SerAPI: Machine-Friendly, Data-Centric Serialization for COQ

Emilio Jesús Gallego Arias

SerAPI: Machine-Friendly, Data-Centric Serialization for COQ

Technical Report

Emilio Jesús Gallego Arias

MINES ParisTech, PSL Research University, France

Abstract

We present SerAPI, a library and protocol for machine-friendly communication with the COQ proof assistant. SerAPI is implemented using Ocaml's PPX pre-processing technology, and it is specifically targeted to reduce implementation burden for tools such as Integrated Development Environments or code analyzers.

SerAPI tries to address common problems that tools interacting with COQ face, providing a uniform and data-centric way to access term representations, proof state, and an extended protocol for document building.

SerAPI is work in progress but fully functional. It has been adopted by the jsCoq and PeaCoq Integrated Development Environments, and supports running inside a web browser instance. For the near future, we are focused on extending the document protocol and providing advanced display abilities to clients.

Keywords serialization; interactive protocols; user interfaces; program verification; theorem prover implementation

1. Introduction

Historically, the interface to the outside world of the COQ proof assistant has been its *toplevel*, a **R**ead **P**rint **E**val **L**oop. COQ's **REPL** is targeted at human users, which issue proof building commands — to progress towards proving the "current goal" — intermixed with control commands and information requests. The limitations that a command-based model suppose for proof development quickly become apparent, and proof development environments such as Proof General [2] appeared to provide a more convenient *document-based* model on top of the **REPL**.

Nobody would deny the success of this approach, however, interacting with the **REPL** interface brings some challenges and limitations. Interpreting prover output is complicated, especially in the presence of COQ's extensible parsing and printing mechanism. Similarly, information internal to the theorem was not exposed in a systematic way. Queries for such data — often needed by the tools — were performed by custom-purpose commands, often with semi-adhoc syntax and output format. The need to modify the core command language of COQ implied that new functionality was costly to implement, requiring a new release of system.

Until recently, the only alternative to the **REPL** was the use of *plugins*, written in Ocaml. Plugins provided full access to COQ's internals, however they were hard to distribute and restricted to Ocaml-friendly environments.

Fortunately, the situation has recently changed with the introduction of the **S**tate **T**ransactional **M**achine [3] in COQ 8.5 and a new, XML-based, interactive protocol. The new protocol was inspired by related proof IDE efforts [9], and several new developments are based on it, including an experimental port of Proof General [7, 4, 10].

Our particular first contact with COQ's **STM** came when we started the development of jsCoq [5], a port of COQ to the browser platform. jsCoq started as a Ocaml application, then transpiled to JavaScript by the js_of_ocaml [8] tool. The event-based nature of the **STM** API proved to be well suited to the browser environment, allowing the completion of a first prototype. However, we soon realized that the next step for jsCoq — running the prover process inside a web worker thread — would require the use of a protocol. The obvious choice would have been to use the existing XML protocol, however after careful consideration we chose to build our own extension. The main reasons were:

- jsCoq was developed using the **STM** Ocaml's API. However, there is significant impedance with the way the state machine is exposed through the XML protocol. Adapting would have required a complete rework of jsCoq.
- Some of the ambitions jsCoq goals — advanced term display, asynchronous communication — were not well supported by the XML protocol.
- The amount of serialization boilerplate was high. COQ exposes hundredths of datatypes and the XML protocol was designed prior to the introduction of the Ocaml's PPX meta programming facilities. The opportunity of designing a new system based on meta programming seemed worth the effort to us.
- The final factor was the realization that the existence of such a tool could be potentially useful to other developers and users.

This way, SerAPI was born.

2. SerAPI overview

The SerAPI philosophy is to use ML-style datatypes as the schema specification, using PPX to automatically generate a machine-friendly serialization. In our opinion, the cost of maintaining a separate data representation is too high. This has the effect to link SerAPI data representation to Ocaml's API stability, however care is taken in order to ensure that the more advanced representation are opt-in. For example, a client may request to receive all objects pretty printed as strings.

Once we have encoded the required datatypes, the API we want to expose are reified to — usually small — domain specific languages, with their corresponding interpreter serving as the link between the reified version and the actual COQ procedures. The DSL representation is serialized in the same way.

We briefly survey the components of the current distribution.

The Serialization Library The base component of SerAPI is SerLib, the serialization library. SerLib provides an overlay of COQ's modules, extending them with the corresponding serialization functions.

Serialization is performed by the `ppx_sexp_conv` package, with help from `ppx_import`. For example, we declare the `vernac_expr` datatype to be serializable to sexps with:

```
type vernac_expr =
  [%import: Vernacexpr.vernac_expr]
  [@@deriving sexp]
(* will generate: *)
val vernac_expr_of_sexp : Sexp.t → vernac_expr
val sexp_of_vernac_expr : vernac_expr → Sexp.t
```

The current distribution serializes 328 COQ datatypes by including this boilerplate in the corresponding modules. For the few private datatypes of interest — such as `constr` — we provide our own view and translation functions.

The SerAPI Protocol The second component is the SerAPI control protocol, an extension of COQ's STM API which is used to build and execute proof scripts. Given the IDE origins of SerAPI, it was natural to provide IDE support first. The current control protocol is specified by a DSL and its full description goes beyond the scope of this abstract. A peek on the definitions is:

```
type control_cmd =
  | StmAdd     of add_opts  * string
  | StmCancel  of Stateid.t list
  | StmObserve of Stateid.t
  ...
type answer_kind =
  | StmAdded     of Loc.t
  | StmCanceled  of Stateid.t list
  ...
```

Several changes over the standard STM model have been performed, mainly to facilitate the work of the clients and to be more robust w.r.t. race conditions. Events or *feedback* from COQ are forwarded to the client unchanged.

The SerComp Coq Processor The last component is SerComp, an experimental batch processor for COQ files, supporting data aggregation and exporting of COQ files to a machine-friendly format.

3. A Data-Centric View

A central goal of SerAPI is to expose information known to COQ using a principled, "database-inspired" API. The SerAPI "Query" protocol is completely separated from the control protocol, and still in heavy development. An important feature is that changes in the protocol aggregation or query functions don't require changes in COQ at all. Thus, we enjoy freedom to experiment without interfering with core parts of the system.

The base datatype is the `coq_object` datatype that simply adds a tag to COQ types. Then, each tag can be queried for, with common options including output format, paging, and filtering:

```
type coq_object =
  | CoqConstr   of Constr.constr
  | CoqExpr     of Constrexpr.constr_expr
  | CoqTactic   of KerName.t * Tacenv.ltac_entry

type query_opt = {
  preds : query_pred sexp_list;
  limit : int sexp_option;
  sid   : Stateid.t [@default current_state()];
  pp    : print_opt [@default {pp_fmt = PpSer}];
  }

type query_cmd = Option | Tactics | Goals | ...
```

A key difference with the current XML protocol is that SerAPI won't produce any non-control information unless queried. This implies a slightly different control flow that we prefer.

4. Current Status and Goals

SerAPI is open source and available at [1]. The current prototype is fully functional, and we provide a functional web version[1]. Experience with jsCoq gives us moderate confidence on the robustness of the approach; our main goal at this stage is to gather more interest and get feedback from users, helping us to converge on the design. The two most important tasks for the near future are:

- Finalize the control protocol: Experience implementing IDEs on top of SerAPI has led us to extend the original STM protocol in a few ways. However, some tricky details remain to be finalized, such as the definitive error handling procedure. We hope to complete a first draft of the extended STM protocol soon, and hopefully some of the ideas would be suitable for incorporation in COQ upstream.
- Principled data API: COQ internally stores many useful data, however there is no principled interface for access. We believe however that for the most part, the ML API is not so far from it. We hope we can contribute to improve and uniformize COQ's upstream API too, covering some important use cases such as efficient search and completion.

A medium term goal is to improve the COQ's printing system so tools can offer a richer interpretation of notations (see for example [6]). Some preliminary work is already in place but more research is needed before completing a fully-functioning prototype.

Acknowledgements: We would like to thank Clément Pit–Claudel, Valentin Robert, and Enrico Tassi for very valuable feedback and testing.

References

[1] E. J. G. Arias. *Coq Serializable Protocol (SerAPI)*. `https://github.com/ejgallego/coq-serapi`. Accessed: 2016-09-13.

[2] D. Aspinall. "Proof General: A Generic Tool for Proof Development". In: *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*. Vol. 1785. Springer, 2000, pp. 38–42.

[3] B. Barras, C. Tankink, and E. Tassi. "Asynchronous Processing of Coq Documents: From the Kernel up to the User Interface". In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. Vol. 9236. Springer, 2015, pp. 51–66.

[4] A. Faithfull et al. "Coqoon: An IDE for interactive proof development in Coq". In: *TACAS*. Eindhoven, Netherlands, Apr. 2016.

[5] E. J. Gallego Arias, B. Pin, and P. Jouvelot. "jsCoq: towards hybrid theorem proving interfaces". In: *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers*. 2016.

[6] C. Pit-Claudel and P. Courtieu. "Company-Coq: Taking Proof General one step closer to a real IDE". In: *CoqPL'16: The Second International Workshop on Coq for PL*. Jan. 2016.

[7] P. Steckler. *Proof General with XML Protocol Support*. `https://github.com/psteckler/ProofGeneral`. Accessed: 2016-09-13.

[8] J. Vouillon and V. Balat. "From bytecode to JavaScript: the Js_of_ocaml compiler". In: *Softw., Pract. Exper.* 44.8 (2014), pp. 951–972.

[1] `https://x80.org/rhino-hawk/`

[9] M. Wenzel. "Asynchronous User Interaction and Tool Integration in Isabelle/PIDE". In: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Vol. 8558. Springer, 2014, pp. 515–530.

[10] M. Wenzel. "PIDE as front-end technology for Coq". In: *CoRR* abs/1304.6626 (2013).