Asynchronous processing of Coq documents: from the kernel up to the user interface

Bruno Barras, Carst Tankink, and Enrico Tassi

Inria

{bruno.barras, carst.tankink, enrico.tassi}@inria.fr

Abstract. The work described in this paper improves the reactivity of the Coq system by completely redesigning the way it processes a formal document. By subdividing such work into independent tasks the system can give precedence to the ones of immediate interest for the user and postpone the others. On the user side, a modern interface based on the PIDE middleware aggregates and presents in a consistent way the output of the prover. Finally postponed tasks are processed exploiting modern, parallel, hardware to offer better scalability.

1 Introduction

In recent years Interactive Theorem Provers (ITPs) have been successfully used to give an ultimate degree of reliability to both complex software systems, like the L4 micro kernel [7] and the CompCert C compiler [8], and mathematical theorems, like the Odd Order Theorem [5]. These large formalization projects have pushed interactive provers to their limits, making their deficiencies apparent. The one we deal with in this work is *reactivity*: how long it takes for the system to react to a user change and give her feedback on her point of interest. For example if one takes the full proof of the Odd Order Theorem, makes a change in the first file and asks the Coq prover for any kind of feedback on the last file she has to wait approximately two hours before receiving any feedback.

To find a solution to this problem it is important to understand how formal documents are edited by the user and checked by the prover. Historically ITPs have come with a simple text based Read Eval Print Loop (REPL) interface: the user inputs a command, the system runs it and prints a report. It is up to the user to stock the right sequence of commands, called script, in a file. The natural evolution of REPL user interfaces adds a very basic form of script management on top of a text editor: the user writes his commands inside a text buffer and tells the User Interface (UI) to send them one by one to the same REPL interface. This design is so cheap, in terms of coding work required on the prover side, and so generic that its best incarnation, Proof General [2], has served as the reference UI for many provers, Coq included, for over a decade.

The simplicity of REPL comes at a price: commands must be executed in a linear way, one after the other. For example the prover must tell the UI if

a command fails or succeeds before the following command can be sent to the system. Under such constraint to achieve better reactivity one needs to speed up the execution of each and every command composing the formal document. Today one would probably do that by taking advantage of modern, parallel hardware. Unfortunately it is very hard to take an existing system coded in a imperative style and parallelize its code at the fine grained level of single commands. Even more if the programming language it is written in does not provide light weight execution threads. Both conditions apply to Coq.

If we drop the constraint imposed by the REPL, a different, complementary, way to achieve better reactivity becomes an option: process the formal document *out-of-order*,[1] giving precedence to the parts the user is really interested in and postpone the others. In this view, even if commands do not execute faster, the system needs to execute fewer of them in order to give feedback to the user. In addition to that, when the parts the document is split in happen to be independent, the system can even process them in parallel.

Three ingredients are crucial to process a formal document out-of-order.

First, the UI must not impose to the prover to run commands linearly. A solution to this problem has been studied by Wenzel in [13] for the Isabelle system. His approach consists in replacing the REPL with an *asynchronous interaction loop*: each command is marked with a unique identifier and each report generated by the prover carries the identifier of the command to which it applies. The user interface sends the whole document to the prover and uses these unique identifiers to present the prover outputs coherently to the user. The asynchronous interaction loop developed by Wenzel is part of a generic middleware called PIDE (for Prover IDE) that we extend with a Coq specific back end.

Second, the prover must be able to perform a *static analysis of the document* in order to organize it into coarse grained tasks and take scheduling decisions driven by the user's point of interest. Typically a task is composed of many consecutive commands. In the case of Coq a task corresponds to the sequence of tactics, proof commands, that builds an entire proof. In other words the text between the `Proof` and `Qed` keywords.

Third, the system must feature an *execution model* that allows the reordering of tasks. In particular we model tasks as pure computations and we analyze the role their output plays in the checking of the document. Finally we implement the machinery required in order to execute them in parallel by using the coarse grained concurrency provided by operating system processes.

In this work we completely redesigned from the ground up the way Coq processes a formal document in order to obtain the three ingredients above. The result is a more reactive system that also performs better at checking documents in batch mode. Benchmarks show that the obtained system is ten times more reactive when processing the full proof of the Odd Order Theorem and that it scales better when used on parallel hardware. In particular fully checking such

[1] In analogy with the "out-of-order execution" paradigm used in most high-performance microprocessors

proof on a twelve core machine is now four times faster than before. Finally, by plugging Coq in the PIDE middleware we get a modern user interface based on the jEdit editor that follows the "spell checker paradigm" made popular by tools like Eclipse or Visual Studio: the user freely edits the document and the prover constantly annotates it with its reports, like underlining in red problematic sentences. The work of Wenzel [11, 12, 13, 14] on the Isabelle system has laid the foundations for our design, and has been a great inspiration for this work. The design and implementation work spanned over three years and the results are part of version 8.5 of the Coq system. All authors were supported by the Paral-ITP ANR-11-INSE-001 project.

The paper is organized as follows. Section 2 describes the main data structure used by the prover in order to statically analyze the document and represent the resulting tasks. Section 3 describes how asynchronous, parallel, computations are modeled in the logical kernel of the system and how they are implemented in the OCaml programming language. Section 4 describes how the document is represented on the UI side, how data is aggregated and presented to the user. Section 5 presents a reactivity benchmark of the redesigned system on the full proof of the Odd Order Theorem. Section 6 concludes.

2 Processing the formal document out-of-order

Unlike REPL in the asynchronously interaction model promoted by PIDE [13] the prover is made aware of the whole document and it is expected to process it giving precedence to the portion of the text the user is looking at. To do so the system must identify the parts of the document that are not relevant to the user and postpone their processing. The most favorable case is when large portions of the document are completely independent from each other, and hence one has complete freedom on the order in which they must be processed. In the specific case of Coq, *opaque proofs* have this property. Opaque proofs are the part of the document where the user builds a proof evidence (a proof term in Coq's terminology) by writing a sequence of tactics and that ends with the `Qed` keyword (lines four to seven in Figure 1). The generated proof term is said to be opaque because it is verified by the kernel of the system and stored on disk, but the term is never used, only its corresponding statement (its type) is. The user can ask the system to print such term or to translate it to OCaml code, but from the viewpoint of the logic of Coq (type theory) the system commits not to use the proof term while checking other proofs.[2]

The notion of proof opacity was introduced a long ago in Coq version 5.10.5 (released in May 1994) and is crucial for us: given that the proof term is not used, we can postpone the processing of the piece of the document that builds it as much as we want. All we need is its type, that is the statement that is spelled out explicitly by the user. Proofs are also lengthy and usually the time spent in processing them dominates the overall time needed in order to check

[2] In the Curry-Howard correspondence Coq builds upon lemmas and definitions are the same: a term of a given type. An opaque lemma is a definition one cannot unfold.

```
1 (* global *) Definition decidable (P : Prop) := P \/ ~ P.
2
3 (* branch *) Theorem dec_False : decidable False.
4 (* tactic *) Proof.
5 (* tactic *) unfold decidable, not.
6 (* tactic *) auto.
7 (* merge  *) Qed.
```

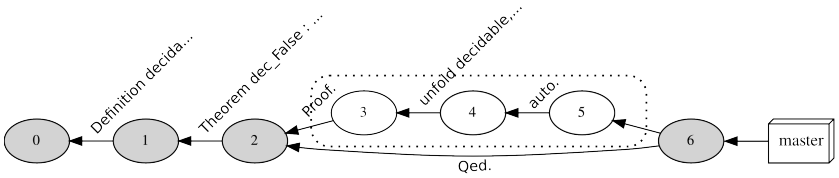

Fig. 1. Coq document and its internal representation

the entire document. For example in the case of the Odd Order Theorem proofs amount to 60% of the non-blanks (3.175KB over 5.262KB) and Coq spends 90% of its time on them. This means that, by properly scheduling the processing of opaque proofs, we can increase the reactivity of the system of a factor of ten. In addition to that, the independence of each proof from the others makes it possible to process many proofs at the same time, in parallel, giving a pretty good occasion to exploit modern parallel hardware.

In the light of that, it is crucial to build an internal representation for the document that makes it easy to identify opaque proofs. Prior to this work Coq had no internal representation of the document at all. To implement the `Undo` facility, Coq has a notion of *system state* that can can be saved and restored on demand. But the system used to keep no trace of how a particular system state was obtained; which sequence of commands results in a particular system state.

The most natural data structure for keeping track of how a system state is obtained is a Directed Acyclic Graph (DAG), where nodes are system states and edges are labeled with commands. The general idea is that in order to obtain a system state from another one, one has to process all the commands on the edges linking the two states. The software component in charge of building and maintaining such data structure is called State Transaction Machine (STM), where the word transaction is chosen to stress that commands need to be executed atomically: there is no representation for a partially executed command.

2.1 The STM and the static analysis of the document

The goal of the static analysis of the document is to build a DAG in which proofs are explicitly separated from the rest of the document. We first parse each sentence obtaining the abstract syntax tree of the corresponding command in order to classify it. Each command belongs to only one of the following categories:

commands that start a proof (*branch*), commands that end a proof (*merge*), proof commands (*tactic*), and commands that have a global effect (*global*). The DAG is built in the very same way one builds the history of a software project in a version control system. One starts with an initial state and a default branch, called master, and proceeds by adding new commits for each command. A commit is a new node and an edge pointing to the previous node. Global commands add the commit on the main branch; branching commands start a new branch; tactic commands add a commit to the current branch; merging commands close the current branch by merging it into master. If we apply these simple rules to the document in Figure 1 we obtain a DAG where the commands composing the proof of `dec_False` have been isolated. Each node is equipped with a unique identifier, here positive numbers. The edge from the state six to state five has no label, it plays the role of making the end of the proof easily accessible but has not to be "followed" when generating state six.

Indeed to compute state six, or anything that follows it, Coq starts from the initial state, zero, and then executes the labeled transactions until it reaches state six, namely "`Definition`...", "`Theorem`..." and `Qed`. The nodes in gray (1, 2 and 6) are the ones whose corresponding system state has been computed. The nodes and transactions in the dotted region compose the proof that is processed asynchronously. As a consequence of that the implementation of the merging command has to be able to accommodate the situation where the proof term has not been produced yet. This is the subject of Section 3. For now the only relevant characteristic of the asynchronous processing of opaque proofs is that such process is a pure computation. The result of a pure computation does depend only on its input. It does not matter when it is run nor in which environment and it has no visible global effect. If we are not immediately interested in its result we can execute it lazily, in parallel or even remotely via the network.

Tactic commands are not allowed to appear outside a proof; on the contrary global commands can, an in practice do, appear in the middle of proofs, as in Figure 2. Mixing tactics with global commands is not a recommend style for finished proof scripts, but it is a extremely common practice while one writes the script and it must be supported by the system. The current semantics of Coq documents makes the effect of global commands appearing in the middle of proofs persist outside proof blocks (hence the very similar documents in Figure 1 and Figure 2 have a different semantics). This is what naturally happens if the system has a single, global, imperative state that is updated by the execution of commands. In our scenario the commands belonging to opaque proofs may even be executed remotely, hence a global side effect is lost. To preserve the current semantics of documents, when a global command is found in the middle of a proof its corresponding transaction is placed in the DAG twice: once in the proof branch at its original position, and another time on the master branch. This duplication is truly necessary: only the transaction belonging to the master branch will retain its global effect; the one in the proof branch, begin part of a pure computation, will have a local effect only. In other words the effect of the "`Hint`..." transaction from state 3 to 4 is limited to states 4 and 5.

```
1 (* global *) Definition decidable (P : Prop) := P \/ ~ P.
2
3 (* branch *) Theorem dec_False : decidable False.
4 (* tactic *) Proof.
5 (* global *) Hint Extern 1 => unfold decidable, not.
6 (* tactic *) auto.
7 (* merge  *) Qed.
```

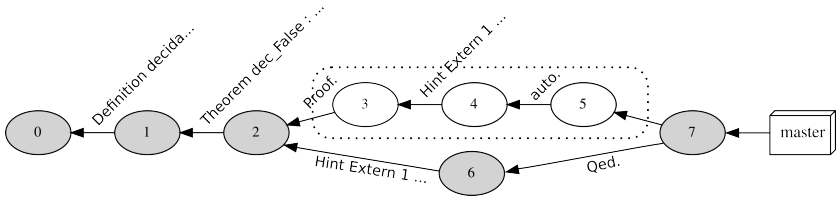

Fig. 2. Coq document and its internal representation

Branches identifying opaque proofs also define compartments from which errors do not escape. If a proof contains an error one can replace such proof by another one, possibly a correct one, without impacting the rest of the document. As long as the statement of an opaque proof does not change, altering the proof does not require re-checking what follows it in the document.

3 Modelling asynchronously computed proofs

The logical environment of a prover contains the statements of the theorems which proof has already been checked. In some provers, like Coq, it also contains the proof evidence, a proof term. A theorem enters the logical environment only if its proof evidence has been checked. In our scenario the proof evidences may be produced asynchronously, hence this condition has to be relaxed, allowing a theorem to be part of the environment before its proof is checked. Wenzel introduced the very general concept of proof promise [11] in the core logic of Isabelle for this precise purpose.

3.1 Proof promises in Coq

Given that only opaque proofs are processed asynchronously we can avoid introducing in the theory of Coq the generic notion of a sub-term asynchronously produced. The special status of opaque proofs spares us to change the syntax of the terms and lets us just act on the logical environment and the Well Founded (WF) judgement. The relevant rules for the WF judgements, in the presentation style of [1], are the following ones. We took the freedom to omit some details, like T being a well formed type, that play no role in the current discussion.

$$\frac{E \vdash \text{WF} \quad E \vdash b : T \quad d \text{ fresh in } E}{E \cup (\textbf{definition } d : T := b) \vdash \text{WF}} \qquad \frac{E \vdash \text{WF} \quad E \vdash b : T \quad d \text{ fresh in } E}{E \cup (\textbf{opaque } d : T \mid b) \vdash \text{WF}}$$

Note that the proof evidence b for opaque proofs is checked but not stored in the environment as the body of non-opaque definitions. After an opaque proof enters the environment, it shall behave exactly like an axiom.

$$\frac{E \vdash \text{WF} \quad d \text{ fresh in } E}{E \cup (\textbf{axiom } d : T) \vdash \text{WF}}$$

We rephrase the WF judgement as the combination of two new judgements: Asynchronous and Synchronous Well Founded (AWF and SWF respectively). The former is used while the user interacts with the system. The latter complements the former when the complete checking of the document is required.

$$\frac{E \vdash \text{AWF} \quad E \vdash b : T \quad d \text{ fresh in } E}{E \cup (\textbf{definition } d : T := b) \vdash \text{AWF}} \qquad \frac{E \vdash \text{AWF} \quad d \text{ fresh in } E}{E \cup (\textbf{opaque } d : T \mid [f]_E) \vdash \text{AWF}}$$

$$\frac{E \vdash \text{SWF}}{E \cup (\textbf{definition } d{:}T := b) \vdash \text{SWF}} \qquad \frac{E \vdash \text{SWF} \quad b = \text{run } f \text{ in } E \quad E \vdash b{:}T}{E \cup (\textbf{opaque } d{:}T \mid [f]_E) \vdash \text{SWF}}$$

$$\frac{E \vdash \text{AWF} \qquad E \vdash \text{SWF}}{E \vdash \text{WF}}$$

The notation $[f]_E$ represents a pure computation in the environment E that eventually produces a proof evidence b. The implementation of the two new judgements amounts to replace the type of opaque proofs (`term`) with a function type (`environment -> term`) and impose that such computation is pure. In practice the commitment of Coq to not use the proof terms of theorems terminated with `Qed` makes it possible to run these computations when it is more profitable. In our implementation this typically happens in the background.

As anticipated the Coq system is coded using the imperative features provided by the OCaml language quite pervasively. As a result we cannot simply rely on the regular function type (`environment -> term`) to model pure computations, since the code producing proof terms is not necessarily pure. Luckily the `Undo` facility lets one backup the state of the system and restore it, and we can use this feature to craft our own, heavyweight, type of pure computations. A pure computation c_0 pairs a function f with the system state it should run in s_0 (that includes the logical environment). When c_0 is executed, the current system state s_c is saved, then s_0 in installed and f is run. Finally the resulting value v and resulting state s_1 are paired in the resulting computation c_1, and the original state s_c is restored. We need to save s_1 only if the computation c_1 needs to be chained with additional impure code. A computation producing a proof is the result of chaining few impure functions with a final call to the kernel's type checker that is a *purely functional* piece of code. Hence the final system state is always discarded, only the associated value (a proof term) is retained.

The changes described in this section enlarged the size of the trusted code base of Coq by less than 300 lines (circa 2% of its previous size).

3.2 Parallelism in OCaml

The OCaml runtime has no support for shared memory and parallel thread execution, but provides inter process communication facilities like sockets and data marshaling. Hence the most natural way to exploit parallel hardware is to split the work among different worker processes.

While this approach imposes a clean message passing discipline, it may clash with the way the state of the system is represented and stocked. Coq's global state is unstructured and fragmented: each software module can hold some private data and must register a pair of functions to take a snapshot and restore an old backup to a central facility implementing the `Undo` command. Getting a snapshot of the system state is hence possible, but marshalling it and sending it to a worker process is still troublesome. In particular the system state can contain a lot of unnecessary data, or worse data that cannot be sent trough a channel (like a file descriptor) or even data one does not want to send to a precise worker, like the description of the tasks he is not supposed to perform.

Our solution to this problem is to extrude from the system state the unwanted data, put a unique key in place of it and finally associate via a separate table the data to the keys. When a system state is sent to a worker process the keys it contains lose validity, hence preventing the worker process to access the unwanted data.

The only remaining problem is that, while when a system state becomes unreferenced it is collected by the OCaml runtime, the key-value table still holds references to a part of that system state. Of course we want the OCaml runtime to also collect such data. This can be achieved by making the table "key-weak", in the sense that the references it holds to its keys do not prevent the garbage collector from collecting them and that when this happens the corresponding data has also to be collected. Even if the OCaml runtime does not provide such notion of weak table natively, one can easily code an equivalent finalization mechanism known as *ephemeron* (key-value) pairs [6] by attaching to the keys a custom finalization function.

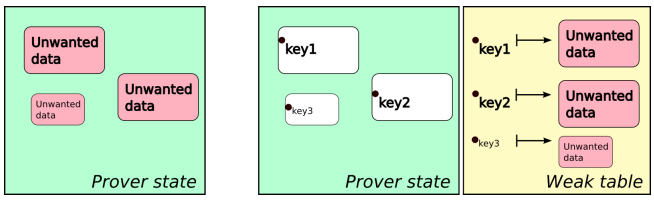

Fig. 3. Reorganization of the prover state

3.3 The asynchronous task queue and the quick compilation chain

The STM maintains a structured representation of the document the prover is processing, identifies independent tasks and delegates them to worker processes. Here we focus on the interaction between Coq and the pool of worker processes.

The main kind of tasks identified by the STM is the one of opaque proofs we discussed before, but two other tasks are also supported. The first one is queries: commands having no effect but for producing a human readable output. The other one is tactics terminating a goal applied to a set of goals via the "`par:`" goal selector.[3] In all cases the programming API for handling a pool of worker processes is the `AsyncTaskQueue`, depicted Figure 4. Such data structure, generic enough to accommodate the three aforementioned kind of tasks, provides a priority queue in which tasks of the same kind can be enqueued. Tasks will eventually be picked up by a worker manager (a cooperative thread in charge of a specific worker process), turned into requests and sent to the corresponding worker.

```
module type Task = sig
  type task
  type request
  type response
  val request_of_task : [ `Fresh | `Old ] -> task -> request option
  val use_response : task -> response -> [ `Stay | `Reset ]
  val perform : request -> response
end
module MakeQueue(T : Task) : sig
  val init : max_workers:int -> unit
  val priority_rel : (T.task -> T.task -> int) -> unit
  val enqueue_task : T.task -> cancel_switch:bool ref -> unit
  val dump : unit -> request list
end
module MakeWorker(T : Task) : sig
  val main_loop : unit -> unit
end
```

Fig. 4. Programming API for asynchronous task queues

By providing a description of the task, i.e. a module implementing the `Task` interface, one can obtain at once the corresponding queue and the main function for the corresponding workers. While the `task` datatype can, in principle, be anything, `request` and `response` must be marshalable since they will be exchanged between the master and a worker processes. `request_of_task` translates a `task` into a `request` for a given worker. If the worker is an `` `Old `` one, the representation of the request can be lightweight, since the worker can save the context

[3] The "`par:`" goal selector is a new feature of Coq 8.5 made possible by this work

in which the tasks take place and reuse it (for example the proof context is the same for each goal to which the `par:` goal selector applies). Also a `task`, while waiting in the queue, can become obsolete, hence the option type. The worker manager calls `use_response` when a response is received, and decides whether the worker has to be reset or not. The `perform` function is the code that is run in the worker. Note that when a task is enqueued a handle to cancel the execution if it becomes outdated is given. The STM will flip that bit eventually, and the worker manager will stop the corresponding worker.

This abstraction is not only valuable because it hides to the programmer all the communication code (socket, marshalling, error handling) for the three kinds of tasks. But also because it enables a new form of separate *quick compilation* (batch document checking) that splits such job into two steps. A first and quick one that essentially amounts at checking everything but the opaque proofs and generates an (incomplete) compiled file, and a second one that completes such file. An incomplete file, extension `.vio`, can already be used: as we pointed out before the logic of Coq commits not to use opaque proofs, no matter if they belong to the current file or to another one. Incomplete `.vio` files can be completed into the usual `.vo` files later on, by resuming the list of requests also saved in the `.vio` file.

The trick is to set the number of workers to zero when the queue is initialized. This means that no task is processed at all when the document is checked. One can then `dump` the contents of the queue in terms of a list of requests (a marshalable piece of data) and stock it in the `.vio` files. Such lists represents all the tasks still to be performed in order to check the opaque proofs of the document. The performances of the quick compilation chain are assessed in Section 5.

4 The user side

A prover can execute commands out-of-order only if the user can interact with it asynchronously. We need an interface that systematically gives to the prover the document the user is working on and declares where the user is looking at to help the system take scheduling decisions. This way the UI also frees the user from the burden of explicitly sending a portion of its text buffer to the prover. Finally this interface needs to be able to interpret the reports the prover generates in no predictable order, and display them to the user.

An UI like this one makes the user experience in editing a formal document way closer to the one he has when editing a regular document using a mainstream word processor: he freely edits the text while the system performs "spell checking" in the background, highlighting in red misspelled words. In our case it will mark in red illegal proof steps. Contextual information, like the current goal being proved, is displayed according to the position of the cursor. It is also natural to have the UI aggregate diverse kinds of feedbacks on the same piece of text. The emblematic example is the full names of identifiers that are displayed as an hyperlink pointing to the place where the identifier is bound.

4.1 PIDE and its document model

To integrate user interfaces with asynchronous provers in a uniform way, Wenzel developed the PIDE middleware [13]. This middleware consists of a number of API functions in a frontend written in the Scala programming language, that have an effect on a prover backend. The front-, and backend together maintain a shared data structured, PIDE's notion of a *document*. Figure 5 depicts how the front and backend collaborate on the document, and its various incarnations. The portions of the figure in dark red represent parts of PIDE that were implemented to integrate Coq into the PIDE middleware.

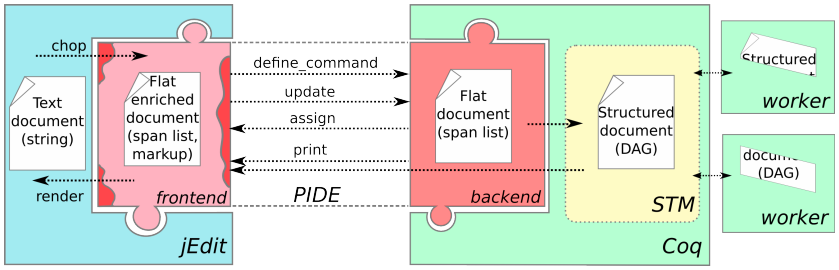

Fig. 5. PIDE sitting between the prover and the UI

This document has a very simple, prover-independent, structure: it is a flat list of spans. A span is the smallest portion of text in which the document is chopped by a prover specific piece of code. In the case of Coq it corresponds to a full command. The frontend exposes an *update* method on the document, which allows an interface to notify the prover of changes. The second functionality of the document on the frontend is the ability to query the document for any markup attached to the command spans. This markup is provided by the prover, and is the basis for the rich feedback we describe in Section 4.4.

The simple structure of the document keeps the frontend "dumb", which makes it easy to integrate a new prover in PIDE: for Coq, this only required implementing a parser that can recognize, chop, command spans (around 120 lines of Scala code) and a way to start the prover and exchange data with it (10 more lines). The prover side of this communication is further described in Section 4.2, which also describes the representation of the document text in the PIDE backend for Coq.

The way an interface uses the PIDE document model is by updating it with new text, and then reacting to new data coming from the prover, transforming it in an appropriate way. For example, Coq can report the abstract syntax tree (AST) of each command in the proof document, which the interface can use to provide syntax highlighting which is not based on error-prone regular expressions. Beyond the regular interface services, the information can also be used to

support novel interactions with a proof document. In the case of an AST, for example, the tree can be used to support refactoring operations robustly.

4.2 Pidetop

Pidetop is the toplevel loop that runs on the Coq side and translates PIDE's protocol messages into instructions for the STM. Pidetop maintains an internal representation of the shared document. When its update function is invoked, the toplevel updates the internal document representation, inserting and deleting command phrases where appropriate. The commands that are considered as new by pidetop are not only those that have changed but also those that follow: because the client is dumb, it does not know what the impact of a change is. Instead it relies on the statical analysis the STM performs to determine this.

Finally PIDE instructs the STM to start checking the updated document. This instruction can include a *perspective*: the set of spans that frontend is currently showing to the user. This perspective is used by the STM to prioritize processing certain portions of the document.

Processing the document produces messages to update PIDE of new information. During the computation, PIDE can interrupt the STM at any moment, to update the document following the user's changes.

4.3 Metadata collection

The information Coq needs to report to the UI can be classified according to its nature: persistent or volatile. *Persistent information* is part of the system state, and can be immediately accessed. For this kind of information we follow the model of asynchronous prints introduced by Wenzel in [14]. When a system state is ready it is reported to (any number of) asynchronous printers, which process the state, reporting some information from it as marked up messages for the frontend. Some of these printers, like those reporting the goal state at each span, are always executed, while others can be printed on-demand. An example of the latter are Coq's queries, which are scheduled for execution whenever the user requests it. *Volatile information* can only be reported during the execution of a command, since it is not retained in the system state. An example is the resolution of an identifier resulting in an hyper link: the relation between the input text and the resulting term is not stored, hence localized reports concerning sub terms can only be generated while processing a command. Volatile information can also be produced optionally and on demand by processing commands a second time, for example in response to the user request of more details about a term. From the UI perspective both kind of reports are asynchronously generated and handled homogeneously.

4.4 Rich Feedback

A user interface building on PIDE's frontend can use the marked up metadata produced by the prover to provide a much richer editing experience than was

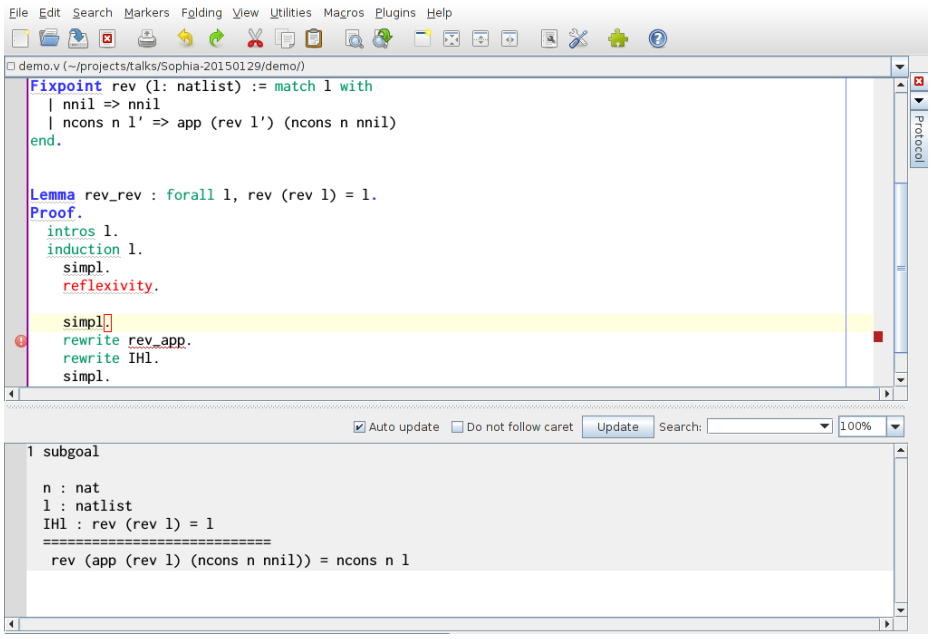

Fig. 6. jEdit using PIDE

previously possible: the accessible information is no longer restricted to the output of the last executed command. For example jEdit, the reference frontend based on PIDE, uses the spell checker idiom to report problems in the entire proof document: instead of refusing to proceed beyond a line containing an error the interface underlines all the offending parts in red, and the user can inspect each of them.

This modeless interaction model also allows the interface to take different directions than the write-and-execute one imposed by REPL based UI. For example it is now easy to decorrelate the user's point of observation and the place where she is editing the document. A paradigmatic example concerns the linguistic construct of postfix bookkeeping of variables and hypotheses (the `as` intro pattern in Coq). The user is encouraged to decorate commands like the one that starts an induction with the names of the new variables and hypotheses that are available in each subgoal. This decoration is usually done in a sloppy way: when the user starts to work on the third subgoal she goes back to the induction command in order to write the intro pattern part concerning the third goal. In the REPL model she would lose the display of the goal she is actually working on because the only goal displayed. would be the one immediately before the induction command. By using a PIDE based interface to Coq the user can now leave the observation point on the third goal and see how it changes *while* she is editing the intro pattern (note the "Do not follow caret" check box in Figure 6).

These new ways of interaction are the first experiments with the PIDE model for Coq, and we believe that even more drastically different interaction idioms can be provided, for example allowing the user to write more structured proofs, where the cases of a proof are gradually refined, possibly in a different order than the one in which they appear in the finished document.

The PIDE/jEdit based interface for Coq 8.5 is distributed at the following URL: `http://coqpide.bitbucket.org/`.

5 Assessment of the quick compilation chain

As described in Section 3.3 our design enables Coq to postpone the processing of opaque proofs even when used as a batch compiler. To have a picture of the improvement in the reactivity of the system we consider the time that one needs to wait (we call that latency) in order to be able to use (`Require` in Coq's terminology) the last file of the proof of the Odd Order Theorem after a change in the first file of such proof. With Coq 8.4 there is nothing one can do but to wait for the full compilation of the whole formalization. This takes a bit less than two and a half hours on a fairly recent Xeon 2.3GHz machine using a single core (first column). Using two cores, and compiling at most two files at a time, one can cut that time to ninety minutes (second column). Unfortunately batch compilation has to honour the dependency among files and follow a topological order, hence by adding extra 10 cores one can cut only twelve minutes (third column). Hence, in Coq 8.4 the best latency is of the order of one hour.

Thanks to to the static analysis of the document described in Section 2.1, the work of checking definitions and statements (in blue) can be separated by the checking of opaque proofs (in red), and one can use a (partially) compiled Coq file even if its opaque proofs are not checked yet. In this scenario, the same hardware gives a latency of twelve minutes using a single core (fourth column), eight minutes using two cores (fifth column) and seven minutes using twelve cores (sixth column). After that time one can use the entire formalization. When one then decides to complete the compilation he can exploit the fact that each proof is independent to obtain a good degree of parallelism. For example checking all proofs using twelve cores requires thirteen extra minutes after the initial analysis, for a total of twenty minutes. This is 166% of the theoretical optimum one could get (seventh column). Still with 12 cores the latency is only six minutes, on par with the theoretical optimum for 24 cores.

The reader may notice that the quick compilation chain using 1 core (4th column) is slightly faster than the standard compilation chain. This phenomenon concerns only the largest and most complicated files of the development. To process these files Coq requires a few gigabytes of memory and it stresses the OCaml garbage collector quite a bit (where it spends more than 20% of the time). The separation of the two compilation phases passes trough marshalling to (a fast) disk and un-marshaling to an empty process space. This operation trades (non blocking, since the size of files fits the memory of the computer)

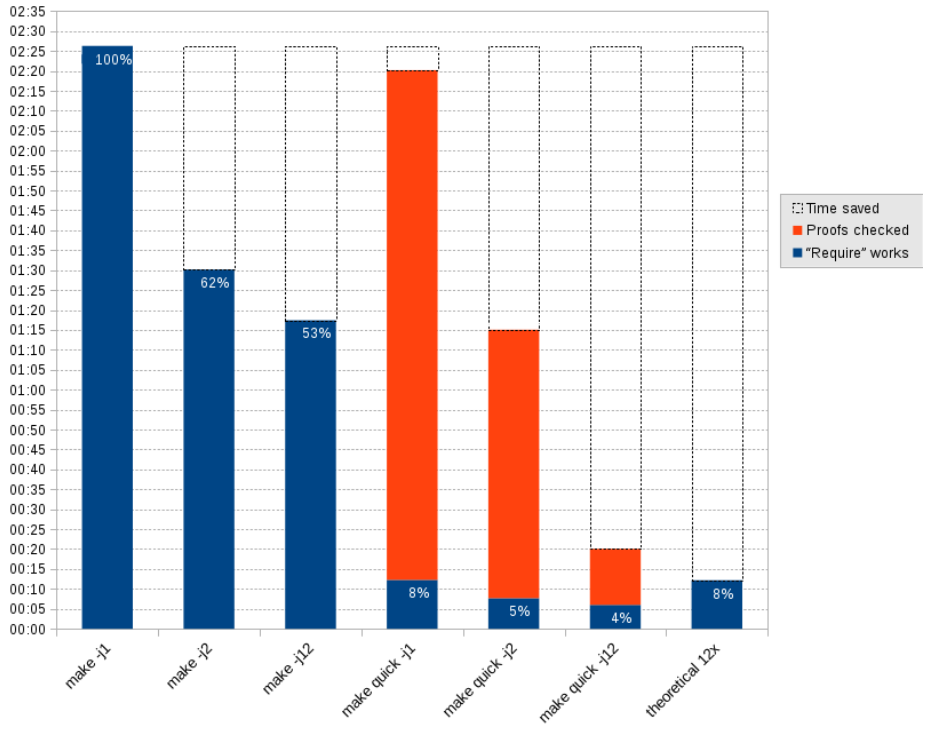

Fig. 7. Benchmarks

disk I/O for a more compact and less fragmented memory layout that makes the OCaml runtime slightly faster.

6 Concluding remarks and future directions

This paper describes the redesign Coq underwent in order to provide a better user experience, especially when used to edit large formal developments. The system is now able to better exploit parallel hardware when used in batch mode, and is more reactive when used interactively. In particular it can now talk with user interfaces that use the PIDE middleware, among which we find the one of Isabelle [9] that is based on jEdit and the Coqoon [3, 4] one based on Eclipse.

There are many ways this work can be improved. The most interesting paths seem to be the following ones.

First, one could make the prover generate, on demand, more metadata for the user consumption. A typical example is the type of sub expressions to ease the reading of the document. Another example is the precise list of theorems or local assumptions used by automatic tactics like `auto`. This extra metadata could be at the base of assisted refactoring functionalities the UI could provide.

Another interesting direction is to refine the static analysis of the document to split proofs into smaller, independent, parts. In a complementary way one could make such structure easier to detect in the proof languages supported by the system. The extra structure could be used in at least two ways. First, one could give more accurate feedback on broken proofs. Today the system stops at the first error it encounters, but a more precise structure would enable the system to backup by considering the error confined to the sub-proof in which it occurs. Second, one could increase the degree of parallelism we can exploit.

Finally one could take full profit of the PIDE middleware by adapting to Coq interesting user interfaces based on it. For example clide [10] builds on top of PIDE and provides a web based, collaborative, user interface for the Isabelle prover. The cost of adapting it to work with Coq seems now affordable.

References

[1] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. A compact kernel for the Calculus of Inductive Constructions. *Sadhana*, 34(1):71–144, 2009.

[2] D. Aspinall. Proof general: A generic tool for proof development. In *Proceedings of TACAS*, volume 1785 of *LNCS*, pages 38–42, 2000.

[3] J. Bengtson, H. Mehnert, and A. Faithfull. Coqoon: Eclipse plugin providing a feature-complete development environment for Coq. Homepage: `https://itu.dk/research/tomeso/coqoon/`, 2015.

[4] A. Faithfull, C. Tankink, and J. Bengtson. Coqoon – An IDE for interactive proof development in Coq. Submitted to CAV 2015.

[5] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A Machine-Checked Proof of the Odd Order Theorem. In *Proceedings of ITP*, volume 7998 of *LNCS*, pages 163–179, 2013.

[6] B. Hayes. Ephemerons: A new finalization mechanism. In *Proceedings of OOPSLA*, pages 176–183. ACM, 1997.

[7] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, June 2010.

[8] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[9] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[10] M. Ring and C. Lüth. Collaborative interactive theorem proving with clide. In *Proceedings of ITP*, volume 8558 of *LNCS*, pages 467–482, 2014.

[11] M. Wenzel. Parallel proof checking in Isabelle/Isar. In *Proceedings of PLMMS*, pages 13–21, 2009.

[12] M. Wenzel. Isabelle as document-oriented proof assistant. In *Proceedings of ICM/MKM*, volume 6824 of *LNCS*, pages 244–259, 2011.

[13] M. Wenzel. READ-EVAL-PRINT in parallel and asynchronous proof-checking. In *Proceedings of UITP*, volume 118 of *EPTCS*, pages 57–71, 2013.

[14] M. Wenzel. Asynchronous user interaction and tool integration in Isabelle/PIDE. In *Proceedings of ITP*, pages 515–530, 2014.