CONSTRUCTOR UNIVERSITY BREMEN

Bachelor of Science
Computer Science

Andrei Kozyrev

Equality saturation for solving equalities of relational expressions

Bachelor's Thesis

Scientific supervisor:
professor Anton Podkopaev

Reviewer:

Bremen
2023

Abstract

Modern CPUs are being developed exceptionally fast, and the number of cores is increasing rapidly. This has led to the development of multithreading, which is a technique that allows for the execution of multiple threads on a single CPU. Memory models are a fundamental aspect of multithreading and describe how memory is ordered at runtime in relation to source code. Currently, existing memory models are unsatisfactory and there is a need for new models that can be rigorously proven. In order to achieve this, formal verification using the Coq proof assistant is utilized, which enables automated proof checking and ensures the accuracy of results. Specialists in weak memory are continuosly improving the results in this domain.

One of the big and common problems in weak memory is the proof of equivalence of several memory models. Memory models are represented as expressions over relational language.

This thesis focuses on the automation of proving equalities over relational expressions in Coq. We are utilizing the techniques of equality saturation and E-graph data structure to generate proof of equaivalence for a given pair of terms. By automating these proofs, we can greatly increase the efficiency and accuracy of the proof process in weak memory.

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

1 Introduction

1.1 Motivation

Weak memory specializes on research that aims to improve the results in memory modeling. Weak memory focuses on determining the relative strength of different models. This is why one of the common challenges is to show equaivalences between several models. Given two memory models `A` and `B` we might want to check if model `A` is stronger then model `B`. If that condition holds, then the consistency of the model `A` would imply the consistency of the model `B`.

Memory models and propositions about them are usually represented as expressions over relational language. However, proofs of such the propositions are typically massive and very error prone. There were several cases of incorrect result in submited and published papers. Batty et al. [4] suggested an incorrect fix for the semantics of SC calls in C++, which was later documented by Lahav et al. [8]. Moreover Pichon-Pharabod et al. in 2016 suggested an incorrect proof of compilation in their paper [11].

Weak memory proofs are usually written in Coq [5], which is a proof assistant system. Coq helps to grant the correctness of the proof process. The underlying language of Coq is called *Calculus of Inductive Constructions* (CIC). In CIC, 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 the standard of development in the field of weak memory, helping to unify and verify already complex proofs.

Nevertheless, weak memory proofs, even with a 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 equaivalences between several memory models. As already mentioned, memory models are defined as relations. Let's denote `r` and `r'` the two memory models or relations over a given set `A`. Now we may consider common op-

erations 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 thorem `rt_cr` twice:

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

After `rt_cr` being 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 equaivalence 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 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.

1.2 Approach

Our approach to solve 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 equaivalence 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 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 equaivalent and, moreover, to find a proof of their equivalence. Having a sequence of rewrites, we can apply them within Coq proof view. Just as we would do by hand, applying the theorems one by one.

1.3 Core implementation components

We have chosen fast and lightweight `egg` [12] rust library as an equality saturation engine for our project. `Egg` is a new and ambitious project, that has already proven its worth and was used in tools like *Ruler* [9].

In Coq you can add new functionality by writing a plugin. Plugins are written in Ocaml language. We have set the communication between Ocaml and Rust using the `ocaml-rs` [2] foreign function interface (FFI). To prove theorems in Coq, you use proof mode, which is activated when you start a proof, for example, by using the `Theorem` command. The proof state contains one or more unproven goals. Each goal consists of a conclusion, which is the statement to be proven, and a local context. When the proof is complete, you exit proof mode by using the `Qed` command. During proof mode, you use so called *tactics*, to gradually transform the proof. Tactics specify how to transform the proof state of an incomplete proof and are used to eventually generate a complete proof.

Commands in Coq are similar to tactics, but accessible from outside of proof mode, e.g. command `Print`, which prints the given term to console. A Coq plugin typically consists of a set of commands and tactics, that are

used from Coq as a front-end to the plugin. We have defined multiple tools inside our plugin, that are used to interact with the `egg` library:

- `Cegg solve` — a tactic that simplifies the lhs of the equation. It takes the conclusion of the goal, retrieves the lhs of the equivalence, parses it into an s-expression[1] and passes it to `egg`. `Egg` builds an e-graph E for it and extracts the "best" term t from E. A sequence of rewrites to achieve t is passed back to Ocaml and is applied to the lhs of the equivalence inside Coq proof mode.
- `Cegg solve eq` — a tactic that tries to prove the equivalence between the lhs and rhs of the equation, using `egg`.
- `Cegg config` — a command, that allows to configure the ruleset for `egg`. It takes a user-defined list of rewrite rules and caches it for the later use in `Cegg solve` and `Cegg solve eq`.

The source code and documentation for the thesis is available at:

https://github.com/K-dizzled/relations-via-egg

[1] S-expressions represent tree-like data

2 Objectives

This thesis aims to automate solving equaivalences over relational expressions in Coq. The goal is to manage to use the `egg` library and automate the proof process. The main objectives are:

- Develop a framework for communication between Rust and the `Coq` interactive proofs. We need to connect two completely stadalone systems, Coq and `egg`, together to enable their communication and data interchange.

- Make a rule-parameterized algorithm on top of the `egg` library for producing a series of rewrites, which prove relational equivalence. Given two expressions and a rewriting system, there may be multipl eapproaches to prove the equivalence. The naive approach to build an e-graph for one expression and search for another may have problems unrevealed in Chapter ?.

- Experiment on existing proofs and used lemmas to come up with a usable and efficient rule set. Firstly, to analyse existing weak memory proofs and provide users a useful interface. Secondly, as big rule sets result in huge e-graphs, that take long to be built, we aim to research on the efficient and substantive rule set, that would be small enough to be used to build e-graphs in practice.

3 Related Work

3.1 Coq

This section aims to provide an overview of Coq and its significance in the field of theorem proving. We will outline the advantages of using Coq compared to other proof assistants and methods. Additionally, we will delve deeper into the Coq proving process.

3.1.1 Coq Overview

Coq is a formal proof management system. It provides a language called Gallina, which is used to define mathematical objects and write formal proofs. The formalism behind Coq is the Calculus of Inductive Constructions (CIC) [10]. In CIC types are used to ensure the correctness of proofs. Each theorem is essentially a type, and its proof is a value inhabiting that type. It might also be useful to think of proofs as functions from hypothesis to conclusion. For example, assume we have a context Γ, a proposition φ and we want to prove $\Gamma \vdash \varphi$. That would mean that the proof we are looking for is a mapping, which for any argument of type Γ constructs a value of type φ.

3.1.2 Coq Proof mode

As mentioned earlier, proofs can be viewed as functions and this is one of the ways to make a proof in Coq. We will provide an example for a better understanding. Consider the following theorem: given value of type A and a function $f : A \to B$, we can construct a value of type B.

```
Definition test (A B : Prop) :
  A -> (A -> B) -> B.
```

To prove such theorem we need to apply the value a of type A to the function f and conclude the proof:

```
Definition test (A B : Prop) :
  A -> (A -> B) -> B
```

```
:= fun (a : A) (f : A -> B) => f a.
```

Constracting proof terms by hand may be useful to learn Coq, but it is very inconvenient to write bigger proofs in such manner. In a bigger proof, as like as in paper proofs, you want to iteratively modify the environment and step by step achieve the goal. Coq enters proof mode when you begin a proof, such as with the `Theorem` command:

```
Theorem test (A B : Prop) :
  A -> (A -> B) -> B.
Proof.
```

When you enter a proof mode, you are able to always see current unfinished goals and an up-to-date hypothesises:

```
A, B : Prop
============================
A -> (A -> B) -> B
```

Now we can proof the theorem using tactics. Tactics implement backwards reasoning, so when tactic is applied, all hypothesises it uses to modify the goal are added to the context.

Firstly, we call an `intros` tactic, which introduces all propositions on the left side of the implication as assumptions:

```
A, B : Prop
H : A
H0 : A -> B
============================
B
```

We have a hypothesis H of type A, but we need B. We call an `apply H0` tactic, which proves us B, but adds A as a new goal. Now if we call `apply H`, we will conclude the proof.

```
Lemma test' (A B : Prop) :
  A -> (A -> B) -> B.
Proof.
  intros.
```

```
  apply H0.
  apply H.
Qed.
```

3.1.3 Coq's Significance

There are other Proof Assistants continuing to appear, but Coq has been developed and improved since 1989. The heart of a proof assistant is its kernel, which is the primary source of reliability of the tool. Unfortunately, regarding of the kernel's size, as any other software, it has issues. However, the amount of work done by the Coq community to recheck and validate the kernel, makes Coq the most trustworthy proof assistant available.

Moreover, Coq's huge strength is the amount of impressive results, obtained using it. The most famous result, achieved using Coq, is the Four Color Theorem proof. The Four Color Theorem states that any map can be colored using only four colors, so that no two adjacent regions are colored the same. There were several paper proofs, that were all proven to be incorrect after a while. A computer-assisted proof was proposed by Kenneth Appel and Wolfgang Haken in 1976 [3]. The proof reduced the problem to the analysis of a smaller number of options and their enumeration by a computer for many hours. Consequently, the mathematical community was not inclined to trust the proof. The first formal checked proof was proposed by George Gonthier et al. in 2005 [6], which showed the community that Coq is ready for huge and complex proofs.

3.1.4 Proof Automation

This thesis focuses on automating the proofs, whilst a question may arise whether such approach makes proofs less clear to read. There are several proof styles in Coq. A framework called `SSReflect`[2] exists. Conceptually, `SSReflect` differs from ordinary `Tactics`[3] in that proofs are written with almost no automation, and the tactics language is much more expressive.

[2] https://coq.inria.fr/refman/proof-engine/ssreflect-proof-language.html
[3] https://coq.inria.fr/refman/proof-engine/tactics.html#tactics

Even though `SSReflect` has less automation, proofs anyway tend to be confusing. From the other hand, when classical Coq `Tactics` are used, proof is usually broken down into a huge number of lemmas and sub-claims, definitions of which make the idea clear. Lemmas that are deep down inside the proof can typically be automated without sacrificing the readability, because their proofs are self-evident.

3.2 Solutions

In 2023 Kokologiannakis et al. presented the tool called `Kater` [7]. `Kater` is a sound and automated way to answer memory-model questions. `Kater` is a useful, but standalone tool. Firstly, that means we must believe in its correctness, whereas any functionality integrated into Coq is protected by the Coq's typechecker from producing incorrect results. Secondly, proofs in weak memory are ofter more diverse than just reasoning about relational expressions. For instance, we may want to prove correctness of the compilation scheme, e.g. from C to Assembly language. In such cases, the semantics of the assembly code must be related to the original operational semantics of the instructions in our language. While relations are still relevant, they are only a small part of the problem. The main focus is on proving properties about the compilation process. Therefore, `Kater` is suitable only for a limited set of problems, where relations are the primary concern, and the proof process does not involve other complex components. In contrast, Coq provides a comprehensive framework for formal reasoning that allows us to tackle a wide range of problems.

3.3 Equality Saturation

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] Zach Shipko et al. "ocaml-rs: OCaml extensions in Rust". In: (2021). URL: https://crates.io/crates/ocaml.

[3] Kenneth I. Appel. "The Use of the Computer in the Proof of the Four Color Theorem". In: *Proceedings of the American Philosophical Society* 128.1 (1984), pp. 35–39. ISSN: 0003049X. URL: http://www.jstor.org/stable/986491 (visited on 03/29/2023).

[4] 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.

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

[6] Georges Gonthier. "Formal Proof—The Four- Color Theorem". In: 2008. URL: https://www.ams.org/notices/200811/tx081101382p.pdf.

[7] Michalis Kokologiannakis, Ori Lahav, and Viktor Vafeiadis. "Kater: Automating Weak Memory Model Metatheory and Consistency Checking". In: *Proc. ACM Program. Lang.* 7.POPL (2023). DOI: 10.1145/3571212. URL: https://doi.org/10.1145/3571212.

[8] 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.

[9] Chandrakana Nandi et al. "Rewrite Rule Inference Using Equality Saturation". In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021). DOI: 10.1145/3485496. URL: https://doi.org/10.1145/3485496.

[10] Christine Paulin-Mohring. "Introduction to the Calculus of Inductive Constructions". In: *All about Proofs, Proofs for All*. Ed. by Bruno Woltzenlogel Paleo and David Delahaye. Vol. 55. Studies in Logic

(Mathematical logic and foundations). College Publications, Jan. 2015. URL: https://hal.inria.fr/hal-01094195.

[11] 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.

[12] Max Willsey et al. "Egg: Fast and Extensible Equality Saturation". In: *Proc. ACM Program. Lang.* 5.POPL (2021). DOI: 10.1145/3434304. URL: https://doi.org/10.1145/3434304.