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

Statutory Declaration

Family Name, First Name	Kozyrev, Andrei
Matriculation number	30006595
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

. . . .

Bremen, May 16th, 2023 Andrei Kozyrev

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. 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 continuously 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

I would like to express my deepest gratitude to my Academic Supervisor Anton Podkopaev for his invaluable guidance, expertise, and unwavering support throughout the entire duration of this thesis. His weekly meetings, interest in the topic, and willingness to provide advice and assistance made the research process both stimulating and enjoyable. With such an academic supervisor, the work was a joy and a huge pleasure.

Additionally, I would like to extend my sincere thanks to my supportive friends Ivan Klimov, Semen Panenkov, Maksim Martynov, Nikita Mittcev and Ivan Kabashnyi, who provided invaluable assistance with specific aspects of my research contributed insightful ideas and offered unwavering emotional support throughout the entire process. Their encouragement and support were instrumental in keeping me motivated and focused.

Finally, I would like to thank my family for their unconditional love and support. Their unwavering belief in me and my abilities has been a constant source of inspiration and motivation, and I am truly grateful for their presence in my life.

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) [10]. 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 [3]. 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. [5] suggested an incorrect fix for the semantics of SC calls in C++, which was later documented by Lahav et al. [9]. Moreover Pichon-Pharabod et al. in 2016 suggested an incorrect proof of compilation in their paper [15].

Given that writing and maintaining proofs about Weak Memory on paper is very difficult, it is considered good practice to write them in Coq [6], 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.

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

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 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.

2 Objectives

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

- Develop a framework for communication between Rust and `Coq` interactive proofs.
- 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 multiple approaches to prove the equivalence.
- Experiment with existing proofs and used lemmas to come up with a usable and efficient rule set. Firstly, to analyze existing weak memory proofs and provide users with a useful interface. Secondly, as big rule sets result in huge e-graphs, that take a long to be built, we aim to research the efficient and substantive rule set, that would be small enough to be used to build e-graphs in practice.

3 Related Work

First part of this chapter focuses on introducing the reader to Coq proof assistant, its core concepts and significance. Then other solutions to the problem are discussed in Section 3.2. Finally, we focus on the details of our particular approach and cover details regarding equality saturation and the `egg` library in Section 3.3.

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) [14]. 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 a 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.
```

Constructing proof terms by hand may be useful to learn Coq, but it is very inconvenient to write bigger proofs in such a 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 the current unfinished goals and an up-to-date hypothesizes:

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

Now we can prove the theorem using tactics. Tactics implement backwards reasoning, so when a tactic is applied, all hypothesizes 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 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

Other Proof Assistants are continuing to appear, e.g. Lean, Agda, Arend, 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 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, testimony to 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 [4]. 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 [7], 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 an approach makes proofs less clear to read. There are several proof styles in Coq. A framework called `SSReflect`[3] exists. Conceptually, `SSReflect` differs from ordinary `Tactics`[4] in that proofs are written with almost no automation, and the tactics language is much more expressive. Even though `SSReflect` has less automation, proofs anyway tend to be confusing. On the other hand, when classical Coq `Tactics` are used, the 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 Alternative solutions to solving relational equations

In 2023 Kokologiannakis et al. presented a tool called `Kater` [8]. `Kater` is a tool that allows one to automatically 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 Coq's type checker from producing incorrect results. Secondly, proofs in weak memory are often 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 the properties of 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

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

tackle a wide range of problems.

3.3 E-graphs and Equality Saturation

E-graphs are a generalization of a set-union data structure [16]. E-graphs were first introduced by Greg Nelson and Derek C. Oppen in 1980 [13]. Since then, e-graphs have been used for successful mechanical theorem proving and program verification, e.g. in Stanford Pascal Verifier in 1979 [11].

We will now formally define an e-graph. Let Σ be a set of function symbols. Let Σ_n be a subset of Σ consisting of symbols of arity n. Let Id be a set of unique identifiers $\text{id}_1, \text{id}_2, \ldots, \text{id}_k$, called the e-class **Id**s. Then **e-node** is a function symbol $f \in \Sigma$ and a list of n e-class Ids. E-node is denoted $f(\text{id}_1, \text{id}_2, \ldots, \text{id}_n)$.

Definition 3.1 *(Definition of an E-graph [17]). An e-graph is a tuple (U, M, H) where:*

- *U is a union-find data structure over e-class Ids.*
- *M is a map from e-class Ids to e-classes.*
- *H is a map from e-nodes to e-class Ids.*

E-graphs provide an interface that is a superset to DSU. Operations `add`, `find` and `merge` allow to query the data structure and `ematch` allows to perform an e-matching [12] algorithm that searches for the pattern in the e-graph. The pattern is an expression with variable placeholders, which is used in rewrites, e.g. in rewrite $\texttt{x} \times \texttt{2} \longrightarrow \texttt{x} \ll \texttt{1}$, `x` is the pattern variable. Information about the e-graph invariants can be found in egg paper [17].

Here we give an example of how a simple e-graph is built. Consider such a simple language:

```
enum SimpleLanguage {
    Num(i32),
    ''+'' = Add([Id; 2]),
    ''*'' = Mul([Id; 2]),
    ''/'' = Div([Id; 2]),
    ''<<'' = Bitwise([Id; 2]),
    Symbol(Symbol),
}
```

It is denoted as if we were to use it in `egg` library. Elements of the enum are function symbols with a name and a given arity. Notation explained:

$$\underbrace{\texttt{''+''}}_{1} \; \texttt{=} \; \underbrace{\texttt{Add}}_{2}\texttt{(}\underbrace{\texttt{[Id; 2]}}_{3}\texttt{)}$$

1 — A string literal to automatically generate a parser for the language

2 — A unique identifier of a function symbol

3 — Arity of the function symbol

Let us now consider the following arithmetic expression: $\texttt{(a} \times \texttt{2) / 2}$. The resulting e-graph is shown in Figure 1. E-classes are shown in dashed boxes and e-nodes are circles with their function symbols. Edges represent the parent-child relationship between e-nodes. If the node represents an operator, its children are the arguments. In Figure 1, no rewrite rules were yet added, so each e-node is in its own e-class.

When the e-graph is built we can run an equality saturation algorithm. Consider a rewriting system S, containing rules of form **`lhs`** $\longrightarrow$ **`rhs`**, where **`lhs`** and **`rhs`** are expressions in our language. On a high level algorithm does the following:

1. Iterate over rules $r \in S$. For each rule r:

    (a) Try to match the pattern from r's **`lhs`** with e-nodes present in the graph.

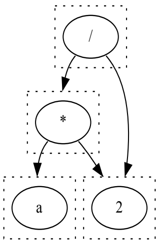

Figure 1

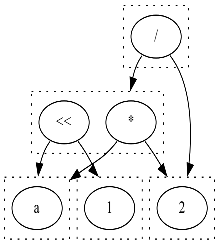

Figure 2

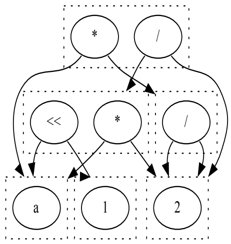

Figure 3

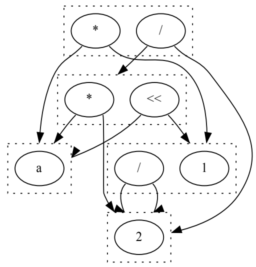

Figure 4

   (b) If a match was found: if `rhs` was not present in the graph, add it. Then merge `lhs` and `rhs` e-classes.

2. Loop step 1 until each new iteration introduces new equivalences into the graph. After the process is finished, graph is called saturated.

Figures 1–4 illustrate how the algorithm proceeds. In Figure 2 the rule $\texttt{x} \times \texttt{2} \longrightarrow \texttt{x} \ll \texttt{1}$ was applied. In Figure 3 the rule $\texttt{(x} \times \texttt{y)/z} \longrightarrow \texttt{x} \times \texttt{(y/z)}$ was applied. In Figure 4 rules $\texttt{x/x} \longrightarrow \texttt{1}$ and $\texttt{1} \times \texttt{x} \longrightarrow \texttt{x}$ were applied.

4 Implementation

Let us summarize the technical problem and the main components of the developed system. Given a proposition about relations in Coq, we want to prove it using equality saturation, performed by the `egg` library in Rust.

A Coq plugin is a tool with external functionality, added to Coq. There are two main approaches to writing Coq plugins:

- `Ltac`[5] and `Ltac2`[6] are languages that help to write simple plugins, combining basic combinations of tactics and actions into a single tactic. It is useful, but not powerful enough to write complex plugins.
- The most traditional way of building new complex tactics is to write a Coq plugin in OCaml.

Coq's compiler is written in OCaml, so plugins written in OCaml allow to extend Coq's grammar, along with adding complex logic to the tactic, e.g. using FFI to call external libraries. That is exactly what we need. To have a better understanding of how all the components interact with each other in our work, we provide a diagram in Figure 5.

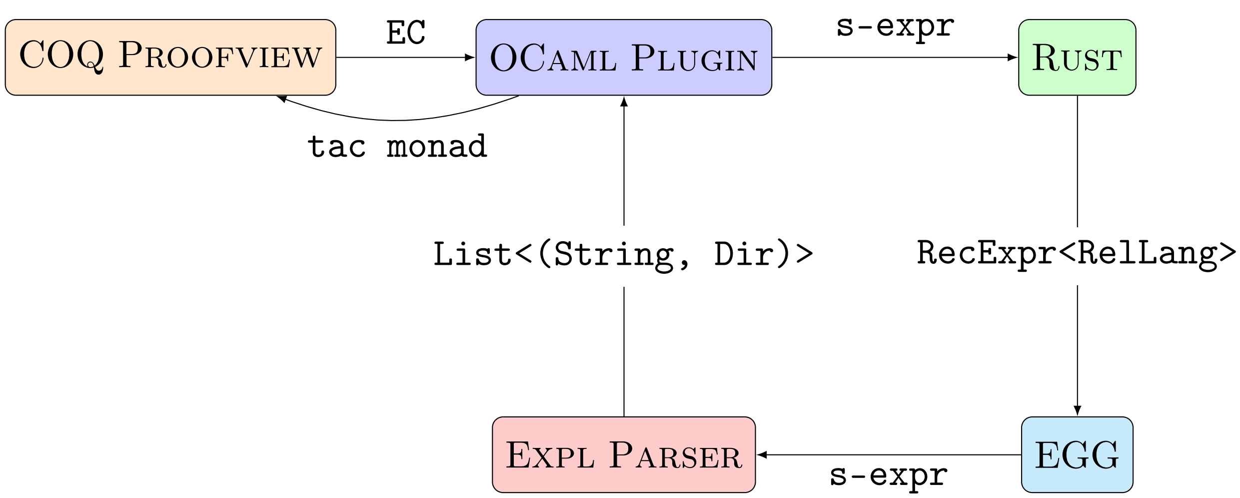

Figure 5: Components of the system

[5] https://coq.inria.fr/refman/proof-engine/ltac.html#ltac
[6] https://coq.inria.fr/refman/proof-engine/ltac2.html#ltac2

A term with a proposition to prove is extracted from Coq Proofview using Coq-Api. We receive it in OCaml as an EConstr (EC), which is a basic type to store terms in Coq. Expression is parsed into a smaller custom type and transferred to Rust. Rust makes data ready to be used by egg and runs an equality saturation algorithm. After that, the proof is constructed and returned to OCaml, conveniently packed. Finally, OCaml performs changes in Coq Proofview.

4.1 Vernacular Commands

To extend Coq's grammar with a new tactic or command, one should write a `.mlg` file, where the tactic's syntax is defined. Consider an example:

```
DECLARE PLUGIN "coq-via-egg-plugin.plugin"

VERNAC COMMAND EXTEND cegg_config CLASSIFIED AS QUERY
| [ "Cegg" "config" reference(r) ] -> { ... }
END

TACTIC EXTEND cegg_solve
| [ "Cegg" "solve" ] -> { ... } (* Paring and interpretation rule *)
END
```

In listing 4.1, we define a command called `cegg_config` and a tactic called `cegg_solve`. A command is marked as `QUERY`, meaning it is a *pure* function. Otherwise, it would have been marked as `SIDEFF`.

Definition 4.1 (Pure function) *A pure function is a function where the return value is only determined by its input values, without observable side effects.*

After the `|` symbol, we define the parsing rule and the interpretation rule, separated by `->`. The parsing rule itself is a set of terminals that are matched against a string of tokens. The interpretation rule is a function that is called when the parsing rule is matched. More on the `.mlg` file format can be found in the Coq's plugin guide [7].

[7] https://github.com/coq/coq/blob/master/doc/plugin_tutorial/tuto2/src/g_tuto2.mlg

4.2 OCaml plugin

When our tactic is called from inside the Coq proof, firstly, we need to extract the goal. Consider an example:

```
Lemma example (r : relation A) :
    r^* ;; r^? ≡ r^*.
Proof.
    Cegg solve eq.
```

When we enter the interactive proof process, we see the following proof state:

```
A : Type
r : relation A
============================
r^* ;; r^? ≡ r^*
```

Current hypothesizes are located above the line and the conclusion of the goal — is below. To interact with Coq from OCaml, we use the Coq-Api[8], which provides the widest spectrum of functionality. To extract that information about the goal in OCaml we first enter the Proofview monad.

Definition 4.2 (Monad) *Monads serve as a representation of computations. Consider a computation to be similar to a function that transforms an input into an output, but with an additional component. This component represents the effect that the function has as a consequence of being executed.*

In the following function, `t` denotes the goal. `enter` applies the goal-dependent tactic, with `(t -> unit tactic)` type in each goal independently.

```
val enter : (t -> unit tactic) -> unit tactic
```

Now we see that our tactic, all in all, should be a function that takes a goal (`Proofview.Goal.t`) as input and return a tactic monad (`unit tactic`). When we have such a function, `enter` will apply it to each goal.

[8]https://coq.github.io/doc/V8.16.0/api/coq-core/index.html

Having a `gl : Proofview.Goal.t` as input, we need to split it into the conclusion, hypotheses, and the environment, using functions `concl`, `hyps`, and `env` respectively. The most import for us is the `concl`, which we will get. It has an `EConstr.constr`, which is the most important datatype in Coq, namely the kernel term. It is used to represent expressions, which are built using a set of constructors that correspond to the different types and operations.

Our next task is to prepare the conclusion for the `egg` use. `Econstr.t` (similar to `Econstr.constr`) has a huge list of constructors, but we are not expecting to handle most of them. For example, constructors such as `Forall` or `LetIn` are not something we expect to see in a proposition about relations. From the limitations of what kind of rules we can pass to `egg` we can infer that we want to handle particular relations and various operations on them, i.e. applications of functions. Moreover, after analyzing the Hahn library and the imm code base, we have decided to add some concrete relations as constants: An empty relation, denoted as `(fun_ _ => False)`, which means that any two elements of the given set are related, and a full relation, denoted as `(fun _ _ => True)`. Thereby the constructors we are interested in are as follows:

```
type Econstr.constr =
    | App of 'constr * 'constr array
    | Var of Names.Id.t
    | Lambda of Names.Name.t Context.binder_annot * 'types * 'constr
    | Ind of Names.inductive * 'univs (* Inductive constructors, e.g. True or
       False *)
```

The conclusion of the goal is split by the equivalence sign into the `lhs` and the `rhs` of the equation. Both sides are individual `Econstr.t`'s that will be passed to Rust as two terms separately. `Econstr.t` is parsed into a smaller type. If unexpected constructors occur in the expression, an exception is raised and an error is shown to the user. Inductive constructors are handled only in case of `True` or `False`, which are parsed as Symbols. A Data type is an s-expression over strings, with addition of lambdas:

```
type goal_s_expr =
```

```
| Symbol of string
| Application of string * goal_s_expr list
| Lambda of goal_s_expr * goal_s_expr
```

The next step is to pass an object of type `goal_s_expr` to Rust for further processing. We use the `ocaml-rs` [2] Rust library to set up communication between OCaml and Rust. A similar type as `goal_s_expr` is defined in Rust:

```
pub enum GoalSExpr {
    Symbol(String),
    Application(String, LinkedList<GoalSExpr>),
    Lambda(Box<GoalSExpr>, Box<GoalSExpr>),
}
```

4.3 Using Egg

Now we want to define an e-graph, which `egg` will operate with. EGraphs are parameterized over the Language given by the user. We define a language in the same notation as introduced in Section 3.3. We denote the following language, which represents operations on relations:

```
define_language! {
    pub enum RelLanguage {
        ''top'' = Top,                       // Full relation
        ''bot'' = Bot,                       // Empty relation
        ''complete_set'' = CompleteSet,      // Full set
        '';;'' = Seq([Id; 2]),               // Relational composition
        ''+'' = CT(Id),                      // Transitive closure
        ''?'' = RT(Id),                      // Reflexive closure
        ''*'' = CRT(Id),                     // Transitive-reflexive closure
        ''eqv_rel'' = Eqv(Id),               // Equivalence relation
        ''clos_sym'' = CS(Id),               // Symmetric closure
        ''-1'' = Transpose(Id),              // Relational transposition
        ''clos_refl_sym'' = CRS(Id),         // Symmetric-reflexive closure
        ''||'' = Union([Id; 2]),             // Union of relations
        ''&&'' = Inter([Id; 2]),             // Intersection of relations
        ''sminus'' = SetMinus([Id; 2]),      // Set difference
        Symbol(Symbol),                      // Concrete relations
    }
}
```

Rust component receives two terms (`lhs` and `rhs` of the goal conclusion) as input and translates both of them into `RelLanguage`. Then `egg` builds an e-graph for the `lhs`. After that, we aim to saturate the e-graph with equivalences, so that it will represent a set of relations, equivalent to the `lhs`. The equality saturation algorithm is parameterized with a set of rules. We take useful theorems about relations from the Hahn library and define a rewriting system in `egg`'s notation:

```
vec![
    rewrite!(''ct_rt''; ''(;; (+ ?r) (* ?r))'' <=> ''(+ ?r)''),
    rewrite!(''rt_ct''; ''(;; (* ?r) (+ ?r))'' <=> ''(+ ?r)''),
    rewrite!(''cr_seq''; ''(;; (? ?r) ?r')'' <=> ''(|| ?r' (;; ?r ?r'))''),
    // ...
]
```

A rewriting system consists of named rules with patterns to search for in the e-graph and terms to replace them with. `?r` denotes an arbitrary term named `r` and the `<=>` sign is a syntactic sugar for a pair of rules: `a <=> b` is equivalent to `a => b` and `b => a`. Currently, we have chosen 51 rules for the system. This is a trade-off between the completeness of the system and the size of the e-graph it produces (more in Section 4.4). When the set of rewrites is provided we run the equality saturation algorithm. It iteratively searches for an `lhs` pattern to apply and expands the e-graph. More on equality saturation in `egg` could be found in the paper [17] and a pseudocode is given in `egg`'s tutorial [9]. Having a saturated graph, to check two terms for equivalence we can call a built-in function `egraph.equivs(expr1, expr2)`.

4.4 Naive implementation complexity

If we follow the naive algorithm strategy to build an e-graph, we can bump into a problem. Unfortunately, choice of the rules has a massive impact on performance. Even though e-graph as a data structure was designed to store a potentially infinite number of terms, some rule combinations can lead to uncontrolled growth. Consider an example:

```
vec![
    rewrite!(''rt_begin''; ''(;; (? ?r) (* ?r))'' <=> ''(* ?r)''),
]
```

[9]https://docs.rs/egg/latest/egg/tutorials/_01_background/index.html

If we have a rewrite system with just one rule: Listing 4.4, and we build an e-graph for expression `r* ;; r?`, we get a tiny e-graph with 5 nodes, e-graph is shown in Figure 6.

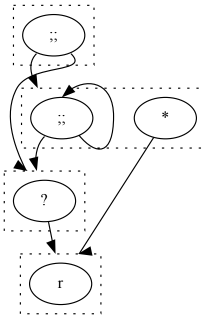

Figure 6: Saturated e-graph for `r* ;; r?` with just one rule

However, if we add another rule to the system, the saturation algorithm will not terminate, if we do not set the node/iteration limit. We add an associativity rule for the `;;` operator, for simplicity it will be unidirectional:

```
vec![
    rewrite!("rt_begin"; "(;; (? ?r) (* ?r))" <=> "(* ?r)"),
    rewrite!("seqA"; "(;; ?a (;; ?b ?c))" => "(;; (;; ?a ?b) ?c)"),
]
```

With associativity, e-graph explodes with new e-classes, that cannot be defined recursively anymore. For example, we get the following set of equivalences:

$$\begin{aligned} \texttt{r*} &\equiv \texttt{(r? ;; (r? ;; r?)) ;; r*} \\ &\equiv \texttt{r? ;; r*} \\ &\equiv \texttt{(r? ;; r?) ;; r*} \\ &\equiv \texttt{((r? ;; r?) ;; (r? ;; r?)) ;; r*} \end{aligned}$$

The highlighted subterms in the left parts of the expressions are not equivalent to each other, but they appear in the e-graph during the saturation.

All in all, the problem is as follows. Expanding rules in tandem with rules that reorder nodes, such as commutativity or associativity, can lead to an infinite growth of the e-graph. There are several ways to solve this problem. We have come up with various approaches and will discuss them in detail in Section 4.4.1. Some of the strategies are implemented in our plugin and are available for the user.

4.4.1 Proof Strategies

A less general, but more efficient solution may be to manually schedule, how many times particular rules should be applied. If we are trying to prove that `a` $\equiv$ `b` and to do so we build an e-graph for `a` and search for `b` in it, we can use "expanding" rules only for terms that are smaller than `b`. However, in this thesis, we will focus on more general, but more heuristic approaches. We are still working on providing the user with various trade-offs between efficiency and completeness, all of which can be easily toggled within the plugin.

All solutions assume that we completely opt out of bidirectional rules and orient all of them in the direction of the smaller term. It shrinks the range of problems we can solve but significantly reduces the time complexity of the algorithm. To prove `a` $\equiv$ `b`, we can build an e-graph for `a` and search for `b` in it and vice versa. Alternatively, we can build both e-graphs and check their intersection to be non-empty. If we find an element that is equivalent to both `a` and `b`, we can conclude that `a` $\equiv$ `b` and construct a proof.

4.5 Retrieving proofs in Coq

After we finish saturating the e-graph and find out whether two expressions are equivalent in it, we can use `egg`'s `explain_equivalence(expr1, expr2)` function to get the proof. The output will be a series of s-expressions annotated with the rewrite being performed. Consider an example, showing how `(/ (* (/ 2 3) (/ 3 2)) 1)` (or $(\frac{2}{3} \cdot \frac{3}{2})/1$) can be simplified to `1`:

```
(/ (* (/ 2 3) (/ 3 2)) 1)
(Rewrite<= div-one (* (/ 2 3) (/ 3 2)))
(* (Rewrite=> unsafe-invert-division (/ 1 (/ 3 2))) (/ 3 2))
(Rewrite=> cancel-denominator 1)
```

This sequence of s-expressions is parsed to a list of tuples with names of theorems to apply and directions (`forward` or `backward`) in which to apply them. This data is returned to OCaml, OCaml consequently applies the `rewrite` tactic with given theorems inside the goal and concludes the proof using the `reflexivity` tactic. This is how we automate the proof of equivalence of two expressions in our plugin.

4.6 Plugin configuration

By that moment we have described the ideas and the most important parts behind the algorithm that checks two terms for equivalence using `egg`. Along with that, the plugin supports the functionality to provide `egg` with user-pre-defined rewriting rules, in complement to the ones we have defined ourselves. For that, the vernacular command (more on vernacular commands in Section 4.1) `cegg_config` is provided. The set of theorems with which we configured the egg consists of statements about abstract relationships. In addition to them, in order to prove more substantial facts, it is necessary to use axioms of specific relationships with a stable meaning. Such axioms are usually combined into a set called *well-formed* (WF). Consider the following example.

```
Variable rf : A -> A -> Prop.
Variable mo : A -> A -> Prop.
Notation ’’’fr’’’ := ( rf⁻¹ ;; mo).

Record Wf :=
{
   rf_mo : rf ;; mo ≡ ∅ ;
   rf_rf : rf ;; rf ≡ ∅ ;
   mo_rft : mo ;; rf⁻¹ ≡ ∅ ;
   mo_fr : mo ;; fr ≡ ∅ ;
   fr_fr : fr ;; fr ≡ ∅ ;
}.
```

```
Cegg config Wf.
```

Listing 1: Well-formed example

The object with axioms is created, then passed to the OCaml plugin, WF is checked to contain only relational equivalences, then it is transferred to Rust, where it is cached in the build folder. Afterwards, these axioms would be used along with pre-defined ones to prove theorems.

4.7 Evaluation

Unfortunately, we cannot demonstrate the usability of the created tool to prove any popular substantive results. An example of such a result could be proving the equivalence of two definitions of the `eco` relation. Consider a slightly modified Listing 1.

```
Variable rf : A -> A -> Prop.
Variable mo : A -> A -> Prop.
Notation '''fr''' := ( rf⁻¹ ;; mo).

Notation '''eco1''' := (rf ∪ mo ∪ fr)^+.
Notation '''eco2''' := (rf ∪ (mo ;; rf^?) ∪ (fr ;; rf^?)).

Record Wf :=
{
    mo_trans : mo ;; mo ⊆ mo ;
    rf_mo : rf ;; mo ≡ ∅ ;
    rf_rf : rf ;; rf ≡ ∅ ;
    mo_rft : mo ;; rf⁻¹ ≡ ∅ ;
    mo_fr : mo ;; fr ≡ ∅ ;
    fr_fr : fr ;; fr ≡ ∅ ;
}.

Implicit Type WF : Wf.

Lemma eco_eq WF :
    eco1 ≡ eco2.
Proof.
```

Listing 2: Eco equivalent definitions

Despite the fact that we need to prove the equivalence of two relations, one of the properties we need to use along the way is `mo_trans`, which was added to the WF in Listing 2. In the process of proof, we inevitably need to use inclusions of one relation into another, which unfortunately our plugin cannot do automatically. Based on our work, we have concluded that for verified automated proofs of equivalences, we frequently require to operate with inclusions of relations, which our approach is not capable of. In order to adapt the equality saturation approach to our task, we need to improve the e-graphs so that they can store asymmetric rewrites.

Nevertheless, the created solution can solve simple subtasks and simplify work in a limited set of cases, when all intermediate steps involve only equivalence rewrites, for example, consider taking Listing 1 and trying to prove the following Lemma:

```
Lemma eco_sub WF (r : relation A) :
    (fr ∪ mo) ;; (fr ∪ mo) ≡ fr ;; mo ∪ mo ;; mo.
Proof.
    rewrite -> seq_union_r.
    rewrite -> seq_union_l.
    rewrite -> fr_fr.
    rewrite -> mo_fr.
    rewrite -> union_false_l.
    rewrite -> union_false_l.
    rewrite -> seq_union_l.
    all: auto.
Qed.

Lemma eco_sub' WF (r : relation A) :
    (fr ∪ mo) ;; (fr ∪ mo) ≡ fr ;; mo ∪ mo ;; mo.
Proof. Cegg solve eq. Qed.
```

It takes 7 rewrite calls to prove this lemma by hand, but it takes only about 600ms for our tool to solve it automatically. Here we have used the proof strategy where all rules are bidirectional. Strategies without expanding rules (described in Section 4.4.1) are usually executed in less than 100ms.

Here are some other simple examples of what we can solve using the developed tool:

```
Lemma test_with1 (r : relation A):
    ((r ;; r^*)^?)^+ ;; ((r^?)^?)^+ ≡ r^*.
Proof. Cegg solve eq. Qed.

Lemma test_with2 (r : relation A):
    (r^?)^+ ;; ((r^?)^?)^+ ≡ (r^?)^+ ;; (r^?)^*.
Proof. Cegg solve eq. Qed.

Lemma test_with3 (r : relation A):
    ((r^?)^+ ;; ((r^?)^?)^+)^+ ;; r^+ ≡ r^+.
Proof. Cegg solve eq. Qed.
```

5 Conclusion and future work

We have implemented a Coq plugin, which uses fast and lightweight `egg` Rust library, to automatically prove relational equivalences in Coq. We have developed a framework for communication between Coq and Rust and made a rule-parameterized algorithm on top of the `egg` library for producing a series of rewrites that is used in Coq interactive proofs. Moreover, we have experimented with various proving strategies and rewriting systems to implement an efficient and useful tool.

However, our experiments and benchmarks show that egg, along with its approach of employing e-graphs to solve rational equivalences, has not been performing up to our expectations when tested on our initial examples and problems. Therefore, in order to enhance the tool's efficacy, we suggest utilizing a different data structure that has the capability to preserve asymmetric relations as the basis for the saturation algorithm. Alternatively, a completely different approach should be taken into consideration.

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, 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 configuring 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`.

As part of our future work, we plan to generalize our solution and extend its applicability to solve similar problems encountered by the `autorewrite` tactic in Coq. It takes a library of theorems as input and automatically tries to use it in order to prove the goal. Equality saturation has potential benefits and can outperform the existing approaches.

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

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

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] 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.

[4] 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).

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] David C. Luckham et al. *Stanford Pascal Verifier User Manual.* Tech. rep. Stanford, CA, USA, 1979.

[12] Leonardo de Moura and Nikolaj Bjørner. “Efficient E-Matching for SMT Solvers”. In: vol. 4603. July 2007, pp. 183–198. ISBN: 978-3-540-73594-6. DOI: 10.1007/978-3-540-73595-3_13.

[13] Greg Nelson and Derek C. Oppen. “Fast Decision Procedures Based on Congruence Closure”. In: *J. ACM* 27.2 (1980), pp. 356–364. ISSN: 0004-5411. DOI: 10.1145/322186.322198. URL: https://doi.org/10.1145/322186.322198.

[14] 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.

[15] 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.

[16] Robert Endre Tarjan. “Efficiency of a Good But Not Linear Set Union Algorithm”. In: *J. ACM* 22.2 (1975), pp. 215–225. ISSN: 0004-5411. DOI: 10.1145/321879.321884. URL: https://doi.org/10.1145/321879.321884.

[17] 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.