Integrating graphical proofs in Coq

Pablo Donato
pablo.donato@polytechnique.edu
LIX, Institut Polytechnique de Paris
Palaiseau, France

Benjamin Werner
benjamin.werner@inria.fr
LIX, Institut Polytechnique de Paris
Palaiseau, France

Kaustuv Chaudhuri
kaustuv@chaudhuri.info
Inria & LIX, Institut Polytechnique de Paris
Palaiseau, France

Abstract

We present Actema, a prototype of graphical user interface where formal proofs can be built through gestural actions, and its integration within Coq through the coq-actema plugin. The latter relies on a client-server architecture, where the plugin requests proof steps from the interface, and then compiles them to appropriate tactic calls. To avoid cluttering the proof script with these calls, graphical proofs are stored separately on disk, and recompiled upon execution of the script. This should also allow to "replay" proofs visually in the interface, which is vital for readability and maintenance. The plugin is usable with any Coq IDE, and its design adapted to any goal-directed interactive theorem prover supporting intuitionistic first-order logic.

Keywords: graphical user interfaces, coq plugin, client-server protocol

1 Introduction

Interactive Theorem Provers allow the user to incrementally construct formal proofs through an interaction loop. One progresses through a sequence of *states* corresponding to incomplete proofs. Each of these states is itself described by a finite set of *goals* and the proof is completed once there are no goals left. From the user's point of view, a goal appears as a sequent, in the sense coined by Gentzen. In the case of intuitionistic logic that is:

- One particular proposition A which is *to be proved*, we designate it as the goal's *conclusion*,
- a set of propositions Γ corresponding to *hypotheses*.

The user performs *actions* on one such goal at a time, and the actions transform the goal, or rather replace the goal by a new set of goals. When this set is empty, the goal is said to be solved.

In the dominant paradigm, these commands are provided by the user in text form; since Robin Milner and LCF [8] they are called *tactics*.

The present work is a form of continuation of the *Proof-by-Pointing* (PbP) effort, initiated in the 1990s by Gilles Kahn, Yves Bertot, Laurent Théry and their group [3]. Both works share a main idea which is to replace the textual tactic commands by *physical actions* performed by the user on a graphical user interface. In both cases, the *items* the user performs actions on are the current goal's conclusion and hypotheses. What is new in our work is that we allow not only to *click* on subterms of these items, but also to *drag-and-drop* (DnD) one subterm onto another. This enriches the language of actions in, we argue, an intuitive way.

We have implemented a small web-based prototype called Actema to demonstrate and explore this approach, that should be usable directly in any web browser[5]. It is based on a custom proof engine for intuitionistic multi-sorted first-order logic (hereafter designated as iFOL), which for ease and speed of prototyping does not certify proofs in a trusted kernel. While the choice of iFOL as a logical framework simplifies the semantics of proof actions and makes our approach applicable to most interactive theorem provers, it is not well suited to the complete formalization of rich theories. Hence the main focus of the interface has been put on proposing graphical ways to perform logical inferences, and there is currently no way to handle databases of definitions, lemmas and proofs.

To address the previous limitations, and thus enable a confrontation of our paradigm to real mathematical developments, we are building coq-actema, a Coq plugin that directly connects Actema to a running Coq instance. The idea is that Actema should act as an enhanced graphical, interactive proof view that integrates in the usual text-based workflow of proof scripts. An illustration of this is shown in the screenshot of figure 1.

In the following, we discuss briefly some motivations that lead to the development of coq-actema, and describe its overall design and architecture. For a review of the interface of Actema, we refer to previous work [6] and online documentation [5].

2 Motivations

Usually, integrated development environments for Coq live in an independent process, and exchange data with Coq through a high-level communication protocol: either coqtop's command line interface, Coq's default XML protocol, or its improved superset SerAPI [7]. In particular, SerAPI emerged from the development of jsCoq [1], an IDE that runs entirely in web browsers by embedding a version of Coq compiled with js_of_ocaml [9]. Since Actema is also web-based and

CoqPL 2023, January 21, 2023, Boston, MA, USA
.

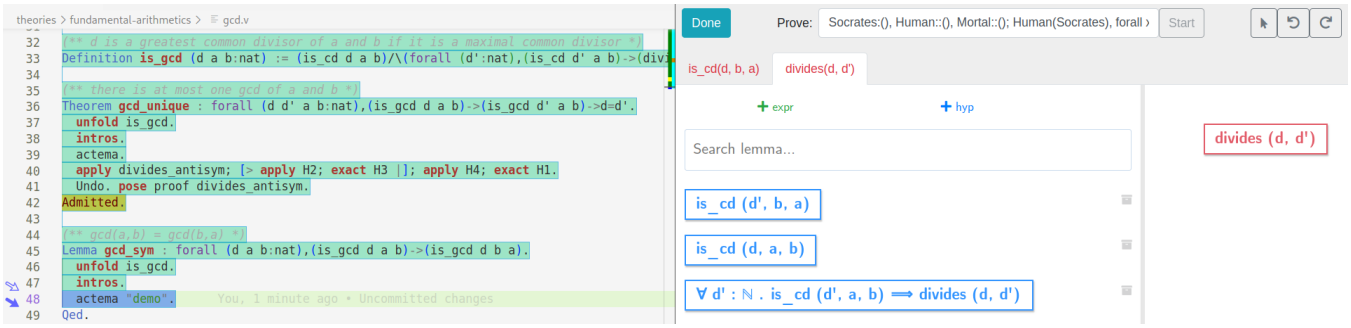

Figure 1. Developing proofs graphically within a textual setting. On the left, the usual interactive view of the proof script, in the VsCoq IDE. On the right, the graphical proof view of Actema. Blue items correspond to hypotheses, the red item corresponds to the conclusion, and tabs allow to switch focus between subgoals.

uses js_of_ocaml, our first idea was essentially to fork jsCoq and replace its interface by that of Actema. However as noted by E. J. G. Arias, the SerAPI protocol — and in fact all the other protocols turn out to be too high-level for our purpose. Typically we need to (partially) translate Coq goals into iFOL goals, which can be done much more easily with a direct access to Coq's low-level API for manipulating kernel terms. We also heavily rely on unification to interactively suggest valid actions on subterms of the goal, and none of the protocols implement unification queries. [1].

Now, remember that Actema is not meant as a full-fledged IDE that can manage the state of the proof script and vernacular commands, but only as an enhanced proof view for manipulating already-parsed logical terms. Considering this and all the above, the solution of a Coq plugin made a lot more sense, with the important benefit of ensuring compatibility with all existing IDEs. This would also entail easier adoption of Actema into existing Coq developments and workflows.

In this setting, Actema still runs in a process independent from Coq. It is already made of two layers: a JavaScript frontend which handles the graphical layout and interaction with the user, and an OCaml backend for the proof engine. We add to this picture a third layer, namely a HTTP server that handles communication of requests from, and responses to the Coq plugin. Of course the server runs in a process of its own, to avoid any delay in the interface. Then we bundle everything in an Electron application, so that the interface can easily be run locally on most operating systems.

[1]Note that currently, we still perform unification in the backend of Actema. But having access to Coq's unification shall prove useful in the future to support the full logic of Coq, while limiting Actema's role to that of a graphical frontend.

3 Plugin

On the plugin side, there remains the question of how to integrate concretely the graphical proofs of Actema into textual proof scripts. In fact this begs for a deeper question: how do we represent statically a sequence of graphical actions, let alone a single action? For a machine representation, we can just dump the user inputs that triggered the action, typically the paths that lead to selected subterms in a drag-and-drop action. But finding a human-readable representation that an average user can quickly manipulate and reason about is a lot more delicate. The most direct way may be to abandon text altogether, and just replay the action on the interface through a graphical animation. This is an intrisically temporal and dynamic representation, akin to a mathematician unfolding her demonstration on the blackboard.

For now we dispense with such considerations, and choose to represent a full sequence of actions as a single call to an `actema` tactic. When run, it executes essentially the following interaction protocol:

1. if the current goal $\mathcal{G}_0$ does not have any saved action sequence associated to it, the plugin sends an `action` request to Actema with $[\![\mathcal{G}_0]\!]$ as its body, where $[\![\cdot]\!]$ is a translation function from Coq goals to the iFOL goals of Actema;
2. then the user can either:
   - perform a proof action on $[\![\mathcal{G}_0]\!]$ in Actema: this returns an `action` response withholding the machine representation $\mathcal{A}_0$ of the action. The plugin then compiles $\mathcal{A}_0$ into a tactic $[\![\mathcal{A}_0]\!]$, runs it on $\mathcal{G}_0$, and sends the new set of goals in another `action` request. This might give back a new action $\mathcal{A}_1$, but this time with an additional index n_1 to indicate the goal/tab being focused in Actema, and thus on which goal $[\![\mathcal{A}_1]\!]$ must be applied;
   - click on the Done button (top-left corner of fig. 1): this returns a `done` response, which ends the interaction loop. The sequence $(\mathcal{A}_i, n_i)$ of actions is saved

on disk in a file $h(\llbracket \mathcal{G}_0 \rrbracket)$, where h is a hash function on Actema goals.

Then the next time the `actema` tactic is called on goal $\mathcal{G}_0$, the plugin will recompile the saved $(\mathcal{A}_i, n_i)$ into the corresponding Coq tactics, without having to put them explicitly in the proof script. The user can also overwrite the previous sequence with the `actema_force` variant, and store multiple sequences for the same goal by supplying as argument to the actema tactics an arbitrary Coq string, which plays the role of an identifier.

Acknowledgments

We would like to thank E. J. G. Arias for his help and advice on the (now abandoned) option of embedding Coq in the browser as a backend for Actema. We also thank Luc Chabassier for fruitful discussions and ideas concerning the design and implementation of coq-actema.

References

[1] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. 2017. jsCoq: Towards Hybrid Theorem Proving Interfaces. *Electronic Proceedings in Theoretical Computer Science* 239 (jan 2017), 15–27. https://doi.org/10.4204/eptcs.239.2

[2] Edward W. Ayers, Mateja Jamnik, and W. T. Gowers. 2021. A Graphical User Interface Framework for Formal Verification. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:16. https://doi.org/10.4230/LIPIcs.ITP.2021.4

[3] Yves Bertot, Gilles Kahn, and Laurent Théry. 1994. Proof by pointing. In *Theoretical Aspects of Computer Software*, Masami Hagiya and John C. Mitchell (Eds.). Vol. 789. Springer Berlin Heidelberg, here, 141–160. https://doi.org/10.1007/3-540-57887-0_94 Series Title: Lecture Notes in Computer Science.

[4] Kaustuv Chaudhuri. 2021. Subformula Linking for Intuitionistic Logic with Application to Type Theory. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, here, 200–216. https://doi.org/10.1007/978-3-030-79876-5_12

[5] Pablo Donato, Pierre-Yves Strub, and Benjamin Werner. 2022. *The Actema prototype (online version).* https://www.actema.xyz.

[6] Pablo Donato, Pierre-Yves Strub, and Benjamin Werner. 2022. A Drag-and-Drop Proof Tactic. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA) *(CPP 2022)*. Association for Computing Machinery, New York, NY, USA, 197–209. https://doi.org/10.1145/3497775.3503692

[7] Emilio Jesús Gallego Arias. 2016. SerAPI: Machine-Friendly, Data-Centric Serialization for COQ. (Oct. 2016). https://hal-mines-paristech.archives-ouvertes.fr/hal-01384408 working paper or preprint.

[8] Robin Milner. 1984. The use of machines to assist in rigorous proof. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences* 312, 1522 (1984), 411–422. https://doi.org/10.1098/rsta.1984.0067 arXiv:https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.1984.0067

[9] Jérôme Vouillon and Vincent Balat. 2014. From bytecode to JavaScript: the Js_of_ocaml compiler. *Software: Practice and Experience* 44, 8 (Aug. 2014), 951–972. https://doi.org/10.1002/spe.2187