CONSTRUCTOR UNIVERSITY BREMEN

Master of Science
Computer Science

Andrei Kozyrev

CoqPilot: LLM-based generation of Coq proofs

Master's Thesis

Scientific supervisor:
professor Anton Podkopaev

Reviewer:

Bremen
2024

Abstract

We present CoqPilot, a VS Code extension designed to help auto- mate writing of Coq proofs. The plugin collects the parts of proofs marked with the admit tactic in a Coq file, i.e., proof holes, and combines LLMs along with non-machine-learning methods to gen- erate proof candidates for the holes. Then, CoqPilot checks if each proof candidate solves the given sub-goal and, if successful, replaces the hole with it.

The focus of CoqPilot is twofold. Firstly, we want to allow users to seamlessly combine multiple Coq generation ap- proaches and provide a zero-setup experience for our tool. Secondly, we want to deliver a plat-form for LLM-based experiments on Coq proof generation. We developed a benchmarking system for Coq generation methods, available in the plu-gin, and conducted an ex- periment using it, showcasing the framework's possibilities.

Contents

Introduction

Memory model is an abstraction, invented to describe the behavior of a program in a multithreaded environment. A memory model describes the behavior of a concurrent program on a particular system. A memory model dictates how memory operations interact with each other.

A well-established approach to define a memory model is to use declarative semantics, where a program execution is depicted as a graph, where nodes represent memory operations and edges denote the order on these operations. Some examples of binary relations between instructions are *Program Order*, which sequantially binds events in one thread, and *reads-from*, which relates writes to reads, reading from them.

The most traditional and conservative memory model is *Sequential Consistency* (SC) [6]. A nice way to think about SC is as a switch. At each time step, the switch selects a thread to run, and runs its next event completely.

> *"The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each processor appear in the order specified by its program."* (Leslie Lamport (1979))

However, SC fails to describe the real-world concurrent systems due to compiler and CPU optimizations. We can only run a single instruction at a time, so we lose the benefit of running a program on multiple threads. Due to that, models that are more complex, but yield better performance, are developed and used in practice. Memory models that are more relaxed than Sequential Consistency are called *weak memory models*. While weak memory models give compilers and processors more flexibility to optimize, they introduce more problems to the programmer.

If we reason about weak memory models in a declarative way, we use relations to describe models and facts about them [2]. As memory models are stated with relational expressions, proofs of various model properties

are also done reasoning in relational language. Proofs of such propositions are typically massive and very error-prone. There were several cases of incorrect results in submitted and published papers. Batty et al. [3] suggested an incorrect fix for the semantics of SC calls in C++, which was later documented by Lahav et al. [5]. Moreover Pichon-Pharabod et al. in 2016 suggested an incorrect proof of compilation in their paper [7].

Given that writing and maintaining proofs about Weak Memory on paper is very difficult, it is considered good practice to write them in Coq [4], which is a proof assistant system. Coq helps to grant the correctness of the proof process. In Coq, proofs are expressed as formal mathematical objects, and the correctness of the proof is ensured by checking that the proof is consistent with the axioms and rules of logic. Coq is a tool that facilitates the creation of mechanically-verifiable proofs, in contrast to paper-based proofs which are prone to errors.

Weak memory proofs, even with the use of Coq, tend to be huge and convoluted. This is why in this thesis, we are focusing on automating a specific part of weak memory proofs, namely the proofs of equivalences between several memory models. As already mentioned, we can define memory models as axioms over relational language. Consider an example of a proposition about relations. Let us denote `r, r'` $\subset$ `A` $\times$ `A` — two relations over a given set `A`. Now we may consider common operations in relational language, e.g. transitive closure (`r*`), reflexive closure (`r?`), or composition (`r ;; r'`). An example of a proposition we want to prove may look like this:

$$\texttt{(r* ;; r?) ;; r?} \equiv \texttt{r*}$$

If we were to prove this statement in Coq by hand, we would have to consequently apply multiple theorems to rewrite both sides of the equivalence relation until they are syntactically equal. In this particular example, we would have to apply the theorem `rt_cr` twice:

```
rt_cr : forall (A : Type) (r : relation A), r* ;; r? = r*
```

After `rt_cr` is applied to the left-hand side (lhs) two times, the two sides become syntactically equal. `rt_cr` and other theorems that help to reason about relations are provided by the Hahn library [1].

The general problem discussed in this thesis is as follows: given a rewriting system, i.e. a set of theorems, and an equivalence relation between two expressions, we want to find a sequence of rewrites that can be applied to both sides of the relation to make them syntactically equal. This is a wider problem than reasoning about relational expressions, thereafter the solution we propose could be adapted and used to solve other problems that involve derivability in two-sided associative calculus over a given language in Coq. This thesis, in turn, focuses on applying the proposed technique to automate weak memory, where the rewriting system we use is the Hahn library.

As a basis for various proofs in the area of weak memory, the weakmemory[1] organization on GitHub, specifically the imm[2] repository, was utilized. The work on the thesis involves analyzing present proofs and their possible patterns to automate the process and attempt to shorten them using the developed tool.

1.1 Approach

Our approach to solving equivalences is based on the technique called *equality saturation* which utilizes the data structure, called an *e-graph*. E-graph stores an equivalence relation over terms of some language and allows to store potentially exponential amount of terms in a compact way. E-graph is a set of equivalence classes (*e-classes*) and e-class is a set of *e-nodes*, which represent equivalent terms in the language. Whilst e-nodes are function symbols, associated with a list of e-classes.

Equality saturation is a process of iteratively applying a set of rewrite rules to the e-graph until rewrites bring no more new information to the

[1] https://github.com/weakmemory
[2] https://github.com/weakmemory/imm

graph. That point is called *saturation*. The saturated e-graph represents a set of all possible terms equivalent to the origin, that can be obtained by applying the rewrite rules to the initial term.

In a saturated e-graph it becomes algorithmically easy to check if two terms are equivalent and, moreover, to find a proof of their equivalence. Having a sequence of rewrites, we can apply them within the Coq proof view. Just as we would do by hand, applying the theorems one by one.

References

[1] Viktor Vafeiadis et al. "Hahn: a Coq library that contains a useful collection of lemmas and tactics about lists and binary relations". In: (2018). URL: https://github.com/vafeiadis/hahn.

[2] Jade Alglave, Luc Maranget, and Michael Tautschnig. "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory". In: *ACM Trans. Program. Lang. Syst.* 36.2 (July 2014), 7:1–7:74. ISSN: 0164-0925. DOI: 10.1145/2627752. URL: http://doi.acm.org/10.1145/2627752.

[3] Mark Batty, Alastair F. Donaldson, and John Wickerson. "Overhauling SC atomics in C11 and OpenCL". In: (2016). URL: https://doi.org/10.1145%2F2837614.2837637.

[4] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[5] Ori Lahav et al. "Repairing Sequential Consistency in C/C++11". In: *SIGPLAN Not.* 52.6 (2017), pp. 618–632. ISSN: 0362-1340. URL: https://doi.org/10.1145/3140587.3062352.

[6] Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Transactions on Computers* C-28.9 (1979), pp. 690–691. DOI: 10.1109/TC.1979.1675439.

[7] Jean Pichon-Pharabod and Peter Sewell. "A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-Air Executions". In: *SIGPLAN Not.* 51.1 (2016), pp. 622–633. ISSN: 0362-1340. DOI: 10.1145/2914770.2837616. URL: https://doi.org/10.1145/2914770.2837616.