Coqoon

An IDE for Interactive Proof Development in Coq

Alexander Faithfull[1(✉)], Jesper Bengtson[1], Enrico Tassi[2], and Carst Tankink[2]

[1] IT University of Copenhagen, Copenhagen, Denmark
alef@itu.dk
[2] Inria, Sophia Antipolis, France

Abstract. User interfaces for interactive proof assistants have always lagged behind those for mainstream programming languages. Whereas integrated development environments—IDEs—have support for features like project management, version control, dependency analysis and incremental project compilation, "IDE"s for proof assistants typically only operate on files in isolation, relying on external tools to integrate those files into larger projects. In this paper we present Coqoon, an IDE for Coq developments integrated into Eclipse. Coqoon manages proofs as projects rather than isolated source files, and compiles these projects using the Eclipse common build system. Coqoon takes advantage of the latest features of Coq, including asynchronous and parallel processing of proofs, and—when used together with a third-party OCaml extension for Eclipse—can even be used to work on large developments containing Coq plugins.

1 Introduction

In the last decade, computer-aided proof development has been gaining momentum. Interactive proof assistants allow their users to state a mathematical theorem in a language that the system understands and then prove that theorem within the system. As long as the proof assistant's verification code is free from bugs, this guarantees that all proofs are actually correct, that no details have been overlooked, and that no mistakes were made. Mechanizing proofs in this way makes very large proofs feasible and protects against subtle and hard-to-notice human errors. Two recent milestones in computer science include the verification of an optimising C compiler [6] and of a micro-kernel [17]. Proof assistants have also been used to verify advanced results in mathematics, such as the Odd Order Theorem, using Coq [12], and the proof of the Kepler conjecture, using HOL-Light and Isabelle [14].

Meanwhile, on the other side of the great chasm between theory and practice, software developers too have come to appreciate computer assistance as they work. For a developer, though, that assistance comes not in the form of a proof assistant, but of an *integrated development environment* (IDE).

C. Tankink—Funded by the Paral-ITP ANR-11-INSE-001 project.

© Springer-Verlag Berlin Heidelberg 2016
M. Chechik and J.-F. Raskin (Eds.): TACAS 2016, LNCS 9636, pp. 316–331, 2016.
DOI: 10.1007/978-3-662-49674-9_18

The IDE combines many important tools of the trade—such as editors, compilers, refactorers, profilers, debuggers, and project and release managers—into a single unified toolbox for working with code. At a glance, the developer has an overview of every aspect of a project, and the repercussions of changes in one area can be shown in every other affected area, allowing the developer to make any necessary corrections. Many IDEs can even abstract away the build process entirely, automatically inferring the relationships between source files and libraries and rebuilding them when necessary.

The workflows of interactive proof assistants are sufficiently similar to conventional programming languages that one might expect IDEs to exist for them as well, but this is not the case. Even though proof assistants are gaining in popularity, there are still no real IDEs for them—none of them are truly *integrated*. Coq is one of the most widely-used proof assistants available, but its proofs are most often written using either Proof General or CoqIDE; these specialised text editors operate only on individual files, leaving project management entirely to developers. Projects are typically built using Makefiles, which in practice require a POSIX-like environment; file dependencies are supported via command-line tools; and complex inter-project dependencies are not supported at all, leaving the work of building and linking projects together up to the user. Moreover, both Proof General and CoqIDE have a workflow, often referred to as the *waterfall* model, in which the editor is only aware of the state at one specific point: to view the state elsewhere, the user must either execute all commands up to the desired point or explicitly revert to an earlier point in the document, throwing away all the computations back to that point in the process. This workflow is not only alien to software developers, who are used to being able to edit their files at arbitrary points and receive immediate feedback from the IDE on what effect these changes had on the rest of their development, it is also very slow (although upcoming versions of CoqIDE improve this situation somewhat).

We argue that the lack of tool support for proof assistants is to the detriment of both theoreticians and software developers with an interest in verification. Requiring that developers learn an old-fashioned workflow in order to try out formal methods is unquestionably a deterrent, but that workflow is also a waste of time and effort for those who have grown accustomed to it. Integration and automation have made life easier for programmers: why should the same not also be true for proof authors?

In this paper we present Coqoon—an Eclipse-based IDE for proof development using the Coq proof assistant. Coqoon includes support for Coq projects, much like Eclipse's built-in support for Java projects: users can create Coq projects, structure these projects using folder hierarchies, and add Coq source files to these folders, and the Eclipse automatic build system will keep track of the project dependencies behind the scenes. Whenever a file is changed, moved, or renamed, everything that depends on it is automatically recompiled, and any errors are reported to the user.

Coqoon does away with the waterfall model, instead allowing the user to make changes anywhere in a file—and automatically and asynchronously reproving

only the parts that are affected by that change. In this way, Coqoon behaves a lot more like an IDE that software developers are familiar with than the tools available to Coq developers today.

Coqoon is also an *integrated* development environment in the fullest sense of the term. Eclipse has a wide variety of plugins available, ranging from version control plugins like EGit to entire development environments like OcaIDE for OCaml, which can be used alongside Coqoon. The combination of Coqoon and OcaIDE is particularly useful, as it brings support for complex Coq developments that contain both proofs and OCaml plugins.

Coqoon depends on features added to Coq in version 8.5. Architectural changes to Coq 8.5 allow it to support Wenzel's PIDE library for asynchronous proof developments [25], which powers Coqoon's replacement for the waterfall model. Coq 8.5 also adds a two-step compilation process, known as the *quick compilation chain*, that can optionally produce `.vio` files in place of standard Coq `.vo` libraries; this new format produces larger, faster files whose proofs are unchecked, but which can later be efficiently compiled into the traditional format by checking the remaining proofs in parallel.

As a test case, we have imported the mechanised proof of the Odd Order Theorem into Coqoon, which is one of the largest Coq 8.5-compatible developments available. Previous versions of this project took over two hours to compile, but, using the quick compilation chain, the project can be compiled into `.vio` form in just seven minutes, and then into `.vo` form in a further twenty minutes. Coqoon is the first IDE to include native support for the quick compilation chain—indeed, no other IDE for Coq has an integrated build system—which makes it possible to work with even the most complex projects at speeds that were hitherto unimaginable.

We have also adapted Pierce's course on Software Foundations to be compatible with this version. This development contains plenty of exercises that demonstrate a wide variety of features of Coq, and can be used to try out Coqoon's capabilities in a smaller setting than the Odd-Order Theorem.

Download links and installation instructions for Coqoon, along with prepackaged example projects, can be found at https://coqoon.github.io/tacas2016.

2 Coqoon, Structured Projects, and the Build System

Coqoon is a family of plugins for the Eclipse framework that together implement an IDE for Coq developments. It has support for structuring these developments easily using Eclipse workspace projects, folders, and files, and for automatically managing the verification and build processes in the face of changed dependencies. To allow more interactive development of proof scripts, Coqoon processes them in the background, showing Coq feedback directly in the proof text editor using idioms familiar to programmers (e.g., by underlining errors in red).

As Coqoon is implemented on top of the Eclipse framework, it interoperates with other Eclipse components: version control plugins like EGit [9], for example, can be used with Coqoon projects.

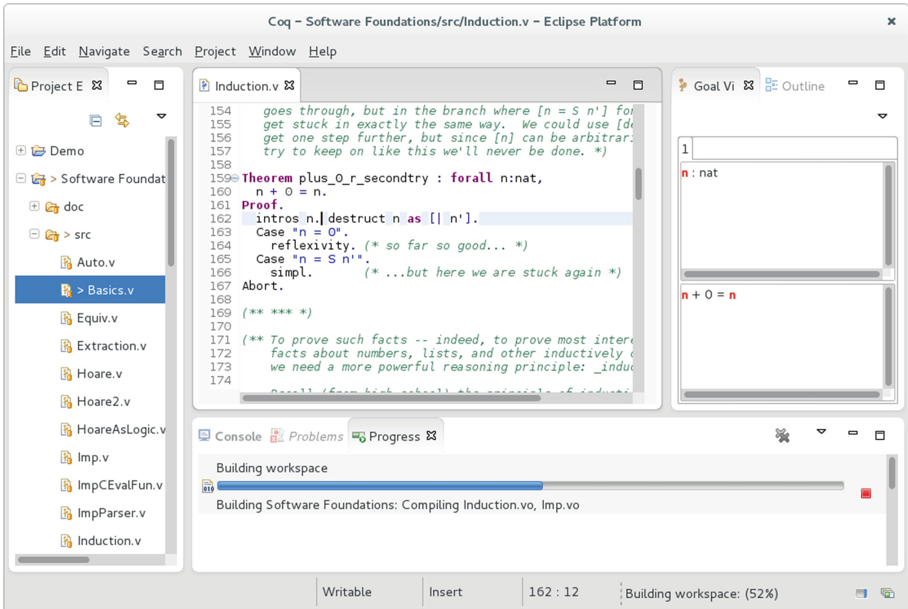

Fig. 1. A screenshot of Coqoon, showing the project viewer, a Coq editor with syntax highlighting, and the goal viewer. The progress bar at the bottom of the screen shows the Coq project builder at work.

2.1 Structured Projects

Coqoon provides a more structured environment than Coq programmers are accustomed to. From the moment the user first creates a Coq project in Coqoon, it already has a complete build system; Coq source code must be placed in designated source folders, and when files start to depend on other files, they will automatically be marked for recompilation when their dependencies are moved or changed, even if those dependencies are in other projects. A progress bar—visible at the bottom of Fig. 1—displays the state of any build operations scheduled by the builder.

This need for structure is not just the IDE being difficult: it is precisely this structure that makes more sophisticated behaviour possible. The use of the Coqoon integrated build system allows Coq projects to support dependencies on other projects, or on external developments, whilst simultaneously freeing the developer from the need to think about the build system and making it work on all operating systems supported by Eclipse.

2.2 The Coq Model

Replacing unstructured collections of files with structured projects is a start, but most IDEs go further. They transform source code files into a more structured

representation (known as a *model*), providing a higher-level way of searching and manipulating code than simple operations on plain text. Eclipse's Java model, for example, presents Java documents as abstract syntax trees; nodes in the tree that represent identifiers can also be "resolved" through the model to see what they refer to in that particular context.

Coqoon provides a similar model for Coq code. Most Coq-specific operations on files begin by using the Coq model to convert an Eclipse file handle into a Coq model file handle, which presents an alternative view of the file as a sequence of parsed and tagged Coq sentences. The model also serves as a central place to cache these sequences, so files whose content has not changed do not need to be reparsed.

In general, the goal of this model is to provide a useful, Coq-centric view of the contents of an Eclipse project. Coq project handles, for example, have methods for retrieving and modifying project configuration information, and projects can be traversed using the visitor pattern [11] to find named lemmas and definitions, making search algorithms easy to build.

The design of Coqoon's Coq model is heavily based on that of Eclipse's own Java model, which has led to the internal use of some Java concepts in areas where Coq lacks any particular convention: the Coqoon model considers projects to consist of Java-style package roots (top-level source and output directories) containing package fragments (those subdirectories of a root which contain source and output files), for example, although the concept of a package is not one native to Coq.

2.3 Coq Interaction

Coqoon's integrated Coq editor communicates using the PIDE library. Originally developed for the Isabelle proof assistant, PIDE frees the user from having to explicitly direct the prover to make progress through a source file. Proofs handled by PIDE may be evaluated in parallel or out-of-order, and Coq's state after the evaluation of each sentence is saved, making it quick and easy to see how tactics affect the state of a proof. Section 3 explains the operation of the PIDE protocol in more detail.

When using PIDE, the operation of the prover in the background is transparent to the user. Any status messages and goal information associated with a command will automatically be displayed when the user moves the text cursor onto it, and errors are highlighted when the user makes a mistake.

2.4 The Coq Build Process

When a Coq file is added to, modified in, or removed from a Coq project, the integrated Coq builder is activated. The builder is responsible for compiling all of a project's Coq proofs into library files.

Whenever the Coqoon builder is activated, Eclipse provides it with a summary of the changes to the project since the last activation. The builder then uses the Coq model to extract the new dependency information from the changed files; it then rebuilds all the changed files and their dependents in an appropriate order, postponing the compilation of a file until its dependencies are also up-to-date.

This behaviour is common to virtually all IDEs, and—although it is not supported by existing Coq interfaces—many projects have built ad-hoc emulations of it for themselves. At the time of writing, for example, the CompCert project contains a pair of shell scripts—one for use with CoqIDE, and one for use with Proof General—which compiles the dependencies of a file, opens it in the appropriate editor, and recompiles that file when the editor is subsequently closed.

A Second Implementation. As the integrated build system might conceivably also be useful outside of Coqoon, we also provide a Python reimplementation of it. This reimplementation is included by default in all Coqoon projects to make it possible to work on them even in the absence of Eclipse.

Projects and Hybrid Projects

All Eclipse projects are collections of files coupled with Eclipse build system metadata which expresses that project's specific requirements. A Coqoon project consists of Coq source code, information about the project's internal structure and its dependencies, and an instruction to the Eclipse build system explaining that the project is a Coq development under the control of the Coqoon builder.

This mechanism is sufficiently general that a project's metadata can have multiple instructions for the Eclipse build system—for example, a Coq project might declare that a bundled plugin is to be built with an OCaml builder. Indeed, a copy of Eclipse equipped with Coqoon and OcaIDE, an OCaml IDE for Eclipse[7], serves as a complete development environment for Coq projects with OCaml plugins; Sect. 4.1 describes such a scenario in more detail.

Project Dependencies

The Coqoon builder uses a project's load path to resolve the dependencies present in that project's source code, and also provides a user interface for manipulating those dependencies. This functionality is loosely inspired by the Java class path management found in Eclipse Java projects.

Coqoon supports five different kinds of load path entries:

- folders in Eclipse projects that contain source code files;
- folders in Eclipse projects that contain compiled source code;
- other Coqoon projects in the Eclipse workspace;
- folders in the local file system containing projects neither built with nor managed by Coqoon; and
- "abstract" entries which are likely to be available everywhere but whose location cannot be known in advance, like the Coq standard library.

The builder calculates how each of these kinds of entry should be represented in the Coq load path, and uses this information to resolve the dependencies of each file in the project.

The dependency resolution process is sophisticated enough to recognize when to prefer files that have yet to be compiled to files that are already available: if a project contains a file called `List.v`, for example, then other files in that project can safely depend on its compiled form by depending on `List`, even though the Coq standard library contains an identically named file which could potentially satisfy that dependency.

Abstract Dependencies. When Coqoon encounters an "abstract" entry, it looks at that entry's identifier and searches an internal registry for a class that knows how to handle that identifier. These classes can then run arbitrary code to resolve the identifier; for example, the handler for the Coq standard library finds it by running a `coqtop` process with the `-where` option.

As this internal registry is also exposed through the standard Eclipse extension mechanism, it can also be used to add new kinds of dependencies to Coqoon. We discuss one possible application of this technique in Sect. 5.2.

Aggressive Rebuilding

Coqoon's internal dependency analysis behaves like that of `make`: when a file is older than one of its dependencies, it becomes a candidate for recompilation. As a result, making changes to a file with many transitive dependencies will trigger the recompilation of many other files.

As Coq proofs do not have a clean separation between their externally-visible interface and the internal implementation, this is the only safe way of ensuring that changes to a fundamental proof are appropriately reflected throughout a project. Coqoon offers two different mitigations to make this more palatable for large developments: the builder can be configured to recompile projects only when the user explicitly requests it, or it can be told to use the Coq 8.5 quick compilation chain, speeding up compilation drastically by postponing the evaluation of proofs.

Neatness and Namespaces

Coq developments do not typically have a clean separation between source and output folders. In a simpler setting, the resulting clutter is merely annoying; in an IDE, however, compiled libraries and other derived files are normally entirely hidden from the user, which is a much harder task when these files are not systematically separated from source code.

The Coqoon builder emulates the behaviour of the Java builder to provide this separation: the Coq source file `src/SoftwareFoundations/Basics.v`, for example, is compiled into the library `bin/SoftwareFoundations/Basics.vo`, the fully-qualified name of which would be `SoftwareFoundations.Basics`. Although there are as yet no conventions for managing the Coq library namespace, this approach is flexible enough to support any convention that might be chosen in a future version of Coq.

3 PIDE: Coqoon's Interaction with Coq

PIDE is a middleware layer originally developed by Wenzel [25] to bridge the gap between the Isabelle system, implemented in PolyML, and its user interface, written in Java.

For both historical and technical reasons, many proof assistants are written in a programming language that is a descendant of ML, a language conceived with that particular application in mind. IDEs, on the other hand, are more usually built atop industry-standard platforms like Java or .NET; a layer like PIDE is thus necessary to enable provers to talk to the outside world.

3.1 PIDE in a Nutshell

PIDE consists of a relatively prover-agnostic frontend library, implemented in Scala, and a prover-specific backend in the prover's own implementation language (i.e., PolyML for Isabelle, or OCaml for Coq). These two components co-operate to ensure that frontend and backend both agree on the content and structure of the user's document. In this model, the backend has a complete view of the document, and so is free to evaluate its commands in any order it sees fit; its half of the PIDE implementation then relays the resulting—potentially out-of-order—status and feedback messages back to the frontend (and thus to the user). The frontend can also interrupt the backend with an update to the document, or direct the backend to focus its attention on a different region.

To bring PIDE support to Coq, Tankink wrote an OCaml implementation of the PIDE backend for use with Coq 8.5, also making some minor changes to the Scala library in the process [4]. Although Coqoon has benefited greatly from this work, it was not carried out with Coqoon in mind—it was originally intended for use with jEdit, a more limited text editor used as the main interface for Isabelle.

Even though jEdit and Eclipse are two very different environments, adding PIDE support to Coqoon has required only minor changes to PIDE, which shows that the library is not tied to one particular style of frontend.

4 Test Cases

To assess the maturity of the tool we apply it to two Coq developments: the *Odd Order Theorem*, a large formalization that comprises both Coq theories and a Coq extension, and the widely used teaching course in *Software Foundations* by Pierce.

4.1 The Odd Order Theorem and the Math.Comp. Library

The Odd Order Theorem by Feit and Thompson is a masterpiece of modern mathematics for which its last author received the Fields medal in 1970 and the Abel prize in 2008. This result was not only famous because of its profound influence on the last fifty years of research in group theory, but also for its

length, weighing in at more than two hundred and fifty pages. Indeed, its length and intricacy caused many to raise concerns about the correctness of its entire argument.

In 2012 a team of fifteen people, led by Gonthier [12], completed a formal verification of the proof, and of the mathematical theories it builds upon, using the Coq system. The project took six years to complete (including three years of work on the part of this paper's third author). The resulting body of formalized mathematics is divided into two main parts: the so called Mathematical Components library (Math.Comp. for short), that covers many general purpose mathematical theories (group theory, linear algebra, character theory ...) and the main proof which builds upon them.

The entire development sums up to 125 Coq modules for a total of 161,000 lines of code: 93 modules and approximately 121,000 lines for the Math.Comp. library, and 32 modules and 40,000 lines for the main proof. All Coq modules are written in a custom language, called SSReflect, that is provided by a plugin for Coq. The plugin, itself a 7,500-line OCaml program, is also part of the Math.Comp. library. The entire source code amounts to 7.4 MB.

This code base constitutes one of the largest developments for Coq, and pushes the system close to its limits; as a consequence, building it and working on it has never been a pleasant experience for the user. The dependency graph of its components, for example, is too large to be printed in this paper,[1] and building the entire project takes around two hours. This time is how long one needs to wait in order to build on top of the Math.Comp. library, or browse it comfortably, or simply to be able to go back to work after having made a minor change to one of the core modules. Despite that, other formalization projects have started depending on (parts of) the Math.Comp. library, inheriting along with it the complexity and time consumption of its build process. In particular, building the SSReflect plugin by hand has always been a source of trouble for its users, and the long time required to build the entire library eventually pushed the authors of the library to provide reduced versions of it for those users who did not need all of its power.

Importing this gargantuan project into Coqoon revealed a few deficiencies in our implementation. Coqoon's dependency resolver, for example, was overwhelmed by the size of the dependency graph, in some cases taking more than ten seconds to work out a file's dependencies. Luckily, this was easily remedied by the addition of a simple cache.

To spare the user from a prolonged compilation process, support for the quick compilation [4] chain, a new feature provided by Coq 8.5, was also added to Coqoon. This process separates Coq compilation into two phases: the first is very quick, checking only definitions and statements, while the second, slower, phase completes the compilation by checking the proofs. As the first phase produces intermediate files that can be used in place of traditional Coq libraries, it only takes around seven minutes of computation on an ordinary laptop computer before the entire set of 125 modules is usable.

[1] The interested reader can browse it online: http://coqfinitgroup.gforge.inria.fr/doc/.

As the user does not need to wait for the second phase in order to work with the development, it can typically be run as a background task. Unlike the first phase, it can take great advantage of parallel hardware, because each proof can be checked independently of the others: on a computer with a dozen cores, the proofs for the whole development can be completed in as few as 15 min.

In addition to that, we added support to OCaml modules to the Coq builder. Combined with the OCaml builder provided by OcaIDE, this has made it possible to build both the SSReflect plugin and the Coq modules that depend on it in a single integrated build process.

Finally, the PIDE backend for Coq was made more responsive and robust when dealing with long modules. Most of the files in the Mathematical Components library are more than a thousand lines of code in length, and some are more than four thousand lines. For comparison, in the CompCert compiler, [6] another Coq flagship project, composed of 5.2 MB of sources, more than 30 % of the modules are longer than a thousand lines.

As a result of these changes, we believe Coqoon represents the best platform for working on such large developments. In particular, at the time of writing, no other IDE for Coq can handle projects that contain both OCaml and Coq code, and Coqoon is the only one to incorporate the quick compilation chain as an integrated part of an automatic build system.

4.2 Software Foundations

This is a relatively small Coq development that complements the Software Foundations book by Pierce et al. It is a widely adopted course that touches on topics like logic, functional programming, interactive theorem provers, and techniques for software verification. Universities in the United States, Japan and Europe use it in their curricula.

Coqoon has been used at the IT University of Copenhagen in conjunction with the Software Foundations teaching material for three years. We have found that the use of a more familiar development environment makes Coq much more user-friendly for students, showing that building on top of an IDE brings advantages for beginners and experts alike.

5 Building on Coqoon

5.1 Embedding Coq

Our work with OcaIDE shows that Coqoon can already interoperate with other development environments built on the Eclipse platform. The next step in our work is to provide even tighter integration between the Coq and Java development environments.

Java projects can already be turned into “hybrid” projects containing both Java programs and Coq proofs about those programs, but this is only a start. There are already several tools that embed assertions and proofs directly into

the source code that they describe, like Dafny [18], Spec# [3], and VeriFast [16]. The IDE seems like an obvious home for this functionality: that is, it should be possible to extend the Java editor already present in Eclipse with Coqoon-powered Coq proofs.

In fact, Coqoon's predecessor, Kopitiam, provided just such an environment. (We discuss Kopitiam in more detail in Sect. 6.1.) However, this environment was built using cruder integration techniques and predated the introduction of PIDE. A prototype inspired by Kopitiam, but built using Coqoon, PIDE, and a custom text editor more aware of the interleaving between Coq and Java code, is under development at the IT University.

5.2 Abstract Load Paths and OPAM

At the heart of Eclipse is an implementation of the OSGi component model [10], which provides a platform for dynamically loading and unloading Java archives. Eclipse extends these archives—known as "plugins" in the Eclipse context—with extra metadata defining *extension points*, services which plugins can declare that they contribute extensions to. Extension point definitions can require arbitrary information from extensions, but this will typically include the fully-qualified name of a class implementing a particular interface; in this way the original plugin can instantiate, configure and use code contributed by one of its extensions.

Coqoon provides a number of Coq-specific extensions to Eclipse's own core plugins: for example, it contributes the Coqoon project builder to the resource management plugin, and the PIDE document editor to the text editors plugin. However, it also defines an extension point of its own, which allows other plugins to add new handlers for abstract load path entries to Coqoon.

The Coq ecosystem has not traditionally provided any way of packaging and distributing projects, which has made building on other people's work difficult and fragile. This is, however, about to change: Coq 8.5 will be distributed alongside a repository of ready-to-install Coq projects for OPAM, the OCaml Package Manager.

Making the abstract load path mechanism extensible means that Coqoon is ready to adapt to this change. We expect to include a plugin with future versions of Coqoon that will use the abstract load path mechanism to support direct dependencies on OPAM projects, but this will not require any changes to Coqoon itself.

6 Related Work

Over the last thirty years, there have been multiple attempts to make the interaction with proof assistants easier. Initially, all interaction was through a Read-Eval-Print loop (REPL), a command-line interface that interprets each command typed by the user and prints out the resulting goal state (or an error) before requesting new commands. Some proof assistants, such as HOL [13] and HOL Light [15], still use this as their primary mode of interaction.

6.1 Waterfall Interaction

Proof General [1], based on Emacs, was the one of the first interfaces that offered more than just a REPL, and is the only one of the early interfaces that endures until today, going so far as to define the *de facto* standard method of interaction with Coq: the waterfall model. Although this still required the user to direct proof processing manually, it was nevertheless a significant improvement over the bare REPL.

The Proof General model of interaction has been duplicated by several other Coq tools, including CoqIDE, which is a GTK+-based interface bundled with Coq [23], and three Eclipse plugins. The first was created by Aspinall as an attempt to port Proof General itself to Eclipse [2]; the other by Charles and Kiniry who, as part of the Moebius project, built the plugin ProverEditor for Coq in Eclipse [8]; the third, Kopitiam [20], by Mehnert, is Coqoon's immediate predecessor.

CoqIDE is a custom cross-platform text editor. It does not add any truly new interaction features, beyond some Coq-specific code templates and the ability to invoke `make` and the Coq verifier from the interface. While it allows the user to have multiple buffers open, there is no relation between the contents of the buffers. The version of CoqIDE shipped with Coq 8.5 was improved by Tassi to support processing the waterfall in parallel. However, the fundamental interaction with Coq will not change: the user still needs to manually direct Coq to process parts of the active document.

The Proof General plugin for Eclipse was only available for Isabelle, and has not been under active development since 2010, based on its Eclipse update site at http://proofgeneral.inf.ed.ac.uk/eclipse/products/. It offered interaction based on the waterfall model, and a high-level overview of individual proofs, but did not provide any support for structured projects.

Conversely, the ProverEditor plugin for Eclipse was only available for Coq. Its project support consisted of automatic Makefile generation and support for invoking `make`—unlike Coqoon, it did not integrate into Eclipse's build system. ProverEditor was discontinued in 2009, when the last update to their GitHub page was made.

Kopitiam targeted the 8.3 series of Coq, which had no structured way of sending and receiving messages: Coq 8.4 introduced an XML-based protocol for executing commands, and Coq 8.5 allows developers to add support for entirely new protocols such as PIDE, but Coq 8.3 only supported interaction through the standard Coq REPL. This was extremely brittle, and required constant polling to read responses from Coq. Kopitiam had no support for Coq projects.

Kopitiam offered one unconventional extension to the waterfall model: it allowed Coq proofs to be interleaved with Java source code. Using aspect-oriented programming to hook into the internals of the Java editor, it added Coq-like controls to step through decorated Java programs: stepping over a Java command would cause it to be 'executed' in an environment based on a separation logic framework built by Bengtson et al. [5]. As the waterfall does not map cleanly onto any Java concepts, this was fragile and difficult to use, but it was an interesting extension—and one which we intend to reintroduce in the future using PIDE.

6.2 PIDE and Asynchronous Editors

With the introduction of PIDE, Wenzel ushered in the third generation of proof assistant interaction: instead of requiring the user to micromanage the system's execution, it allows asynchronous interfaces, such as Coqoon. The flagship application of the PIDE approach for Isabelle is Isabelle/jEdit [26], which is now the standard frontend to the Isabelle system.

Because PIDE and Isabelle/jEdit have been developed in tandem, the editor makes full use of the features we have described in Sect. 3: the editor allows asynchronous interaction with Isabelle, and marks up the proof document using information obtained during interpretation. Isabelle/jEdit has been partially adapted to support Coq by Tankink [4]—the resulting combination being called Coq/jEdit—but this adaptation does not have the full power of an IDE.

jEdit is an extensible text editor, not an IDE, and the way it was extended by Wenzel in order to support entire developments is Isabelle specific. Following the original design of provers of the HOL family, the Isabelle system does not provide a notion of separate compilation: files are just loaded by a single prover instance one after the other, with an option of using concurrent threads to speed up the process. The PIDE protocol is even able to multiplex multiple text buffers to the same prover instance, and expects the prover to sort that out.

The way Coq works is closer to how traditional programming languages work. The Coq compiler can deal with one file at a time, and unrelated files can be processed by different instances of the compiler, possibly in parallel. As a result Coq/jEdit can only work with a single file and relies on the user to provide their own build system for larger projects. Coqoon is able to take care of the entire build process of large developments, even when they include custom Coq plugins, as described in Sect. 2.4.

Another limitation of Coq/jEdit is that, while Isabelle/jEdit maintains a model of proof documents using PIDE, Coq/jEdit does not. The design of Isabelle's language makes it much easier to integrate that model with jEdit's syntax-and-text oriented views. Isabelle's proof language, Isar, is a two-tiered language, that consists of an outer syntax that gives structure to proof documents, the Isar language proper, and numerous inner syntaxes used for specification and proof methods, the most notable being the Higher Order Logic; HOL. The transition between these languages is syntactically indicated using quotation marks. The outer syntax has a simple structure and can easily be parsed by the Scala library of PIDE. The inner syntaxes are more versatile, and the parsing and processing is handled by the Isabelle side of PIDE. It is this outer syntax that is exposed to jEdit. In Coq's language, there is no syntactical separation between the different languages used, making it difficult to implement a Scala-side parser that exposes the structure. (Coqoon's model takes some steps in this direction, but it is necessarily full of special cases and heuristics.) This lack of a Scala-implemented parser for Coq means that jEdit plugins that rely on such a parser do not work.

Finally, because jEdit is not under active development, its plugins have also grown stale, not being updated to new models and tools. For Isabelle/jEdit,

Wenzel already had to change the core of jEdit to allow the PIDE plugin to paint text when semantic information comes in. This means that Isabelle/jEdit is a small fork of jEdit itself, and that it requires its users to install the entire client, instead of just a plugin. Coqoon works on standard Eclipse distributions.

A second client in the PIDE ecosystem is Isabelle/Eclipse [24]. The development of this Eclipse plugin is on hiatus at the moment, but the version that is available, emulates the Isabelle/jEdit interaction model in Eclipse: it does not provide any 'Eclipse-specific' features like project management or compilation of single files. In its current state, it behaves much like Isabelle/jEdit, but using Eclipse to provide the visual elements for the interface.

Clide [22] is another system that builds upon the PIDE architecture for Isabelle. It is a web interface that is mainly aimed at real time collaboration on proof documents. In a similar fashion to Google Docs, several users can work on the same document, seeing each other's modifications and the responses from Isabelle. It supports projects, but only as a way of grouping together collaborations with others; as such, files in a project are not verified when one file changes, and errors in a proof document are not shown until it is opened.

6.3 Another Approach: The ALF Tradition

The ALF proof assistant [19] and its modern-day descendants–chief amongst them Agda [21]—take a rather different approach to the construction of proofs. Whereas Coq proofs consist of a sequence of invocations of tactics, each of which manipulates Coq's internal representation of a proof term, a proof in the ALF tradition consists simply of the finished proof term: the indirect manipulations performed by tactics in the Coq world are replaced by direct modifications of potentially incomplete terms in source files. Compared to ALF-style proofs, Coq proofs are thus somewhat akin to an edit script: they enumerate the steps taken by the user to arrive at a complete proof term, which are analogous to the steps that the user would perform directly in ALF.

Using this approach in practice requires a more intelligent interface than a simple text editor. Agda proofs, for example, are typically written using an advanced Emacs mode equipped with the ability to rewrite regions of the document according to the transformations supported by the prover. However, this mode shares many of the other drawbacks of tools built on extensible editors: in particular, it has no support for project management.

7 Conclusion

This paper presents Coqoon, an IDE for the interactive proof assistant Coq in Eclipse. Coqoon moves away from traditional synchronous proof development and towards an asynchronous model that allows any part of a proof document to be modified and rechecked without having to retract unrelated proofs. It also supports Coq projects that are fully integrated into the Eclipse build system: files can be added, deleted, and moved at will, and Coqoon will track these

changes and rebuild affected files whenever necessary. Coqoon can also make use of the large number of plugins already available for Eclipse, such as the OCaml plugin OcaIDE, turning Coqoon into a complete development environment for even the most complex Coq projects, or the version control plugin EGit.

Coqoon also brings support for Coq projects to other Eclipse projects and plugins, paving the way for complete IDEs for software verification where programs and proofs of their correctness can be maintained within the same project—or even in the same file.

Together, these features represent a significant advance: a truly integrated and comprehensive proof assistant IDE, bringing to the world of proof assistants a workflow that software developers have enjoyed for decades.

References

1. Aspinall, D.: Proof general: a generic tool for proof development. In: Graf, S. (ed.) TACAS 2000. LNCS, vol. 1785, p. 38. Springer, Heidelberg (2000)
2. Aspinall, D., Lüth, C., Winterstein, D.: A framework for interactive proof. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM/CALCULEMUS 2007. LNCS (LNAI), vol. 4573, pp. 161–175. Springer, Heidelberg (2007)
3. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
4. Barras, B., Tankink, C., Tassi, E.: Asynchronous processing of Coq documents: from the kernel up to the user interface. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 51–66. Springer, New York (2015)
5. Bengtson, J., Jensen, J.B., Sieczkowski, F., Birkedal, L.: Verifying object-oriented programs with higher-order separation logic in Coq. In: Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 22–38. Springer, Heidelberg (2011)
6. Boldo, S., Jourdan, J.-H., Leroy, X., Melquiond, G.: A formally-verified C compiler supporting floating-point arithmetic. In: ARITH, pp. 107–115. IEEE Computer Society (2013)
7. Bros, N., Cerioli, R.: OcaIDE. http://www.algo-prog.info/ocaide/
8. Charles, J., Kiniry, J.R.: A lightweight theorem prover interface for eclipse. In: UITP Workshop proceedings (2008)
9. Eclipse Foundation. EGit. http://www.eclipse.org/egit/
10. Eclipse Foundation. Equinox. http://www.eclipse.org/equinox/
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1st edn. 20th printing (1994)
12. Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux, S., Mahboubi, A., O'Connor, R., Ould Biha, S., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., Théry, L.: A machine-checked proof of the odd order theorem. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 163–179. Springer, Heidelberg (2013)
13. Gordon, M.J.C., Melham, T.F.: Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press, New York (1993)
14. Hales, T.C.: Dense Sphere Packings - a blueprint for formal proofs. Cambridge University Press (2012)

15. Harrison, J.: HOL light: an overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 60–66. Springer, Heidelberg (2009)
16. Jacobs, B., Piessens, F.: The VeriFast program verifier. CW Reports CW520, Department of Computer Science, K.U.Leuven (2008)
17. Klein, G., Andronick, J., Elphinstone, K., Murray, T.C., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst. **32**(1), 2 (2014)
18. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
19. Magnusson, L., Nordström, B.: The Alf proof editor and its proof engine. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 213–237. Springer, Heidelberg (1994)
20. Mehnert, H.: Kopitiam: modular incremental interactive full functional static verification of java code. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 518–524. Springer, Heidelberg (2011)
21. Norell, U.: Towards a practical programming language based on dependent type theory. PH.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007
22. Ring, M., Lüth, C.: Collaborative interactive theorem proving with clide. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 467–482. Springer, Heidelberg (2014)
23. The Coq Development Team. The Coq Reference Manual. http://coq.inria.fr/doc
24. Velykis, A.: Isabelle/Eclipse. http://andriusvelykis.github.io/isabelle-eclipse
25. Wenzel, M.: Asynchronous user interaction and tool integration in isabelle/PIDE. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 515–530. Springer, Heidelberg (2014)
26. Wenzel, M.: System description: Isabelle/jEdit in 2014. In: UITP (2014)