This associates the identifiers with a modified source location, accounting for the reduced source location information that is available when executing the tactic input box's contents.

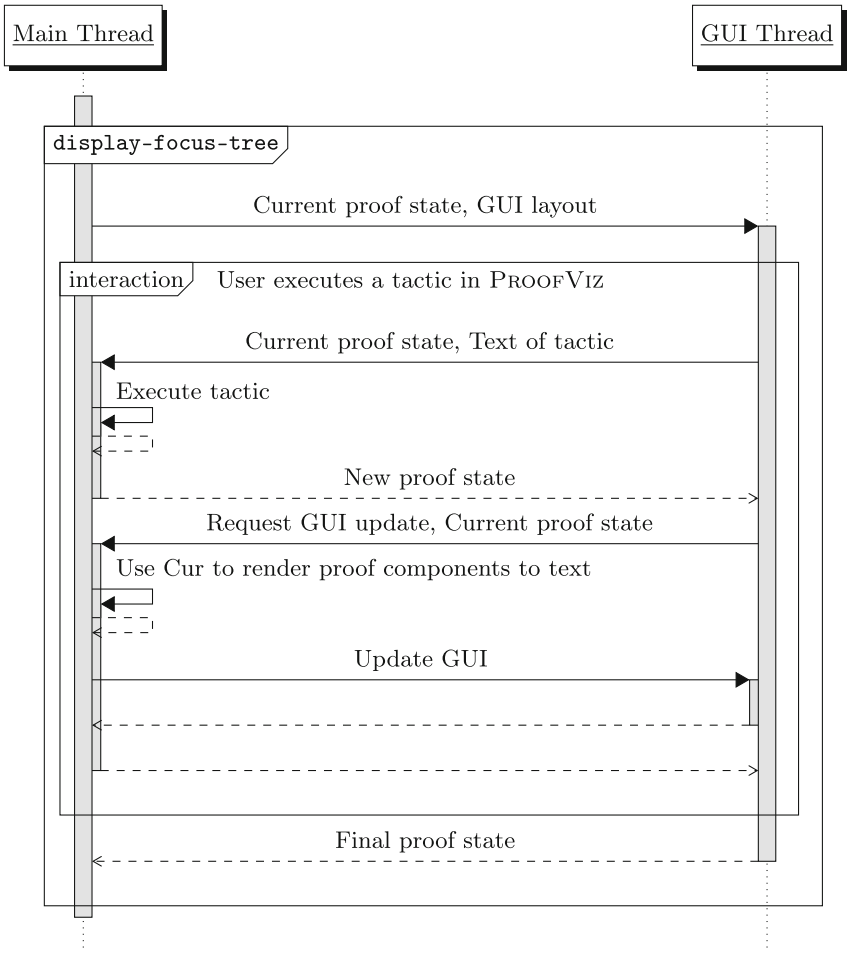


Fig. 10. The threading model for ProofViz, in `display-focus-tree` mode, resulting from constraints on where GUI code and Cur tactics are each allowed to execute.

Threading. Additionally, to further ensure the proper context, ProofViz must be careful when executing tactics in a multi-threaded environment. Specifically, the GUI must run in a second thread because it may start before the main (proof script) program finishes executing. But tactics, to have the same context as the rest of the proof script, must be executed in the main thread. Additionally, during GUI updates, ProofViz must call certain Cur library functions to obtain textual representations of internal tree structures. Due to implementation restrictions, these functions must be run on the main thread as well.

Thus, the tool uses a bidirectional channel to communicate between the main thread and GUI thread. While the GUI is open, the main thread waits on a message from this channel. One such message notifies the main thread that the GUI window has closed, and includes the current state of the proof tree. If `display-focus-tree` was used, the tactic that follows in the proof script receives this proof state as input. The channel also allows the GUI to send arbitrary code to execute on the main thread; this channel ensures that subsequent tactics are evaluated in the correct context. Figure 10 shows the complete threading behavior of ProofViz.

Tree Node Origin Tracking. Most of the features of ProofViz required no changes to any other components of Cur or `ntac`, but the node origin tracking feature required the addition of substructures for each node type. These substructures each add a generic "tag" field, and are interchangeable with the original structures. ProofViz then uses this extra field to associate each node of the proof tree with the node that generated it. However, when `ntac` changes the focus of the tree proof, an implementation detail in `ntac` causes tree nodes to be occasionally deconstructed and reconstructed in the process. The function that does this was modified to detect whether it had destructed a tagged node, and to add the tag back to the new node if it did so. We emphasize that no modifications to Cur's trusted core were necessary to achieve this.

5.3 Unresolved Challenges

Automatically Generated Names. Several tactics automatically generate names for internal variables, often with no way to provide manually-written identifiers. This is typically not an issue, as these names generally would not be exposed to the user. However, when ProofViz encounters one of these identifiers, it can only display the names that have been given to it. To fix this, individual tactics would need to be rewritten to produce meaningful intermediate names. A notable example of this are the names generated by the `by-inversion` tactic, as shown in Fig. 11. In general, we are working to refactor some tactics so they may be more ideally presented in ProofViz.

Ergonomics and Complexity. Relatedly, large proof trees and proof terms can lead to a high information density, or to a GUI which requires excessive scrolling. To combat this, the tree view includes the functionality to collapse

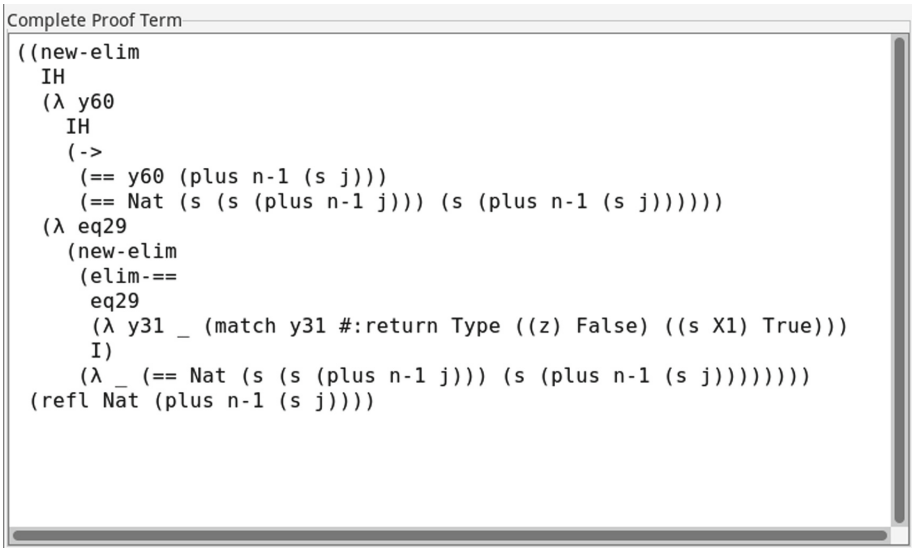


Fig. 11. A sample proof term generated by `by-inversion`. Here, it is used instead of `by-apply` or `f-equal` in the `add1+=+add1` case study. ProofViz may not be as useful when exploring tactics in which the generated term itself is difficult to interpret.

subtrees which are unrelated to the current focus, and the information about an "apply" node only shows how its direct subterms are used locally. Further work is needed to completely solve the problem, especially as proofs get larger. Fortunately, our presented use cases—assisting beginning students, and creating or debugging new tactics—typically involve smaller, more manageable proofs.

6 Related Work

Visualizing proofs as trees is not new. Even textual proof assistant IDEs, e.g., Proof General, typically support some tree-structured organization of proofs, via the "bullet" system. Proof General also includes some tool-support for graphical visualization [17], which itself is based on a visualization tool in PVS [13]. These proof script visualization tools, however, seem to be exactly that: a visualization of the tactic script that is currently entered in the buffer. While this visualization can help users see the logical organization of a tactic-based proof, this style of visualization doesn't help the user relate the tactic script to the generated proof term, and may be more useful for users who only deal with tactic-based scripts, rather than the users coming from a no-tactics theorem prover. We believe, as illustrated by the previous sections, that an explicit correspondence between the proof term and tactic script is useful for some audiences.

Proof visualizers also exist for non-dependently-typed theorem provers, such as the $\mathcal{LOUI}$ [16] tool. The Sparkle [12] theorem prover also provides an IDE-like editing environment for in-progress proofs of a non-dependent functional language. It supports proving properties of many functional features such as laziness, and it interactively shows the context and goals at each point of the proof. It is not based on the Curry-Howard correspondence, however, and thus the proofs generated by its tactics are quite different from the proofs one would construct in a dependently-typed theorem prover based on Curry-Howard.

Several tools visualize the proof tree as a sequent calculus "stack" [9,10], but these tools usually focus solely on the context and goals of each node in the proof tree, rather than the proof term that is generated, and thus don't provide the same intuition to users transitioning from a manual-construction style.

The `Show Proof` command in Coq prints out the partial proof term in the middle of a proof script, but this command lacks the interactivity found in ProofViz. For example, `Show Proof` does not allow the user to view which parts of the proof term are generated by which tactic, and it doesn't display the context and goals of each node in the tree. Furthermore, a call to `Show Proof` must be manually inserted (and removed) at each location where the user is interested in seeing the partial term, while ProofViz enables the user to view and step through all intermediate proof states.

Alectryon [15] aims to allow proof script authors to annotate their proofs and create interactive documentation, enabling readers of this documentation to easily step through the proof state at the current focus. This tool doesn't directly address the generated proof term, which we believe is important for users transitioning to tactic-based proof assistants. However, the tool's motivation highlights many of the same pitfalls of tactic-based programming that can be difficult for such users, thus demonstrating the need for these kinds of tools.

Ultimately, other visualization tools do not aim to address the same issues as ProofViz. Further, they are often tightly coupled with the IDE itself, i.e., the tool must be maintained in sync with the IDE. In contrast, ProofViz is a modular component in the `ntac` tactic system and is independent of how programmers edit their programs. Creating our tool required minimal changes to existing code, and no changes to unrelated tactics. As such, not only is ProofViz itself extensible and well-positioned for future enhancements, it will seamlessly accommodate new tactics, and potentially even changes in the core Cur language.

7 Evaluation, Future Work and Conclusion

The full implementation of ProofViz required approximately 1,000 lines of Racket code. With ProofViz installed, the existing Cur test library of approximately 11,000 lines of code continues to pass. In the course of developing ProofViz, the first author successfully used the tool for dozens of hours, and stepped through thousands of proof script lines. It has greatly enhanced their understanding of dependently typed languages, tactics, and the implementation of theorem provers.

ProofViz continues to be a work in progress. One potential enhancement could be to display available tactics to the user; a related improvement would be the autocompletion of variable bindings or types. A more involved but useful addition could be to illustrate the effect of a given tactic on the proof tree by providing a more direct comparison of the states before and after the tactic is applied. This feature could utilize our existing functionality for relating proof tree nodes to the tactics which generated them. ProofViz could also be modified to automatically save the proof history to a file when closing, instead of printing this to standard output. Finally, extending the undo functionality to full undo/redo-trees would prevent items in the redo buffer from being lost when a tactic is written in the interaction panel. With these and many more enhancements, we hope we will be able to help many more proof assistant users to come.

References

1. Chang, S., Ballantyne, M., Turner, M., Bowman, W.J.: Dependent type systems as macros. In: Proceedings ACM Programming Language 4(POPL), December 2019. https://doi.org/10.1145/3371071
2. Chang, S., Knauth, A., Greenman, B.: Type systems as macros. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. POPL 2017, pp. 694–705. Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009886
3. Curry, H.B.: Functionality in combinatory logic. Proc. Nat. Acad. Sci. **20**(11), 584–590 (1934). https://doi.org/10.1073/pnas.20.11.584
4. Dybjer, P.: Inductive families. Formal Aspects Comput. **6**(4), 440–465 (1994)
5. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 1202–1219, May 2019. https://doi.org/10.1109/SP.2019.00005
6. Felleisen, M., et al.: The racket manifesto. In: 1st Summit on Advances in Programming Languages (SNAPL 2015), pp. 113–128 (2015)
7. Friedman, D.P., Christiansen, D.T., Bibby, D., Harper, R., McBride, C.: The Little Typer. The Massachusetts Institute of Technology, Cambridge, Massuchesetts (2018)
8. Howard, W.A.: The Formulae-as-Types Notion of Construction. To HB Curry: essays on combinatory logic, lambda calculus and formalism **44**, 479–490 (1980)
9. Kawabata, H., Tanaka, Y., Kimura, M., Hironaka, T.: Traf: a graphical proof tree viewer cooperating with Coq through proof general. In: Ryu, S. (ed.) APLAS 2018. LNCS, vol. 11275, pp. 157–165. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02768-1_9
10. Libal, T., Riener, M., Rukhaia, M.: Advanced proof viewing in ProofTool. Electron. Proc. Theoretical Comput. Sci. **167**, 35–47 (2014). https://doi.org/10.4204/EPTCS.167.6
11. McBride, C.: Dependently Typed Functional Programs and Their Proofs. Ph.D. thesis, University of Edinburgh (2000)
12. de Mol, M., van Eekelen, M., Plasmeijer, R.: Theorem proving for functional programmers. In: Arts, T., Mohnen, M. (eds.) IFL 2001. LNCS, vol. 2312, pp. 55–71. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46028-4_4

13. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_217
14. Pierce, B., et al.: Logical Foundations, Software Foundations, vol. 1, September 2020. https://softwarefoundations.cis.upenn.edu/lf-current/index.html
15. Pit-Claudel, C.: Untangling mechanized proofs. In: Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering. SLE 2020, pp. 155–174. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3426425.3426940
16. Siekmann, J., et al.: LΩUI: lovely ΩMEGA user interface. Formal Aspects Comput. **11**(3), 326–342 (1999). https://doi.org/10.1007/s001650050053
17. Tews, H.: Prooftree: Proof Tree Visualization for Proof General (2017). http://askra.de/software/prooftree/. Accessed 05 Nov 2020
18. Wadler, P.: Propositions as types. Commun. ACM **58**(12), 75–84 (2015). https://doi.org/10.1145/2699407
19. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: Hacl*: a verified modern cryptographic library. In: Conference on Computer and Communications Security (CCS) (2017). https://doi.org/10.1145/3133956.3134043