Efficient Term-Rewriting Super-Optimisation

Specialising Rulesets to Reduce Time Requirements for Compiler Optimisation

Mark Ardman[1]
Supervisor(s): Soham Chakraborty[1]**, Dennis Sprokholt**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Mark Ardman
Final project course: CSE3000 Research Project
Thesis committee: Soham Chakraborty, Dennis Sprokholt, Burcu Özkan

Abstract

Term-rewriting super-optimisation during compilation uses rewrite rules in order to restructure a provided code expression into the optimal form, comparing different expressions using a cost function. To reduce the compilation time taken by term-rewriting, the ruleset can be optimised by combining rules that were commonly chained during previous optimisation runs. With an any-time super-optimiser a specialised ruleset makes it possible to attain the optimal code expression, under the ruleset constraints, with a lower time requirement.

We found rule chaining methods to be effective at reducing the time necessary to obtain the canonical form. The methods performed well on synthetic benchmarks, as well as ones representative of real C projects. The frequency of the chained rule use and their uniqueness compared to the other compound rules directly dictates how great of a performance improvement its addition provides. Optimised sets demonstrated poor generality, indicating over-specialisation.

1 Introduction

Compilers are a crucial aspect of the lifecycle of an application. They may perform many tasks the primary one is to translate code from the source into the target. For example, compilers may translate C into machine code, thereby enabling the code to be run, interpreted, etc. Compile times grow with the project, the more dependencies a project utilises, the more code it has, etc. Increasing amount of time taken up by compilation is best spent elsewhere. Optimising different aspects of the compiler's architecture can, provided enough improvements, significantly reduce the compilation time.

A process that many compilers include is code optimisation. While there are various strategies for performing optimisations during compile time, this paper focuses on rewrite-driven optimisation. Research on super-optimization via term-rewriting aims to improve the code performance by performing expression rewrites. This occurs through re-formulating the provided code according to predetermined rules to obtain shorter, and ideally more optimised expressions. Rewrite-driven approach involves exploring alternatives for each sub-expression in the code according to a defined set of rules, and selecting the most optimised expression by means of calculating its approximate cost.

The efficiency and effectiveness of a term-rewriting super-optimizer, like the one realised by the EGG library [1], is primarily governed by the provided set of rules. They vary by language and collecting an effective base set is difficult. This problem is addressed by [2], which proposes the usage of equality saturation to infer new rewrite rules to augment existing sets. An effective set of rewrite rules can contribute to the performance of the super-optimiser and the code itself.

Efficiency of super-optimisers has been continuously improving since the initial implementation, addressed with techniques such as creeper trace transducers. Creeper trace transducers enable term-rewriting super-optimisers to skip intermediate steps in certain scenarios, such as multiplication by 0, in order to arrive at the result more efficiently [3]. However, an aspect that is relatively unexplored the post execution analysis of the rule usage, to combine rules which are often used in sequence with the goal of speeding up the process of optimisation. As described earlier, research on rewrite rulesets is primarily conducted with the aim of improving the effectiveness of the optimisers, where it also has the potential to improve the efficiency equally. 'Specialising' or 'optimising' the ruleset used by the super-optimiser for the specific codebase on which it operates, can potentially reduce

the time necessary for the super-optimiser to re-formulate the code to the lowest possible cost.

This paper investigates the following research question: '**How can an optimised ruleset affect the performance of a term-rewriting super-optimiser?**' In order to construct optimised rulesets, the rule utilisation statistics of the super-optimiser must be examined and interpreted with a well-defined method that creates a compound rule with which the ruleset can be augmented. This allows the investigation of the effect of optimising the ruleset on the performance of the super-optimiser and the generality of the optimised ruleset. Here, generality is defined as the degree to which the ruleset performs well on any randomly chosen piece of code (excluding the codebase on which it was optimised). This research necessitates a methodology to compare super-optimiser performance fairly.

Investigating 'specialising' or 'learning' term-rewriting super-optimisers has the potential to reduce the time taken by the compilation step and potentially improve the performance of the optimised program. Learning super-optimisers can potentially standardize high-performance code patterns in their rewrite rulesets, making them more efficient and effective.

To answer the research question, the paper first provides the necessary background information to understand term-rewriting optimisation, the decisions made in the methodology, and the limitations of the research. Further, the paper explains the methodology using which the research was conducted along with providing details for the experimental setup for the purpose of reproducing the attained results. After displaying and discussing the results, the paper covers discussion relating to research ethics and reproducibility. The results are then summarised in the conclusion, along with limitations and future suggestions.

2 Background

In order to provide background, this section will cover the basics of super-optimisation and the kind of optimisation techniques this paper focuses on investigating. It also covers the basics of the EGG [1] library in relation to the paper.

2.1 Super-Optimisation

The aim of the super-optimisation step in a compiler, and for the purpose of this paper, is to produce functionally equivalent code that is improved in some aspect. It should be noted that despite sounding quite close, super-optimisation and optimisation are by definition not interchangeable. Super-optimisation attempts to find the optimal, canonical, form of an expression [4]. On the other hand, compiler level optimisation attempts to improve the code in one aspect or another, without necessarily aiming to achieve optimality at all times. The functionality of the programme that is constructed and investigated in this research paper, lies somewhere in between classical compiler optimisers and super-optimisers.

Super-optimisation was initially conducted via a brute force search of the possible instruction sequences and satisfiability solvers [5]. However, with the use of E-Graphs the library EGG [1] has emerged. Its use of equality saturation for rewrite-driven optimisation allows for the super-optimisation process to be conducted significantly faster. This enables an optimiser with aspects of super-optimisation to be a viable compile-time step.

2.2 Rewrite-driven optimisation

One of the strategies for optimising code is by applying rewrites to it. At its core, this means having a certain set of rules to allow for the conversion of expression forms into more optimised ones.

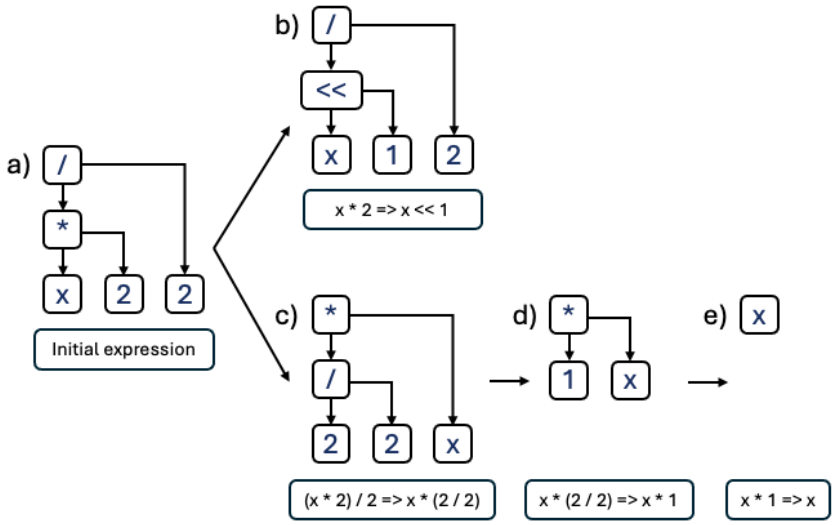

Figure 1: Rewrite driven optimisation of an initial recursive expression a) into more optimal forms b) or e). e) is achieved through the intermediate steps c) and d).

Given in Figure 1 is a simple example of rewrite-driven optimisation. Provided a simple recursive arithmetic expression labeled a), it can be rewritten into an arguably more optimal expression b), where a left shift by 1 is used instead of multiplication by 2. However, a) can also be rewritten into c) by using arithmetical properties of fractions, which then is further rewritten into d) as anything divided by itself is 1. d) is then rewritten to e) by using multiplicative identity property. This is a showcase of a rewrite-driven optimisation that optimises an arithmetic expression. Similar to this, optimisations can be performed on boolean expressions, control flow statements and other language specific features, given that it can be represented as an expression with parameters.

In order to perform rewrite-driven optimisation on expressions as showcased above, a set of rewrite rules is necessary. In its essence, it is a list of items where each item contains an initial and final expression. The initial expression is what the optimiser would try to match for, in the expression being optimised, and when found, replace it by the final expression. An example of a rewrite ruleset can be found in Appendix A.

Rewrite rulesets are constructed on a case-by-case basis for a specific optimiser and language based on their specification. They directly influence the efficiency and effectiveness of the optimiser. Larger rewrite rulesets usually implies longer optimisation time with a more optimal resulting expression. The more feature-rich the language, the more rewrite rules can be created for it. Languages with recursive expressions are well-equipped for rewrite based optimisation via EGG due to its recursive syntax. However, an abundance of features in a language makes it difficult to create a ruleset that encompasses them all and is effective. Control-flow languages, like C, are not as well-suited as recursive languages for the EGG

system, although it has a significantly simpler feature-set and thus makes it reasonable to construct and infer an effective ruleset.

Term-rewriting optimisation efficiency can be improved in various ways, using trace transducers [3] to skip intermediate expressions or discrimination nets [6], etc. Since large rulesets usually impede optimisations, they are rarely considered as points to improve optimiser efficiency. Despite that it may be possible to reduce the time needed to arrive at the final optimal expression by chaining rules together if they are reused with any degree of commonality. For example, as seen in Figure 1, chaining rules applied in c), d) and e) may allow for the program to skip intermediate steps and arrive at the final optimisation under an even lower time constraint.

2.3 E-Graphs Good(EGG) Library

The previously mentioned E-Graphs Good(EGG) library provides a framework for constructing rewrite-driven optimisers. In other words, the library provides a proprietary way[1] to represent code as recursive expressions. They are then converted to a tree-like E-Graph structure to which it then repeatedly attempts to match rewrite rules to find more optimised versions of the sub-expressions [1]. EGG also supplies a way to define rewrite rulesets at compile time. The expressions are ranked by means of a cost heuristic which EGG provides a way to define.

EGG will only attempt to match rewrite rules to the expressions until it meets certain limits which are defined when the optimiser is constructed. After that, it will return the lowest cost expression that has been found at the moment constraints have been reached. This makes it an any-time algorithm, and fit to use time as an independent variable by explicitly setting a time limit.

EGG is also observable, which furthers its suitability for the purposes of investigating rewrite rule chaining. It can produce an explanation of how a certain expression was attained by providing the rules which have been applied to the expression and all its sub-expressions to attain the final results.

3 Methodology and Experimental Setup

This section includes the detailed summary of the approach to data collection taken, along with the summary of the experimental setup used during the course of the experiment.

3.1 Methodology

To perform research on rule-rewriting super-optimisations it is essential to first choose a language and devise the rewrite rules for it. For the purposes of this research paper we chose C to conduct the experiment. Reasons for that include:

- C has minimal, yet workable syntax, allowing for a compact ruleset with room for optimisations (as discussed in the background section).
- C is a very mature language with an extensive number of codebases written in it which may used for experimentation.

[1] `https://docs.rs/egg/latest/egg/struct.RecExpr.html`

- There is an extensive ecosystem of various tools which allow us to manipulate C code as needed.

Rewrite rules for this project were compiled from various sources, a part of the ruleset is provided in Appendix A, full ruleset can be found in the attached repository[2]. C is a language that implements basic arithmetic operators(+, -, *, /), here are some examples of the implemented rules:

- Associative addition: "(+ ?a (+ ?b ?c))"=>"(+ (+ ?a ?b) ?c)"
- Annihilating element of multiplication: "(* ?a 0)"=>"0"
- Negation of negation: "(! (! ?a))"=>"?a"

Alongside arithmetic operations, boolean algebra operations were also implemented into the ruleset. Some C-specific features such as enhanced assignments and pointers have also been taken into account when formulating the base set of rewrite rules. Simple definitive control flow rewrites have also been implemented into the set.

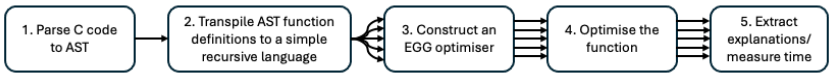

Figure 2: Outline of the pipeline utilised to conduct the experiment

The super-optimiser itself was constructed using the Rust programming language, as it is a fast language frequently used for writing build tools. It also allows direct access to the EGG library which is well-suited for conducting this experiment, it implements an any-time algorithm for rule-rewriting using E-Graphs. This allows for the creation of a fast, flexible and observable super-optimiser while allowing for the optimisation process to be limited on multiple constraints which makes it suitable for conducting this experiment. The process of super-optimisation is summarised in Figure 2.

1. Parse the C codebase using the lang-c crate[3] for Rust. lang-c is a lightweight and minimal crate that pre-processes files with GCC.
2. Lossily transpile the C abstract syntax tree (AST) into a simplified recursive language that can be interpreted by EGG. Only function definitions are extracted and transpiled. Make use of dataflow analysis by resolving obvious variable bindings to increase optimisation opportunities.
3. Populate a vector of rewrite rules. In parallel for each extracted function construct the EGG optimiser with explanations enabled. Each following step is performed in parallel per function.
4. Run the EGG optimiser for each function individually and extract the optimised expression and it's final cost.
5. Extract the EGG TreeExplanation data structure containing the data on the utilised rewrite rules. Benchmark the amount of time the optimiser takes to achieve minimal cost.

[2] https://github.com/MarkArdman/term_rewriting_rule_chaining/blob/master/src/lib.rs
[3] https://github.com/vickenty/lang-c

```
1 void main() {
2     int a = 1;
3     a = ((8 + 2) - 2) + (1 * a);
4     a = ((9 + 2) - 2) + (1 * 0);
5     int b = (2 * (2 + 4) - (2 * 2));
6 }
```

Listing 1: Example of a C programme to be optimised

To exemplify the described pipeline, consider Listing 1. It presents a simple C programme which does not serve any function, simply as an example. According to the first step of the pipeline, the code will be parsed and converted into the C AST. After that the functions from the parsed program will be transpiled into a simple recursive language, this code can be seen in Listing 2. The syntax is reminiscent of Lisp. At the beginning of an expression the function is specified, for example: `=` assignment, then the arguments for it: the binding name and the value. The `compound` function denotes a list of expressions to be executed in sequence, in order to simulate control flow.

```
1 (compound
2     (declaration a 0)
3     (= a (+ (- (+ 8 2) 2) (* 1 a)))
4     (= a (+ (- (+ 9 2) 2) (* 1 0)))
5     (declaration b (- (* 2 (+ 2 4) (* 2 2)))
6 )
```

Listing 2: C programme Listing 1 transpiled to a simple recursive language

Notably, the optimiser does not transpile the final EGG expression back into a C AST, as it is intended to primarily be used for the purpose of investigating the optimisation time. Due to this and the lossy transpilation mentioned in step 2, which impedes the reconstruction of the original AST, it is not possible to verify the correctness of optimised or transpiled expressions.

As mentioned in step 2, the transpiler makes use of data-flow analysis in an effort to create larger expressions which can ultimately be optimised to have a lower total cost, simultaneously producing more opportunities for rule-chaining. As an example, in Listing 1 line 2 the variable a is bound to the value 0. Later, in line 3 the binding a is used in the expression and can immediately be resolved to 0 upon transpilation. This may lead to artificial inflation of the total cost of the expression, however, for the purpose of investigating the time spent on optimisation the cost is irrelevant.

The TreeExplanation[4] that was noted to be produced by EGG earlier is analysed to extract the chains of rewrite rules that are used consecutively. This is the primary aim of the research paper: investigate the effect that the combination of these rules have on the performance of the super-optimiser. The rules are to be combined and added to the ruleset as they are extracted. TreeExplanation data structure provides a list of expressions where the first one is the initial form and the last one is the final form. For each expression a rule is provided that was used to transform it to the consecutive one. Even for the most minute of rewrites the data structure ends up being quite large. For example, consider Listing 2 line 4. The sub expression `(* 1 0)` can be simplified into `0`, a simple example of how an EGG explanation for this tranformation might look can be found in Listing 3.

[4] `https://docs.rs/egg/latest/egg/type.TreeExplanation.html`

```
1 [
2     { Mul, child_proofs: [{ Num(1) }, { Num(0) }] },
3     { Num(0), forward_rule: Some("Annihilating element of multiplication") }
4 ]
```

Listing 3: Simplified TreeExplanation of (* 1 0) to 0 rewrite

There are multiple ways to extract compound rules from the explanation provided by EGG and devising new rewrite rules from them. One method is to naively take any chain of rules which decreases the cost of the expression and is longer than one rule and formulate a chain from that. For example in Listing 2 the expression on line 5, can be optimised to `(declaration (b) (* (2) (4)))`. This involves the application of distributive law, associative law and addition and substraction of the same element as seen in Listing 4. This optimisation can be extracted and saved as a rule "(- (* ?a (+ ?a ?b) (* ?a ?a))" => "(* ?a ?b)".

```
1 (declaration b (- (* 2 (+ 2 4) (* 2 2)))
2     (Distributive law) =>
3 (declaration b (- (+ (* 2 2) (* 2 4)) (* 2 2)))
4     (Associative law) =>
5 (declaration b (- (+ (* 2 4) (* 2 2)) (* 2 2)))
6     (Addition and substraction of same) =>
7 (declaration b (* 2 4))
```

Listing 4: Naive rule chaining example

Another way to extract compound rules is to track rule chains as described previously, but only extend the ruleset with rule chains that are utilised more than once by the optimiser. An example of such rule usage can be seen in Listing 2 lines 3 and 4. Here it can be seen that addition and substraction of the same element, annihilating element of multiplication and identity element of addition are chained together, this rule chaining is exemplified in Listing 5. Since this rule chain is utilised in 2 different places, the following compoud rewrite rule can be extracted: "(+ (- (+ ?c ?a) ?a) (* ?b 0))" => "?c".

```
1 (= a (+ (- (+ 9 2) 2) (* 1 0)))
2     (Addition and substraction of same) =>
3 (= a (+ 9 (* 1 0)))
4     (Annihilating element of multiplication) =>
5 (= a (+ 9 0))
6     (Identity element of additon) =>
7 (= a 9)
```

Listing 5: Common rule chaining example

As noted earlier, EGG is an any-time optimiser, it will run indefinitely given the option, and terminates only upon reaching one of the few conditions set when initialising the optimiser, such as time, iteration or node limit. As the interest of this paper primarily lies in investigating the efficiency of the optimiser, the parameter that is going to be the limiting factor is maximum time taken. To find the optimal cost of the program given the base ruleset and the most complete TreeExplanation the optimiser must be run with very high iteration and time constraints, shrink the permitted time for each function until the optimiser produces a cost lower than the optimal one found in earlier runs. After that, TreeExplanation can be analysed accordingly to augment the base ruleset.

To measure the effects of a new rewrite ruleset on the optimisation time the optimiser must retain high iteration, node and other limits to ensure time as the only limiting factor.After that, the time should be continuously and gradually shrunken until the optimiser fails to achieve the optimal cost found in the base run.

3.2 Experimental setup summary

The experiment is performed on a Windows machine with the following specifications: Ryzen 7800X3D CPU, 32GB RAM and ASRock B650m motherboard. The environment for the experiment is the Windows Subsystem for Linux(WSL) running the Ubuntu distribution of version 5.15.146.1-microsoft-standard-WSL2.

When performing the shrinking process, the optimiser programme is always ran with the –release flag to ensure realistic performance.

4 Results and Analysis

This section will include the raw data collected from both the naive and common extraction methods of ruleset optimisation. Both methods were tested on multiple benchmark C codebases. They include:

- Synthetic codebase constructed for purposes of testing the optimiser in ideal conditions and verifying it's correctness manually[5]
- An aggregation of various Olympiad problem solutions[6]
- GZIP source code[7]
- DMC, an open source unrar library[8]

We trialed each datapoint at least 100 times to minimise the effects of external factors such as CPU temperature, load, memory layout and etc. on the accuracy and precision of the measurements. This section will also include the analysis and interpretation of the presented data.

4.1 Synthetic Codebase Rule Extraction Performance

In Figure 3 subplot A the naive ruleset optimisation technique is compared against the base ruleset. A roughly 2 times improvement can be seen in the time required by the super-optimiser to perform the optimisations. This codebase was specifically created for testing and to provide the optimiser with ideal conditions. It included many opportunities for optimisation, therefor the performance increase falls inline with the expectations.

In subplot B, the common rule technique shows a comparable improvement to the naive technique, though it appears to be more optimal on average. While this result is initially puzzling, it likely occurs due to the fact that the other compound rules present in the naively optimised ruleset are only slightly differing from the common rule extracted. For example the compound rule chain `"(= ?a (+ (- (+ ?a ?b) ?b) (* ?c 0)))" => "(ignore)"` is only

[5] `https://github.com/MarkArdman/term_rewriting_rule_chaining/tree/master/codebases/synthetic`
[6] `https://infoarena.ro/arhiva-acm`
[7] `https://people.csail.mit.edu/smcc/projects/single-file-programs/`
[8] `https://github.com/nothings/single_file_libs`

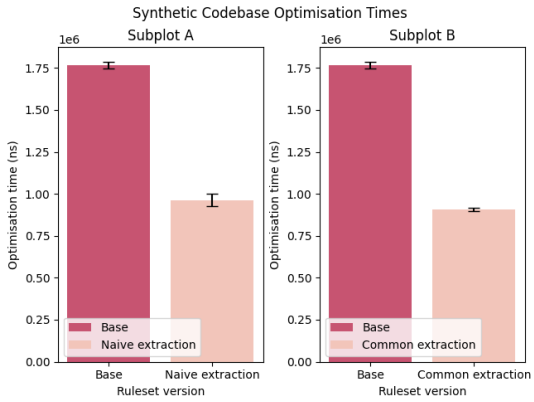

Figure 3: Synthetic codebase comparison of base to naive (Subplot A) and common (Subplot B) ruleset optimisation performances

one rule away from the `"(+ (- (+ ?a ?b) ?b) (* ?c 0))"=>"?a"` compound rule chain. As the common ruleset contains less rules that the optimiser needs to iterate through, and the fact that the common rule partially fills the role of the naive compound rules, the slight time reduction increase seen in subplot B compared to A is reasonably expected.

4.2 Olympiad Codebase Rule Extraction Performance

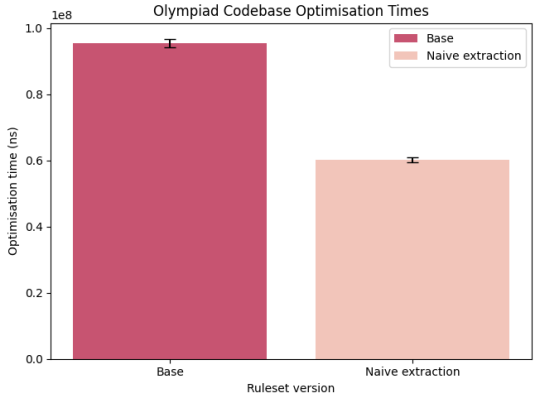

Figure 4: Olympiad codebase comparison of base to naive ruleset optimisation performances

In Figure 4 there is only one subplot comparing the base ruleset performance to the naively optimised one, this is due to the fact that the olympiad codebase did not produce any rules that were reused more than once. This has occurred due to the fact that the codebase is a compilation of multiple different solutions, from various people, likely resulting in little to no overlap in the coding style and patterns.

The naively optimised ruleset improves on the performance of the base ruleset by around 1.6 times. This is a smaller increase than seen in Figure 3 for the synthetic codebase. As this optimisation was performed on functional code which is more representative of real C code,

it is expected that there are less large inline expressions. This implies a smaller number opportunities for rewrites to be chained, even considering binding resolution being utilised.

4.3 GZIP Codebase Rule Extraction Performance

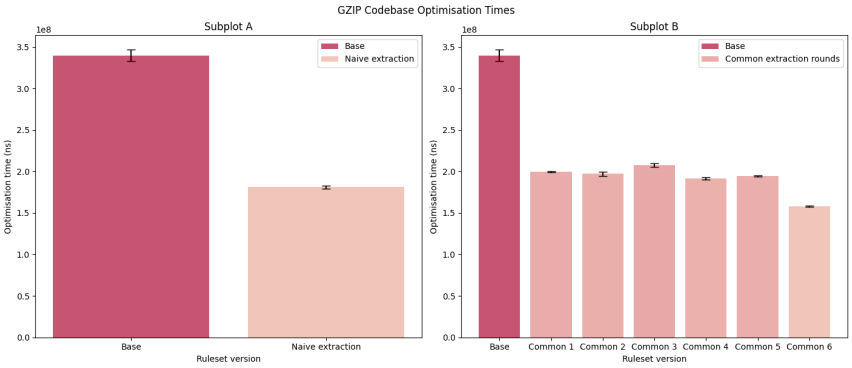

Figure 5: GZIP codebase comparison of base to naive (Subplot A) and common (Subplot B) ruleset optimisation performances

As Figure 5 depicts in subplot A, the naive optimisation for the ruleset nets the super-optimiser a roughly 1.9 performance improvement. This result exceeds the hypothesised performance effects. This is likely due to a multitude of factors. The binding resolution technique has likely significantly increased the amount of opportunities for optimisations to be made. Due to the limitations of the current pipeline and the inability to verify correctness of the output there is a possibility that binding resolutions have created incorrect sub-expressions, more on this in Section 6.2 and 7.2.

Although this result exceeds expectations, it is in line with the results obtained on the synthetic and the olympiad codebase. Even if most of the functions in the codebase do not provide opportunities for optimisations to be performed they do not affect the required time. The optimiser does not spend time optimising these functions if no cost reductions are possible. It is therefore likely that parts of the codebase that provide optimisation opportunities simultaneously provide rule chaining opportunities.

In Figure 5 subplot B is a bar chart where the bar labeled 'Base' is referring to the time taken for optimisation by the base ruleset. 'Common n' bars refer to the most commonly used common rules being added to the common ruleset in sequence, from most commonly used to least commonly used. As seen in the graph, adding the initial most commonly used rule increases the optimiser performance by roughly 1.7 times. This is likely due to the fact, as discussed for the synthetic codebase common ruleset, that this rule is commonly used by other compound rules. This significantly shortens the path to the optimal expression.

Another notable point can be seen at the bars labeled 'Common 3' and 'Common 5', where adding another rule worsens the perfromance. This is likely due to the fact that the added compound rules are closely related to the previously added ones, and are not used frequently enough. This creates more rules for the super-optimiser to iterate through on each expression, without providing sufficient speed up in situations when these rules can be applied. This is likely similar to what occurs in the synthetic codebase naive vs common optimisation technique comparison in Figure 3.

The common extraction technique outperforms the naive technique as seen in Figure 5, this likely occurs due to the fact that the compound rules that are only used once do not provide enough speed up to compensate for the overhead caused by the optimiser attempting to match them elsewhere.

4.4 Generality

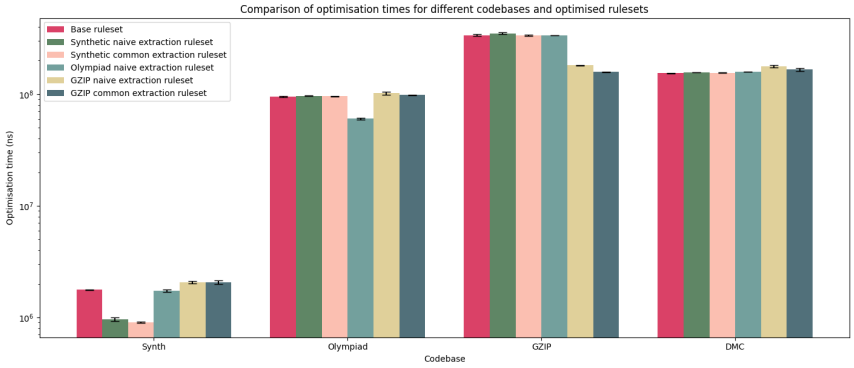

Figure 6: Performance comparison of all optimised rulesets for the synthetic, olympiad solutions, GZIP and DMC codebases

Both techniques exhibited limited performance when applied to datasets other than the ones they were trained on as can be seen in Figure 6. The Figure additionally includes testing conducted on codebase of DMC. It is an unrar library which is one of GZIP's functions. Performance measurements of optimised rulesets on DMC demonstrate the poor generality of optimised rulesets, additionally it shows that the overlapping functionality between GZIP and DMC does not allow for overlapping optimisation opportunities.

The poor generality is likely a result of rules that can be applied infrequently increasing the search space for the optimiser. The larger search space introduces an overhead time which likely contributes to worse performance for the optimised rulesets in comparison to the base ruleset. This is reinforced by Figure 5 Subplot B where rules that are used infrequently introduced a performance decrease.

It is worth noting that the increase in the time budged needed for optimisation is not as large as initially expected. This is likely due to EGG deliberately coordinating its rule application process to avoid needlessly applying expensive and expansive rules [1].

5 Discussion

This section includes the discussion of the data analysed in the previous section. It aims to place the results in a broader context by discussing the perceived utility of the methods, their potential drawbacks and how they would perform in a real world context.

5.1 Effectiveness of Rule Extraction Techniques

The naive rule extraction methodology demonstrates effectiveness in ideal conditions. As can be seen in Figure 3 subplot A it shows around 2 time performance increase over the

unoptimised ruleset. This demonstrates that the naive technique can be effective in reducing the amount of time spent on optimisation, in codebases with large inline expressions. Similar results are seen when applying the technique to GZIP and olympiad solutions codebases as seen in Figure 5 subplot A and Figure 4 respectively. The technique's effect on performance exceeded the hypothesised expectations when applied to real life situations. Results demonstrate that an optimiser that is able to ignore sections of code that it knows cannot be optimised, only focusing on sections with optimisation opportunities, naive rule chaining is capable of shortening the optimisation time. The technique can be utilised to reduce the time requirements of optimisation step during compilation, further discussed in Section 7.2.

Common extraction technique showcases similar performance to the naive technique, however it does demonstrate an advantage. While the naive extraction does in principle provide more rules which may shorten the optimisation process, the results in Figure 5 Subplot B showcase that removing rules with low usage or ones that are insufficiently different from the existing ones, in favour of a smaller ruleset, can in fact lower the required time for optimisation. This is further supported by Figure 6 which demonstrates that unused rules negatively affect performance. Common extraction can be utilised in a real world setting and provide effective results as testing has showcased. Further investigation is warranted into rule utility and de-duplication, it may potentially improve upon the common extraction method further, allowing for more general, yet well specialised rulesets. This application is further discussed in Section 7.2.

5.2 Optimised Ruleset Generality

The investigation revealed that both optimized rulesets lack generality, performing well only on the codebases they were trained on. The hyper-specialization of the rulesets limits their applicability to other codebases, as demonstrated by the cross-performance measurements of rulesets, and their performance on the DMC codebase in Figure 6. This limitation suggests that while optimized rulesets can improve performance for specific codebases, their broader utility outside is limited.

6 Responsible Research

This section will provide a discussion on the ethics, reproducibility and other important aspects of the research performed in this paper.

6.1 Representativeness

A significant aspect that must be considered when performing this type of research is that the sample code that is used for testing is representative of the general C code that one would write or find in a repository into which they are looking.

This is important to avoid synthetic benchmarks causing selective reporting of data that produces unrealistic expectations of the technique's effect on performance. While results would be technically correct and reproducible, they would not be representative of how a technique performs on a randomly selected piece of code produced by a real software engineer in a real piece of software.

In order to ensure that the technique effectiveness is represented fairly, and the results are representative of the general expected performance improvement of the optimiser testing is conducted on various codebases. In addition to synthetic benchmarking, the GZIP and

DMC codebases were tested. They are large and mature, frequently used projects that do not provide much room for optimisation and represent code written by experienced programmers. Testing is also conducted on programming Olympiad solutions in an effort to be representative of smaller, less mature projects written by less experienced programmers, to evenly demonstrate effects of technique on efficiency, and potentially efficacy, across a broader spectrum of scenarios.

6.2 Ethics

Due to the fact that this paper involves developing a technique for compiler architecture, compiler correctness is a relevant issue. Incorrect compilation outputs can potentially cause catastrophic crashes, lead to hard-to-spot bugs, vulnerabilities and malfunctions, which can have severe negative consequences, especially in safety-critical systems like medical devices, automotive software, or financial systems [7].

Due to the aforementioned lossy transpilation of the AST, this project is notably lacking in ability to verify correctness of the output, aside from manual verification which is rarely reliable. Errors can potentially occur in the base ruleset, or during the transpilation when the optimiser resolves obvious bindings. Further discussion on this can be found in the Future Recommendations section.

6.3 Reproducibility

Reproducibility is highly relevant to all experimental research and this is no exception; the ability to repeat and verify experimental results has been ensured by taking the following steps:

- The background and methodology section provide in full detail the tools utilised for the research performed, as well as the reasoning for why these tools were selected.
- Methodology section summarises in precise detail the exact specifications of the hardware, as well as the environment and compiler parameters under which the experiment was performed.
- The research makes use of open-source tools like the EGG library and lang-c crate. The codebases used for testing are directly referenced in the report. Open source codebase and dependencies in a public repository further the reproducibility, making an effort to make reproducing the research as accessible and effortless as possible for others.

7 Conclusions and Future Work

This section will summarise the general conclusions that can be drawn from the discussion and result interpretations above. It will also go over the notable limitations of the research conducted and potential further research on this topic.

7.1 Conclusion

To conclude on the question of '**How can an optimised ruleset affect the performance of a term-rewriting super-optimiser?**' From the results found during the course of the

experiment it is clear that optimising rulesets can directly reduce the time required by the optimiser to attain the optimal form of the expression. This is true for both the naive and the common extraction technique.

Contrary to the original hypothesis, the techniques produced measurable reduction in required optimisation time in synthetic codebases, and more mature and established codebases.

The naive method is outperformed by the common extraction method, demonstrating the importance of tracking the usage of the generated compound rules. It is also observed that the increased size of the ruleset is more detrimental to the optimisation time than a compound rule with no usages throughout the codebase. For the same reason it is important to track the similarities and duplications of the compound rules. Many rules that are not significantly different from each other directly leads to a decline in performance.

Both the naive and common extraction optimisation methods showed no generality; they are exclusively useful for the codebases on which they were optimised. The hyper-specialisation causes the compound rules added to the ruleset to rarely be useful anywhere else, only by chance, thereby worsening the performance of optimised rulesets on other codebases.

7.2 Limitations and Future Recommendations

One of the most significant limitations of this research is the lossy AST transpilation, and the subsequent unverifiability of the correctness of the produced expressions. Compiler correctness is of absolutely critical importance to the compilation process, optimisations performed by the compiler do not matter if the program is reproduced incorrectly. Future research could focus on implementing lossless transpilation and verifying the correctness of the binding resolution, rewrite rules and the optimiser in its entirety.

Another limitation that is worth mentioning is the rule usage analysis and extraction methodology. The optimisation and generality of the set directly depend on the compound rule selection methodology, further investigation of various analysis and extraction methods can prove useful in improving the performance of the ruleset for the codebase it is being run on, while maintaining generality, potentially allowing for certain compound rewrite rules to be solidified as a permanent part of the optimiser. Tracking rule usage and effects of newly added rules on performance can produce more optimised sets as demonstrated by the common extraction technique. Investigations into potentially removing rules that are shown to be too similar to others in the set or ones that are not used frequently enough could possibly increase the effectiveness further.

It is also worth further investigating the usage of these techniques in context of other programming languages. Haskell or Lisp could provide significantly more optimisation opportunities due to their recursive syntax, similar to one EGG uses for its internal language, meaning that the rule chains could potentially occur significantly more frequently and be much longer than the ones in C. This could mean that the learning super-optimisers may be significantly more relevant for them and provide more utility. Furthermore, investigating the methodologies performance in other language can provide valuable data on their performance on other codebases, to further refute or support their effectiveness.

Implementing the rule extraction techniques into real compilation processes that use term rewriting optimisations can potentially provide more data. It can highlight flaws and benefits of the techniques that were previously unexplored, or have gone unnoticed. Furthermore, it can demonstrate whether the techniques conflict with currently existing optimisations.

References

[1] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "egg: Fast and extensible equality saturation," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3434304

[2] C. Nandi, M. Willsey, A. Zhu, Y. R. Wang, B. Saiki, A. Anderson, A. Schulz, D. Grossman, and Z. Tatlock, "Rewrite rule inference using equality saturation," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485496

[3] R. Erkens, "Optimizing term rewriting with creeper trace transducers," *Journal of Logical and Algebraic Methods in Programming*, p. 100987, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352220824000415

[4] H. Massalin, "Superoptimizer: a look at the smallest program," *SIGARCH Comput. Archit. News*, vol. 15, no. 5, pp. 122–126, oct 1987. [Online]. Available: https://doi.org/10.1145/36177.36194

[5] A. Jangda and G. Yorsh, "Unbounded superoptimization," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 78–88. [Online]. Available: https://doi.org/10.1145/3133850.3133856

[6] K. Ogata, S. Ioroi, and K. Futatsugi, "Optimizing term rewriting using discrimination nets with specialization," in *Proceedings of the 1999 ACM Symposium on Applied Computing*, ser. SAC '99. New York, NY, USA: Association for Computing Machinery, 1999, pp. 511–518. [Online]. Available: https://doi.org/10.1145/298151.298431

[7] S. Baase and T. Henry, *A gift of fire: Social, legal, and ethical issues for Computing Technology*. Pearson, 2019.

A Rewrite Rules

Please find here a rewrite ruleset example. The rules are given in the following format: "Name/Short description": "Initial pattern" => "Final pattern". => indicates that the rule is uni-directional, aka can only be used to transform the initial pattern into the final one, <=> indicates that the rule is bi-directional. ?a indicates any sub expression [9].

- Commutative addition: "(+ ?a ?b)"=>"(+ ?b ?a)"
- Commutative multiplication: "(* ?a ?b)"=>"(* ?b ?a)"
- Associative addition: "(+ ?a (+ ?b ?c))"<=>"(+ (+ ?a ?b) ?c)"
- Associative multiplication: "(* ?a (* ?b ?c))"<=>"(* (* ?a ?b) ?c)"
- Distributive multiplication over addition: "(* ?a (+ ?b ?c))"<=>"(+ (* ?a ?b) (* ?a ?c))"

[9] `https://docs.rs/egg/latest/egg/macro.rewrite.html`

- Identity element of addition: "(+ ?a 0)"=>"?a"
- Identity element of multiplication: "(* ?a 1)"=>"?a"
- Annihilating element of addition: "(+ ?a 0)"=>"?a"
- Annihilating element of multiplication: "(* ?a 0)"=>"0"
- Sum of 2 to multiplication: "(+ ?a ?a)"=>"(* ?a 2)"
- Difference of 2 to 0: "(- ?a ?a)"=>"0"
- Difference of 0 to self: "(- 0 ?a)"=>"(- ?a)"
- Difference of self to 0: "(- ?a 0)"=>"?a"
- Add and subtract the same: "(- (+ ?a ?b) ?b)"=>"?a"

Note: The provided rewrite rules are examples and do not represent the complete set used in the research. The complete list can be found in the attached repository [10].

[10] `https://github.com/MarkArdman/term_rewriting_rule_chaining/blob/master/src/lib.rs`