Integrating Dependency Building with Document Checking in Coq

Emilio Jesús Gallego Arias
Université Paris Cité, CNRS, Inria, IRIF, F-75013
Paris, France
emilio-jesus.gallego-arias@inria.fr

Bhakti Shah
University of Chicago
Chicago, USA
bhaktishah@uchicago.edu

Recent years have seen a trend towards more integrated tooling in programming environments. For example, Rust's Salsa [2] combines the Rust compiler with an incremental build system in order to provide a query-based reactive architecture that language servers and tools can build on. In this abstract, we explore the integration of a build system with `coq-lsp` [1], an incremental document checking system for Coq. We believe our approach opens the door to significant usability and performance improvements. We guarantee that the user will never `Require` an out of date library, an undesired action that is a common source of frustration among Coq users. On the performance front, we can share `.vo` file parsing among *all* the files in a theory, which saves a significant amount of time. Moreover, this integration allows for users to have very different build strategies tailored to their particular needs. We have implemented a prototype of this system for `coq-lsp`; the implementation relies on algebraic effects in OCaml 5.0 [5], dispatching an effect every time a `Require` statement is found.

1 Building Coq Theories Today

Coq users usually organize their developments (theories) as a set of vernacular `.v` files. Such files, referred to as *libraries*, can be *required* from other files using the `Require` command — in a non-cyclic fashion — allowing users to access a wide range of objects such as definitions, notations, etc. However, before a user can access these objects, the file needs to be processed by `coqc` to turn the text-based `.v` file into a `.vo` file, which is a binary processed format obtained by dumping Coq's runtime object representation.

For many years, Coq users have relied on the `make` build system to fully build their theories. A `coq_makefile` wrapper generates a `Makefile` that first runs `coqdep`, a special-purpose lexer that tries to infer the set of libraries a **.v** file depends on by resolving logical Coq names to file names. The dependency information generated by `coqdep` will then be used by `make` to determine the order in which the `.v` files should be built.

While quite mature, this `make`-based setup has some downsides: `make` itself is limited in today's context. For example, `make` does not clean up stale object files, which is problematic when there are unwanted object files in context that the user is unaware of. Moreover, `make` is based on a notion of timestamps, that prevents many useful optimizations such as build caches. Further, the user must rebuild their project manually every time it is updated.

Proof General [4] does provide a helper to build dependent files automatically. This comes at the cost of re-implementing a small build system inside Emacs, which can be hard to maintain, and is often decoupled from upstream changes in Coq. Editor support for automatically running `make` tends not to be usable, as it is often too heavy.

There exists support for building Coq theories with more advanced build systems such as Dune, a build system for OCaml [3]. While Dune helps with some common `make` pitfalls via a hash-based build and a shared build cache, the user must still manually rebuild their project. This current approach of manual building is cumbersome; it breaks the natural proof writing workflow that the user has settled into. Moreover, we introduce transient unsoundness into the workspace via inconsistencies between the binaries and `.v` files.

Deeper integration between build systems and compilers has been explored recently in languages such as OCaml and Rust; we believe that the Coq user experience may benefit a lot from explorations in this direction.

Our aim is to provide a Coq user experience where everything "Just Works™"; to this end, we integrate `.vo` file building with the `coq-lsp` document manager for Coq.

2 Implementation Outline

We implement our proposal on top of `coq-lsp`, a new document checking engine for Coq based on the notion of an effectful, memoized interpreter. `coq-lsp` has been influenced by the literature on build systems and incremental computing. The process of checking a Coq file interactively with `coq-lsp` can already be seen as the process of *building* a document, and so it is natural to extend it with file-building capabilities.

There are key points that make this process non-trivial. In particular, *coq-lsp* is specialized for the often linear dependency graph that is found inside proof documents, and will avoid propagation of dependencies — which is often very costly in this setup — by preferring a full memoized re-check, assuming all the sentences after a change in the document where impacted by it. This doesn't work well with libraries for two reasons: we cannot assume that a library update impacts all files in the theory, as this is expensive, and doing

```
type _ Effect.t =+ Require_eff : DirPath.t -> VoContents.t Effect.t

let eval_doc ~st = ...
  | AstRequire lib ->
    let vo = Eff.perform (Require_eff lib) in
    let st = Memo.Interp.eval_require ~vo ~st in
    ...
```

Figure 1. Require effect and its use in the interpreter.

this could lead to cycles, which, in the document case, are prevented by construction.

Instead, we use algebraic effects to implement a more classical incremental computing setting. Changes to the interpreter are detailed in Fig.1. The key idea is that we will yield control to an upper layer using the `Require`$_{eff}$ effect, which will at some point resume document checking, providing the interpreter with a handle representing the contents of the file in a way that can be efficiently memoized.

The effect is handled in the *theory manager* component of `coq-lsp`, which knows what files are in scope, and maintains this list in a table. We have extended the information for each file in the theory with a new field, storing information on the files that it is required by. We also use the existing field that signals completion status to mark files as "dirty", to indicate that recompilation will be required. Files can be in one of two states: `Completed`, which indicates that the object file is up to date, or `Stopped`, which indicates that recompilation is required.

Then, when a `Require`$_{eff}$ effect is received, we first register the newly discovered dependency in the file table, if it wasn't already present. We then have two cases: the required file is in a `Completed` state, and its handle can be served, or the file is in the `Stopped` state and requires rebuilding, in which case, we push it to the front of the build queue, and resume checking.

The last bit of our implementation is when the document manager receives a changes request for a file. In this case, we will mark the files that depend on the changed file as `Stopped` at position 0, which will force them to be rechecked when some information is demanded from them. Note that this process is memoized, and hence more efficient than the naive implementation.

3 Perspectives and Future Work

Our implementation is still in a prototype stage; however, we think our strategy does open up the door for further experimentation, and we'd like to gather feedback from the community. In particular, we think there is a wide range of design options in terms of checking strategies that could benefit actual Coq users. For example, we don't necessarily have to wait for the compilation of a `.vo` file to resume the checking of a document that depends on it. We could emit a warning and allow the user to continue working on their file with a warning; when the file is ready, we can (transparently) re-check the document; that may be fast if the changes made in the depender play well with the document cache. Other options are also possible.

There are important work items that we hope to complete soon: for example, we have not provided a graphical UI to provide feedback to the user about the status of file building yet. Our implementation requires OCaml 5.0 and changes to Coq; changes need to be upstreamed to make our setup work with released Coq versions, and Ocaml 4.x (`coq-lsp` has a way to "emulate" effects as to remain compatible with the OCaml 4.x series).

Of particular importance is to ensure that our setup works well with compilation setups where the proofs are skipped. While this is not possible yet, we hope that requiring a file where the opaque proofs have been skipped, and then upgrading this file to fully check proofs, plays well with the memoization strategy, that is to say: theorems are only rechecked if the dependee file really changed the global universe graph in an important way.

Another important thing to understand is the cost of change propagation; this can become a dominant factor for large projects, given how frequent changes are in interactive use. The spectrum of mitigations here is large, and we must be able to measure the cost and associated patterns so as to design and implement a propagation method that works well in practice.

References

[1] coq-lsp, a LSP-compliant document manager for Coq, 2023.

[2] Salsa, a generic framework for on-demand, incrementalized computation., 2023.

[3] Andrey Mokhov, A. A. Memo: an incremental computation library that powers Dune. Presented OCaml 2022, 2022.

[4] Aspinall, D. 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* (2000), S. Graf and M. I. Schwartzbach, Eds., vol. 1785 of *Lecture Notes in Computer Science*, Springer, pp. 38–42.

[5] Sivaramakrishnan, K. C., Dolan, S., White, L., Kelly, T., Jaffer, S., and Madhavapeddy, A. Retrofitting effect handlers onto ocaml. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021* (2021), S. N. Freund and E. Yahav, Eds., ACM, pp. 206–221.