Verifying the Rewrite Rules of Vercors Using Interactive Theorem Provers

CONNOR BLEUMINK, University of Twente, The Netherlands

Vercors uses rewrite rules to check whether code is working correctly, these rewrite rules have yet to be proven. this paper has verified 60 rewrite rules using Coq. for this purpose 9 categories were found that all had a specific approach to proof the rules.

Additional Key Words and Phrases: Rewrite Rules, Validation, Interactive Theorem Provers

1 INTRODUCTION

Software has become omnipresent in people's daily life. With a wide range of applications, such as games, business software, and embedded control software. it is becoming increasingly more important to make sure that the software works as intended. For this purpose tools have been developed to validate the workings of the software. One of these tools is called Vercors, which with the help of annotations made by the programmer will transform the code to the intermediate language of Viper and then uses existing tools to check whether the code is working as intended. Vercor is specifically made to work with concurrent programs, both heterogeneous systems (e.g. Java programs) as homogeneous systems (e.g. GPU kernels) [3, 4] To be able to achieve this as efficient as possible, Vercors changes the lines within the program according to their defined rewrite rules. These rewrite rules have not been formally proven to transform the code into equivalently working but easier to prove code. This is a problem, as Vercor cannot correctly conclude whether the system is working as intended. This is due to the fact that the program can be rewritten into a differently acting system if the rewrite rules are incorrectly defined. This study aims at the verification of these rewrite rules with the help of the interactive theorem prover Coq. The reason for using Coq over other theorem provers is due to Coq having integration with the Iris project, which has been created to help with proving theorems in a concurrent setting.

2 RESEARCH QUESTION

It is important that VerCors rewrite rules are verified, as the usage of incorrect rewrite rules can make it such that a program will be incorrectly validated by VerCors. For this purpose the following research questions will be answered.

- How can the rewrite rules of VerCors be categorized?
- Are the rewrite rules of VerCors Correct?

3 RELATED WORK

To find articles related to the research, the domains Scopus, Google Scholar, and IEEE were used. using terms akin to "verification", " rewrite rules", and "Automatic Theorem Provers" Some papers could be found.

TScIT 37, July 8, 2022, Enschede, The Netherlands

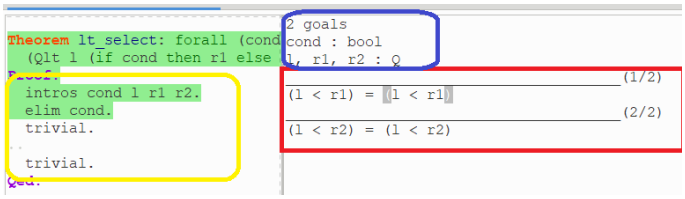

Fig. 1. Example Coq Proof

Winter et al. in 2004 wrote a paper about the use of transformation rules and how they can help verify, data movement within Java class loaders. In the paper, it is also discussed how this application of rewrite rules can be verified. [8]
Arendt et al. in 2005 wrote a paper about the automatic verification of rewrite rules that are used while verifying JAVA source code. [1]
Jaquel et al. wrote a paper in 2011, about the verification of B prove rules using automated theorem checkers. In this paper, they discuss how they approached the verification of rewrite rules.[5]
Morin-Allery et al. have done research on the validation of rewrite rules for assertion languages using automated theorem provers.[6, 7]
Bertot and Casteran wrote an elaborate book about how Coq can be used and the theoretical research behind the proofs that can be made using Coq.[2]

4 BACKGROUND

4.1 Coq

Coq is a program that assists its users with proving theorems, and can also be used to verify programs. When used it provides the user with an interface as shown in Figure 1. To help its users with their proofs it shows the parts that still need proving as goals (red box). To prove theorems the user inputs commands called tactics (yellow box), if a tactic succeeds, it either results in a finished proof or it will change the goal of the proof. The blue box shows the context in which the goals have to be proven, the context in the example consists of variables that are present, but it can also hold hypotheses.

5 METHODOLOGY

To answer how the rewrite rules of VerCors can be categorised, a literature research was used to see how rewrite rules have been categorised by others for the purpose of verification. Then the rewrite rules of VerCors were divided between the categories. After this the rules Were verified using Coq. Coq has been chosen as it came up in other literature about proving rewrite rules, and has good material to learn it. On top of this, it has an active user base and keeps being updated when new discoveries are made in theorem proving. The biggest reason Coq was used is because it has an extension based on the Iris Project. The Iris Project is a framework to help to reasoning about the safety of concurrent programs. After the rewrite rules were proven, another look was given to the chosen categories and

Category	Amount	Proven
Rational Arithmetic	21	21
Rational Division	5	5
Integer Modulo	4	4
Boolean Logic	21	21
Selections	5	5
Comparisons	12	12
Separation Logic	6	2
Out of Scope	42	0

Table 1. Rewrite Rule Distribution

they were refined to better reflect the differences in the approach needed to prove them.

6 RESULTS

Using the variable types and the operations within the rewrite rules, the following categories have been found:

- Rational Arithmetic
  - Rational Division
- Integer Modulo
- Boolean logic
- Selections
- Comparisons
- Separation Logic
- Out of Scope

The following segments will explain what kind of rewrite rules are part of each category and show how they can be proven using Coq.

6.1 Rational Arithmetic

For a rewrite rule to be part of the Rational Arithmetic category its variables can only consist of rationals, and can have any number and combination of addition, subtraction, and multiplication. Some examples would include:

- $\forall i, i \in \mathbb{Q}, i + 0 \equiv i$
- $\forall i, i \in \mathbb{Q}, i \times 0 \equiv 0$
- $\forall i, i \in \mathbb{Q}, i - 0 \equiv i$
- $\forall a, a \in \mathbb{Q}, \forall b, b \in \mathbb{Q}, a \times b + a \equiv a \times (b + 1)$

For proofs using rationals, Coq has the library *QArith*, this library gives access to the rational type *Q*, and contains the logic for operators on rationals. Hence it is first required to have the library imported using *Require Import QArith.*
The proofs of all rewrite rules of this category are done in the same way. It starts by removing the quantifiers over the variables using the *intros* tactic, followed by the automatic tactic of *ring*. *intros* is a variant of the coq tactic *intro*, which repeats *intro* until the proof term has reached a state in which it can no longer be reduced. The tactic *intro* uses head reduction to remove parts of the proof goal and add them to the context; when quantifiers are involved, the variable is added to the context. When implications are involved, it would add the implication to the context as a hypothesis and transform the proof goal into the final state of the implication. The tactic *ring* can be used to automatically solve equations of a ring structure, since this category consists of only rational equations that do not include division, and all have a ring structure.

Manual	Automatic	Outliers
2	1	1

Table 2. Proof Distribution of Integer Modulo

6.2 Rational division

This category is an extension of Rational Arithmetic, because it includes the division, which required a different tactic to be solved. Examples include:

- $\forall i, i \in \mathbb{Q}, i/1 = i$
- $\forall num, num \in \mathbb{Q}, \forall den, den \in \mathbb{Q}, num * (1/den) = num/den$

To prove the rules within this category one starts again with removing the quantifiers using *intros*. Since these rules include division they do not have a ring structure and thus the tactic *ring* cannot be used to automatically solve them. However, they do have a field structure. To deal with these structures Coq has the *field* tactic which is part of the *Field* library, which can be imported using *Require Import Field*. While the *field* command is able to prove rules using division, it will not immediately result in a full proof for rewrite rules akin to the second example. When used on the second example Coq requires a proof for $den = 0 \rightarrow False$ which states that den cannot be 0. Since VerCors checks for division by 0, den cannot be 0 when this rule is used. Due to the impossibility of den being 0, this line can be added as an assumption into the context, allowing for the use of the *assumption* tactic to complete the proof.

6.3 Integer Modulo

This category deals with rewrite rules, that contain variable of the Integer type and include the usage of the modulo operator. An example would be:

- $\forall i, i \in \mathbb{I}, i \mod i = 0$

For proofs using integers, Coq has the library *ZArith*. This library gives access to the integer type *Z*, and contains the logic for operators on integers. The start of the proof always starts with the tactic *intros* to remove the quantification. The next step is to either use the tactic *trivial*, which is a tactic that automatically finds a proof for the most simple cases, or to use the tactic *apply*, which when used will transform the proof-state using a term that is part of the context. Since the proofs are dealing with integers, lemmas from the *ZArith* library can be used as an argument for the *apply* tactic. Although modulo works using division, the lemmas that can be used with the *apply* tactic can be used even with division by 0.

However, the category has one rule that could not be solved in the above-described way. This rule was $\forall num, num \in \mathbb{Z}, \forall denom, denom \in \mathbb{Z}, (num/denom) \times denom + (num \mod denom) = num$. For this rule there was a similar lemma within the *ZArith* library, however apply could not be used to prove the rule. This is most likely due to the difference in how they were structured, the lemma started with $num =$ and instead of $((num\ denom) \times denom$ it was $denom \times (num/denom)$. To make up for these differences Coq has the *rewrite* tactic, this tactic can be used to transform parts of the proof goal keeping the

Manual	Automatic	outlier
15	4	2

Table 3. Proof Distribution Boolean Logic

new goal equivalent to the previous one. To solve this outlier, firstly the second difference was rewritten, and after the similar lemma was used to rewrite the goal into *num = num*, which could then be solved using the *trivial* tactic.

6.4 Boolean Logic

This category consists of rewriting rules that use booleans and boolean operations: and, or, implies, and negation. Some examples would include:

- $\forall b, b \in Boolean, b \wedge true = b$
- $\forall b, b \in Boolean, true \vee b = true$
- $\forall b, b \in Boolean, \neg\neg b = b$
- $\forall b, b \in Boolean, b \implies b = true$

For proofs using booleans, Coq has the library *Bool*, which gives access to the boolean type *Bool* and the logic for operators on booleans. The start of a proof always starts with the tactic *intros* to remove the quantification. The next step is to either use the *trivial* tactic or to use the *apply* tactic using lemmas from the *Bool* library.

The boolean logic category has two outliers. The first is similar to the outlier in the integer modulo category where both sides of the equal sign were swapped. This problem could once again by using the *rewrite* tactic followed by the *trivial* tactic. The second outlier was of a different case. For this outlier, there were no similar lemmas. This outlier is: $\forall b, b \in boolean, \forall a, a \in boolean, a \wedge (a \implies b) = a \wedge b$. After using the *intros* tactic to remove the quantification, it was first required to transform $a \implies b$ into $\neg a \vee b$. Then the and could be distributed over the or resulting in $(a \wedge \neg a) \vee (a \wedge b) = a \wedge b$. Finally before it could be solved with the *trivial tactic*, $a \wedge \neg a$ had to be transformed into *false*.

6.5 Selections

This category are rewrite rules that use the if-then-else structure. While in programming these structures take a boolean for the condition, the similarly functioning if-then-else structure of Coq uses the type *Prop*. Since *Prop* functions similarly to how a boolean would otherwise, these were used for the proofs. Some examples of a rewrite rule from this category are:

- $\forall cond, cond \in Boolean, if\ b\ then\ true\ else\ false = cond$
- $\forall cond, cond \in Boolean, \forall l, l \in \mathbb{Q}, \forall r1, r1 \in \mathbb{Q}, \forall r2, r2 \in \mathbb{Q}, l < (if\ cond\ then\ r1\ else\ r2) = if\ cond\ then\ (l < r1)\ else\ (l < r2)$

The start of a proof always starts with the tactic *intros* to eliminate the quantification. Unlike the previous categories, it is useful to give names for the variable as arguments. This is due to the second step involving the use of the *elim* tactic with the name given to cond as an argument. The use of *elim* here is to remove the if-then-else structure and to split the proof into 2 parts. The first part is to prove the resulting statement of when cond is true, the seconds part is to prove the resulting statement of when cond is false. For all rewrite rules of this category both of these could be proven with the *trivial* tactic.

6.6 Comparisons

The category of comparisons deals with rewrite rules using rationals, where a comparison is made between these rationals. examples include:

- $\forall l, l \in \mathbb{Q}, \forall r, r \in \mathbb{Q}, \neg(l > r) \Leftrightarrow l <= r$
- $\forall r, r \in \mathbb{Q}, r < r \Leftrightarrow false$
- $\forall r, r \in \mathbb{Q}, r < (r + 1) \Leftrightarrow true$

The proof begins with the *intros* tactic to remove the quantification. Since these rules use the double arrow, the *split* tactic is required after. The split tactic is used to split the proof goal into all smaller proof goals, in the case of $r < (r + 1) \Leftrightarrow true$ it would be split into $r < (r + 1) \implies true$ and $true \implies r < (r + 1)$ which both would need to be true for the rewrite rule to be correct. After the split, the goals will take 1 of the following forms:

(1) $comparison \implies comparison$
(2) $comparison \implies true$
(3) $comparison \implies false$
(4) $true \implies comparison$
(5) $false \implies comparison$

Goals of the form 1 and 3 required manual solving with the use of the *apply* tactic using lemmas from the *QArith* library. Goals of the form 5 are always true due to the ex falso principle, which states that anything can be proven using incorrect/false assumptions. Coq uses the *contradiction* tactic to proof goals of this form. Goals of form 2 are trivially true, and can thus be proven using the *trivial* tactic. The proof of a goal of form 4 starts with the tactic *intro* to add the true hypothesis to the context, so it was only necessary to prove the comparison, which can be done using the automatic tactic *reflexivity*. This tactic is used for proofs with reflexive or equivalence relations to see whether the relation holds true. Most of the comparisons within the rewrite rules either had a relation of this form or could be transformed into such a form using the *apply* or *rewrite* tactics, using lemmas from the *QArith* library.

6.7 Separation Logic

The category of separation logic has rewrite rules using the separated conjunction operator. An example is:

- $\forall b, b \in resource, true * b * -\!* b$

Coq in itself is not able to reason about these rules, hence Iris and its standard *heaplang* language were used for the proofs. Since only 2 rules were proven it is difficult to say whether a framework exists to prove these rules. Before the rewrite rule can be transformed into the Iris syntax one needs to establish the language they are using, for heaplang this is done by *Context { !heaplangGS Σ }*. A resource is represented by a pointer and value duo as $pointer \mapsto value$. The proof starts with the *intros* tactic to remove the quantifiers. Then the Iris tactic *iSplit* is used to split the goal, similar to how the tactic *split* functions for proofs in the comparisons category.

For the proof of $true * b -\!* b$ the Iris tactic *iIntros "[_H]"* is used. This works similarly to *intros*, however, the part between quotation marks breaks up the conjunction into two different hypotheses, here

the underscore is used to discard one of the hypotheses and the H is the name of the other one, a question mark can be used instead of a word/letter to assign a random name. From this all that is left is to prove b, which is the hypothesis so the Iris tactic *iAssumption* can be used to finish the proof for this goal.

For the proof of $b -\!\!* true * b$ the Iris tactic *iIntro* is used to add the hypothesis _: b to the context. Then the Iris tactic *iSplit* was used to split the conjunction, leaving us to prove the goals *true* and b. For *true* the tactic *done* is used, this tactic is used to tell the program that the proof goal is correct. For the goal b the Iris tactic *iAssumption* was used, this tactic works like the Coq tactic *assumption* and can be applied because we have a hypothesis added by *iIntro*.

6.8 Out of Scope

The out of scope category contains all rewrite rules that have not been fully categorized; this is due to it requiring more advanced knowledge of both Coq and Iris. These rules contain: nesting, dependencies, ranges, $\forall *$ quantifiers, and use variables that do not have a representation in Coq or Iris.

7 CONCLUSION

The rewrite rules of VerCors could be categorized into 8 different categories, of which 5 had a different approach to proving. Integer modulo and Boolean logic had a similar approach, except that the lemmas were found in different libraries. During the proving process there were 3 rewrite rules that did not adhere to any of the proof approaches given by the 7 categories proofs were used, 2 of these were similar to lemmas that Coq can use for proofs but were written slightly differently. The last outlier did not have a similar lemma in the Coq library, instead requiring its own approach. Since all the categories had a distinct approach it could be possible to more easily prove some of the rewrite rules that were categorised under the out of scope category, as well as new rewrite rules that get created. In the end, 70 out of 112 rewrite rules were proven to be correct.

8 FUTURE WORK

Since not all of the rewrite rules were proven, there exists the possibility to continue research to verify the remaining 42 rewrite rules. Furthermore, due to the categorization process, further research can focus on the creation of a script to automatically verify rewrite rules in Coq.

REFERENCES

[1] Wolfgang Ahrendt, Andreas Roth, and Ralf Sasse. 2005. LNAI 3835 - Automatic Validation of Transformation Rules for Java Verification Against a Rewriting Semantics.

[2] Yves Bertot and Pierre Casteran. 2004. *Interactive Theorem Proving and Program Development.* https://doi.org/10.1007/978-3-662-07964-5

[3] Stefan Blom, Marieke Huisman, Saeed Darabi, and Wytse Oortwijn. 2017. *The VerCors Tool Set: Verification of Parallel and Concurrent Software.* Vol. 10510. Springer International Publishing. https://doi.org/10.1007/978-3-319-66845-1

[4] Marieke Huisman and Raúl E. Monti. 2020. On the Industrial Application of Critical Software Verification with VerCors. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 12478 LNCS, 273–292. https://doi.org/10.1007/978-3-030-61467-6_18

[5] Mélanie Jacquel, Karim Berkani, David Delahaye, and Catherine Dubois. 2011. LNCS 7041 - Verifying <TEX>{B}</TEX> Proof Rules Using Deep Embedding and Automated Theorem Proving.

[6] Katell Morin-Allory, Marc Boulé, Dominique Borrione, and Zeljko Zilic. 2008. Proving and disproving assertion rewrite rules with automated theorem provers. *Proceedings - IEEE International High-Level Design Validation and Test Workshop, HLDVT*, 56–63. https://doi.org/10.1109/HLDVT.2008.4695875

[7] Katell Morin-Allory, Marc Boulé, Dominique Borrione, and Zeljko Zilic. 2010. Validating assertion language rewrite rules and semantics with automated theorem provers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29 (9 2010), 1436–1448. Issue 9. https://doi.org/10.1109/TCAD.2010.2049150

[8] Victor L. Winter, Steve Roach, and Fares Fraij. 2004. Higher-order strategic programming: A road to software assurance. *Proceedings of the Eigtht IASTED International Conference on Software Engineering and Applications* (2004), 350–355.