



Untangling Mechanized Proofs

Clément Pit-Claudel

cpitcla@csail.mit.edu

MIT CSAIL

Cambridge, Massachusetts, USA

Abstract

Proof assistants like Coq, Lean, or HOL4 rely heavily on stateful meta-programs called *scripts* to assemble proofs. Unlike pen-and-paper proofs, proof scripts only describe the steps to take (induct on x , apply a theorem, ...), not the *states* that these steps lead to; as a result, plain proof scripts are essentially incomprehensible without the assistance of an interactive user interface able to run the script and show the corresponding proof states.

Until now, the standard process to communicate a proof without forcing readers to execute its script was to manually copy-paste intermediate proof states into the script, as source code comments — a tedious and error-prone exercise. Additional prose (such as for a book or tutorial) was likewise embedded in comments, preserving executability at the cost of a mediocre text-editing experience.

This paper describes a new approach to the development and dissemination of literate proof scripts, with a focus on the Coq proof assistant. Specifically, we describe two contributions: a compiler that interleaves Coq’s output with the original proof script to produce interactive web-pages that are complete, self-contained presentations of Coq proofs; and a new literate programming toolkit that allows authors to switch seamlessly between prose- and code-oriented views of the same sources, by translating back and forth between reStructuredText documents and literate Coq source files. In combination, these tools offer a new way to write, communicate, and preserve proofs, combining the flexibility of procedural proof scripts and the intelligibility of declarative proofs.

CCS Concepts: • Software and its engineering → Documentation; • Theory of computation → Logic and verification.

Keywords: literate programming, formal verification, proof browsing, proof presentation

ACM Reference Format:

Clément Pit-Claudel. 2020. Untangling Mechanized Proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE ’20)*, November 16–17, 2020, Virtual, USA. ACM, New York, NY, USA, 20 pages. <https://doi.org/10.1145/3426425.3426940>

1 Introduction

In Coq [50], as in many other proof assistants, propositions are encoded as types. To prove that a given proposition holds, a user must exhibit an inhabitant of that type (a *proof term*). For example, the following is a Coq proof that the conjunction operator \wedge commutes:

```
Definition and_comm :  $\forall A B, A \wedge B \rightarrow B \wedge A :=$   
fun A B ab => conj (proj2 ab) (proj1 ab).
```

The proofs of complex propositions can be quite large, so Coq proofs terms are typically generated using meta-programs called *proof scripts*. A proof script is a sequence of individual steps (such as invoking a lemma, rewriting equivalent expressions, or performing a case analysis) establishing the correctness of a proposition: each step, or *tactic*, produces a new set of hypotheses and propositions (goals), until all goals have been solved (discharged). Tactics are such a convenient way to write mechanized proofs that they are common to a wide range of proof assistants, spanning all the way from LCF (1979), to Coq (1989), HOL4 (2008), Lean (2017), and F* (2019). As an example, the three Coq proofs below each establish the correctness of the proposition above:

```
Lemma and_comm0 :  $\forall A B, A \wedge B \rightarrow B \wedge A$ .  
  intros A B ab.  
  destruct ab as (a, b).  
  constructor. apply b. apply a.  
Qed.  
Lemma and_comm1 :  $\forall A B, A \wedge B \rightarrow B \wedge A$ .  
  intros ? ? (? & ?).  
  split. all: assumption.  
Qed.  
Lemma and_comm2 :  $\forall A B, A \wedge B \rightarrow B \wedge A$ .  
  tauto.  
Qed.
```

In a sense, the variations between these proofs are immaterial, as the intricacies of a Coq proof do not matter: if Coq’s kernel accepts a proof, users can feel reasonably confident that the proof is correct. For many uninteresting proofs, the readability of the proof script is of little consequence.

And yet, for some proofs, readability matters. This is particularly true when a proof is used to communicate ideas, such as an interesting proof technique or an unexpectedly

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SLE ’20, November 16–17, 2020, Virtual, USA

© 2020 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8176-5/20/11.

<https://doi.org/10.1145/3426425.3426940>

tricky corner case, or when introducing students to formal reasoning. When all that matters is to check that a property holds, it suffices to exhibit a convincing (here, machine-checked) proof — any proof. But when one starts wondering *why* a statement holds, the readability of the proof matters.

Unfortunately, as proofs increase in complexity, proof scripts become hard to read: unlike traditional pen-and-paper proofs, the written part of a proof script records only the *steps* taken, not the *states* that they led to. Concretely, while a pen-and-paper mathematician would write “ $\int_X \int_Y f(x, y) = \int_Y \int_X f(x, y)$ by Fubini’s theorem”, a mathematician writing proofs in Coq would only write “`rewrite fubini`”, and the proof assistant would transform the left-hand side into the right-hand side and display both. Without an interactive user interface to display the goals that tactics operate on, or heavy commenting, or very specific proof-writing styles, non-trivial proof scripts are essentially inscrutable (as an example, in a Coq script like `inversion IH; eauto`, the `inversion` tactic will create some number of goals in an order determined by the definitions of the types that `IH` refers to, and `eauto` will solve some, none, or all of them using a global database of proof hints).

How do authors communicate mechanized proofs to their readers, then? One approach, used in Coq’s reference manual (written in LaTeX [48], then reStructuredText [49]) as well as books such as the *Coq’Art* [3] or *Mathematical Components* [34], is to use a markup language directly, incorporating code or proof scripts in special environments and capturing the prover’s output with varying degrees of automation. Unfortunately, the pleasant prose-writing experience offered by this approach comes at the cost of a significantly degraded proof-writing experience: authors (and readers!) cannot step through the code in an interactive editor without first reassembling a source code file, as the marked-up document is not executable on its own. As such, this approach is poorly suited to ongoing proof developments.

Instead, most proof-heavy Coq literature, including books such as *Software Foundations* [36], *Certified Programming with Dependent Types* [7], or *Programs and Proofs* [42], is distributed as annotated source files compiled to HTML or LaTeX using the *Coqdoc* tool. In this style, determined authors (it takes a special kind of grit to edit an entire book out of source code comments) use special “*literate*” comments to annotate proof scripts with explanatory prose and plain-text snapshots of the intermediate proof states printed by the author’s IDE:

```
(** The [^] operator commutes: **)
Lemma and_comm : ∀ A B, A ∧ B → B ∧ A.
  intros A B ab.
  (** <<
    A, B : Prop
    ab : A ∧ B

    =====
    B ∧ A >> **)
  destruct ab as (a, b). ...
```

Manually copy-pasting goals and messages is cumbersome, error-prone, and fragile: if the output produced by a tactic changes (due to a prover or library update, or a change to a local tactic definition), the hardcoded plain-text snapshots embedded in the proof will become stale.

It’s time for something better. This paper introduces *Alectryon*, a collection of tools intended to make it much easier to write and distribute documents mixing Coq proofs and prose. The core of Alectryon is a wrapper around Coq’s APIs that records goals and messages, plus an HTML backend that interleaves these outputs with the original proof script. This process generates an interactive document that readers can use to explore proofs without running the proof assistant. Additionally, because Alectryon records all of Coq’s output, this document forms a robust record of a proof development, immune to bit rot and suitable for long-term archival. Concretely, authors use lightweight directives to specify what should be displayed by default, and Alectryon runs Coq and generates an interactive webpage. The following listing shows an annotated proof (top) and a snapshot of Alectryon’s output (bottom), which embeds the goals and error messages produced by Coq¹:

```
Lemma ge0 : ∀ n, 0 ≤ n.
  induction n. (* .unfold *)
  - (* n ← 0 *)
    constructor.
  - (* n ← S _ *)
    Fail exact IHn. (* .unfold .fails .no-goals *)
    constructor.
    assumption.
Qed.
```

```
Lemma ge0 : ∀ n, 0 ≤ n. =
  induction n. =
```

$0 \leq 0$

$n : \text{nat} \quad \text{IHn} : 0 \leq n$


$0 \leq S \ n$

$- (* n \leftarrow 0 *) -$
 constructor.
 $- (* n \leftarrow S _ *) -$
 exact IHn.

In environment
 $n : \text{nat}$
 $\text{IHn} : 0 \leq n$
 The term " IHn " has type " $0 \leq n$ "
 while it is expected to have type " $0 \leq S \ n$ ".

constructor.
 assumption.

Qed.

¹**A note on interactivity:** This paper is about generating interactive webpages, but interactivity is hard to convey in PDF. For readers curious to see Alectryon in action, we have uploaded  all listings and webpages to <https://alectryon-paper.github.io/>.


More ambitiously, Alectryon is the first step towards a literate programming environment for Coq. Like Coqdoc, Alectryon invites authors to place prose in special *literate* comments. Unlike Coqdoc, however, Alectryon uses a well-known markup language, reStructuredText (reST) [22], inside these special comments (Coqdoc uses its own markup language), and automatically captures the prover’s output, obviating the need for manual copy-pasting. A bidirectional translation utility allows authors to switch seamlessly between two views of their documents: the Coq view, where prose is embedded in special comments and the whole file is a valid Coq module, and the reST view, where Coq code is embedded into special blocks (directives) and the whole file is a valid reST document. In literate programming parlance [28], instead of working with a literate source format to be *woven* into reST documentation or *tangled* into executable Coq code, Alectryon supports tangling directly from the woven representation (generating Coq files from reST documents), *as well as weaving directly from the tangled representation* (generating reST documents from Coq source files). While this places restrictions on the complexity of the weaving, it allows authors to tackle the distinct tasks of writing prose and writing proofs separately, with optimal tooling: with Alectryon, we do not have to enhance Coq IDEs to teach them about editing reST prose, and we do not have to enhance reST editors to teach them how to step through Coq proofs; instead, authors can simply switch back and forth between woven and tangled views, using appropriate editors for each. Integrating this bidirectional translation mechanism into an IDE such as Emacs is trivial. Our two main contributions are the following:

- An architecture to record and visualize Coq proofs, facilitating sharing and interactive exploration of proof scripts.
- A bidirectional translator between woven and tangled documents, enabling seamless editing of prose and code.

The rest of this paper offers a detailed tour of Alectryon’s features in Section 2, dives into its implementation in Section 3, presents case studies and discusses migration from existing document formats to Alectryon in Section 4, and discusses related work and some of our design decisions in Section 5.

2 A Tutorial


Alectryon can be used as a standalone command line application to compile plain proof scripts, or as a Docutils plugin to compile literate Coq proofs annotated with explanatory prose (Docutils [21] is the standard reST compilation toolchain). The command line application is a standard compiler toolchain, including a number of frontends (coq, coq+rst, coqdoc, rst, json) and backends (coq+rst, rst, snippets-html, webpage, json, and lint, though not all combinations are valid). This section showcases Alectryon by exploring the most useful pipelines.

Recording proofs: coq → webpage The most basic facility offered by Alectryon compiles a Coq document into a standalone webpage. The resulting webpage looks like a regular source code listing at first, except for small bubbles indicating interactive elements (top). Tapping a sentence reveals the corresponding output; a shaded background distinguishes it from neighboring inputs (bottom). 

```
Lemma rev_rev {A} (l: list A) : List.rev (List.rev l) = l. =
Proof. =
  induction l; cbn. =
  - (* l ← [] *) =
    reflexivity.
  - (* l ← _ :: _ *) =
    rewrite rev_app_distr. =
    rewrite IHl. =
    cbn. = reflexivity.
Qed.
```

⚡ (tapping or hovering reveals output)

```
Lemma rev_rev {A} (l: list A) : List.rev (List.rev l) = l. =
Proof. =
  induction l; cbn. =
  - (* l ← [] *) =
    reflexivity.
  - (* l ← _ :: _ *) =
    rewrite rev_app_distr. =
    rev (rev l ++ a :: nil) = a :: l
    rewrite IHl. =
    rev (a :: nil) ++ l = a :: l
    cbn. = reflexivity.
Qed.
```

Finally, keyboard commands can be used to focus on one goal, and with just a bit of CSS we can replicate the multi-pane interface traditionally used in Coq IDEs: 


```
Lemma Gauss:
  ∀ n, 2 * (sum n) = n * (n + 1). =
Proof. =
  induction n. =
  - (* n ← 0 *) =
    reflexivity.
  - (* n ← S _ *) =
    cbn [sum]. =
    rewrite Mult.mult_plus_distr_l. =
    rewrite IHn. =
    change (S n) with (1 + n). =
```

```
n : nat
IHn : 2 * sum n = n * (n + 1)
2 * S n + 2 * sum n = S n * (S n + 1)
```

In most cases, users just need to invoke alectryon.py instead of coqc to generate these webpages, and, in line with Tankink et al. [46], we argue that they represent a much better way to distribute proofs than simply sharing plain Coq

files: unlike plain sources, which are subject to rapid bit rot as new versions of Coq appear, they offer a complete and permanent record of a proof, suitable for archival.

Controlling output For readability, Alectryon recognizes special flags placed in comments to let users control what should be included in the output: `.in` for input sentences, `.goals` and `.messages` (collectively `.out`) for Coq's output, or `.all/.none` to include/hide everything. The exact semantics depend on the polarity of the first inclusion option encountered: `.x .y` means `.none .x .y`, i.e. include `.x`, `.y`, and nothing else; `.no-x .no-y` means `.all .no-x .no-y`, i.e. include everything but `.x` and `.y`. An additional pair of flags (`.fold` and `.unfold`) controls whether the output should be shown or hidden by default (clicking the input toggles the visibility of the output). Lastly, `.fails` can be used to indicate that a command is expected to display an error message.

These annotations allow authors to highlight important steps by unfolding the corresponding goals. It also enables authors to hide irrelevant or redundant bits such as **Require** statements, **Sections**, or the **Show Proof** commands used in the following example to demonstrate the contribution of individual tactics to the construction of a proof term: 

```
Section classical. (* .none *)
Context (excl: ∀ A, A ∨ ¬ A).
Goal ∀ A, ¬¬A → A.
  intros A notnot_A.
  Show Proof. (* .messages .unfold *)
  destruct (excl A) as [a | na].
  Show Proof. (* .messages .unfold *)
  - assumption.
  Show Proof. (* .messages .unfold *)
```

⚡

```
Context (excl: ∀ A, A ∨ ¬ A).
Goal ∀ A, ¬¬A → A. =
  intros A notnot_A. =
```

```
(λ (A : Prop) (notnot_A : ¬ ¬ A), ?Goal)
```


```
destruct (excl A) as [a | na]. =
```

```
(λ (A : Prop) (notnot_A : ¬ ¬ A),
  let o : A ∨ ¬ A := excl A in
  match o with
  | or_introl a => ?Goal
  | or_intror na => ?Goal0
end)
```

```
- = assumption.
```

```
(λ (A : Prop) (notnot_A : ¬ ¬ A),
  let o : A ∨ ¬ A := excl A in
  match o with
  | or_introl a => a
  | or_intror na => ?Goal
end)
```

Compiling reST documents: `rst+coq` → `webpage` To compile reST documents, Alectryon plugs itself into the *Docutils* reST compiler by registering a custom `.. coq::` environment

(*directive*) before handing off control to the standard compilation toolchain. The *Docutils* compiler then delegates processing the contents of these directives back to Alectryon, which gathers their contents, feeds them to Coq, and interleaves the corresponding outputs with the original sources. This process can be used to plug Alectryon into various document preparation systems, blogging platforms, and documentation generators: 

Booleans

The built-in `:coqid:`Boolean <Coq.Init.Datatypes.bool>`` type has two constructors:

```
.. coq:: unfold no-in
  Print bool.
.. topic:: Induction principles
  Coq automatically defines an induction principle:
.. coq:: unfold
  Search (∀ b: bool, _ b) "_ind".
```

⚡

Booleans

The built-in `Boolean` type has two constructors:

```
Inductive bool : Set :=
  true : bool | false : bool
```

Induction principles

Coq automatically defines an induction principle:

```
Search (∀ b: bool, _ b) "_ind". =
```

```
bool_ind:
  ∀ P : bool → Prop,
  P true → P false → ∀ b : bool, P b
```

Compiling literate Coq files: `coq+rst` → `webpage` Alectryon recognizes comments delimited with special markers `(*| ... |*)` as reST fragments, and generates a webpage using the *Docutils* compiler and its own recording facilities to generate an interactive webpage. The example below produces the same output as the reST snippet above:

```
(*|
Booleans
```

```
The built-in :coqid:`Boolean <Coq.Init.Datatypes.bool>`
type has two constructors:
|*)
```

```
Print bool. (* .unfold .no-in *)
```

```
(*|
.. topic:: Induction principles
  Coq automatically defines an induction principle:
|*)
```

```
Search (∀ b: bool, _ b) "_ind". (* .unfold *)
```

Toggling between woven and tangled views In traditional literate programming, a single literate document is woven into a text-oriented format (such as reST or LaTeX) and tangled into executable source code. In Alectryon, there is no single source of truth: both formats are equally valid sources. Woven reST sources can be tangled, and Coq code can be untangled (woven). For this to work, the tangling process preserves all the prose as special `(*| ... |*)` comments, and the untangling turns these special comments back into prose, and wraps code into `.. coq::` directives. This bidirectional translator is exposed through the command line and through a simple Emacs mode. Additionally, in Coq mode, the Emacs mode invokes Alectryon with its lint backend to untangle the document and check it for reST mistakes (such as syntax errors or broken references), and reports these mistakes as overlays in the Coq code (careful position tracking makes it possible to translate error locations in both directions and to keep the author’s position in a document when switching between Coq and reST). A side-by-side screenshot is provided in Figure 9, in the appendix.

Recording and caching Coq’s output: `*` \rightarrow `json` Instead (or in the process) of generating a webpage, Alectryon supports generating a JSON file containing Coq’s responses for further processing — these files can then be loaded by Alectryon’s Docutils module to avoid re-invoking Coq. Previous literature [46] calls this type of record a *movie* (here shown in compact form for space):

```
Let inv b: negb (negb b) = b.
  destruct b. all: reflexivity.
Qed.
Print Assumptions inv.
```

```
[{"sentence": "Let inv b: negb (negb b) = b.",
 "goals": [{"conclusion": "negb (negb b) = b",
 "hypotheses": [{"names": ["b"],
 "type": "bool" }]}], "\n ",
 {"sentence": "destruct b.",
 "goals": [
 {"conclusion": "negb (negb true) = true" },
 {"conclusion": "negb (negb false) = false" }]}], " ",
 {"sentence": "all: reflexivity." }, "\n",
 {"sentence": "Qed." }, "\n",
 {"sentence": "Print Assumptions inv.",
 "responses": ["Closed under the global context" ]}]
```

Caching movies has multiple benefits:

1. Recompiling documents after editing the prose is much faster, since Coq snippets do not have to be re-evaluated; this makes the overhead of compiling documents with Alectryon essentially imperceptible.
2. Deploying a website or recompiling a book does not require setting up a complete Coq development environment. This is particularly useful when multiple authors are responsible for producing different parts of a document, like book chapters or blog posts: each author only needs to install the Coq libraries required for their own

part of the document, and each part can even require different versions of Coq, either because of possibly incompatible plugins or because of bit rot: old proofs may be of historical value but may not compile with the latest releases of Coq, for example. In that sense, movies provide a robust form of long-term storage for Coq documents.

3. Changes in output can be inspected by comparing cached records. Movies contain just enough information to recreate input/output listings, so they can be checked-in into source control, making it easy to assess whether updating a library or Coq itself meaningfully affects a document (it is otherwise easy to miss breakage or subtle changes in copy-pasted or automatically recorded output). JSON records of movies can also be used as a convenient format to interoperate with third-party libraries.

Using custom markup: `coqdoc` \rightarrow `webpage` Alectryon primarily focuses on reST support, but it can also be used with other markup languages. For example, we can compile unmodified Coqdoc documents by using Coqdoc instead of Docutils to compile the prose found in literate comments, while continuing to use Alectryon to render code and proofs (more details are given in section 4):

```
(** * Function types
Implications ( $P \rightarrow Q$ ) are
_universal quantifications_ ( $\forall x, Q$ ): *)
Set Printing All. (* .none *)
Check (False  $\rightarrow$  True). (* .unfold *)
```

Function types

Implications ($P \rightarrow Q$) are *universal quantifications* ($\forall x, Q$):

```
Check (False  $\rightarrow$  True). —
```

```
forall _ : False, True
: Prop
```

3 Implementation


Alectryon is implemented in Python:


- A core module takes a list of code snippets, feeds them to Coq, and records goals and messages (626 lines of code).
- A transforms module implements document transformations used by the various compilation pipelines (377 LoCs).
- An html module formats recorded goals and messages as HTML, which, paired with appropriate CSS (and, optionally, JavaScript), can be explored interactively (202 LoCs).
- A literate module implements translations from Coq to reST and from reST to Coq (554 LoCs).
- A docutils module plugs Alectryon into reST, making it easy to embed Coq snippets in reST documents. A sphinx module wraps this functionality for use with the Sphinx documentation system (593 LoCs).
- A json module serializes recorded output (166 LoCs).
- A cli module ties these tools together (626 LoCs).

This section dives into the details of the implementation of each of these components (the last two are simple enough that we do not discuss them further). Readers interested in the high-level takeaways can skip to the following sections, which evaluate Aletryon and discuss our design decisions in relation to previous work.

3.1 The Core

The core of Aletryon is built on top of a new IDE protocol for Coq called SerAPI [19]. It exposes a minimal interface: a constructor to start an instance of SerAPI and a method to process a fragment of Coq code, returning a list of sentences paired with the corresponding outputs.

Coq's extensible notation system lets users define readable domain-specific languages with complex grammars, but it also makes it much harder to reliably segment a Coq document into sentences without invoking Coq. It would be relatively easy to parse comments, strings, standard Coq sentence terminators like `.` and `...`, bullets `-+*`, and braces `{}` (though see the discussion in subsection 5.1), but Coq does not forbid defining custom notations including periods. Properly handling that case essentially requires running Coq's parser; the following confuses most Coq IDEs, but not the compiler, nor Aletryon: 

Notation `"(a . b)" := (a, b).`
Check `(1 . 2).` 



Aletryon handles it correctly by mixing the segmentation and execution phases and letting Coq handle both. Outputs are computed by feeding the fragments to Coq one by one, processing them, recording sentence boundaries and raw goals and messages, and querying Coq for a pretty-printed representation of the hypotheses and conclusion of each goal. Letting Coq split sentences for us means that sentences and comments cannot span fragment boundaries, but we have not found this to be an issue in practice.


3.2 Document Transforms

Recording Coq's output produces a list of sentences paired with their outputs. Depending on the application, Aletryon applies certain transforms to each input-output pair before formatting the result.

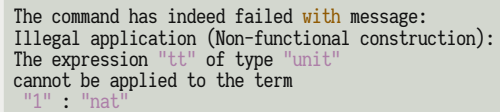
1. *Attach comments to preceding code.* A common pattern in Coq code (popularized by *Software Foundations* [36]) is to add short comments to help readers orient themselves in the proof — this style is useful even when running in a full-featured IDE, since it gives a brief summary of what the reader is looking at:


```
induction n. =
- (* n ← 0 *) = ...
- (* n ← S _ *) = ...
```

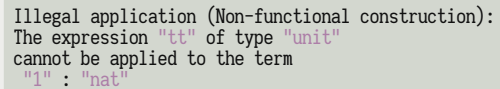
The default segmentation algorithm parses the text following the call to `induction` as a sentence including the first `-` bullet followed by a comment `(* n = 0 *)`. Without post-processing, Aletryon would turn only `-` into an interactive element, making it hard to click or hover. This first transformation merges comments into preceding code sentences, as long as they are on the same line. It can optionally be restricted to bullets.

2. *Group hypotheses by type.* This is done to save space and line up with Coq's default rendering strategy: consecutive hypotheses of the same type are coalesced into a single hypothesis with two names.
3. *Process I/O annotations* (`.in`, `.out`, etc.). Annotation comments are stripped from input sentences and accumulated to determine what should appear in the output, and what should be shown by default. The default is that all output is included and hidden (folded). As this step comes after running Coq, all that can be done is to hide part of the proof script, not prevent it from being executed (this guarantees that running the script in a regular Coq IDE will produce the same results).
4. *Highlight failing commands.* This phase post-processes sentences annotated with `.fails`. We strip the `Fail` prefix from the input, remove the failure header added by Coq to the output, and indicate the failure with a red underline (top: plain; bottom: stripped): 

Fail Check `(tt 1).` 



Check `(tt 1).` 



We discuss the reasoning behind requiring all sentences to succeed in section 5.

5. *Group white space with preceding code.* This is used only by the HTML backend, and it matters when a block of output is unfolded. Briefly: when displaying goals or messages, we want to hide blanks that follow the corresponding input, up to and including the first subsequent new-line, since the output is already presented as a block.
6. *Apply custom filters.* This enables fine grained-control on the output, like mixing Coq syntax with graphics, using \LaTeX math, embedding DSLs with custom syntax highlighting, displaying contextual diffs between proof steps, etc. (examples are given in Appendix A). Simple filters typically use custom Coq notations to identify parts of the proof that require custom processing and some JavaScript code for rendering; more complex filters can

embed arbitrary Python objects into the recorded output and hook into the HTML backend directly to customize the way these objects are rendered.

Alectryon’s core and (to a lesser extent) the transforms and literate modules are the only parts of it that are strongly tied to Coq: to implement support for other proof languages, it would be enough to reimplement the core on top of alternative communication protocols like the IDE frontends of F* [38], Lean [12], or Idris [6], add any relevant language-specific transforms, and adjust the comment delimiters used in the tangling and untangling code. If a standardized protocol emerges to communicate with proof assistants, possibly as an extension of the popular Language Server Protocol [16], porting Alectryon will only be a matter of rewriting the core module and developing appropriate transforms.

3.3 HTML Rendering

Translating input/output pairs into HTML trees is straightforward, except for two points:

Most interactivity is achieved using CSS, not JavaScript. In particular, hovering is detected using CSS, and individual output elements are toggled using hidden checkboxes: Alectryon creates one checkbox per input sentence and uses `<label>`s to make it so that clicking on an input sentence toggles the corresponding checkbox. Then, CSS rules are used to selectively display or hide the output. This technique is usually called the “checkbox hack” [10].

There are multiple advantages to using this “hack”: users do not need to run JavaScript code (in particular, the webpages work fine with all scripts disabled), and browsers automatically keep track of which outputs were toggled by the user, so reloading an Alectryon webpage does not reset the page’s state. Browser support is good: this approach works fine with browsers as old as IE 11 (released in 2013).

A minimal amount of JavaScript (dozens of lines) can be used to further enhance the experience, allowing users to focus on a specific sentence and to replicate the IDE experience of stepping through a document using the keyboard.

The structure of the HTML documents that Alectryon generates is carefully tuned to ensure that Alectryon’s output remains readable when CSS and JavaScript are disabled, as is common when viewing HTML documents in news aggregators (RSS and Atom feed readers; Figure 10). Minimal inline CSS provides visual improvements where supported.

3.4 Tangling and Untangling

In its simplest form, tangling and untangling is relatively easy to implement: from reST to Coq, locate `.. coq::` directives and wrap all other text in `(*) ... (*)` comments. From Coq to reST, simply remove the `(*) ... (*)` markers. There are multiple problems with this approach, so the following goes over the main issues: identifying literate comments, adjusting indentation, and trimming redundant markup.

Parsing Coq code Not all “(*)” tokens in a Coq document are comment delimiters: for example, the command `Check "string (* ... *)"` does not contain a comment, since the delimiters are contained in a Coq string. Thus, to untangle (convert from Coq to reST) Alectryon starts by using a simple recursive-descent parser to segment documents into sequences of code and comment spans (Coq does not let users redefine comment markers, so we do not need to invoke the compiler to parse comments). Separately, when tangling (translating from reST to Coq), we escape “(*)” and “(*)” tokens in prose to avoid opening a nested comment or prematurely closing the surrounding literate comment (these tokens are not rare, since “*” is used to indicate emphasis in reST, *(*)like this**).

Adjusting indentation Indentation is semantically meaningful in reST, so Alectryon needs to preserve enough information when tangling to reconstruct it when untangling. We could simply preserve all indentation, of course, but it would yield poor usability: users would have to carefully preserve indentation carrying no direct connection to the structure of their Coq code. Instead, when toggling between woven reST documents and tangled Coq ones, Alectryon unindents the Coq code by the amount of the preceding `.. coq::` directive; when untangling, the process is reversed and code is indented back under the preceding `.. coq::` directive (top: reST view; bottom: Coq view). With this process, it is safe to automatically reindent tangled Coq using any preferred indentation style — the only constraint is to respect reST conventions inside `(*) ... (*)`.

```
.. note::
    Most commands accept ``Local`` and ``Global``:
.. coq::
    Global Open Scope list_scope.
```

⚡

```
(*)
.. note::
    Most commands accept ``Local`` and ``Global``:
.. coq::
    Global Open Scope list_scope.
```

Minimizing redundant markup The tangling and untangling functions should be bijective (modulo whitespace and redundant markup), and their composition should be idempotent (the first round of `coq` → `rst` → `coq` or `rst` → `coq` → `rst` conversions may normalize spacing and markup, but subsequent roundtrips should not make further changes). To improve readability, the tangling process (reST to Coq) removes all `.. coq::` markers that can be unambiguously reconstructed — that is, all markers that do not have custom flags and whose indentation (nesting) level matches the immediately preceding line. Here is a concrete example (top: reST view; bottom: Coq view):

```

``Let`` introduces a local definition:
.. coq::
  Section A. Let a := 1. End A.
.. coq:: unfold fails
  Fail Check a.
.. note::
  Outside sections, ``Let`` will print a warning.
.. coq::
  Let a' := 1.

```



```

(*)
``Let`` introduces a local definition:
(*)
Section A. Let a := 1. End A.
(*)
.. coq:: unfold fails
(*)
Fail Check a.
(*)
.. note::
  Outside sections, ``Let`` will print a warning.
.. coq::
(*)
Let a' := 1.

```

In the reST view (top), each Coq fragment is prefixed with a `.. coq::` header. In the tangled Coq view (bottom), however, Alectryon removes the first `.. coq::` marker, since it can be inferred. The other two are kept: the first one because it has arguments (unfold fails); the second because it specifies a different nesting (indentation) level than the immediately preceding text (if removed, the second **Let** would be nested inside the note when untangling back to reST).

3.5 Docutils Integration

The standard reST compilation toolchain, Docutils, has extensive support for language extensions, including custom roles (inline markup), directives (block markup), AST nodes, and AST transforms. Alectryon plugs into it using a custom `.. coq::` directive that records its contents into a custom “pending” node. Once the document is fully parsed, a custom AST transform uses Alectryon’s APIs to process the content of each pending node, replacing them with formatted HTML nodes. The implementation of the directive is slightly more complicated than typical Docutils directives: by default, Docutils strips the indentation of a directive’s body, whereas we want to ensure consistent indentation between consecutive Coq blocks. This is visible in the following example, in which Docutils would, if `.. coq::` were a regular directive, line up the **Goal True** and **exact I** statements as if they had the same indentation.

```

.. coq::
  Goal True.
.. coq::
  exact I.

```

A few additional roles allow users to write inline code (not executed) and to link to external definitions (so that `:coqid:Coq.Arith.PeanoNat#Nat.Even` produces a link to the definition of `Nat.Even` in the [Coqdoc rendering of Coq’s standard library](#)).

Combining Alectryon’s Docutils extensions with the untangling support provided by the `literate` module would be enough to compile literate Coq documents, but the experience would be miserable: since tangling and untangling can change line numbers, reST mistakes would be reported at positions relative to the untangled reST document, not the original Coq source file. Instead, Alectryon implements a custom Docutils parser frontend, which produces a Docutils AST by untangling Coq sources while preserving original line and column numbers, allowing for precise error reports.

4 Evaluation

Alectryon has been in development for a bit over a year, but had not been publicly released until now, and thus has not yet seen widespread use. To evaluate it, we plugged it into a reST-based blog engine for our lab’s blog (<https://plv.csail.mit.edu/blog/>), integrated it into the Coq reference manual [5] in replacement of the existing `coqrst` module, and used it to recompile a popular book in its entirety (*Logical Foundations* [36] [5]), to compile chapters manually translated to reST from two other books (*Certified Programming with Dependent Types – CPDT* [7] [5] and *Functional Reasoning About Programs – FRAP* [8] [5]), and to generate webpages for Coq’s standard library [5] and for a popular library, *Flocq* [4] [5].

Compilation times Compiling literate programs with Alectryon generates significant but reasonable overheads over regular compilation. Figure 1 shows compilation times for a randomly selected subset of Coq’s standard library.

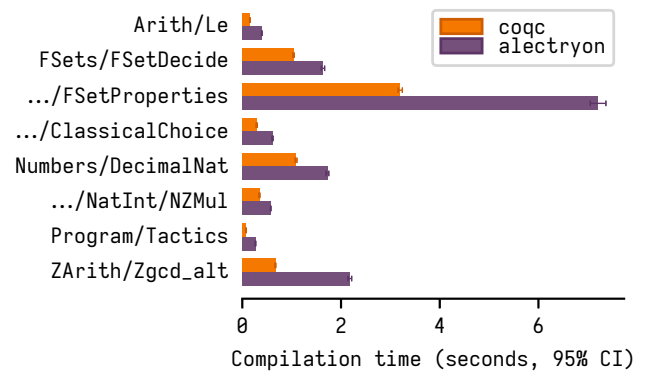


Figure 1. Compilation times with `coqc` and Alectryon for a subset of Coq’s standard library (Intel Core i7-4810MQ CPU @ 2.80GHz, single-threaded). Error bars indicate 95% confidence intervals (95% CI).

Over a larger test set (Coq’s standard library), Alectryon’s per-file overheads vary between $1.2\times$ and $110\times$ (median: $3\times$; 90th percentile: $6.9\times$; 95th: $11.6\times$), and the overall slowdown is about $5.5\times$; Figure 2 shows their distribution. Compiling *LF* takes seconds. In practice, these overheads do not matter much: they are incurred only once before Alectryon’s caching kicks in.

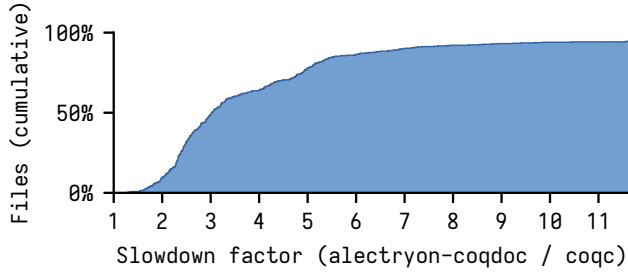


Figure 2. Cumulative distribution of slowdowns (ratios of execution time) incurred by compiling Coq’s standard library (482 files).

Interestingly, there is little correlation between *coqc* and Alectryon’s running times: for files with few goals or simple goals Alectryon’s speed is comparable to that of *coqc*, while for files with numerous or complex goals Alectryon is much slower. To better understand how Alectryon’s overheads break down, Figure 3 offers a detailed look at increasingly complex pipelines on three examples with median and extreme overheads, and confirms our intuition: *Byte.v* is a short file with small goals but long-running, highly automated tactics (145 sentences): Alectryon’s overhead is minimal, because there’s not too much data to retrieve from Coq and to format (Alectryon sends and receives about 1MB). *Ranalysis3.v*, on the other hand, manipulates large proof states (a typical state in that file has 8 goals, each with 40 hypotheses) in rather explicit style (569 sentences), which generates upwards of 165MB worth of communication between Alectryon and SerAPI (unlike *Coqtop*, Alectryon records and formats the hypotheses of all goals at each step, not just the top one).

Porting existing documents A wealth of Coq documents have already been written using Coqdoc’s syntax. For full compatibility, one could write a Docutils frontend for that syntax, but for simplicity we implemented a Coqdoc-specific compilation pipeline instead. We start with a document containing Coqdoc comments, segment it using the first stage of the untangling code, gather all Coqdoc comments into a new document, compile that document to HTML using the *coqdoc* binary (compiling all prose at once allows us to invoke *coqdoc* just once), segment the resulting HTML and finally interleave it with the HTML generated by Alectryon

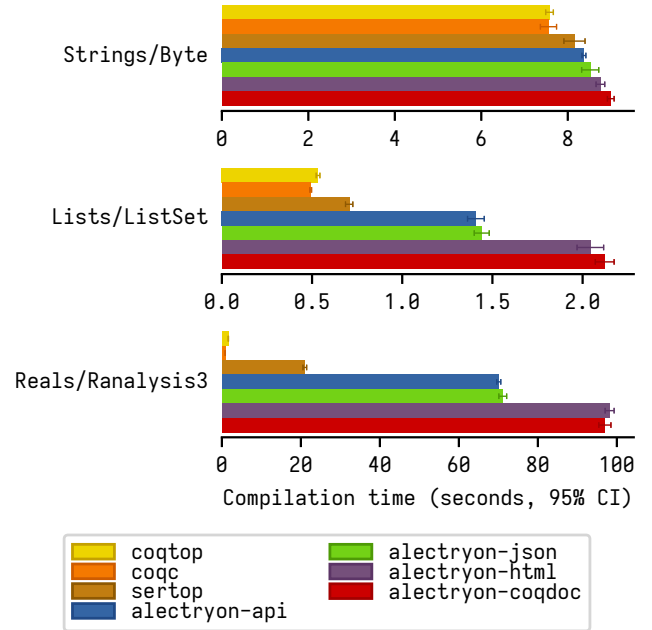





Figure 3. Compilation times for various pipelines. *coqtop* and *coqc* are the usual Coq compilers. *sertop* is SerAPI’s REPL, running on pre-recorded inputs generated by Alectryon (as an example, these inputs total about 90MB worth of SerAPI commands in the case of *Ranalysis3.v*); it measures the cost of running Coq and pretty-printing goals and hypotheses, plus parsing and unparsing overheads in SerAPI. *alectryon-api* stops before producing any output; it adds the overhead of parsing and unparsing on Alectryon’s side. *alectryon-json* serializes results to JSON. *alectryon-html* generates plain webpages. *alectryon-coqdoc* generates webpages in Coqdoc compatibility mode.

for each source code fragment. This approach is fairly robust: it works reliably for *Software Foundations*, one of most widely used Coq textbooks. 

As another experiment, we manually ported a chapter of *CPDT* from Coqdoc to literate Coq with reST comments. This book relies more heavily than *Logical Foundations* on displaying goals, messages, and errors, so it required more significant adjustments to take advantage of Alectryon’s facilities (in *Logical Foundations* it was enough to add a few display annotations and remove a few copy-pasted outputs). A reasonable strategy is to change Coqdoc comments (`(** ... **)`) into Alectryon ones (`([*] ... [*])`), untangle the document into reST, and make appropriate syntax edits before tangling back. Porting to Alectryon required a few minutes per section (a few pages), corresponding roughly to the speed of a careful read-through. 

Finally, we recompiled Coq’s reference manual, using Alectryon as a replacement for *coqrst* (the tool currently

used by Coq’s reference manual) and tweaking its configuration to match coq_{rst}’s behavior. The process was smooth: the only change that we had to perform was prefixing failing sentences with an explicit **Fail** annotation, since Alec_{tryon} rejects erroneous code (the reference manual uses such code to demonstrate errors; adding **Fail** and a `.fails` flag reproduces the pre-existing behavior). For simplicity, we left the `reset` and `abort` flags currently used in the manual unchanged, but a complete port would likely want to change these to explicit `Reset` and `Abort` commands, to ensure that tangling produces a file whose behavior fully matches that of the reST sources. 

5 Related Work and Discussion

This section reviews related work and discusses parts of Alec_{tryon}’s design. The most closely related tools are Coqdoc, a documentation language for Coq distributed as part of Coq’s standard distribution, Proviola [46] a “tool for proof reanimation”, and CoqTeX, a Coq interpreter designed to annotate Coq inputs with the corresponding outputs in LaTeX files.

Coqdoc is the de facto standard for documenting proof scripts in the Coq community. As a literate programming language, authors have used it in a number of popular books [7, 36]. As a documentation language, it is commonly used to attach docstrings to bits of Coq code. Its main limitations are that it uses a custom markup language for its prose and that it does not support embedding Coq’s output into the webpages and LaTeX files that it generates.

Coqdoc’s use of a custom markup language is limiting: Coqdoc documents often include raw LaTeX or HTML code to work around these limitations, and little support exists for editing Coqdoc prose. In contrast, Alec_{tryon}’s reliance on reST and its ability to untangle Coq files into reST documents means that literate Coq files inherit all the features of reST, and that authors can switch to a specialized reST editor to write their prose.

Still, Coqdoc has the edge over Alec_{tryon} in a few cases, especially when it comes to hyperlinking: Coqdoc understands Coq’s `glob` files, which enables it to hyperlink uses of identifiers and notations to their definition — we plan to use corresponding features in Alec_{tryon} once the SerAPI interface exposes sufficient information.

Proviola [46] implements a subset of Alec_{tryon}’s output-recording features: it lets authors record *movies* (XML files capturing input and output from a proof assistant) by means of a *camera* (recording software) and replay them using a *proviola* (such as an interactive webpage). In Proviola terms, Alec_{tryon}’s core is a *camera* built on top of modern Coq APIs; the *movies* it generates are semantically richer (Proviola records Coq’s output as pre-formatted plain text); and its interactive webpages are *proviolae* offering multiple interaction modes, one of which matches the traditional IDE

display used by the original Proviola system. Unlike Proviola, Alec_{tryon} can be used to interleave Coq’s output in proofs that are part of literate documents (Proviola can only be used to emulate the UI of a Coq IDE). Proviola has seen little development since 2013, but its reliance on coq_{top}, Coq’s REPL interface, has granted it surprising longevity (Proviola was originally released in 2010). It is now part of the coq-community repository.

CoqTeX was long used to display input/output examples in Coq’s reference manual — it was used until the manual was migrated to reST and Sphinx, at which point the manual switched to coq_{rst}, a newly developed Docutils plugin that uses coq_{top} to execute Coq sentences and incorporate Coq’s responses into generated documents [37]. Part of the motivation to develop Alec_{tryon} were repeated requests on public discussion forums to make coq_{rst} into a standalone tool usable independently of Coq’s reference manual.

Alec_{tryon}’s core is a clean-slate reimplement of the REPL part of coq_{rst} based on a richer API. Its HTML backend differs from coq_{rst} and CoqTeX in that both of these tools produced fully static webpages, while Alec_{tryon} lets readers browse goals interactively. CoqTeX and coq_{rst} do not support tangling a LaTeX or reST document into a Coq source file; as a result, readers need to manually copy examples from the reference manual into a fresh Coq file to be able to explore them in a proof assistant (in contrast, Alec_{tryon} supports tangling reST files into executable Coq sources).

Alec_{tryon}’s `.. coq::` directive is essentially the same as the `.. coqtop::` directive used in coq_{rst}, though we add support for annotations on individual lines (coq_{rst} only supports I/O configuration for complete directives), and we enforce that all Coq code included in those directives should be valid and error-free. We argue that this last point is a significant improvement. Indeed, coq_{rst}’s original design simply resumed execution at the next sentence every time it encountered an error: this was convenient for showcasing error messages, but it allowed breakage introduced by Coq upgrades to go undetected. To protect against this, a new Coq setting was introduced to force coq_{top} to exit after encountering an error, and the `.. coq::` directive was taught to temporarily disable that setting for blocks marked to allow failing sentences. This design is a significant improvement over the original coq_{rst}, but it does not guarantee that the right sentence, or really any sentence, is failing, leaving potential errors undetected. In contrast, Alec_{tryon}’s single-sentence annotations offer a more robust design: users must prefix failing sentences with an explicit **Fail** command, which suppresses the failure (from Coq’s perspective, the line succeeds); an explicit `.fails` annotation can then be used to strip Coq’s error header, producing the same output as coq_{rst} (one weakness of this scheme is that Coq currently does not print error locations for commands prefixed with **Fail**). Single-sentence annotations can

likewise be used to implement the restart, reset, and abort flags currently supported by `coqrst`: instead of letting `coqrst` insert `Restart`, `Reset Initial`, or `Abort All` commands, we can include them in the actual source listing and hide them with a `.none` annotation, which guarantees that the tangled file will execute correctly. Two more differences enhance the robustness of the Alectryon-based build of the manual: unlike `coqrst`, Alectryon leaves sentence segmentation to Coq itself; and Alectryon can record and save proof movies, making it easy to spot regressions when rebuilding the manual for a new Coq release.

5.1 Discussion

Parsing Coq comments Block comments in Coq are relatively complicated: parsers need to track not just nested comments but also nested strings, an oddity Coq inherited from OCaml (string delimiters in comments must be properly matched, and comment markers within them are ignored). This design was intended to make commenting more robust, so that wrapping a valid bit of code in `(* ... *)` would always produce a comment spanning the whole wrapped fragment. As an example, the following is valid OCaml code:

```
let a = "x *) y" in
(* let a = "x *) y" in *) a
```

Unfortunately, most tools handle this snippet incorrectly (in particular, Pygments, a popular syntax-highlighting library commonly used with Docutils, which we use for syntax highlighting in this paper, gets it wrong). So does Emacs' `tuareg-mode`.

Arguably, the reasoning behind this design does not apply in Coq: unlike in OCaml, a lone `"*)`" token is *not* rejected in Coq, and in fact it is relatively common in proof scripts, as in `split; (reflexivity || rewrite H in *)`, for example. As a result, paired `(* ... *)` delimiters already cannot be used reliably to comment out a block of code.

The development of a system similar to Alectryon, `fsplit`, designed for use with the F* programming language, was significantly simplified by F*'s single-line comments: commenting is more reliable, literate blocks (prefixed with `///` markers) are less invasive than `(*| ... |*)` markers, and comment delimiters inside prose do not need to be escaped. Proposals have been made to enhance Coq (and, in fact, OCaml) with single-line comments, but none has gathered significant support yet.

Docstrings and literate comments Alectryon has no support for attaching documentation to specific bits of code, like definitions, axioms, variables, etc. We claim that this is a different job ("docstrings"), ideally to be handled by Coq itself (similar to how Coq tracks the body and location of definitions). These docstrings could then be referred to using custom reST directives, likely as part of a new extension of the Coq semantic domain for Sphinx used by the reference manual. In the long run, such extensions could be combined

with a proper cross-referencing system to allow authors to present code fragments and proofs in arbitrary order: one would use custom directives to pull definitions and annotated proofs from Coq developments into reST documents (such a tool would leverage Alectryon's recording and interactive presentation features, but it would be orthogonal to its literate programming support: to allow straightforward bidirectional translation between reST and Coq views, Alectryon's literate programming tools do not allow authors to reorder code fragments).

Choosing a markup language Alectryon heavily favors reST — the choice was made for the same reasons as for Coq's manual: the language and its tooling (Docutils) are extensible, battle-tested, and relied upon by many other projects (Python, Haskell, Agda, etc.). On the other hand, its syntax leaves much to be desired: most importantly, its inline markup is fragile and cannot nest. This paper is written in reST and compiled using the standard Docutils toolchain.

In principle, Docutils can compile any markup language, as long as an appropriate frontend is written for it. In practice, the default distribution only includes support for reST, but third-party frontends exist, like `recommonmark` [52] for a dialect of Markdown (it offers limited support for custom directives, piggybacking on the syntax for fenced code blocks). We have not designed a translator to tangle and untangle to and from `recommonmark`'s dialect of Markdown, though it would be relatively simple to do so.

5.2 Other Related Work

Variants of CoqTeX and Coqdoc The limitations of existing tools have prompted the development of a host of other programs for documenting Coq scripts, most of them variants of `coqdoc` or `CoqTeX`. `coq-lit` [59] was built by Jay Wilcox for his blog; it builds upon `Coqdoc` to format goals copy-pasted into comments as floating tooltips. *Mathematical Components* [47] and the *Coq'Art* [3] each use custom systems similar to `CoqTeX`; both record Coq's output within the original TeX document (`CoqTeX` generates a new copy of the document instead) and both can be used to refresh or check the output (interestingly, both checkers are robust to manual changes like tweaking the indentation or removing parts of the output). Like `CoqTeX`, `coq-psv` [18] automatically records Coq's output, but it displays the input and the output side-by-side, in two-column format, and attempts to minimize the size of the output by omitting redundant parts.

In-browser IDEs and notebooks `Coqdoc`'s default output is fully static and omits Coq's output, but it can be enhanced with `jsCoq` [20] to run a copy of Coq in the user's browser, enabling interactive exploration. Related efforts include similar software for Lean [12], HTML visualizations of Mizar proofs based on its XML movie format [54, 55], a new Coq kernel [33] for Jupyter (Jupyter is a browser-based notebook

application [27]), as well as PeaCoq [39] and the more venerable ProofWeb [26], the last two of which support more advanced proof visualizations. We expect that it would be straightforward to plug jsCoq into webpages generated by Alectryon (and that it would complement its pre-recorded outputs nicely), but part of the appeal of Alectryon’s pre-computed webpages is that they minimize the requirements placed on readers: running Coq scripts can require significant amounts of computational resources, especially in a web browser, while Alectryon’s pre-recorded outputs are instantly available and relatively lightweight.

Structured editors and projectional editing Alectryon’s bi-directional translation between Coq and reST is a simple form of *projectional editing*, a style in which programs are written as abstract objects edited by projecting them into various representations [17, 44] (for example, a fluid dynamics simulation package might include mathematical formulae edited using a graphical equations editor, models edited in a 3D modeling program, a graphical user interface edited using a UI-building toolkit, and a grammar for its configuration files projected into railroad diagrams editable using a structured graphical editor). In projectional editors, the storage format of the program is usually distinct from its projections; in Alectryon, Coq and reST projections are directly converted into one-another instead and no separate storage format is ever materialized.

Literate programming Literate programming systems are too numerous to enumerate here, though interested readers should consult Knuth’s seminal work on the topic [29, 30], as well as reviews from its early days [45].

Interactive theorem provers present a special challenge for literate programming, however, because proof scripts are typically written with the help of a specialized environment (at the very least a REPL). This constraint means that proof-assistant languages do not usually lend themselves to the usual literate-programming style: without additional tooling, mixing prose and code in a non-executable document format and then deriving executable code by tangling the original sources precludes interactive code edition.

Still, the most robust solution in that style for Coq is likely the Coq Babel extension of Org mode, an “*Emacs plain-text system*” [41]. The experience is similar to using CoqTeX or coqrst, but all of Coq’s output is placed together at the end of each code block instead of being interleaved with the corresponding inputs. Extensions can be used to refer to a fragment of Coq code in an external file [40]. Org mode supports tangling, can preserve the original prose in comments inside tangled files, and even supports a form of untangling, using special “link” comments to connect sections of the tangled code to the source blocks they originated from. Unfortunately, unlike in Alectryon, the tangling and detangling process are lossy: tangling does not preserve the full structure of the prose, and detangling only mirror modifications

made to the code, not to the prose included in comments. Of course, to preserve interactivity while sticking with the usual literate programming paradigm, we could enhance Coq’s compiler and all existing Coq UIs to teach them to ignore the prose parts of a reST, Org, or LaTeX document and compile only the code contained within it. This is essentially the approach taken by Literate Haskell and Literate Agda: both compilers include preprocessors which tangle Literate programs into regular code prior to compiling them, and IDEs are extended to recognize the special format of literate Haskell and Agda files. Unfortunately, this comes at significant costs in complexity: enhancing Proof General (an Emacs mode for Coq programming [1]) to ignore reST prose would be a non-trivial endeavor, for example. Instead, Alectryon’s support for bidirectional translation between Coq and reST allows authors to benefit from the full flexibility and convenience of specialized reST and Coq IDEs, with no additional implementation effort in the compiler or the UIs.

Visualizations and example-centric programming The early applicative language Poplar [35] (1980) encouraged authors to annotate source code with automatically checked comments recording of the result of running each subexpression of a program on a well-chosen example to help readers grasp the meaning of the program. These ideas have persisted in many languages and environments, and led to advanced execution and proof visualizations [11, 15, 51] as well as UIs able to automatically generate and display these annotations, in a style called *example-centric programming* [14]. Alectryon’s way of showing goals and messages inline with corresponding proof scripts inherits from this tradition, though without difficulty of picking representative inputs since each proof script runs on a specific proof.

Calculational, structured, and declarative proofs The tactic-based proof style pioneered by LCF [23] is sometimes called *imperative* or *procedural*, in contrast to the *declarative* style introduced by Mizar [53] and later implemented in proof assistants like HOL88 [25], Isabelle/Isar [57], TLA+ [31], HOL Light [58], and even in program verifiers like Dafny [32].

Declarative proofs tend to more closely mirror pen-and-paper proofs, and are generally more readable in isolation than plain imperative proof scripts. They derive from *structured calculational* proofs [2, 13, 24, 56], in which calculations (sequences of propositions chained by logical connectors, each annotated with a succinct justification of the corresponding step, as in $A = \{\text{because } X\} B \Rightarrow \{\text{because } Y\} C$) are structured through logical cuts and typographical means such as indentation and explicit nesting marks.

A declarative proof language was added to Coq in version 8.1 [9] (2006), but it saw only limited use and was removed in version 8.7 (2017). Coq 8.4 instead introduced structure to imperative proof scripts using bullets and braces, giving scripts most of the *structured* part of *structured calculational*

proofs. While it is possible to emulate the *calculational* part in imperative proofs using tactics to re-state the current goal and hypotheses after each step in a chain of rewrites, this “checked comments” style is not particularly common in Coq (rich tooling existed in Isabelle to support this style of proofs before the introduction of Isar and the switch to declarative style [43]). Aletryon’s automatic annotations, combined with careful structuring of proof scripts and judicious use of logical cuts (`assert ... by ...` or SSReflect’s `have`), offer the readability benefits of declarative proofs without the burden of adding output annotations.

6 Conclusion

Authors hoping to discuss proofs and programming techniques are stuck between prose-first formats that make it hard for users to explore the code and code-first formats that require authors to manually annotate proof scripts with copy-pasted snapshots of the prover’s output and write whole books out of source code comments. It does not have to be that way.

Aletryon embodies a new approach to the development and dissemination of literate proof scripts, which we hope will enable authors to share interesting proofs and produce textbooks, tutorials, blogs, and discussions more smoothly. Through examples, discussions, and benchmarks, we have shown that Aletryon offers robust and flexible tools to produce automatically annotated proof scripts, in the form of interactive webpages; that its output format offers a stable and self-contained archival medium for mechanized proofs; that it integrates seamlessly into existing documentation systems; that it scales to large documents, like Coq’s reference manual; and that its support for bidirectional tangling and untangling enables authors to offer an excellent experience to their readers without compromising on their own prose-writing experience. Lastly, we have provided a detailed description of the system, and demonstrated its flexibility and extensibility by enhancing it with a Coqdoc-compatibility mode.

Aletryon is distributed under a permissive (MIT/Expat) license at <https://github.com/cpitclaudel/aletryon/>. The artifact accompanying this paper is available at <https://zenodo.org/record/4033626>.

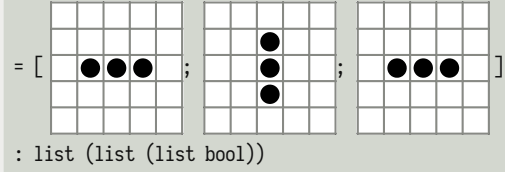
Acknowledgments

Many thanks to the reviewers for their insightful comments and suggestions, and to the many readers who provided valuable feedback on this paper, including (in alphabetical order) Thomas Bourgeat, Arthur Charguéraud, Paolo Giarrusso, Jasper Haag, Olivier Hercend, Shachar Itzhaky, Benoît Pit-Claudel, Valentin Robert, Reyus Sakakibara, Alan Schmitt, Anton Trunov, and Théo Zimmerman.

A Advanced Rendering

```
Definition blinker := [[0;0;0;0;0;];
                      [0;0;0;0;0;];
                      [0;1;1;1;0;];
                      [0;0;0;0;0;];
                      [0;0;0;0;0;]].
```

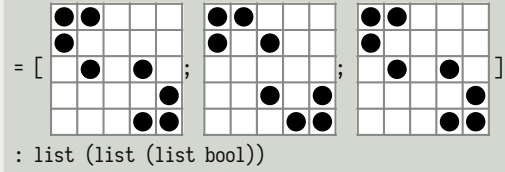
```
Compute take 3 (iter conway_life blinker). =
```



```
= [
  [ [0;0;0;0;0;];
    [0;0;0;0;0;];
    [0;1;1;1;0;];
    [0;0;0;0;0;];
    [0;0;0;0;0;] ];
  [ [0;0;0;0;0;];
    [0;1;1;1;0;];
    [0;0;0;0;0;];
    [0;0;0;0;0;];
    [0;0;0;0;0;] ];
  [ [0;0;0;0;0;];
    [0;0;0;0;0;];
    [0;1;1;1;0;];
    [0;0;0;0;0;];
    [0;0;0;0;0;] ] ]
: list (list (list bool))
```

```
Definition bipole := [[1;1;0;0;0;];
                     [1;0;0;0;0;];
                     [0;1;0;1;0;];
                     [0;0;0;0;1;];
                     [0;0;0;1;1;]].
```

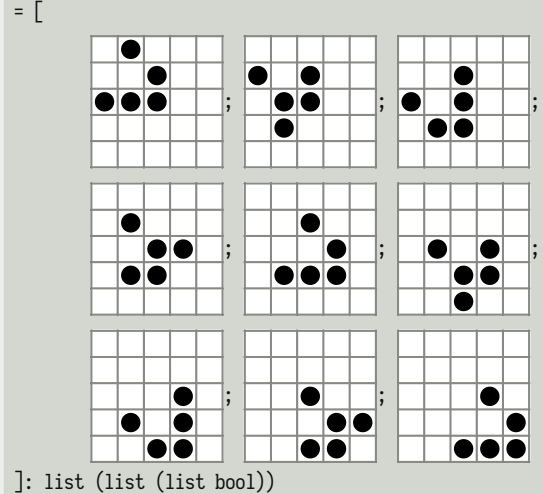
```
Compute take 3 (iter conway_life bipole). =
```




```
= [
  [ [1;1;0;0;0;];
    [1;0;0;0;0;];
    [0;1;0;1;0;];
    [0;0;0;0;1;];
    [0;0;0;1;1;] ];
  [ [1;1;0;0;0;];
    [1;0;0;0;0;];
    [0;1;0;1;0;];
    [0;0;0;0;1;];
    [0;0;0;1;1;] ];
  [ [1;1;0;0;0;];
    [1;0;0;0;0;];
    [0;1;0;1;0;];
    [0;0;0;0;1;];
    [0;0;0;1;1;] ] ]
: list (list (list bool))
```

```
Definition glider := [[0;1;0;0;0;];
                     [0;0;1;0;0;];
                     [1;1;1;0;0;];
                     [0;0;0;0;0;];
                     [0;0;0;0;0;]].
```

```
Compute take 9 (iter conway_life glider). =
```



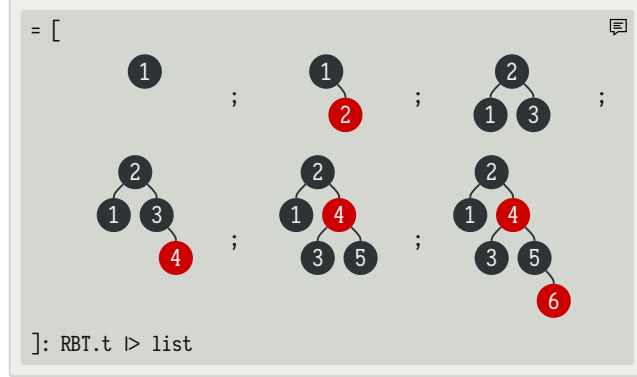
```
= [
  [ [0;1;0;0;0;];
    [0;0;1;0;0;];
    [1;1;1;0;0;];
    [0;0;0;0;0;];
    [0;0;0;0;0;] ];
  [ [0;1;0;0;0;];
    [0;0;1;0;0;];
    [1;1;1;0;0;];
    [0;0;0;0;0;];
    [0;0;0;0;0;] ];
  [ [0;1;0;0;0;];
    [0;0;1;0;0;];
    [1;1;1;0;0;];
    [0;0;0;0;0;];
    [0;0;0;0;0;] ];
  [ [0;1;0;0;0;];
    [0;0;1;0;0;];
    [1;1;1;0;0;];
    [0;0;0;0;0;];
    [0;0;0;0;0;] ];
  [ [0;1;0;0;0;];
    [0;0;1;0;0;];
    [1;1;1;0;0;];
    [0;0;0;0;0;];
    [0;0;0;0;0;] ];
  [ [0;1;0;0;0;];
    [0;0;1;0;0;];
    [1;1;1;0;0;];
    [0;0;0;0;0;];
    [0;0;0;0;0;] ];
  [ [0;1;0;0;0;];
    [0;0;1;0;0;];
    [1;1;1;0;0;];
    [0;0;0;0;0;];
    [0;0;0;0;0;] ];
  [ [0;1;0;0;0;];
    [0;0;1;0;0;];
    [1;1;1;0;0;];
    [0;0;0;0;0;];
    [0;0;0;0;0;] ];
  [ [0;1;0;0;0;];
    [0;0;1;0;0;];
    [1;1;1;0;0;];
    [0;0;0;0;0;];
    [0;0;0;0;0;] ] ]
]: list (list (list bool))
```

Figure 4. Images. Aletryon recordings can be post-processed to customize the rendering of Coq’s output. Here, we replaced binary matrices by SVG graphics to help readers visualize the evolution of various configurations of Conway’s *Game of Life*. 

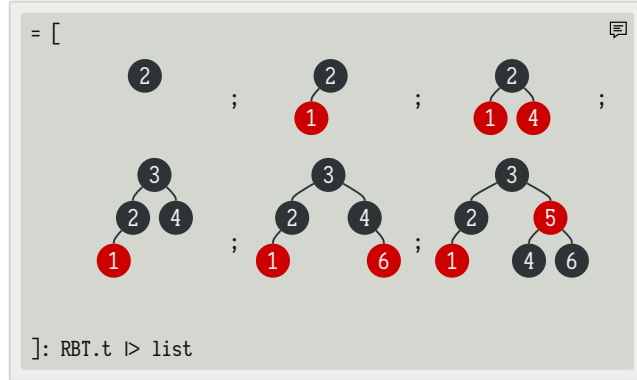
```
Module RBT := MSets.MSetRBT.Make Nat.as_OT.
```

```
Definition build_trees (leaves: list nat) :=
  List.fold_left (fun trs n =>
    RBT.add n (hd RBT.empty trs) :: trs)
    leaves [] > List.rev.
```

```
Compute build_trees [1;2;3;4;5;6]. -
```



```
Compute build_trees [2;1;4;3;6;5]. -
```



```
Compute build_trees [6;5;4;1;2;3]. -
```

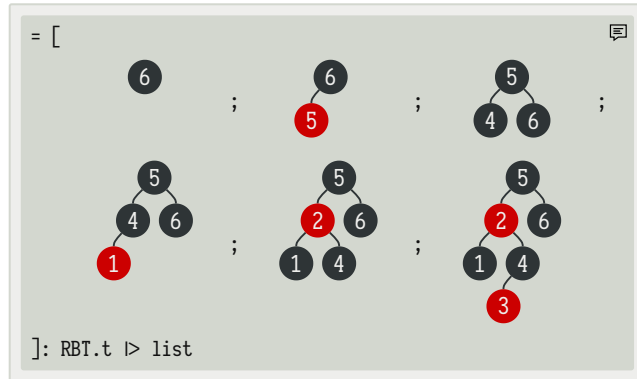


Figure 5. Graphs. Introductory Coq exercises often revolve around data-structure proofs. Here, to help hypothetical students understand rebalancing operations on red-black trees, we rendered three examples of incremental tree construction using the d3.js library [5].

```
Lemma Gauss: ∀ n,
  2 * (nsum n (fun i => i)) =
  n * (n + 1). -
```

$$\forall n : \mathbb{N}. 2 \times \sum_{i=0}^n i = n \times (n + 1)$$

```
induction n; cbn [nsum]. -
```

$$2 \times 0 = 0 \times (0 + 1)$$

$$2 \times (S n + \sum_{i=0}^n i) = S n \times (S n + 1)$$

```
- (* n ← 0 *) -
```

$$2 \times 0 = 0 \times (0 + 1)$$

```
reflexivity.
```

```
- (* n ← S _ *) -
```

$$n : \mathbb{N} \quad \text{IHn} : 2 \times \sum_{i=0}^n i = n \times (n + 1)$$

$$2 \times (S n + \sum_{i=0}^n i) = S n \times (S n + 1)$$

```
rewrite Mult.mult_plus_distr_l. -
```

$$2 \times S n + 2 \times \sum_{i=0}^n i = S n \times (S n + 1)$$

```
rewrite IHn. -
```

$$2 \times S n + n \times (n + 1) = S n \times (S n + 1)$$

```
ring.
```

```
Qed.
```

Figure 6. Mathematics. To improve the readability of this simple proof, we used a custom filter that translates Coq's output to \LaTeX . The resulting formulae are rendered with MathJax, a JavaScript display engine for mathematics. The same script without \LaTeX is partly shown in section 2.

Lemma `Qle_pairwise` :

$\forall a\ b\ c\ d,$
 $0 < a \wedge 0 < b \wedge 0 < c \wedge 0 < d \rightarrow$
 $(a + c)/(b + d) \leq a/b + c/d. \text{ --}$

$$\forall a\ b\ c\ d : \mathbb{Q}. 0 < a \wedge 0 < b \wedge 0 < c \wedge 0 < d \Rightarrow \frac{a+c}{b+d} \leq \frac{a}{b} + \frac{c}{d}$$

Proof with `Qeauto`. --

`intros a b c d H.` --

$$a, b, c, d : \mathbb{Q} \quad H : 0 < a \wedge 0 < b \wedge 0 < c \wedge 0 < d$$

$$\frac{a+c}{b+d} \leq \frac{a}{b} + \frac{c}{d}$$

`field_simplify...` --

$$\frac{a+c}{b+d} \leq \frac{a \times d + c \times b}{b \times d}$$

`rewrite <- (Qmult_le_l (b + d)), Qmult_div_r, Qmult_Qdiv_fact...` --

$$a + c \leq \frac{(b + d) \times (a \times d + c \times b)}{b \times d}$$

`rewrite <- (Qmult_le_l (b * d)), Qmult_div_r...` --

$$(b \times d) \times (a + c) \leq (b + d) \times (a \times d + c \times b)$$

`field_simplify.` --

$$(b \times d) \times a + (b \times d) \times c \leq b^2 \times c + (b \times d) \times a + (b \times d) \times c + d^2 \times a$$

`rewrite <- (Qminus_le_l (b * d * a)); ring_simplify.` --

$$(b \times d) \times c \leq b^2 \times c + (b \times d) \times c + d^2 \times a$$

`rewrite <- (Qminus_le_l (b * d * c)); ring_simplify.` --

$$0 \leq b^2 \times c + d^2 \times a$$

`Qeauto using Qsqr_0.`

Qed.

Figure 7. Mathematics. Here, too, we leveraged a custom filter to improve the readability of an arithmetic proof.



```

Definition udiv := {|
  c_source :=
    "#include ""stdint.h""

    typedef struct {
      uint32_t quot; ...";
  riscv_elf :=
    [x7f; x45; x4c; x46; x01; x01; x01; x00; x00; x00; x00; x00; x00; x00; x00;
    x02; x00; xf3; x00; x01; x00; x00; x00; x54; x00; x01; x00; x34; x00; x00;
    x0c; x02; x00; x00; x00; x00; x00; x00; x34; x00; x20; x00; x01; x00; x28;
    x06; x00; x05; x00; x01; x00; x00; x00; x00; x00; x00; x00; x00; x01; ... ] |}.

```

Compute udiv.(c_source). =

```

=
#include "stdint.h"

typedef struct {
  uint32_t quot;
  uint32_t rem;
} udiv_t;

udiv_t udiv(uint32_t num, uint32_t denom) {
  uint32_t q = 0;
  while (num ≥ denom) {
    num -= denom; ++q;
  }
  return (udiv_t) { .rem = num, .quot = q };
}

: string

```


Compute udiv.(riscv_elf). =

```

=
00010054 <udiv>:
10054: 00 05 07 93      mv      a5,a0
10058: ff 01 01 13      addi    sp,sp,-16
1005c: 00 00 05 13      li      a0,0
10060: 00 b7 f8 63      bgeu    a5,a1,10070 <udiv+0x1c>
10064: 00 07 85 93      mv      a1,a5
10068: 01 01 01 13      addi    sp,sp,16
1006c: 00 00 80 67      ret
10070: 40 b7 87 b3      sub     a5,a5,a1
10074: 00 15 05 13      addi    a0,a0,1
10078: fe 9f f0 6f      j       10060 <udiv+0xc>

: list Byte.byte

```

Figure 8. Pretty-printing. In this example, we used a custom transform to improve the readability of the inputs and outputs of a hypothetical C compiler. When invoked, the transform syntax-highlights the value of `udiv.(c_source)` as C code using Pygments (this part starts from a string, but it would also be reasonable to start from an abstract syntax tree). It then reconstructs a byte stream from the value of `udiv.(riscv_elf)`, disassembles it by running `objdump -d` in a subprocess, and pretty-prints the resulting assembly listing. Running this transformation automatically at compile time guarantees that the figures remain in sync with the Coq code. 

B Editor Support for Tangling and Untangling

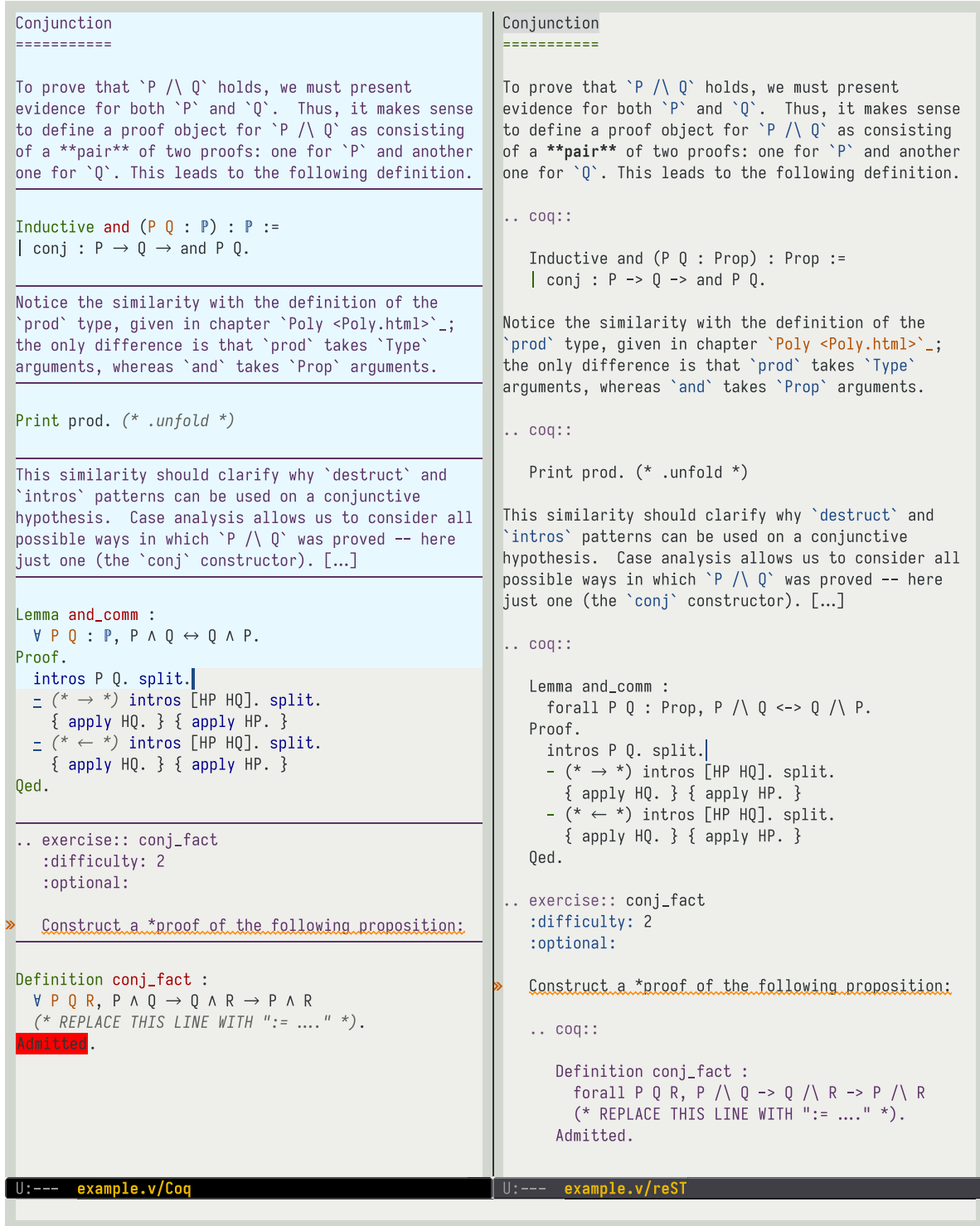


Figure 9. Screenshot of Emacs running Proof General and rst-mode side-by-side. On the left is the Coq view of an edited excerpt of *Software Foundations*; on the right is the reST view of the same excerpt. The conversion is transparent, so editing either view updates the same .v file on disk. Notice the highlight on both sides indicating a reST syntax error. Literate comment markers (“(*)” and “|*|”) are rendered as solid lines.

C Plain-HTML Rendering

Lemma rev_rev {A} (l: list A) : List.rev (List.rev l) = l.

Proof.

induction l; cbn.

A:Type

nil = nil

A:Type

a:A

l:list A

IHl:rev (rev l) = l

rev (rev l ++ a :: nil) = a :: l

- (* l ← [] *)

A:Type

nil = nil

reflexivity.

- (* l ← _ :: _ *)

A:Type

a:A

l:list A

IHl:rev (rev l) = l

rev (rev l ++ a :: nil) = a :: l

rewrite rev_app_distr.

rev (a :: nil) ++ rev (rev l) = a :: l

rewrite IHl.

rev (a :: nil) ++ l = a :: l

cbn.

a :: l = a :: l

reflexivity.

Qed.

Figure 10. Plain-HTML rendering of a simple proof, with all CSS stylesheets disabled. Alectryon uses a careful selection of standard HTML elements to ensure intelligible results in RSS feed readers without resorting to external styling.

References

- [1] David Aspinall. 2000. Proof General: A Generic Tool for Proof Development. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000*, Susanne Graf and Michael Schwartzbach (Eds.). Lecture Notes in Computer Science, Vol. 1785. Springer Berlin Heidelberg, 38–43. https://doi.org/10.1007/3-540-46419-0_3
- [2] Ralph Back, Jim Grundy, and Joakim von Wright. 1997. Structured calculational proof. *Formal Aspects of Computing* 9, 5 (01 Sep 1997), 469–483. <https://doi.org/10.1007/BF01211456>
- [3] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development*. Springer-Verlag Berlin Heidelberg. <https://doi.org/10.1007/978-3-662-07964-5>
- [4] Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011*, Elisardo Antelo, David Hough, and Paolo Ienne (Eds.). IEEE Computer Society, 243–252. <https://doi.org/10.1109/ARITH.2011.40>
- [5] Mike Bostock, Jason Davies, Jeffrey Heer, Vadim Ogievetsky, and community. 2011. D3.js: Data-Driven Documents. (Feb. 2011). <https://d3js.org/>
- [6] Edwin C. Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (Sep 2013), 552–593. <https://doi.org/10.1017/S095679681300018X>
- [7] Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press.
- [8] Adam Chlipala. 2015. *Formal Reasoning About Programs*. Electronic textbook. <http://adam.chlipala.net/frap/>.
- [9] Pierre Corbineau. 2008. A Declarative Language for the Coq Proof Assistant. In *Types for Proofs and Programs*, Marino Miculan, Ivan Scagnetto, and Furio Honsell (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 69–84. https://doi.org/10.1007/978-3-540-68103-8_5
- [10] Chris Coyier. 2011. Stuff you can do with the “Checkbox Hack”. (2011). <https://css-tricks.com/the-checkbox-hack/>.
- [11] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, , and Man-dayam Srivas. 1995. A Tutorial Introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*. Boca Raton, Florida. <http://www.csl.sri.com/papers/wift-tutorial/>
- [12] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Proceedings of the 25th International Conference on Automated Deduction (CADE-25)*. Berlin, Germany, August 2015. Springer International Publishing, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26
- [13] Edsger W. Dijkstra and Carel S. Scholten. 1990. *Predicate Calculus and Program Semantics*. Springer New York. <https://doi.org/10.1007/978-1-4612-3228-5>
- [14] Jonathan Edwards. 2004. Example Centric Programming. *ACM SIGPLAN Notices* 39, 12 (Dec. 2004), 84–91. <https://doi.org/10.1145/1052883.1052894>
- [15] Marc Eisenstadt and Mike Brayshaw. 1990. A fine-grained account of Prolog execution for teaching and debugging. *Instructional Science* 19, 4 (01 Jul 1990), 407–436. <https://doi.org/10.1007/BF00116447>
- [16] Gorkem Ercan. 2016. A common interface for building developer tools. (June 2016). <https://developers.redhat.com/blog/2016/06/27/a-common-interface-for-building-developer-tools/>.
- [17] Martin Fowler. 2008. Projectional editing. (2008). <https://martinfowler.com/bliki/ProjectionalEditing.html>.
- [18] Mario Frank. 2020. The Coq Proof Script Visualiser (coq-psv). In *11th Coq Workshop, colocated with IJCAR '20*.
- [19] Emilio Jesús Gallego Arias. 2016. SerAPI: Machine-Friendly, Data-Centric Serialization for COQ. (Oct. 2016). <https://hal-mines-paristech.archives-ouvertes.fr/hal-01384408>.
- [20] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. 2016. js-Coq: Towards Hybrid Theorem Proving Interfaces. In *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers, UITP 2016, Coimbra, Portugal, 2nd July 2016 (EPTCS, Vol. 239)*, Serge Autexier and Pedro Quaresma (Eds.). 15–27. <https://doi.org/10.4204/EPTCS.239.2>
- [21] David Goodger. 2016. Docutils Project Documentation Overview. (Jan. 2016). <https://docutils.sourceforge.io/docs/index.html>.
- [22] David Goodger. 2020. reStructuredText Markup Specification. (July 2020). <https://docutils.sourceforge.io/docs/ref/rst/restructuredtext.html>.
- [23] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. 1979. Edinburgh LCF. *Lecture Notes in Computer Science* 78 (1979). <https://doi.org/10.1007/3-540-09724-4>
- [24] Jim Grundy and Thomas Långbacka. 1997. Recording HOL Proofs in a Structured Browsable Format. In *Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology (AMAST '97), Sydney, Australia, December 13-17, 1997*. 567–571. <https://doi.org/10.1007/BFb0000500>
- [25] John Harrison. 1996. A mizar mode for HOL. *Theorem Proving in Higher Order Logics* (1996), 203–220. <https://doi.org/10.1007/bfb0105406>
- [26] Maxim Hendriks, Cezary Kaliszyk, Femke Van Raamsdonk, and Freek Wiedijk. 2010. Teaching logic using a state-of-the-art proof assistant. *Acta Didactica Napocensia* 3, 2 (2010), 35–48.
- [27] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. 2016. Jupyter Notebooks — A publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, Fernando Loizides and Birgit Schmidt (Eds.). IOS Press, 87–90. <https://eprints.soton.ac.uk/403913/>
- [28] D. E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (Feb 1984), 97–111. <https://doi.org/10.1093/comjnl/27.2.97>
- [29] Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (01 1984), 97–111. <https://doi.org/10.1093/comjnl/27.2.97>
- [30] Donald E. Knuth. 1992. *Literate Programming*. Center for the Study of Language and Information, USA.
- [31] Leslie Lamport. 2012. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications* 11, 1 (Mar 2012), 43–63. <https://doi.org/10.1007/s11784-012-0071-6>
- [32] K. Rustan M. Leino and Nadia Polikarpova. 2014. Verified Calculations. In *Verified Software: Theories, Tools, Experiments*, Ernie Cohen and Andrey Rybalchenko (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 170–190.
- [33] Eugene Loy. 2018. Jupyter kernel for Coq. (Dec. 2018). https://github.com/EugeneLoy/coq_jupyter.
- [34] Assia Mahboubi and Enrico Tassi. 2020. *Mathematical Components*. Electronic textbook. <https://math-comp.github.io/mcb/>.
- [35] James H. Morris, Eric Schmidt, and Philip Wadler. 1980. Experience with an Applicative String Processing Language. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Las Vegas, Nevada) (POPL '80). Association for Computing Machinery, New York, NY, USA, 32–46. <https://doi.org/10.1145/567446.567450>
- [36] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, Vilhelm Sjöberg, and Brent Yorgey. 2018. *Logical Foundations*. Electronic textbook. <https://softwarefoundations.cis.upenn.edu/> Version 5.5. <https://softwarefoundations.cis.upenn.edu/>
- [37] Clément Pit-Claudel. 2016. coqrs, An experiment in modernizing Coq’s manual. (May 2016). <https://github.com/cpitclaudel/coq-rst/>.
- [38] Clément Pit-Claudel. 2017. Editor support for F*. (April 2017). https://github.com/FStarLang/FStar/wiki/Editor-support-for-F*.

- [39] Valentin Robert. 2018. *Front-end tooling for building and maintaining dependently-typed functional programs*. Ph.D. Dissertation. UC San Diego. <https://escholarship.org/uc/item/9q3490fh>
- [40] Alan Schmitt. 2016. Documenting Coq Code using Org-mode. (June 2016). http://alan.petitepomme.net/tips/documenting_coq.html.
- [41] Alan Schmitt. 2016. Executing Coq Code using Org-mode. (June 2016). http://alan.petitepomme.net/tips/executing_coq.html.
- [42] Ilya Sergey. 2014. *Programs and Proofs: Mechanizing Mathematics with Dependent Types*. Lecture notes with exercises. <https://ilyasergey.net/pnp/>.
- [43] Martin Simons. 1997. Proof presentation for Isabelle. In *Theorem Proving in Higher Order Logics*, Elsa L. Gunter and Amy Felty (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 259–274. <https://doi.org/10.1007/bfb0028399>
- [44] Charles Simonyi, Magnus Christerson, and Shane Clifford. 2006. Intentional Software. *SIGPLAN Not.* 41, 10 (Oct. 2006), 451–464. <https://doi.org/10.1145/1167515.1167511>
- [45] Lisa M. C. Smith and Mansur H. Samadzadeh. 1991. An annotated bibliography of literate programming. *ACM SIGPLAN Notices* 26, 1 (Jan 1991), 14–20. <https://doi.org/10.1145/122203.122204>
- [46] Carst Tankink, Herman Geuvers, James McKinna, and Freek Wiedijk. 2010. Proviola: A Tool for Proof Re-animation. In *Intelligent Computer Mathematics, 10th International Conference, Paris, France, July 5-10, 2010. Proceedings*. 440–454. https://doi.org/10.1007/978-3-642-14128-7_37
- [47] Enrico Tassi. 2018. Roundtrip. (2018). <https://github.com/math-comp/mcb/tree/master/roundtrip>.
- [48] The Coq Development Team. 2002. *The Coq Proof Assistant: Reference Manual, version 7.2*. Technical Report RT-0255. INRIA. 290 pages. <https://hal.inria.fr/inria-00069919>
- [49] The Coq Development Team. 2018. *The Coq Proof Assistant: Reference Manual, version 8.8.0*. <https://doi.org/10.5281/zenodo.1219885>
- [50] The Coq Development Team. 2020. *The Coq Proof Assistant, version 8.11.0*. <https://doi.org/10.5281/zenodo.3744225>
- [51] Hendrik Tews. 2011. Proof tree visualization for Proof General. (April 2011). <https://askra.de/software/prooftree/>.
- [52] Read the Docs. 2013. A markdown parser for docutils. (Aug. 2013). <https://github.com/readthedocs/recommonmark/>.
- [53] Andrzej Trybulec and Howard A. Blair. 1985. Computer Assisted Reasoning with MIZAR. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence. Los Angeles, CA, USA, August 1985*, Aravind K. Joshi (Ed.). Morgan Kaufmann, 26–28. <https://ijcai.org/Proceedings/85-1/Papers/006.pdf>
- [54] Josef Urban. 2005. XML-izing Mizar: Making Semantic Processing and Presentation of MML Easy. In *Proceedings of the 4th International Conference on Mathematical Knowledge Management (MKM '05), Bremen, Germany, July 15-17, 2005*. 346–360. https://doi.org/10.1007/11618027_23
- [55] Josef Urban and Grzegorz Bancerek. 2007. Presenting and Explaining Mizar. *Electronic Notes in Theoretical Computer Science* 174, 2 (May 2007), 63–74. <https://doi.org/10.1016/j.entcs.2006.09.022>
- [56] Antonetta J. M. van Gasteren. 1990. *On the Shape of Mathematical Arguments*. Springer Berlin Heidelberg. <https://doi.org/10.1007/bfb0020908>
- [57] Markus Wenzel. 1999. Isar — A Generic Interpretative Approach to Readable Formal Proof Documents. In *Theorem Proving in Higher Order Logics*, Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 167–183. https://doi.org/10.1007/3-540-48256-3_12
- [58] Freek Wiedijk. 2001. Mizar Light for HOL Light. In *Theorem Proving in Higher Order Logics*, Richard J. Boulton and Paul B. Jackson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 378–393. https://doi.org/10.1007/3-540-44755-5_26
- [59] Jay Wilcox. 2014. Literate Coq Blogging. (Dec. 2014). <https://github.com/wilcoxjay/coq-lit>.