TU WIEN Informatics

Delta Debugging for CUE Configurations

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Markus Bointner, BSc
Matrikelnummer 11808221

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Wien, 23. Jänner 2025

Markus Bointner   Jürgen Cito

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Informatics

Delta Debugging for CUE Configurations

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Markus Bointner, BSc
Registration Number 11808221

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Vienna, January 23, 2025

Markus Bointner   Jürgen Cito

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Markus Bointner, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. Jänner 2025

Markus Bointner

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Danksagung

Ich bin dankbar für die Unterstützung des gesamten CUE-Teams. Marcel, der mir viele Einzelheiten der Sprache erklärte und Ideen für Heuristiken vorschlug, Matthew, der sich die Zeit nahm, eine frühe Version auszuprobieren und viele Anregungen zur Verbesserung des Tools gab, Dominik für die anfängliche Koordination und Aram und Daniel für die Bereitstellung von Testinstanzen und die Überprüfung meiner Grammatik für CUE. Ein großes Dankeschön geht auch an meinen Betreuer Jürgen, der mir viele Anregungen für die Richtung der Arbeit gegeben, und mir gleichzeitig viel Freiheit und Flexibilität geschenkt hat. Selbst aus einigen tausend Kilometern Entfernung hat die Betreuung weiter funktioniert, was nicht selbstverständlich ist und ich sehr zu schätzen weiß.

Der größte Dank geht an meine Familie, die mich während meines gesamten Studiums immer mit motivierenden Worten unterstützt hat. Sie haben die Höhen und Tiefen erlebt und mich in jeder Situation geholfen soweit sie konnten. Ein besonderes Dankeschön geht an meine Großeltern, die in mir die Neugierde an der Technik geweckt haben. Zu guter Letzt, ein großes Dankeschön an Philipp, die einzige Konstante während des Studiums. Du hast mich vom ersten Tag an motiviert, immer mein Bestes zu geben und ich bin sehr froh darüber. Es war nicht immer einfach und es hat einige Zeit gedauert, aber wir haben es geschafft.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Acknowledgements

I am thankful for the support I received from the whole team at CUE. Marcel, who explained many intricacies of the language and proposed ideas for heuristics, Matthew, who took the time to try an early version and provided much input on how to improve the tool, Dominik for the initial coordination and Aram and Daniel for providing test instances and checking my grammar for CUE. A big thanks also to my supervisor, who provided many inputs on the direction of the thesis while allowing for much freedom and flexibility, what I greatly appreciate (even from far away).

The biggest thanks has to go to my family, who always had motivating words and supported me during all of my studies. They endured the ups and downs and supported me no matter what. A special thank you goes to my grandparents, who ignited the spark of technical curiosity. Last but not least, a thank you to Philipp, who pushed me from day one to give my best. It was not always easy and it took some time, but we did it.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Program-Reduction hat sich als wirksames Mittel erwiesen, um die Compilerentwicklung zu beschleunigen bzw. Programme zu verkleinern, um sie sicherer zu machen. Kernstück vieler Reduktionsalgorithmen ist Delta-Debugging, ein Algorithmus zur systematischen Entfernung von Elementen aus einem Programm, währenddessen eine gegebene Eigenschaft erhalten bleiben soll. Da das Problem der Suche nach einem globalen Minimum als NP-complete eingestuft wird, definiert Program-Reduction den Begriff der "1-minimalen Lösung", bei der das Programm beim Entfernen eines einzelnen Elements die gewünschte Eigenschaft verlieren würde. Verbesserte Versionen von Delta-Debugging nutzen die baumartige Struktur des Quellcodes zur Reduzierung der notwendigen Tests, die formale Syntax von Programmiersprachen zur Begrenzung des Suchraums und probabilistische Modelle zur Beschleunigung des Reduktionsprozesses.
Reduktionsalgorithmen werden in zwei Gruppen unterteilt: sprachunabhängige Algorithmen verwenden allgemeine Transformationen, um Programme in jeder beliebigen Sprache zu verkleinern, während sprachspezifische Reduktionsalgorithmen Domänenwissen nutzen, um spezifische und effektive Transformationen zu implementieren. Die meisten dieser Algorithmen werden auf prozedurale Sprachen angewandt.
Diese Arbeit schlägt ein neuartiges Framework namens Seru vor, das sprach-agnostische Reduktionsalgorithmen erweitert und sprachspezifische Heuristiken auf modulare Weise integriert. Der Ansatz kann auf mehrere Sprachen angewandt werden, während das semantische Verständnis einer bestimmten Sprache genutzt wird, um Programme weiter zu reduzieren.
Seru wird mit der Sprache CUE evaluiert, die starke Einflüsse von logischen Programmiersprachen enthält. Ergebnisse zeigen Verbesserungen für beide genutzten Reduktionsalgorithmen, Perses und Vulcan, mit bis zu 20,88% bzw. 16,94% kleineren Dateien.
Der kombinierte Ansatz aus sprachunabhängigen Reduktionsalgorithmen und semantischen Heuristiken erwies sich als effektiv, aber zeitaufwändig in der Ausführung. Die Programmreduktion ist nicht nur für prozedurale Sprachen effektiv, sondern erweist sich auch als nützliches Werkzeug zur Reduktion logischer Programmiersprachen. In zukünftigen Arbeiten könnten Large Language Models verwendet werden, um die Entwicklung von Heuristiken für weitere Sprachen zu beschleunigen.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Program reduction proved to be an effective way to speed up compiler development or shrink programs to make them more robust. At the core of many reduction algorithms is delta debugging, a technique to systematically remove elements from a program while ensuring a required property holds. Since finding a global minimum is an NP-complete problem, program reduction aims to find a 1-minimal result, where any removal of one element causes a test failure. Delta debugging was further improved by utilizing the tree-like structure of source code, by leveraging the formal syntax of programming languages to limit the search space and by building a probabilistic model to speed-up the reduction process.

Reducers are partitioned into two groups: language-agnostic reducers use general transformations to shrink programs of any language while language-specific reducers leverage domain knowledge to implement specific and powerful transformations. Most of these reducers focus on procedural languages.

This work proposes a novel framework called Seru, that extends agnostic reducers and adds the ability to integrate language-specific heuristics in a modular way. The generality of the approach is kept, while the semantic understanding of a specific language is utilized to further shrink inputs.

Seru is evaluated on the language CUE, which has its roots in logic programming. The results show improvements for both base reducers, Perses and Vulcan, ranging up to 20.88% and 16.94%, respectively.

The combined approach of language-agnostic reducers and semantic heuristics proved effective, yet time-consuming. Program reduction is not only effective for procedural languages, but shows to be a useful tool to reduce logic programming languages as well. In future work, large language models could be used to accelerate development of heuristics for more languages.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung xi

Abstract xiii

Contents xv

1 Introduction 1
1.1 Motivating Example . . . 2

2 Background 5
2.1 Program reduction . . . 5
2.2 Logical Programming Languages . . . 8

3 Related work 11

4 Approach 15
4.1 Framework . . . 16
4.2 Versatile Heuristics . . . 22
4.3 Logical Heuristics . . . 26
4.4 CUE-specific Heuristics . . . 27

5 Evaluation 31
5.1 Methodology . . . 32
5.2 Results . . . 34

6 Discussion 41
6.1 Effectiveness . . . 41
6.2 Queries . . . 42
6.3 Reduction Rate . . . 42
6.4 Efficiency and Execution Time . . . 44
6.5 Ablation study . . . 44
6.6 Order of Heuristics . . . 45
6.7 Implementation details . . . 46
6.8 Future work . . . 46

TU Bibliothek, Your knowledge hub, WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

7 Conclusion **49**

List of Figures **51**

List of Tables **53**

List of Algorithms **55**

Glossary **57**

Acronyms **59**

Bibliography **61**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

In addition to regular programming activities, developers frequently need to perform debugging tasks during their work. Alaboudi et al. [1] found that a debugging session occurs approximately once every eight minutes. These sessions last from a few minutes to more than one hour. The software projects in their study range in size from 2k up to 4M lines of code. A significant amount of time is used to browse the source code files and to identify the location of a bug.

Compilers and interpreters typically encourage or even require bug reporters to provide a small reproducing program and a test that verifies the existence of a bug to start debugging (GCC [6], Java [7], CUE [5]). Smaller reproducers enable compiler developers to find and fix bugs in less time. However, finding a smaller reproducing program from a real-world instance is not an easy task and requires knowledge of the particular programming language and the specific program instance.

The need for automation of this process emerged and led to research to apply Automatic Program Reduction (APR) to a wide range of applications [21, 23, 26, 27, 31, 33–36]. While most approaches aim to speed up compiler development by shrinking a bug-inducing input, program reduction can generally be used to reduce programs to fit any specification. Heo et al. [14] use APR to reduce the size of a program by removing unwanted features and thus reducing the attack vectors against it. This use-case is also interesting for devices with limited storage such as Internet-of-Things devices or for cloud applications, since reduced programs require less storage space.

Program reducers are typically partitioned into two groups: language-agnostic program reducers (AGRs) and language-specific program reducers (SPRs) [23, 27]. The first kind supports a multitude of programming languages and applies general transformations to shrink the size of a program, while the latter exploits the characteristics of a single language to perform powerful, yet specific transformations. The generality of AGRs is a desired property, yet SPRs typically perform better.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

This work aims to combine the two approaches and incorporate the generality of AGRs and the semantic knowledge and performance of SPRs into one framework, SeRu. The approach utilizes existing state-of-the-art AGR solutions and applies additional language-specific transformations to further improve the results. While existing program reducers are mostly designed for procedural languages such as C and Java [26, 27], we explore the paradigm of logic programming languages as the target for program reduction. In particular, SeRu implements 14 language-specific heuristics for CUE [9], a modern, open-source data validation and configuration language with roots in logic programming.

1.1 Motivating Example

When CUE language developers are faced with bug reports regarding issues in a new beta version of the language, they typically get a program to reproduce the issue. These bug-inducing programs can have hundreds or thousands of lines of code. In some occurrences, a program cannot be supplied since it is proprietary.

An automatic program reducer can solve both of these issues: it reduces the size of a particular source code file thus enabling faster debugging, and can be used by language users to shrink proprietary code for easier obfuscation to include it in a bug report.

Figure 1.1 shows part (hidden lines given at start and end) of a CUE configuration file which defines a custom command called `foo`. It defines multiple variables using the `let` keyword, utilizes external modules such as `strings` and `path` and performs a complex loop operation.

```
1  ... 12 lines
2  command: {
3    foo: {
4      let json_indent = "    " & strings.MinRunes(4) & strings.MaxRunes(4)
5      let dir_operations = path.FromSlash("_operations/github", path.Unix)
6      let file_tf_json = "config.tf.json"
7
8      orgs: {
9        for orgName, orgTerraform in target.terraform.github.org
10       let dir_org = path.Join([dir_operations, orgName], _goos)
11       let file_org_config = path.Join([dir_org, file_tf_json], _goos) {
12         "generate config \(file_org_config)": cli.Print & {
13           text: json.Indent(json.Marshal(orgTerraform.config), "", json_indent) + "\n"
14         }
15       }
16     }
17   }
18 }
19 ... 57 lines
```

Figure 1.1: Part of a longer CUE configuration file (instance panic/2490/v1)

When a program reducer such as Perses [27] is applied on the input file, many of the tokens can already be removed, as shown on the left in Figure 1.2. Hidden lines at the bottom reduced from 57 to 21. Nested objects are removed in the body of `command`, yet many statements were not changed. The AGR used performs syntax-guided reduction and has no understanding of specific concepts like the `for` loop or `let` statements.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1  ... 11 lines
2  command: foo: {
3      let json_indent = "    " & strings.MinRunes(4)
4      let dir_operations = path
5      let file_tf_json = "config.tf.json"
6      for orgName, orgTerraform in github
7      let dir_org = dir_operations
8      let file_org_config = file_tf_json {
9          cli.Print & {
10             text: json.Indent(json.Marshal(
                   orgTerraform), "", json_indent)
11         }
12     }
13 }
14 ... 21 lines
```

```
1  ... 8 lines
2  command: foo: {
3      file_tf_json: "config.tf.json"
4      for orgName, orgTerraform in github
           {
5          cli.Print & {
6              text: json.Marshal(
                   orgTerraform)
7          }
8      }
9  }
10 ... 14 lines
```

Figure 1.2: Results of a reduction process (instance panic/2490/v1). Left shows part of the result for Perses and right shows part of the result of Seru+Perses

When semantic understanding is added to the reduction process, additional tokens are removed from the input, as shown on the right of Figure 1.2. Unnecessary declarations are removed, `let` definitions are transformed to simple field declarations and the `for` loop is simplified. Dependencies on external modules such as `path` are no longer used and therefore removed. The input instance was reduced from the original 420 tokens to 152 by Perses, with a final result of 85 tokens when the semantic component is added. This reduction to ≈20% of the input size implies less work for a developer to find the culprit.

The next chapters discuss the background and preliminary knowledge in Chapter 2, followed by related work in Chapter 3. A detailed description of the approach is given in Chapter 4. We present results in Chapter 5 and discuss them in Chapter 6, before concluding in Chapter 7.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Background

This chapter focuses on the preliminary knowledge required to interpret the results of this thesis. It continues to explain program reduction in Section 2.1 and provide information on logical programming languages and CUE in Section 2.2.

2.1 Program reduction

Program reduction is a technique to remove as much unnecessary information from a source code file while still exhibiting a property (often a bug). Ideally, a program contains as few tokens as possible after the reduction process and therefore simplifies the debugging process for developers, since most of the irrelevant information is gone.

Formally, program reduction takes an input program P that exhibits a property ψ and reduces it to a smaller program P' (or p_{min}) that also exhibits the property ψ. We use the definition of program reduction from Sun et al. [27]:

$$\psi : \mathbb{P} \rightarrow \mathbb{B} \tag{2.1}$$

$$\underset{p \in \mathbb{P} \wedge \psi(p)}{\arg\min} |p| \equiv \{p \mid p \in \mathbb{P} \wedge \psi(p) \wedge \forall x \in \mathbb{P}. |p| \leq |x|\} \tag{2.2}$$

With $\mathbb{B} = \{true, false\}$ and $\mathbb{P}$ as the search space of all programs that can be derived from P using a concrete program reduction algorithm such as Perses [27] or SeRu. The property test function $\psi : \mathbb{P} \rightarrow \mathbb{B}$ is defined such that $\psi(P) = true$ and for any program $p \in \mathbb{P} : \psi(p) = true$, if the program p exhibits the property ψ and $\psi(p) = false$, if not.

Finding the smallest possible program, i.e. a global minimum, is a very hard problem. In fact, Misherghi et al. [23] defined the program reduction problem on a tree structure (the Abstract Syntax Tree (AST) of the source code) and showed that it is NP-complete. To find a practical solution, the problem has to be relaxed by using another variant of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

minimality called 1-minimality. First described by Zeller et al. [35] and later adapted for tree structures by Misherghi et al. [23], a program $p \in \mathbb{P}$ is 1-minimal if any removal of a single element (or node, respectively) would cause the property test to fail, i.e. $\psi(p) = false$. Existing program reducers find 1-minimal solutions, or local minima, according to their capabilities [19, 27, 34, 36].

Our approach using semantically informed heuristics to reduce programs aims to escape these local minima and improve the overall result.

2.1.1 Delta debugging

The delta debugging algorithm *ddmin* was introduced by Zeller and Hildebrandt [35] and is still used as the core of many program reduction algorithms [3, 16, 19, 21, 23, 27, 31, 33, 34].

ddmin intuitively consists of 3 steps:

1. *Reduce to subset*
   The program is split into n subsets (starting with $n = 2$), followed by a property test for each. If one of these subsets passes the property test, it is treated as the new result, and Step 1 is repeated.

2. *Reduce to complement*
   If none of the derived subsets passes the property test, their complements are checked. A complement contains all elements of the current result, except the elements in the subset. If a complement passes the property test, it is used as the new result and the granularity is set to $\max(n-1, 2)$. The algorithm continues at Step 1.

3. *Increase granularity*
   If none of the derived subsets pass the property test, the granularity is increased to $2n$ and the algorithm continues at Step 1.

It is easy to observe that *ddmin* will return a 1-minimal result. Before it terminates, the granularity is increased such that all configurations with 1 missing element are tested. The algorithm shows similarities to binary search, since half of the elements are removed in each step in a best-case scenario, much like binary search trims half the search space.

Misherghi et al. [23] observed that *ddmin* does not consider the tree-like structure often found in source code. Therefore, they propose the Hierarchical Delta Debugging (HDD) algorithm. It works directly on the AST of source code and utilizes the hierarchy to run the *ddmin* algorithm on each level of the tree. Note that Delta Debugging (DD) and HDD are instances of AGRs, i.e. the algorithms are applicable to any programming language.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.1.2 Syntax-Guided Reduction

HDD poses a significant improvement over DD, but has to reject many generated candidates, since they do not conform to the syntax of a programming language. In practice, only solutions with valid syntax are of interest to developers. Sun et al. [27] solved this issue by proposing a syntax-guided AGR based on DD and HDD with Perses. Their approach utilizes a programming language grammar in Backus-Naur form (BNF) to reduce the number of generated candidates with invalid syntax to zero, while DD and HDD essentially waste a lot of time generating syntactically wrong programs. A program reducer tries to find a minimal solution in $\mathbb{P}$, containing all the programs a reducer can derive from the original input P. This search space is partitioned in

$$\mathbb{P} = \mathbb{P}_{valid} \uplus \mathbb{P}_{invalid} \tag{2.3}$$

with $\mathbb{P}_{valid}$ containing all syntactically valid candidates and $\mathbb{P}_{invalid}$ all programs with invalid syntax. The approach of Perses to utilize a program grammar ensures $\mathbb{P}_{invalid} = \varnothing$. With the insights gained by the grammar, syntax-guided reduction is able to provide more transformation opportunities, by removing optional parts of a rule for instance, increasing $\mathbb{P}_{valid}$. Therefore, syntax-guided reduction improves over DD and HDD in size and time.

This approach is extended by Vulcan [34] to push the 1-minimality of a reduction result by using additional transformations that produce a smaller or non-1-minimal result. T-Rec [33] uses the lexical syntax to further reduce a result using fine-grained techniques on the token level.

2.1.3 Language-specific Reduction

In contrast to AGRs, SPRs apply deep insights from a particular programming language to implement transformations for it. They can utilize the semantics of language constructs and are able to implement complex transformations, such as function inlining, where the body of a function is copied to all calling locations with correct parameters substituted.

A popular instance of an SPR is C-Reduce [26]. It focuses on C/C++ program reduction an implements transformations such as:

- Parameter removal from function signatures
- Inlining a function
- Removing unused code
- Changing return types of functions
- Simplifying variable declarations

While SPRs provide good performance on one programming language, they are not suited for general use.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.2 Logical Programming Languages

Logical programming started to develop in the late 1960s and early 1970s and uses a paradigm conceptually different from procedural languages [20]. Problems are described in a declarative way using facts and rules. Horn clauses are at the basis of logic programming. They are written in the form

$$A \leftarrow B_0 \land B_1 \land \cdots \land B_n \quad \textit{where } n \geq 0 \tag{2.4}$$

A is called the head of the clause while $B_0 \land \ldots$ is the body where each B_i is an atomic formula. $\leftarrow$ is read as *if* and $\land$ as *and*. This yields a natural way to read a Horn clause as a rule: "A if B_0 and B_1 and ...". The fact A holds if and only if all of B_i hold. Combining multiple rules into a logic program yields a way to express knowledge and compute new facts and it was shown that logic programming is Turing complete by Tärnlund [30].

One of the most widely known logic programming languages is Prolog [4]. It is a general-purpose language and is often used for the development of complex applications in the field of artificial intelligence. A Prolog program consists of facts and rules. Figure 2.1 shows two facts on the first two lines: *charlie* is a dog and *roxie* is a cat. Line 3 is an example of a rule, which in this case defines that every dog is also an animal.

```
1 dog(charlie).
2 cat(roxie).
3 animal(X) :- dog(X).
```

Figure 2.1: Example of a Prolog program.

If this program is loaded, various queries can be performed to check whether a fact holds. The query **?-animal(charlie)** would return *true*, while the query **?-animal(roxie)** yields *false*. It is also possible to query for all solutions to an open predicate: **?-animal(X)** would give the answer **X=charlie**.

While Prolog uses a top-down evaluation model, Datalog [22] generally uses a bottom-up approach. Datalog is another instance of a logic programming language and its syntax is a subset of Prolog. In contrast to Prolog being a general-purpose language, Datalog is mainly used as a query language for databases and is not Turing complete. Some differences compared to Prolog include: a missing *cut* operator, no negation and no complex terms as arguments of predicates.

2.2.1 CUE

CUE [9] (Configure Unify Execute) is an open-source data validation language which also includes an inference engine and is heavily inspired by logic programming. It has its roots in GCL, a configuration language designed at Google to configure the predecessor of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Kubernetes. Similar to Datalog, CUE is not a general-purpose language, but has distinct applications. It aims to simplify configuration tasks, supports data validation, templating and queries, and can be used for code generation as well as executing custom commands.

The most prominent property of CUE is the unification of types and values into a single concept, which orders them in the same hierarchy. Data and schema do not have to be separate languages, but are all defined in CUE. For instance, a field can be assigned to the type `int`, the constraint `>5` or the value `10`. CUE unifies all occurrences of an identifier and checks if all constraints are fulfilled.

The syntax of CUE is a super-set of JSON. CUE programs (also called configurations) are compatible with JSON and YAML files, both as input and as output. It is suitable to unify and simplify large configurations typically found for cloud applications. Developers can reduce redundancy, add constraints to existing configuration properties and generate specific application configurations within one tool.

A typical CUE configuration consists of a collection of fields which link their label to a value. `foo`, `bar` and `out` are labels for the fields in Figure 2.2.
The field on line 1 has a string value `"Hello"`, while the field `bar` has a struct as its value. A struct is the CUE equivalent of a JSON object and itself contains fields. A typical JSON file would start and end with curly braces "`{}`" to define an object. These braces can be omitted in CUE.
The value in line 3 contains a string interpolation. The syntax "`\()`" can be used inside a string literal to evaluate any expression. Therefore, `out` results in the value `"Hello world!"`.

```
1 foo: "Hello"
2 bar: {
3     out: "\(foo) world!"
4 }
```

Figure 2.2: Small instance of a CUE configuration

Unification is a key concept in CUE. It allows multiple definitions of the same identifier across multiple files which are merged into one upon evaluation. Figure 2.3 shows an example of this behavior. It shows multiple definitions for `foo`, two different fields for a struct in `bar` and an abstract and concrete list definition for `baz`. These fields are merged into the output on the right. If any of the constraints are not compatible, e.g. change line 2 to `foo: <5`, the evaluation process would throw an error.

Templating is a major use-case for CUE and is often use for cloud configurations. A simplified example is given in Figure 2.4. It shows a template for the label `app` on line 1. `[X=string]` is a placeholder expression and expects any string to be a filed directly below `app` and stores its name in `X`. The field `name` on line 2 uses the stored name `X` and the value of the sibling field `kind` to create a string value. `apiVersion` is a constant value while `kind` is constrained to one of two values. Since fields `name` and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
foo: int
foo: >5
foo: 10

bar: {
  a: 5
}
bar: {
  b: 2
}

baz: [...int]
baz: [1, 2, 3]
```

```
foo: 10
bar: {
    a: 5
    b: 2
}
baz: [1, 2, 3]
```

Figure 2.3: Unification process of fields in CUE

`apiVersion` can be inferred, a user only has to define a concrete value for `kind` on line 8-9. The output on the right side is shown in YAML format.

```
app: [X=string]: {
  name: "\(X)-\(kind)"
  apiVersion: "v1"
  kind: "app" | "service"
}

app: {
  database: kind: "app"
  order: kind: "service"
}
```

```
app:
  database:
    name: database-app
    apiVersion: v1
    kind: app
  order:
    name: order-service
    apiVersion: v1
    kind: service
```

Figure 2.4: Example of templating in CUE

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

CHAPTER 3

Related work

Delta debugging was first introduced by Zeller and Hildebrandt [35], who proposed *ddmin*, an algorithm to reduce the input of a program while still passing a predefined test, which has great use in compiler development, software maintenance, and general program reduction. The algorithm tries to reduce the input by removing elements from the original input. It does so by removing a fixed-size sequence of elements from the input, starting with the size $\frac{n}{2}$, where n is the number of tokens in the input. The reduced input sequences and their complements are tested and excluded from further testing when the given test passes, otherwise the size is halved.

This language-agnostic program reducer (AGR) was improved by Misherghi and Su [23] with the Hierarchical Delta Debugging (HDD) algorithm that builds on top of *ddmin* and exploits the tree structure of programming and markup languages to reduce both the final output size and the execution time. They also showed that program reduction to a global minimum is an NP-complete problem. Improvements to HDD were made with modern HDD by Hodovan and Kiss [16], coarse HDD by Hodovan et al. [17] and HDDr by Kiss et al. [19].

A state-of-the-art tool for AGR is *Perses* by Sun et al.[27]. It utilises the formal syntax of programs to only produce syntactically valid and smaller solutions. The framework *Vulcan*, proposed by Xu et al. [34], leverages the result of state-of-the-art reducers like *Perses* and adds auxiliary reducers to the pipeline. Using various transformation and reduction methods, the auxiliary reducers manipulate the result to create a different, but not necessarily smaller, result which is again processed by the main reducer. *Vulcan* achieves smaller results in a variety of programming languages but needs more execution time.
T-Rec by Xu et al. [33] is another improvement for AGRs. It uses a fine-grained reduction technique that does not treat tokens as atomic, unlike other AGRs. Their approach uses the lexical grammar of programming languages to explore the reduction opportunities within tokens. The results of *Perses* and *Vulcan* are improved by 65 and 53

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

percent in bytes. However, it is inevitable that this approach takes more execution time, since additional steps are taken. The execution times are increased by 24-56 percent using *Perses* and by 1-5 percent with *Vulcan*, depending on the particular programming language.

The work of Wang et al. [31] aims to improve the core *ddmin* algorithm, which is used as a basis in most works on top of the original delta debugging paper. Instead of following a predefined sequence in testing, it builds a probabilistic model during execution, which yields a probability for inclusion of each element in the final result. This approach improves the execution time in practical cases and reduces the asymptotic worst-case performance from $O(n^2)$ to $O(n)$ with n being the size of the input. They later expanded this work to apply this probabilistic approach to ASTs [32].

In contrast to AGRs, SPRs use fine-tuned rules and knowledge about a specific programming language to find a smaller result. A state-of-the-art SPR for C and C++ is *C-Reduce* proposed by Regehr et al. [26]. They implement 30 unique transformations utilizing the specific properties of the C and C++ languages. These source code transformation can simplify types and variable declarations, inline functions and remove parameters from function signatures, to name a few examples. *ddSMT 2.0* implements automatic program reduction for SMT-LIBv2, a language for evaluating satisfiability modulo theories [21]. It improves on the first version by implementing a hybrid approach, using both the *ddmin* strategy and an orthogonal hierarchical strategy.

A recent work by Zhang et al. [36] combines the the ideas behind AGRs and SPRs using the advancements in the field of Large Language Models (LLMs) [18]. They propose LLM-aided program reduction (*LPR*) to perform language-specific transformations with a language-agnostic tool. A multi-level prompt is sent to an LLM to first identify possible locations for a source code transformation and then perform them to retrieve new candidate programs. An AGR is used in a reduction loop to efficiently remove unnecessary tokens. *LPR* significantly outperforms Vulcan in terms of token size, however, it dynamically prompts an LLM which requires much computing resources and potentially takes a long time.

This work introduces SeRu, which aims to combine the benefits of AGRs and SPRs similar to *LPR* [36], but removes the high resource requirements of calling a LLM during the reduction process. We introduce a total of 14 heuristics to implement a semantic reduction component in addition to utilizing state-of-the-art AGRs. While some of these heuristics are specific to a single language, we include versatile heuristics that are applicable to any language and paradigm-specific heuristics, which apply to one language paradigm, specifically logic programming languages. This work implements these algorithms manually, but we also propose the idea of creating a human-in-the-loop approach to generate specific implementations for various languages using LLMs. This step would reduce the manual labor required while limiting the high resource requirements of prompting a LLM multiple times during each reduction run.
SeRu also focuses on logic programming languages, while most of the state-of-the-art algorithms focus on procedural languages with very specific language constructs or are

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

focused on general-purpose program reduction. We specifically explore CUE [9] and the reduction possibilities using versatile heuristics as well as fine-tuned reductions based on the logical rules of CUE. While many programming paradigms, such as procedural languages, have an explicit control flow defined by constructs like conditionals, loops and function calls, logic programming languages are declarative and have its execution order determined by logical inference engines. Our approach builds on top of existing state-of-the-art solutions to incorporate rules tailored for logic programming languages, specifically CUE, to advance AGRs in this specific area.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 4

Approach

The current state-of-the-art as described in Chapter 3 regarding automatic program reduction can be categorized into two main approaches:

- Generalized approaches with a focus on the structure of the AST of a program. We will refer to these reducers as *syntactic reducers*.
- Specialized approaches focusing on the details and semantics of the source code. We will refer to them as *semantic reducers*.

Syntactic reducers achieve high generality by working directly on the source code as text or on the AST level. An important example of this group is delta debugging [35] which processes individual tokens from source code files to achieve efficient reduction. Modern examples of syntactic reducers use delta debugging or a modern variant at its core, for instance HDD [23] exploiting the tree-like structure of source code to gain efficiency, or Perses [27] which uses the grammar of a language to only generate valid candidates. (other examples [16, 17, 19, 31, 32, 34]) While syntactic reducers improved over time, they still face one issue: they cannot perform semantically informed reductions.

The approach of semantic reducers utilizes domain knowledge of a specific programming language to reduce source code. One such tool is called C-Reduce [26]. This tool utilizes a collection of transformations specifically designed for the C programming language, to create a smaller instance based on the input. However, since such transformations are designed for one programming language, language-specific reducers lack the generality of syntactic reducers.

Our approach merges the generality of syntactic reducers with the effectiveness of semantic reducers to create a modular and extensible framework which we call *SeRu*.

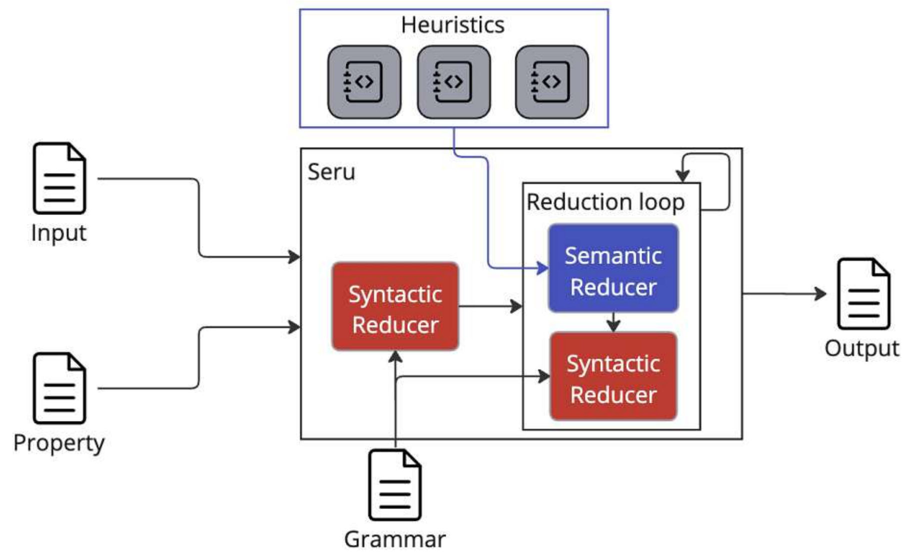


Figure 4.1: The SeRu reduction framework in a simplified view

4.1 Framework

The goal of program reducers is to create a new instance with fewer tokens from an input while maintaining a property ψ to hold during and after the process. The SeRu framework behaves similarly to other reducers, with one input instance being reduced to a new, smaller output instance.

The run configuration parameters of the system consist of an input program and a property of this program, as seen at the left side of Figure 4.1. The input program contains all relevant source code of the original program and fulfills the property. This program is processed by the framework to create a new, smaller output by modifying the input using reduction transformations. The property must hold for the input program and is used to determine interesting outputs, also candidates, of the reducers. A candidate is only interesting iff the property holds and will be discarded otherwise. The check of the property ψ for each candidate during the reduction process ensures that the property also holds for the output of the framework.

The main reduction loop (center in Figure 4.1) of the framework runs syntactic and semantic reducers on the input instance and combines their respective results. Both reducer components require some language-specific configuration to support a language.

Syntactic reducers (red in Figure 4.1) use general transformations that can be applied to any language. These transformations must only remove tokens from the input, they must not add any tokens. Their output is therefore equal to or smaller than the input. Syntactic reducers must only return syntactically valid programs and therefore require a grammar of a programming language to support it.

Semantic reducers (blue in Figure 4.1) implement a set of heuristics specific to a language.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

These heuristics use domain knowledge to remove or transform expressions in a program instance. Their abstract goal is to identify expressions that can be represented in a different way, potentially adding redundancy. To achieve this goal, a semantic reducer also requires the grammar of the concrete language and must generate syntactically valid output. Since these heuristics can understand the semantics in the source code, they can provide more powerful transformations, such as removing semantically linked expressions or inlining constants. Therefore, semantic reducers are able to remove, modify, or add tokens to the input with the goal of providing more reduction opportunities for the syntactic reducer.

While the language configuration is defined once to support all instances of a certain language, the run configuration parameters must be supplied for every new reduction instance.

4.1.1 Main reduction loop

The main reduction loop in Figure 4.1 is presented in more detail in Algorithm 4.1. According to the definition of program reduction in Section 2.1, the algorithm takes an input program P and a property function ψ and returns a minimal program p_{min}. We use the definition of 1-minimality in Section 2.1 for the minimal program.

The main algorithm calls two functions: `SyntacticReducer` (see Section 4.1.2) and `SemanticReducer` (see Section 4.1.3). Both of these functions take a program $p \in \mathbb{P}$ and the property ψ as input and return a candidate program. While `SemanticReducer` can return a candidate with $|candidate| > |p|$, `SyntacticReducer` always returns a candidate equal to or smaller than the input. `SemanticReducer` performs all transformations provided by the implemented heuristics sequentially. Both reducer functions return the input program when no reduction could be achieved.

The main reduction loop runs as long as the combination of `SyntacticReducer` and `SemanticReducer` finds a smaller result.

4.1.2 Syntactic Reducer

Let $\mathbb{P}$ be the set of all programs that can be derived from P during the reduction process. Then

$$\mathbb{P} = \mathbb{P}_{valid} \uplus \mathbb{P}_{invalid} \tag{4.1}$$

where $\mathbb{P}_{valid}$ is the set of all programs with valid syntax in the target language and $\mathbb{P}_{invalid}$ the set of all programs with invalid syntax [27]. A syntactic reducer takes a program p and the property function ψ and returns a modified version

$$p' \in \mathbb{P}_{valid} \quad s.t. \quad (|p'| < |p| \wedge \psi(p')) \vee p' = p \tag{4.2}$$

If the syntactic reducer cannot find a smaller result exhibiting property ψ with the given input, it returns the input. Otherwise, tokens are removed from the input as long as

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 4.1: Main algorithm of the SeRu framework

```
    Input: P: ℙ, input program
    Input: ψ: ℙ → 𝔹, property function
    Output: p_min: a minimum program p ∈ ℙ with ψ(p) holding
 1  best ← SyntacticReducer(P)
 2  candidate ← ∅
 3  while best ≠ candidate do
 4  |   candidate ← SemanticReducer(best)
 5  |   candidate ← SyntacticReducer(candidate)
 6  |   if |candidate| < |best| then
 7  |   |   best ← candidate
 8  |   end
 9  end
10  return best
```

possible, and the so generated program p' will be returned. To achieve $p' \in \mathbb{P}_{valid}$, a syntactic reducer requires the grammar of a concrete programming language to filter invalid candidate programs.

A syntactic reducer implements one or more transformation algorithms that are applicable regardless of the concrete programming language of a concrete reduction instance. We define a syntactic transformation as

$$t_{syn} : \mathbb{P} \to \mathbb{P} \quad \forall p \in \mathbb{P}.t_{syn}(p) \subseteq p \tag{4.3}$$

Informally, a syntactic transformation identifies and removes irrelevant parts of a program. An irrelevant or redundant part is a token or a set of tokens that is not required for the program to exhibit property ψ.

We give a simple algorithm for a syntactic reducer in Algorithm 4.2 as an example. It is one instance of syntactic reducers and not necessarily the only option to fulfill above requirements.

In addition to the input program P and the property function ψ, a syntactic reducer also takes a syntax checker function S.

$$S(p) = \begin{cases} true, & p \in \mathbb{P}_{valid} \\ false, & \text{otherwise, i.e. } p \in \mathbb{P}_{invalid} \end{cases} \tag{4.4}$$

The first step of Algorithm 4.2 initializes a list T with all available transformations of the concrete reducer. These transformations are instances of t_{syn} as defined in Equation 4.3. The program of minimal size *best* is initialized with P. Then, all transformations are executed sequentially. If a transformation does not find a new candidate smaller than the best, which exhibits ψ and has valid syntax, the next transformation is used.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Algorithm 4.2: An example algorithm for *SyntacticReducer*

```
   Input: P: ℙ input program
   Input: ψ: ℙ → 𝔹 property function
   Input: S: ℙ → 𝔹 syntax checker
   Output: p′: ℙ smaller program derived from P
 1 T ← list of available transformations t_syn
 2 best ← P
 3 for t in T do
 4 |  candidate ← ∅
 5 |  while best ≠ candidate do
 6 |  |  candidate ← t(best)
 7 |  |  if ψ(candidate) and S(candidate) and |candidate| < |best| then
 8 |  |  |  best ← candidate
 9 |  |  end
10 |  end
11 end
12 return best
```

The so-generated result is returned to the main algorithm. The identity function gives a trivial, yet ineffective, syntactic reducer that matches the definition.

A modified version of the original delta debugging algorithm [35] is an example of a syntactic reducer. It processes input files line by line and removes tokens in a structured way (as mentioned in Chapter 3 and Chapter 2) Since delta debugging processes lines, it does not require any other input than the source code and works on any language. However, to fulfill the requirement of returning syntactically valid programs, an additional syntax check is necessary to filter invalid candidates. Recent approaches improved on delta debugging utilizing the grammar of a language and applying the algorithm on the AST representation of source code. HDD[23] is an example which runs the delta debugging algorithm on each level in the tree. Perses and Vulcan[27, 34] follow a syntax-guided approach which only generates candidates with valid syntax.

4.1.3 Semantic Reducer

Semantic reducers use heuristics tailored to a specific programming language. These range from widely-applicable ideas optimized for a specific language to algorithms leveraging concrete program constructs. The term *semantic* refers to informed decisions which are made using an understanding of the semantics of a programming language, and do not require a known relationship between the semantics of an input program and the semantics of transformed programs, different to metamorphic testing [8]. It is not required to maintain the semantics of a program, since the only required property of the output program is ψ.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

We categorize the heuristics for semantic reducers into three groups:

1. *Versatile heuristics* are transformations that can be applied to a multitude of programming languages, in various paradigms. They utilize very common constructs like loops and conditions, which enables implementations for many languages.

2. *Paradigm-specific heuristics* describe algorithms focused on one programming paradigm.

3. *Language-specific heuristics* use domain knowledge specific to one programming language. Such heuristics do not generalize and directly use language constructs defined in the grammar.

Like syntactic reducers, semantic reducers also take an input program P and a property function ψ, as shown in Algorithm 4.3.

Algorithm 4.3: *SemanticReducer*

Input: P: $\mathbb{P}$ input program
Input: ψ: $\mathbb{P} \rightarrow \mathbb{B}$ property function
Output: p'': $\mathbb{P}$ program derived from P

```
1  H ← list of semantic heuristics h_sem
2  best ← P
3  i ← 0
4  while i < |H| do
5  |   h ← H[i]
6  |   candidates ← h(best)
7  |   candidates_valid ← filter(candidates, ψ)
8  |   if candidates_valid not empty then
9  |   |   best ← min(candidates_valid)
10 |   else
11 |   |   i++
12 |   end
13 end
14 return best
```

The output of Algorithm 4.3, p'' is a combination of candidates generated by the heuristics. A loop iterates over all implemented heuristics and applies them to the current best to generate new program candidates. In each iteration of the *while* loop, the current heuristic h creates candidates based on the current *best*. Each heuristic defines its own criteria to determine where it is applicable and how many candidates are generated, i.e., the loop unrolling heuristic (Section 4.2.2) is only applicable to *loop* constructs. In general, heuristics are applied to all possible locations. Candidates are not required to exhibit property ψ, therefore, they are filtered to only maintain candidates with

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

$\psi(candidate) = true$. If the current heuristic produced one or more such candidates, the one with minimum size is used as the new *best*. Otherwise, if the current heuristic does not create any candidates exhibiting property ψ, the algorithm will use the next heuristic for the next iteration, until all heuristics are exhausted.

A heuristic h_{sem} is not required to return a result smaller in size compared to the input. We define h_{sem} as

$$h_{sem} : \mathbb{P} \rightarrow \mathbb{P} \quad \forall p \in \mathbb{P}. h_{sem}(p) \cap p \neq \varnothing \tag{4.5}$$

Similarly to t_{syn}, h_{sem} is a function that transforms a program into another program. A heuristic is required to produce a program which has at least one expression in common with the original program.

Since the definition of a heuristic is lenient with one required property, we provide informal goals of heuristics for a semantic reducer.

A goal of this work is a high reduction in a single step. A syntactic reducer is not informed about the semantics of expressions and is unable to connect semantically linked expressions together. However, a heuristic with semantic knowledge can link expressions and perform a transformation of such linked expressions. A concrete example is the elimination of a single variable in a program. A variable typically has a declaration, definition, and one or more usages. The corresponding heuristic can identify and link these statements and perform a transformation which removes all usages together with the declaration and definition in a program and replaces usages with the variable value or an algorithmically chosen value.

Another variant of heuristics transforms expressions to exhibit more redundancy compared to the input program. Such heuristics can potentially add to the size of a program, which contradicts the overall objective of program reduction: generating a smaller program exhibiting a property. Although these heuristics would not benefit the reduction process on their own, the combination with syntactic reducers, which are designed to eliminate redundancy, provides new reduction possibilities compared to syntactic reducers on their own. A trivial and versatile heuristic with this property is constant propagation. If a program assigns a literal value to a variable, the heuristic can inline this value for all usages, potentially growing the program while leaving the variable definition redundant. A syntactic reducer is now able to remove the variable definition.

A third approach to heuristics transform single expressions to simpler program constructs. Control structures like loops and conditions often contain boolean expressions. These expressions can be simplified or (partially) evaluated, depending on the available context.

CUE We instantiated the SeRu framework for the CUE language. Figure 4.2 shows the framework in more detail. The concrete inputs consist of a CUE file as the target for the reduction and a shell script which determines if a CUE file exhibits the property ψ. The semantic reducer is implemented using the official CUE API [10] written in GO and implements a total of 14 heuristics. Perses [27] and Vulcan [34] are the options for the

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

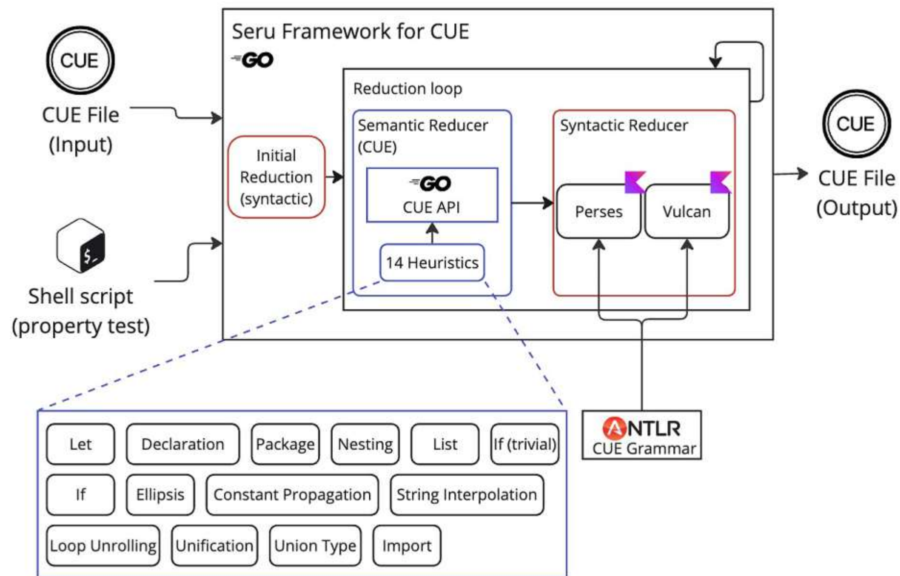


Figure 4.2: The Seru framework instantiated for the CUE language [9]. The syntactic reducers require an ANTLR [2] grammar to generate a parser for CUE. Heuristics for the semantic reducer plugin are implemented using the official CUE API for GO [10].

syntactic reducer and require a program grammar in the ANTLR format [2]. After the reduction process terminated, a reduced CUE file is created by the framework.

As some heuristics serve a similar purpose, they are grouped together:

- `if` and `if(trivial)` in Section 4.3.1
- `nesting`, `package`, `declaration` and `ellipsis` in Section 4.4.3

All detailed descriptions of all heuristics follow in Sections 4.2-4.4.

4.2 Versatile Heuristics

We consider a heuristic to be versatile, if the algorithm can be applied to various programming languages, over different paradigms. These algorithms use common control structures such as conditions and loops which occur in many programming languages. However, the concrete implementation is specific to one programming language.

4.2.1 Constant Propagation

In constant propagation, variables are backtracked to their assigned constant or value. This can either be another variable or a literal value. All usages of a variable is then replaced with the found constant value, or in other words, the constant is *propagated*. The

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

```
1 foo: 3
2 bar: foo
3
4 x: {
5     y: bar
6 }
7 z: x
```

```
1 foo: 3
2 bar: 3
3
4 x: {
5     y: 3
6 }
7 z: {
8     y: 3
9 }
```

Figure 4.3: An example of constant propagation in CUE. Variables are resolved recursively to their respective value over multiple steps. Effectively removing variable dependencies.

conceptual idea of constant propagation is similar to the strategy *Variable Elimination* used by Zhang et al. [36].

We give an example of constant propagation in Figure 4.3. The example defines some variables in CUE with literal values, references to other variables, or struct definitions. The constant propagation algorithm recursively resolves all variables to their respective values. A variable `bar` resolves its reference to `foo` to the literal `3`. Another example is variable `z`. It references `x`, which transitively references `bar` and `foo`, respectively. `x`, `bar` and `foo` can be considered dependencies of `z`. All these references are resolved to their value, which removes these transitive dependencies from `z`. If a property ψ depends on `z`, all other variables can be removed after this step.

A general algorithm for constant propagation is given in Algorithm 4.4.

Algorithm 4.4: Constant propagation

Input: P: $\mathbb{P}$ input program
Output: t: set of transformed programs $p \in \mathbb{P}$

1 $t \leftarrow \varnothing$
2 $I \leftarrow FindIdentifierUsages(P)$
3 **for** *ident in I* **do**
4 $\quad value \leftarrow ResolveValueInScope(P, ident)$
5 $\quad$ **if** *found value* **then**
6 $\quad\quad t \leftarrow t \cup \{ReplaceIdentWithValue(P, ident, value)\}$
7 $\quad$ **end**
8 **end**
9 **return** t

The function `FindIdentifierUsages()` traverses the AST and returns references to all identifiers used while skipping the declarations of such identifiers. Each usage of an identifier is returned separately. The algorithm does not replace declarations, since it is the first occurrence of an identifier and replacing it would remove it from the set of known identifiers in a program. Each iteration of the loop resolves a value

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

for a given identifier. Depending on the concrete programming language, there could be multiple values associated with the same identifier. In such cases, the function `ResolveValueInScope()` resolves values first from the most local scope and tries every parent scope until it reaches the global scope. If a value could be resolved, Algorithm 4.4 copies the input program, replaces the current identifier and stores the transformed program in `t`.

4.2.2 Loop Unrolling

The heuristic of loop unrolling was also inspired by Zhang et al. [36]. Loops are common in many programming languages and repeat a set of expressions until a condition is reached.

Algorithm 4.5: Loop unrolling

Input: P: $\mathbb{P}$ input program
Output: t: set of transformed programs $p \in \mathbb{P}$

1 $t \leftarrow \varnothing$
2 $L \leftarrow FindLoops(P)$
3 **foreach** *loop in L* **do**
4     $iterationValues \leftarrow ResolveValuesByIteration(P, loop)$
5   **if** *iterationValues is empty* **then**
6   **continue**
7   **end**
8     $statements \leftarrow \varnothing$
9   **foreach** *i in iterationValues* **do**
10         $statements \leftarrow statements \cup \{LoopBodyWithValue(loop, i)\}$
11   **end**
12     $t \leftarrow t \cup \{ReplaceLoopWithInlinedStatements(P, loop, statements)\}$
13 **end**
14 **return** t

If it is known that a loop will execute its loop body `n` times, the loop can be replaced by `n` copies of the loop body with all loop variables substituted to the value in the respective iteration. This action is known as loop unrolling (Algorithm 4.5) and can provide opportunities for program reduction. If the property ψ depends on a specific loop iteration and all others are irrelevant, program reduction can remove all irrelevant iteration after loop unrolling was performed, as shown in Figure 4.4.

4.2.3 Dependency Optimization

Many programming languages have a mechanism to share code, e.g. modules in NodeJS [24] or GoLang [13]. To use shared code, a program usually has to import it using a special keyword, e.g. `import` in GoLang. Using static analysis, a heuristic can identify

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
foo: {
    a: 2
    b: 0
}

bar: {
    for k, v in foo {
        (k): 10 / v
    }
}
```

```
foo: {
    a: 2
    b: 0
}

bar: {
    a: 10 / 2
    b: 10 / 0
}
```

```
bar: {
    b: 10 / 0
}
```

Figure 4.4: Motivating example for loop unrolling. The property to keep is a division by zero error. The loop in the input on the left is unrolled by the heuristic first, and reduced by a syntactic reducer to only retain the lines contributing to the error.

```
import (
    "strings"
    "encoding/json"
)

foo: {
    a: strings.Repeat("a1", 5)
    a: int
}
bar: json.Marshal(foo)
```

```
foo: {
    a: ""
    a: int
}
bar: ""
```

Figure 4.5: Example for dependency optimization. Variable *a* is defined with type *string* on line 7 and *int* on line 8, causing a conflict. All imported packages can safely be removed, since they do not contribute to the issue.

unused imports or remove used imports by replacing all usages with other expressions. These expressions have one required property: they must be admissible in the current context. To determine whether an expression is admissible and to generate syntactically valid candidates, the grammar of a specific programming language is required. If a programming language has a type system, the modified expression must adhere to all the type constraints of the original expression. For the purpose of program reduction, a reference implementation could replace expressions with a trivial instance of the allowed type. An example is shown in Figure 4.5, where packages `"strings"` and `"json"` are removed from a CUE program by replacing usages with an empty string literal.

4.2.4 List Literal Reduction

Languages like CUE [9] or JavaScript [12] provide literals to define a list or an array. During the evaluation of syntactic reducers such as Perses [27] and PICIRENY [15] (which implements modern HDD), we noticed that list literals are not reduced. This is likely caused by the specifics of the implementation and how list literals are represented

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
foo: 1
if foo < 2: {
    a: 2
}
```

```
foo: 1
if true: {
    a: 2
}
```

```
foo: 1
if false: {
    a: 2
}
```

Figure 4.6: Trivial approach to simplify conditions. A condition is replaced with the literals *true* and *false*

in a grammar, motivating this heuristic.

The list literal heuristic aims to remove as many list items in a literal as possible. To simplify the approach, we implement a variant which removes all items in a list literal.

4.3 Logical Heuristics

Logical heuristics leverage boolean expressions to find reduction opportunities. Since many programming languages have boolean expressions, such as C or CUE, we consider these heuristics to be versatile.

4.3.1 Simplifying conditions

A common control structure in programming languages is a branching statement, referred to as if-then-else. These structures include at least one condition in the form of an expression returning a boolean value and a block of statements that is only executed if the condition yields *true*. A trivial approach to simplify the conditions for program reduction produces two alternatives: once with the condition set to *true* and once set to *false*, as shown in Figure 4.6.

Once a condition consists of a boolean literal, it is trivial for a second heuristic to reduce the tokens of a condition statement. For *true*-literals, the heuristic removes the condition and keeps the body of the clause, whereas conditions with *false*-literals are completely removed.

An alternative and more advanced heuristic tries to evaluate a boolean expression in the current context, and thus simplifies a conditional control structure. This approach requires evaluation of all expressions used in the condition.

4.3.2 Type reduction

Inspired by Data Type Simplification & Elimination used in Zhang et al. [36], we propose a heuristic to reduce types. Boolean expressions occur not only as values in programming languages, there are also some languages where such constructs can be used in the type system. See TypeScript [29], where the | operator is used to combine several types or values to a union type. The type *string* | *number* accepts values of both types *string* and *number* while a type A & B produces the intersection of two types, i.e. all common

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
import "strings"

#foo: "stage" | "prod"
#bar: string \
    & strings.MinRunes(4) \
    & strings.MaxRunes(8)

env: "stage" & #foo
app: "proxy" & #bar
```

```
#foo: "stage"
#bar: string

env: "stage" & #foo
app: "proxy" & #bar
```

```
env: "stage"
app: "proxy"
```

Figure 4.7: Example of type reductions in CUE. A union type *#foo* is reduced to only keep used options while constraints from *#bar* are removed. Both definitions are then inlined in a separate step.

properties are retained. CUE has similar capabilities to create union types, and extends intersections to add constraints to types. Figure 4.7 shows a definition `#foo` that allows the values `stage` and `prod` while the definition `#bar` constraints a string to have a length between 4 and 8.

The union type can be reduced to the only option used `"stage"` and the constraints are removed from the type `#bar`. In a second step, the values `env` and `app` are further simplified, by removing the constraint on `#foo` and `#bar`, respectively.

4.4 CUE-specific Heuristics

All heuristics described above in Sections 4.2 and 4.3 can be applied to several programming languages. In contrast, the heuristics in this section are specific to the CUE programming language [9]. These approaches originate from the experiences of experts at CUE. During the development of CUE, users report issues with various program instances. To find an issue, CUE developers try to reduce the instance as much as possible by leveraging their experience and removing unnecessary tokens from the original instance. During this manual program reduction process and the analysis of their work, they found several strategies for efficient reduction which are refined and described as heuristics in this section.

4.4.1 String Interpolations

CUE allows strings to include so-called interpolations. String interpolations are delimited by an escape sequence: `\(...)`. If these characters occur inside a string literal, e.g. `"hello \(name)"`, the contents inside the parenthesis are evaluated like any other expression. With a variable `name: "john"`, the string will evaluate to `"hello john"`. Any valid CUE expression is allowed in a string interpolation as long as the result is of type string, boolean, number, or bytes.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
1 name: "john"
2
3 fullname: "\(name) doe"
```

```
1 name: "john"
2
3 fullname: "john doe"
```

```
1 fullname: "john doe"
```

Figure 4.8: Simple example of string interpolation. The expression in *fullname* is evaluated, essentially making variable *name* redundant.

```
1  import "strings"
2
3  #env: {
4      name: "stage" | "prod"
5      secret: string \
6          & strings.MinRunes(8)
7  }
8
9  stage: #env & {
10     name: "stage"
11     secret: "superdifficult"
12 }
```

```
1 import "strings"
2
3 stage: {
4     name: "stage" | "prod"
5     secret: string \
6         & strings.MinRunes(8)
7     name: "stage"
8     secret: "superdifficult"
9 }
```

Figure 4.9: Example to remove a type intersection by inlining all constraints.

The heuristic for string interpolations (demonstrated in Figure 4.8) evaluates an expression and replaces the interpolation `\(...)` with the resulting value, if valid.

4.4.2 Type Intersection

Values and types are unified to the same concept in CUE. This property enables CUE configurations to define constraints for fields similarly to the assignment of a concrete value. In CUE, this is typically achieved by using the `&` operator. Figure 4.9 (left) shows an example, where `#env` is a definition used to provide constraints on properties `secret` and `name`. `name` is defined as a string with possible values `stage` and `prod`. `secret` is constrained to a string with length greater than 8. The definition of `stage` on line 9 combines the constraints of `#env` with concrete values. The example shows a valid assignment since all properties adhere to the constraints, i.e. `"stage"` is trivially an instance of `"stage"` or `"prod"` and `"superdifficult"` has length $14 > 8$.

Our proposed heuristic moves the constraints into the same struct as the concrete value definition, thus enabling other heuristics or a syntactic reducer to remove the definition (`#env`) and possibly any statement or constraint unnecessary to retain property ψ. This heuristic retains the same semantics as the original input.

4.4.3 Redundancy Removal

During analysis of preliminary results when applying the syntactic reducer on CUE configurations, we found some obvious redundancy patterns, which are not removed

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
foo: {
    {
        a: 5
    }
}
```

```
foo: {
    a: 5
}
```

```
{
    "foo": {
        "a": 5
    }
}
```

Figure 4.10: Extra nesting will result in the same output when the CUE configuration is evaluated. Both examples on the left result in the JSON output on the right.

```
a: {
    null
    null
}

b: {
    a
    a
}

c: {
    _
    _
}
```

```
a: {
    null
}

b: {
    a
}

c: {
    _
}
```

Figure 4.11: Duplicate declarations containing *null*, an identifier or *top* will be removed.

by the syntactic reducer. These redundant constructs can be removed with targeted heuristics.

Nesting of structs is one such construct. A struct contains one or more fields in CUE. Fields can again contain a struct and create nested structs. If the field name is omitted, it is possible to create redundant nested structs, as in Figure 4.10. Since CUE evaluates such nested structs the same as a struct with one layer, this nesting can be removed.

Package declarations are the first statement in a CUE file with the purpose of defining which files belong to which package. It also defines the default import name. The reduction process targets one file, therefore the package information is not necessary for most configurations to retain property ψ and can be removed by a trivial heuristic.

Syntactic reducers can produce CUE configurations containing redundant declarations. Such declarations do not contribute to the output of a CUE configuration when exported and do not impact the evaluation. An example is a duplicate expression used as a declaration in a struct, as seen in Figure 4.11.

A simple heuristic is used to remove such declarations when they contain: `null`, `top` or an identifier.

An ellipsis ... is used in lists or structs to mark them as open, i.e. it is explicitly allowed to add further elements or properties to the respective value. This is a redundant

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
foo: {
    let x = 42
    y: x * 2
}

for i in [0, 1]
let j = i * 2 {
    "i_\(i)": j
}
```

```
foo: {
    x: 42
    y: x * 2
}

for i in [0, 1]
let j = i * 2 {
    "i_\(i)": j
}
```

Figure 4.12: Example of let expression replacement. A let expression in line 2 is transformed to a regular field, while a let expression inside a clause in line 7 is not modified.

expression for structs, since they are extensible by default. Lists are declared open using ... at the end and can add a type constraint for further elements by adding a type to the ellipsis, e.g. ...*int*. To simplify a concrete CUE configuration and further reduce tokens, these occurrences of ... are removed by a heuristic.

4.4.4 Let Expressions

CUE offers various options to assign a value to a label (or variable). One of these options are let expressions. They are used to bind expressions to an identifier in a local scope and are not exported by CUE [11]. If a let expression is used in a struct definition, it can be simplified to a regular field definition instead. This removes at least one token and provides further reduction opportunities for heuristics working with regular fields only. However, if an expression is used inside a comprehension such as a *for*- or an *if*-clause, they cannot be replaced since the grammar only allows let expressions at this place (see Figure 4.12).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

CHAPTER 5

Evaluation

Our framework SeRu was implemented for two syntactic reducers: Perses [27] and Vulcan [34]. Therefore, it is compared to these two reducers.

We perform the evaluation on a total of 16 real-world CUE configurations. These configurations were provided by CUE developers and originate from GitHub issues in the official CUE repository. CUE developers manually reduced the original configurations of 7 issues to several versions per issue for a total of 16 CUE configurations. The maximum number of versions for a single issue is 3 while the minimum is one.

Some of these configurations included more than one file. Since all reducers under test, including SeRu, only support reduction of a single file, all configurations containing more files were merged using a combination of manual work and the `cue def` command. We refer to merged configurations as the *inlined* version. Evaluation was only performed on inlined configurations for instances with more than one file.

All instances including their inlined versions are provided in the artifact for SeRu. A detailed view of the input instances is given in Table 5.1.

The severity column in Table 5.1 groups issues into error types: semantic, panic, error, and other. Semantic errors occur when the output of a CUE configuration differs from the language specification. A panic is a type of error that occurs within the GO programming language, which is the programming language used to implement CUE. It is used to indicate unexpected behavior and can be recovered. An error occurs when a command fails according to the language specification, e.g. when illegal input is used. Other instances do not fit into any of the above groups.

The instances provide a command that runs successfully on one version of CUE and fails on another. These commands are used during the reduction process to represent the property ψ. If and only if a command runs successfully, the property ψ holds for the instance. Three distinct features of CUE are used in the commands of all test

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Severity	Issue	Version	Tokens	Manual reduction rate	Property Command
semantic	2218	v1 v2 final	63 39 37	58.73% (39/63)	eval
panic	2490	v1 v2 final	420 67 48	11.43% (48/420)	cmd
	2584	v1 v2 final	251 106 25	9.96% (25/251)	eval
error	2209	v1 final	274 124	45.26% (124/274)	export
	2246	v1 final	110 34	30.91% (34/110)	export
	2473	v1 final	67 35	52.24% (35/67)	export
other	2	final	249	-	export*

Table 5.1: Token sizes of CUE instances used for evaluation.
* the command was used with an experimental option

instances: `cue eval` combines all definitions and checks constraints, `cue export` requires concrete values in addition to the evaluation and `cue cmd` runs a custom defined procedure.

Since some of the instances were manually reduced by CUE developers, we include the achieved reduction rate (lower is better), i.e. tokens of the final version divided by the first version of the respective issue.

5.1 Methodology

The evaluation was performed on a 2021 Apple MacBook Pro with an M1 Pro CPU (8 cores) and 16 GB of memory.

We evaluate 4 tools: Perses [27], Vulcan [34], SeRu+Perses and SeRu+Vulcan on all of the 16 CUE input configurations. Each evaluation run is performed 5 times for every instance and framework configuration with a timeout of 4 hours, which sums up to 80 runs per tool and a total of 320 runs.

Two instances - issue 2246 (v1) and issue 2209 (v1) - did not terminate within the timeout with Vulcan configurations and are therefore excluded in the results.

We use version 1.8 for Perses and Vulcan, which can be found implemented in the same tool on GitHub [25]. This version was slightly modified to generate statistics for every run even if it failed unexpectedly.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

To add support for CUE to Perses and Vulcan, we created a grammar specification in the ANTLR [2] format based on the official language specification for CUE and used Perses-adhoc method by Tian et al. [28] to generate the appropriate extension.

5.1.1 Metrics

We measure the following metrics in the evaluation:

- **Tokens**
  Represents the total size of the reduced program, lower is better. Token size is the main metric used to compare reducers, as it is used in many works (non-exhaustive list) [19, 23, 27, 34–36]. A token is a string of characters that has some defined semantics in a programming language. All tokens are counted using the official parser published in the CUE repository on GitHub [9] (specifically the scanner API). Perses and Vulcan use a generated parser based on our ANTLR [2] grammar for CUE and could therefore output different token counts during their respective reduction process or afterwards. To achieve a fair comparison, we count tokens on all output files equally with the official parser.

- **Queries**
  The number of property tests performed during the reduction process [23, 27, 34]. Given the assumption that the property test runs in constant time, this metric gives an indication of the runtime complexity of a specific instance. Therefore, a lower amount of queries is better. Since our proposed framework SeRu runs the configured syntactic reducer one or multiple times, the number of queries is always equal to or greater than the number of queries performed by the syntactic reducer alone.

- **Reduction rate**
  A reduction rate is defined as the ratio of token size after reduction and before reduction (lower is better):

$$RR = \frac{\#output\ tokens}{\#input\ tokens} * 100\% \tag{5.1}$$

  It is used as a relative metric to compare reducer performances without using the absolute token sizes of test instances.

- **Execution time**
  The Execution Time (ET) in seconds measures the total time a reducer uses to generate an output and terminate. For SeRu, an additional metric is introduced: semantic reduction execution time (SET). It measures the time SeRu spends applying heuristics to the input, including any property tests.

- **Efficiency**
  The efficiency of a reducer represents the speed of the reduction process. It is

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

measured in tokens per second and is defined as

$$E = \frac{\#removed\ tokens}{execution\ time} \tag{5.2}$$

A high efficiency is a desired property of a reducer.

5.1.2 Ablation study

To determine whether any heuristic performs better or worse than others, we performed an ablation study. Such studies are commonly applied in machine learning systems but can be applied to any system with multiple components. In this study, we evaluate SeRu+Perses on all input configurations, but enable only one heuristic at a time. With 16 input instances, 5 runs and a total of 14 heuristics, we ran a total of 1120 evaluation runs. These runs took 2.5 hours to finish.

We noticed that one specific heuristic - constant propagation (see Section 4.2.1) - generates more candidates than others. As mentioned in Section 4.1.3, heuristics define their own criteria for applicable locations in a program. The constant propagation heuristic can be applied to usages of identifiers, which occur commonly in our dataset. Since this heuristic can also add tokens to the program and to explore its impact, we decided to do an additional evaluation run with all heuristics except constant propagation.

The focus of this evaluation is the effectiveness of each heuristic and not the best overall configuration for program reduction, therefore, we run the ablation study with one syntactic reducer, Perses.

5.2 Results

Tokens Table 5.2 shows tokens per instance (#T) before and after applying reducers. Both SeRu configurations achieve equal or fewer tokens in all instances compared to the respective syntactic reducer alone (4 equal for Perses and 6 equal for Vulcan). Issue 2584 final is the only instance where none of the tested tools achieves any reduction while also being the smallest instance.

SeRu+Vulcan achieves the fewest tokens in 11 instances while SeRu+Perses achieves the fewest tokens in 9 of all instances (5 of them are equal on both tools). In Issues 2584/v1, 2490/final and 2246/final, SeRu+Vulcan performs equal when added to Vulcan. The same occurs in issue 2246/v1 with Perses performing equal to SeRu+Perses. Issue 2246 is therefore the only instance where SeRu did not affect the results in terms of token size. The absolute and relative improvements achieved using SeRu are shown in Figure 5.1.

Queries The number of property tests (#Q), `queries`, are given in Table 5.2. The query columns for the SeRu configurations show the number of queries performed by SeRu and an estimate of the total number of queries performed by SeRu and the respective syntactic reducer. This estimate is based on the query count of syntactic reducers and the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Instance			Tool								
Severity	Issue	Version	Before	Perses		Vulcan		Seru+Perses		Seru+Vulcan	
			#T	#T	#Q	#T	#Q	#T	#Q	#T	#Q
semantic	2218	v1	63	47	221	42	2535	47	35 (477)	**37**	45 (7650)
		v2	39	32	135	32	1570	**29**	60 (600)	**29**	40 (4750)
		final	37	32	115	32	1486	**29**	60 (520)	**29**	30 (4488)
panic	2490	v1	420	152	2444	130	16218.8 (±0.447)	85	1565 (13785)	**84**	285 (65165)
		v2	67	49	247	49	3387	**44**	130 (1118)	49	35 (6809)
		final	48	44	156	**42**	2028	**42**	75 (543)	**42**	25 (4081)
	2584	v1	251	53.4 (± 0.894)	578 (±1.581)	**26**	1805 (±1.732)	42.2 (± 1.64)	235 (2547)	**26**	20 (3630)
		v2	106	48	365	31	2129	**29**	165 (1625)	31	40 (4298)
		final	25	**25**	21	**25**	609	**25**	25 (67)	**25**	25 (1243)
error	2209	v1	274	145.2 (± 4.025)	1537.6 (±25.491)	-	-	**134**	925 (7181)	-	-
		final	124	100	562	56	5610	83	390 (2638)	**35**	190 (22630)
	2246	v1	110	**45**	485	-	-	**45**	730 (1700)	-	-
		final	34	30	70	**26**	1553	30	150 (290)	**26**	105 (3211)
	2473	v1	67	41	300	33	1926	36	240 (1440)	**28**	180 (5958)
		final	35	31	129	31	788	**30**	75 (462)	**30**	75 (2439)
other	2	final	249	127	982	107	12005	**75**	675 (4603)	77	600 (48620)

Table 5.2: Results of each instance and version. #T represents the number of tokens after reduction or in the input file for column "Before". #Q is the number of queries or property tests. Seru columns show the queries performed by Seru itself and an estimate of the queries performed by Seru plus the syntactic reducer (Perses or Vulcan).

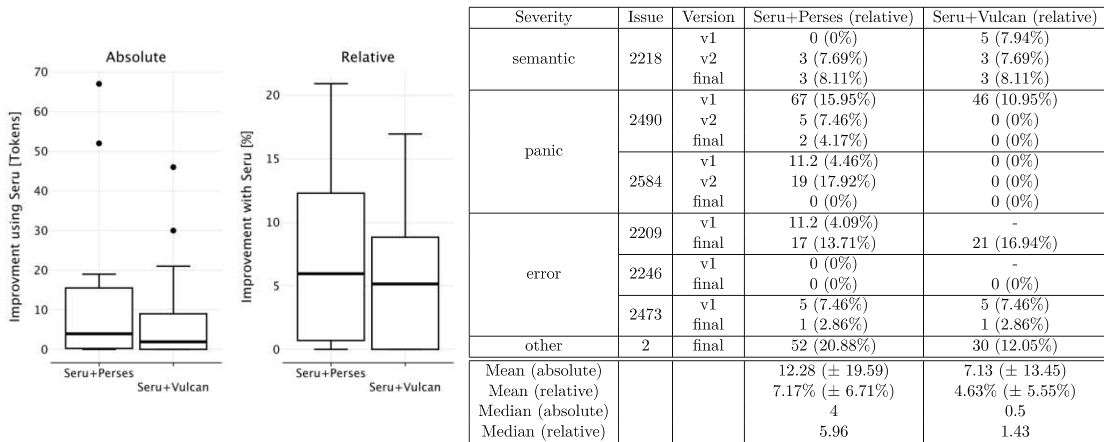
Severity	Issue	Version	Input tokens	Perses [%]	Vulcan [%]	Seru+Perses [%]	Seru+Vulcan [%]
semantic	2218	v1	63	74.60	66.67	74.60	58.73
		v2	39	82.05	82.05	74.36	74.36
		final	37	86.49	86.49	78.38	78.38
panic	2490	v1	420	36.19	30.95	20.24	20
		v2	67	73.13	73.13	65.67	73.13
		final	48	91.67	87.50	87.50	87.50
	2584	v1	251	21.27	10.36	16.81	10.36
		v2	106	45.28	29.25	27.36	29.25
		final	25	100	100	100	100
error	2209	v1	274	52.99	-	48.91	-
		final	124	80.65	45.16	66.94	28.23
	2246	v1	110	40.91	-	40.91	-
		final	34	88.24	76.47	88.24	76.47
	2473	v1	67	61.19	49.25	53.73	41.79
		final	35	88.57	88.57	85.71	85.71
other	2	final	249	51	42.97	30.12	30.92

Table 5.3: Reduction rates of all tested tools (lower is better)

iterations of SeRu per instance. As the configured reducers Perses [27] and Vulcan [34] monotonically reduce an input, the number of queries with the original input represents an upper bound.

The number of queries performed by SeRu is less than that of a syntactic reducer - except in issue 2584/final and 2246/final for SeRu+Perses. Due to SeRu using a syntactic reducer one or multiple times, the total number of queries is greater than the queries

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



Severity	Issue	Version	Seru+Perses (relative)	Seru+Vulcan (relative)
semantic	2218	v1	0 (0%)	5 (7.94%)
		v2	3 (7.69%)	3 (7.69%)
		final	3 (8.11%)	3 (8.11%)
panic	2490	v1	67 (15.95%)	46 (10.95%)
		v2	5 (7.46%)	0 (0%)
		final	2 (4.17%)	0 (0%)
	2584	v1	11.2 (4.46%)	0 (0%)
		v2	19 (17.92%)	0 (0%)
		final	0 (0%)	0 (0%)
error	2209	v1	11.2 (4.09%)	-
		final	17 (13.71%)	21 (16.94%)
	2246	v1	0 (0%)	-
		final	0 (0%)	0 (0%)
	2473	v1	5 (7.46%)	5 (7.46%)
		final	1 (2.86%)	1 (2.86%)
other	2	final	52 (20.88%)	30 (12.05%)
Mean (absolute)			12.28 (± 19.59)	7.13 (± 13.45)
Mean (relative)			7.17% (± 6.71%)	4.63% (± 5.55%)
Median (absolute)			4	0.5
Median (relative)			5.96	1.43

Figure 5.1: Improvement in tokens (relative improvement in parentheses) when using Seru over just using a syntactic reducer (Perses and Vulcan). Vulcan data points for instances 2209/v1 and 2246/v1 are not included since Vulcan does not terminate on those inputs.

performed by a syntactic reducer alone (given in the parentheses of the last two columns of Table 5.2).

Reduction rate All rates are given in Table 5.3. Since Vulcan does not terminate for issues 2209/v1 and 2246/v1, there is no reduction rate for these instances. The absolute and relative improvements when using SeRu over a syntactic reducer alone are given in Figure 5.1. The reduction rates improve significantly ($\alpha = 0.05$, paired t-test) for Perses and Vulcan when SeRu is used, with p-values of 0.0007 and 0.0045, respectively.

Execution time & Efficiency The execution times per instance together with the efficiency are given in Table 5.4. For configurations using SeRu, the time spent in language-specific heuristics is given in column SET or semantic execution time. Execution times for Perses grow for all instances when adding SeRu. This is also reflected in the efficiency metric, where the removed tokens per second are lower in the Seru+Perses configuration. With Vulcan on the other hand, there are 7 instances where the execution time is lower when adding SeRu. The efficiency grows on 9 instances when SeRu is added to Vulcan. The time spent on semantic reduction, i.e. the heuristics applied by SeRu, is greater in 11 instances for the Seru+Perses configuration compared to Seru+Vulcan (+2 instances only terminating with Perses). Issues 2246/final and 2473/v1 take the longest to finish with Vulcan and Seru+Vulcan configurations, with approx. 45 minutes and 2 hours, respectively.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Instance			Perses		Vulcan		Seru+Perses			Seru+Vulcan		
Severity	Issue	Version	ET [s]	E [#/s]	ET [s]	E [#/s]	ET [s]	SET [s]	E [#/s]	ET [s]	SET [s]	E [#/s]
semantic	2218	v1	2.06 ± 0.02	7.77	25.64 ± 0.77	0.819	6.30 ± 2.15	2.07 ± 2.13	2.54	30.40 ± 0.07	1.34 ± 0.02	0.855
		v2	1.31 ± 0.02	5.32	16.17 ± 0.43	0.433	7.10 ± 0.57	2.64 ± 0.57	1.41	18.60 ± 0.09	1.23 ± 0.00	0.538
		final	1.18 ± 0.01	4.24	15.69 ± 0.43	0.319	6.93 ± 0.76	2.63 ± 0.75	1.15	17.78 ± 0.09	0.95 ± 0.08	0.450
panic	2490	v1	22.69 ± 0.12	11.81	243.65 ± 3.19	1.190	76.18 ± 4.66	51.23 ± 4.32	4.40	473.29 ± 14.34	9.78 ± 0.39	0.710
		v2	2.52 ± 0.02	7.14	71.58 ± 1.57	0.251	13.12 ± 1.02	6.93 ± 0.88	1.75	58.38 ± 1.10	1.12 ± 0.07	0.308
		final	1.75 ± 0.18	2.28	56.25 ± 0.74	0.107	7.67 ± 1.18	3.83 ± 1.19	0.78	47.17 ± 0.29	0.84 ± 0.08	0.127
	2584	v1	5.09 ± 0.13	38.79	13.43 ± 0.24	16.754	13.80 ± 0.47	6.47 ± 0.41	15.13	9.90 ± 0.11	0.64 ± 0.09	22.717
		v2	3.52 ± 0.03	16.47	20.30 ± 0.17	3.695	10.54 ± 0.31	4.61 ± 0.28	7.30	15.65 ± 0.67	1.44 ± 0.66	4.793
		final	0.25 ± 0.00	0.00	8.73 ± 0.12	0.000	1.67 ± 0.02	0.69 ± 0.01	0.00	7.62 ± 0.09	0.76 ± 0.09	0.000
error	2209	v1	14.26 ± 0.40	9.04	-	-	44.62 ± 0.40	26.24 ± 0.36	3.14	-	-	-
		final	5.18 ± 0.09	4.63	55.05 ± 0.66	1.235	22.64 ± 0.25	10.82 ± 0.14	1.81	79.31 ± 4.29	6.13 ± 0.53	1.122
	2246	v1	4.83 ± 0.05	13.47	-	-	35.79 ± 6.30	28.97 ± 6.21	1.82	-	-	-
		final	0.92 ± 0.02	4.35	2919.88 ± 76.49	0.003	7.90 ± 0.80	4.58 ± 0.81	0.51	2520.25 ± 101.31	3.36 ± 0.05	0.003
	2473	v1	2.77 ± 0.04	9.39	7512.71 ± 132.75	0.005	12.38 ± 0.28	6.66 ± 0.26	2.50	7062.16 ± 164.57	8.83 ± 0.10	0.006
		final	1.16 ± 0.01	3.45	6.91 ± 0.03	0.579	4.75 ± 0.02	2.01 ± 0.01	1.05	16.27 ± 1.11	2.14 ± 0.14	0.307
other	2	final	10.49 ± 0.17	11.63	171.18 ± 0.90	0.830	36.00 ± 2.67	21.68 ± 2.43	4.83	187.67 ± 0.75	19.09 ± 0.65	0.916
Mean			4.999	9.361	795.511	1.874	19.213	11.379	3.133	753.175	4.117	2.347

Table 5.4: Execution times of all tools per instance. *ET* measures the total execution time until the reduction process of the respective tool terminated in seconds. *E* measures the efficiency or how many tokens are removed per second. *SET* measures the semantic execution time, which is the time spent executing language-specific heuristics within the Seru framework, hence it is only available for Seru configurations.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.2.1 Ablation Results

The results for the ablation study are provided in Figures 5.2 and 5.3. Both figures show the improvement of each heuristic compared to Perses and include the default configuration of SeRu with all heuristics enabled. To see the exact values for all heuristics enabled, refer to Figure 5.1. An additional run is included in the graphs: all heuristics except constant propagation (see Section 4.2.1). As mentioned in Section 5.1, this run was included to isolate the impact of constant propagation, as this heuristic can add more tokens to a result. The output size per instance is displayed as boxplots as well as individual points (scattered for visibility). In each figure, the configurations are ordered by the median difference to Perses. Figure 5.2 shows the absolute difference in tokens compared to Perses. Figure 5.3 displays the difference in reduction rate in percent when each configuration is compared to the Perses baseline.

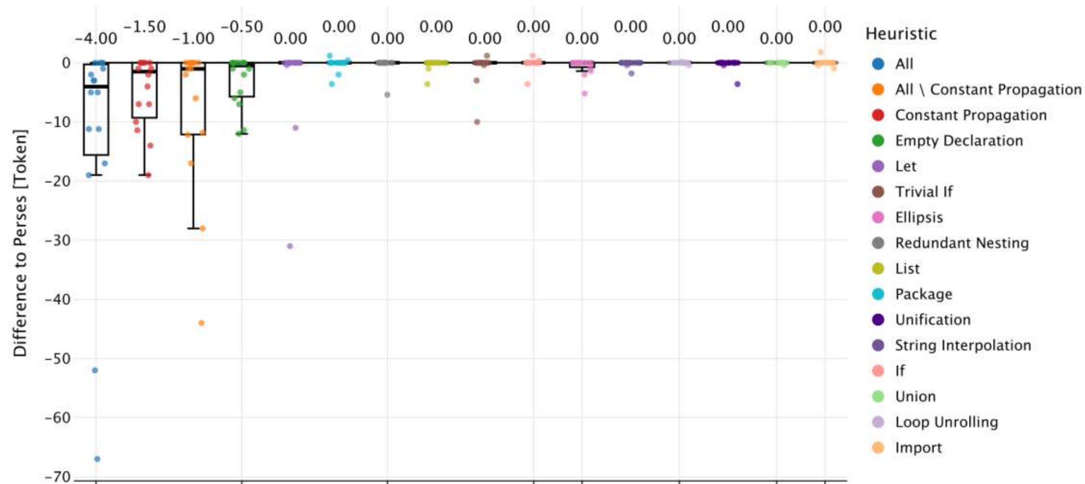


Figure 5.2: The improvement of each heuristic plus an additional run with all heuristics except constant propagation compared to the ground truth reduction performed by Perses. This graph shows the difference of reduced programs in tokens, sorted by the mean value of all instances.

In absolute token difference shown in Figure 5.2, the configuration with *all* heuristics yields the lowest results, with a median value of -4 tokens. With a median difference of -1.5 tokens, *constant propagation* achieves the second lowest result, followed by *all except constant propagation* with -1 and *empty declaration* with -0.5 tokens, respectively. All other heuristics have a median value of zero, with only a few instances resulting in lower values. The *let* heuristic achieves token differences of -31 in issue panic/2490/v1 and -11 in issue other/2/final. The *trivial if* heuristic is able to outperform the more sophisticated *if* heuristic in one instance. Four configurations performed worse on at least one instance compared to Perses: *import* on error/2209/v1,and *package*, *trivial if* and *if* on error/2246/v1. More details on these instances are discussed in Section 6.5.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

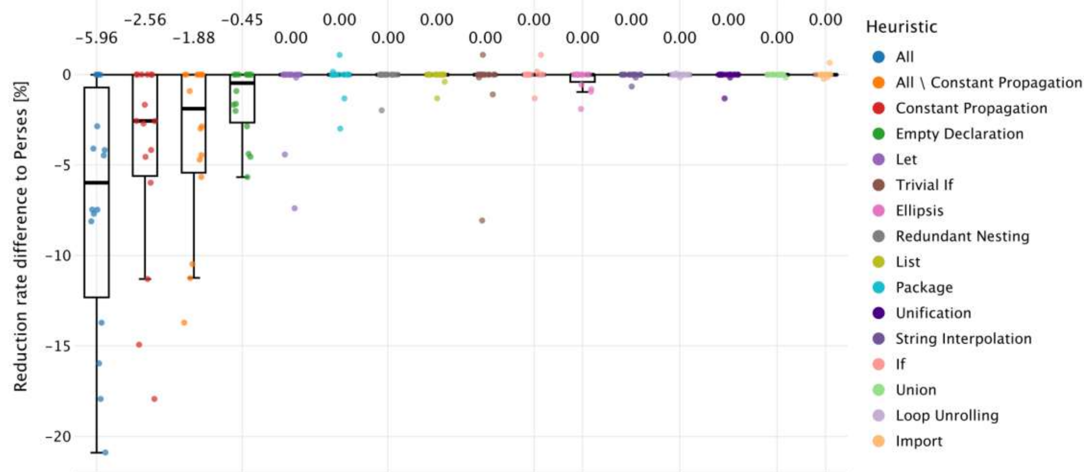


Figure 5.3: Similar to Figure 5.2, but the improvement relative to the reduction rate of the ground truth in percent.

Figure 5.3 provides the improvement of reduction rate for each heuristic. The heuristics are also ordered by their median values, with the lowest on the left.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 6

Discussion

6.1 Effectiveness

As presented in Table 5.2, the SeRu framework performs well and can further improve the results of already very effective reducers such as Vulcan [34]. SeRu improves upon the results of Perses [27] and Vulcan by 7.17 and 4.63 percent, respectively (see Figure 5.1). However, the possible improvement is highly dependent on the concrete instance, which is reflected by the standard deviations of 6.71 and 5.55 percent. The highest improvement for Seru+Perses is instance other/2/final with 20.88 percent, reducing 249 tokens to 75 instead of 127. Seru+Vulcan could improve error/2209/final by 16.94 percent, reducing 124 tokens to 35 instead of 56.

However, not all instances are improved as much and three instances did not improved at all.

1. **panic/2584/final**
   None of the syntactic reducers could improve this instance. So, SeRu has to apply at least one heuristic to achieve further reduction. Only three heuristics were applicable, which generated five candidate programs, but none of which passed the property test, so no further reduction was found.

2. **error/2246/v1**
   Vulcan does not terminate with this instance, so we focus on Seru+Perses for this instance. SeRu started to apply the ConstantPropagation heuristic (see Section 4.2.1), which adds redundancy to a program that is supposed to be removed by a syntactic reducer component. In this case, the intermediate program size was increased from 45 to 104, which was reduced to 73 by Perses. Since 73 is larger than 45, this candidate was dropped and the process was terminated. However, skipping the ConstantPropagation heuristic improves the result to 44 tokens. Applying

heuristics in a different order or just once per iteration could potentially further improve the results, but since finding the perfect order has a large search space, it was left for future work.

3. **error/2246/final**
   Similar to the previous instance SeRu only finds candidates with the ConstantPropagation heuristic. Instead of 68 valid candidates in version 1 of this instance, only 11 are found in this version using Perses and 6 valid candidates when using Vulcan as the syntactic reducer. In this case, skipping this heuristic does not improve the result.

Two of the above-mentioned instances also have the fewest tokens, which is one reason why the reducers and SeRu do not find any improvements. The tokens left in the programs are essential for the program to exhibit the property and can therefore not be removed. The third instance could only be improved by skipping one heuristic and shows that our approach can be further improved, either by using a different application order of heuristics or by adding new heuristics to the framework.

6.2 Queries

The total number of queries is higher for all instances when using SeRu. This is caused by SeRu applying the syntactic reducer at least once and often multiple times during one run. While this approach can improve the reduction rate (see Section 6.1), running the syntactic reducer several times also multiplies the number of property tests. However, in all but three instances, the additional queries performed by SeRu are much fewer than the original query count (often half or a magnitude less). Since the heuristics used by the SeRu framework are targeted at specific language constructs and do not try to remove "random" tokens based on its syntax, fewer candidates are generated.

If the property test is an expensive process, SeRu would not be the best choice.

6.3 Reduction Rate

To facilitate comparison by tool across varying instance sizes, we calculated the reduction rate per instance, which represents the percentage of tokens kept from the original input after the reduction process finished. The results indicate a relation between reduction rate and token size, which is shown in Figure 6.1. The graph shows a tendency in the form of quadratic regression lines for instances with greater token sizes to also have a better (lower) reduction rate, with a slight upwards trend above 400 tokens. However, there is only one instance with more than 400 tokens, therefore the regression might be over-fitted in this area. To make a statement about this relation, studying more instances will be necessary.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The improvement using SeRu is noticeable when comparing the regression lines in Figure 6.1. Seru+Perses in green is lower than Perses in red, and Seru+Vulcan in purple is lower than Vulcan in blue. Although the difference with Perses as the syntactic reducer is greater than Vulcan.

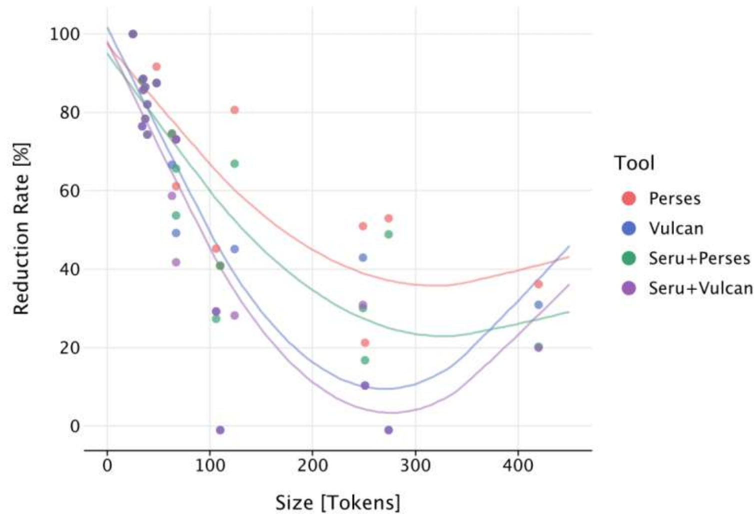


Figure 6.1: Reduction rate by instance size, grouped by tool and with added quadratic regression

Since we have several version per instance and therefore the result of a manual reduction by a domain expert, we compare Perses, Vulcan and SeRu to these results in Table 6.1. Seru+Vulcan achieves an equal reduction rate in instance semantic/2218 and beats the manual reduction rate in instance error/2473. In instance panic/2584, Vulcan and Seru+Vulcan have a reduction rate of 10.36%, exactly 0.4% more than the manual reduction. The Seru+Perses configuration is closest to the manual reduction in instance error/2473, where the reduction rate is 1.49% greater. The instance other/2 has only one version, so there is no manual reduction rate, but it is included as comparison.

Severity	Issue	Manual [%]	Perses [%]	Vulcan [%]	Seru+Perses [%]	Seru+Vulcan [%]
semantic	2218	**58.73**	74.60	66.67	74.60	**58.73**
panic	2490	**11.43**	36.19	30.95	20.24	20.00
	2584	**9.96**	21.27	10.36	16.81	10.36
error	2209	**45.26**	52.99	-	48.91	-
	2246	**30.91**	40.91	-	40.91	-
	2473	52.24	61.19	49.25	53.73	**41.79**
other	2	-	51.00	42.97	**30.12**	30.92

Table 6.1: Comparison of manual reduction rate (final version versus first) and the achieved reduction rates of all tested tools (reduction rate of first version)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.4 Efficiency and Execution Time

Overall, using the SeRu framework increases the total execution time of the reduction process, as shown in Table 5.4. This behavior is expected, since the SeRu framework utilizes a syntactic reducer at least once and up to n times during its main loop. Therefore, the total execution time can be a multiple of the syntactic reducer's execution time.

Table 5.4 includes the time spent by the semantic reduction (heuristics) for SeRu configurations. This metric gives insights about the ratio time spent in syntactic reduction versus time spent in semantic reduction. The semantic reduction takes less time in all instances. Interestingly, there are substantial differences in semantic reduction time between Seru+Perses and Seru+Vulcan configurations. In all but two instances, SeRu spends less time in semantic reduction when using Vulcan, in some cases the difference is up to a factor of 10. One reason for this behavior could be the higher reduction capabilities of Vulcan compared to Perses. Since Vulcan achieves better reduction results than Perses, there are fewer tokens for the semantic reduction to process. Less tokens imply less reduction possibilities and less generated candidates. Therefore, the semantic execution time is shorter when using Vulcan. In other words, adding SeRu to Perses or less potent reducers brings a greater benefit in terms of effectiveness. This is also confirmed by the results given in Figure 5.1 where the improvement with SeRu over Perses is 7.17% on average and 4.63% over Vulcan.

6.5 Ablation study

The ablation study shows that SeRu performs best with all heuristics activated, proving that most heuristics have a positive impact on the output. However, two heuristics do not have an impact when used alone, *loop unrolling* and *union*. While there are some differences for these two heuristics compared to the baseline, they are less than 1 token and are caused by deviating results in the baseline Perses tests, since the heuristics do not find valid candidates during the respective evaluation runs. The *import* heuristic seems to have similar results, but it is successfully applied in one instance to achieve a reduction of a single token.

We want to point out, that the run with only *constant propagation* and the run with all heuristics except constant propagation perform very similarly, in terms of absolute token difference and reduction rate. *Constant propagation* achieves a median difference of -1.5 tokens (-2.56% in reduction rate) compared to Perses while just skipping *constant propagation* with all other heuristics activated achieves a median difference of -1 tokens (-1.88%). Therefore, we conclude that *constant propagation* has the single highest impact of all heuristics, achieving a better result than all other heuristics combined, even though it temporarily increases the size of a reduced program. This fact suggests that heuristics that add more redundancy to a program (like constant propagation) work well in the SeRu framework, were a combination of heuristics and a syntactic reducer is used. We leave the validation with more heuristics as future work.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Interestingly, there is a substantial difference in one result between heuristics *if* and *trivial if*. While *if* tries to evaluate a condition based on available information in the current scope and possibly removes a clause if the condition evaluates to `false`, heuristic *trivial if* simply sets a condition to `true` or `false`. In instance error/2209/final, the more sophisticated *if* heuristic could not generate a valid candidate, but the *trivial if* heuristic could and removed 10 tokens from the instance. These two heuristics confirm the benefit of our approach of combining various heuristics and also shows the application order has impact on the result. In the default configuration of SeRu, the *trivial if* heuristic reduces a condition to `false` and the following *if* heuristic removes the clause completely, leading to further reduction.

The data in the ablation study shows several anomalies in the form of an increase in output size, when adding a heuristic to Perses.

- Instance error/2209/v1 with heuristic *import*
- Instance error/2246/v1 with heuristics *package*, *trivial if* and *if*

In instance error/2209/v1, one of the five evaluation runs deviates in output size, causing the result to contain fewer tokens (138 versus typically 147). Since this deviation did not occur during the runs with only the *import* heuristic enabled, this instance appears to be worse than the baseline.

Similarly, there is a choice in instance error/2246/v1 where the reduction using Perses results in a worse result (51 tokens versus typically 45). In our testing, this result was only observed during runs with heuristics *package*, *trivial if* and *if*, which appears as a result worse than the ground truth for these heuristics.

We consider these deviating results as anomalies and conclude that SeRu inherits properties such as indeterminism in output size from the syntactic reducer it uses.

6.6 Order of Heuristics

The semantic reduction component of SeRu uses a list of heuristics to apply transformations to a program. This list uses the same order for all evaluations. In general, SeRu applies redundancy-removing heuristics first and more complex, potentially size-increasing heuristics last. Heuristics such as *constant propagation* or *loop unrolling* are able to duplicate existing parts of a program and by removing redundancy in these parts as the first steps, we aim to limit the size increase of these heuristics.

We do not require the heuristics to be commutative functions, which opens the question if there exists an optimal order, providing the minimum outputs. Since the ablation study shows that applied heuristics can differ from one instance to another, we believe the optimal application order for heuristics is also instance-specific with no general optimum. We leave the validation of this assumption open for future work.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.7 Implementation details

SeRu is implemented in GOLang 1.23.2. As one goal of this framework is its modularity and extensibility, support for languages is handled by a plugin system. To support a new language, some pre-defined functions must be exposed by the plugin, such as a reduction function and a token counter. A plugin is only responsible for the specific heuristics of a language. SeRu runs the main loop, calls the syntactic reducer and the heuristics and manages intermediate states as well as logging and collection of metrics.

The CUE plugin uses the official GO API for CUE from its source code repository on GitHub [10] (v0.11.1). The syntactic reducers, Perses and Vulcan, use version 1.8 with a small modification allowing the collection of metadata even if the program crashes.

CUE value evaluation The GO API for CUE exposes functions to partially evaluate a CUE program. This API is used to implement several heuristics which utilize evaluated values, such as *if*, *string interpolation*, *loop unrolling*, *unification* and *constant propagation*. However, *constant propagation* uses a simpler approach. It resolves identifiers recursively as long as there is a definition available. With each definition of a struct in CUE, a new scope is created where fields and variables can be defined. *Constant propagation* traverses these scopes from the local scope to the global scope until it either finds a definition for the identifier it tries to replace or fails to do so.

The SeRu framework has the benefit of utilizing native APIs to correctly implement such heuristics because of the modular plugin system.

Grammar During the development of SeRu, we created a program grammar in the ANTLR [2] format for CUE, based on its language specification [11]. This grammar was used to add support for CUE in Perses and Vulcan using the Perses ad-hoc approach [28]. The program grammar must adhere to the specifications for the best reduction results in the syntactic reduction step. Especially complex expressions must be handled correctly, such as CUEs string interpolations, which allow CUE expressions to be inlined and evaluated in a string definition. In an early experimental run, we used a grammar which modeled string interpolations as any other string, causing issues with some instances, as the syntactic reducers generated invalid candidates. Updating the grammar to support interpolations improved the reduction results and fixed any invalid instances.

6.8 Future work

With the concrete implementation of SeRu and the plugin for CUE, we provide an instance where the approach works. However, this implementation only works for one language at the moment. A starting point for future work is to add support for more languages using the existing plugin system. SeRu and the main loop does not have to be re-implemented. To add support for another language, such as Prolog or Datalog, a new

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

plugin containing heuristics for the specific language has to be created and a grammar in ANTLR format has to be supplied to Perses and Vulcan.

Inspired by the work of Zhang et al. [36] and the recent popularity of LLMs for software engineering tasks [18], we experimented with a semi-automated approach to generate SeRu plugins to support further languages. Using versatile heuristics (see Section 4.2), a general prompt can be designed per heuristic to generate code for a language plugin. In our experiments, we saw potential for a very rapid development experience. However, the generated algorithms did include several bugs and often missed edge-cases. A human-in-the-loop approach could be utilized for a future version of SeRu which generates a language plugin with some widely applicable heuristics. This would be a good starting point to support reduction of a language quickly, with the option to refine the process by adding more specific heuristics later on.

Another possible improvement for SeRu is to add more heuristics or reducers. Perses and Vulcan was updated to version 2.0 in January 2025 which includes more fine-grained reduction algorithms which remove parts of tokens [33].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek, Your knowledge hub, WIEN

CHAPTER 7

Conclusion

This work proposes a novel program reduction framework called SeRu. It combines the robust approach of a language-agnostic program reducer (AGR) with language-specific heuristics, that utilize the semantics of a particular language to expose additional reduction opportunities. We implement SeRu and evaluate it with the data validation language CUE. CUE has its roots in logic programming and has declarative syntax, different to most targeted languages within program reduction, such as C or Rust. We propose a total of 14 heuristics, with 4 versatile, 3 logic-specific and 7 CUE-specific heuristics.

SeRu is evaluated on 16 CUE instances with two syntactic reducers (AGRs), Perses and Vulcan. SeRu improves the results of both reducers, with a mean improvement in reduction rate of 7.17% over Perses and 4.64% over Vulcan. The best instance for Perses improves over 20% and 16% for Vulcan. Since SeRu uses an AGR during the reduction framework, it is clear that total reduction times increases.

We performed an ablation study to explore the capabilities of each heuristic separately. The best performing heuristic is *constant propagation* from the set of versatile algorithms. A configuration only applying this one heuristic performs similarly to a configuration with all other heuristics except *constant propagation*. Most CUE-specific heuristics are useful for a handful of instances while posing no benefit for others, however one of them, *empty declaration*, is effective for half instances.

SeRu was manually implemented with heuristics for one programming language, CUE. Other works [36] have already explored the potential of LLMs for program reduction. We believe a future version of SeRu could implement a human-in-the-loop approach to generate specific implementations of versatile heuristics for various programming languages. Thus eliminating the need for manual labor while adding effective reduction transformations and limiting the computing resources required to run a LLM.

We showed that program reduction can successfully be applied to logical languages such as CUE and the combination of syntactic reduction and semantic reduction using heuristics proved to be effective, while adding more execution time. The framework is easily extensible to support more programming languages or use different AGRs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Figures

1.1 Part of a longer CUE configuration file (instance panic/2490/v1) 2
1.2 Results of a reduction process (instance panic/2490/v1). Left shows part of the result for Perses and right shows part of the result of Seru+Perses . . 3

2.1 Example of a Prolog program. 8
2.2 Small instance of a CUE configuration 9
2.3 Unification process of fields in CUE 10
2.4 Example of templating in CUE 10

4.1 The SeRu reduction framework in a simplified view 16
4.2 The Seru framework instantiated for the CUE language [9]. The syntactic reducers require an ANTLR [2] grammar to generate a parser for CUE. Heuristics for the semantic reducer plugin are implemented using the official CUE API for GO [10]. 22
4.3 An example of constant propagation in CUE. Variables are resolved recursively to their respective value over multiple steps. Effectively removing variable dependencies. 23
4.4 Motivating example for loop unrolling. The property to keep is a division by zero error. The loop in the input on the left is unrolled by the heuristic first, and reduced by a syntactic reducer to only retain the lines contributing to the error. 25
4.5 Example for dependency optimization. Variable *a* is defined with type *string* on line 7 and *int* on line 8, causing a conflict. All imported packages can safely be removed, since they do not contribute to the issue. 25
4.6 Trivial approach to simplify conditions. A condition is replaced with the literals *true* and *false* 26
4.7 Example of type reductions in CUE. A union type *#foo* is reduced to only keep used options while constraints from *#bar* are removed. Both definitions are then inlined in a separate step. 27
4.8 Simple example of string interpolation. The expression in *fullname* is evaluated, essentially making variable *name* redundant. 28
4.9 Example to remove a type intersection by inlining all constraints. 28
4.10 Extra nesting will result in the same output when the CUE configuration is evaluated. Both examples on the left result in the JSON output on the right. 29

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.11 Duplicate declarations containing *null*, an identifier or *top* will be removed. 29
4.12 Example of let expression replacement. A let expression in line 2 is transformed to a regular field, while a let expression inside a clause in line 7 is not modified. 30

5.1 Improvement in tokens (relative improvement in parentheses) when using Seru over just using a syntactic reducer (Perses and Vulcan). Vulcan data points for instances 2209/v1 and 2246/v1 are not included since Vulcan does not terminate on those inputs. 36
5.2 The improvement of each heuristic plus an additional run with all heuristics except constant propagation compared to the ground truth reduction performed by Perses. This graph shows the difference of reduced programs in tokens, sorted by the mean value of all instances. 38
5.3 Similar to Figure 5.2, but the improvement relative to the reduction rate of the ground truth in percent. 39

6.1 Reduction rate by instance size, grouped by tool and with added quadratic regression 43

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Tables

5.1 Token sizes of CUE instances used for evaluation. * the command was used with an experimental option 32
5.2 Results of each instance and version. #T represents the number of tokens after reduction or in the input file for column "Before". #Q is the number of queries or property tests. Seru columns show the queries performed by Seru itself and an estimate of the queries performed by Seru plus the syntactic reducer (Perses or Vulcan). 35
5.3 Reduction rates of all tested tools (lower is better) 35
5.4 Execution times of all tools per instance. *ET* measures the total execution time until the reduction process of the respective tool terminated in seconds. *E* measures the efficiency or how many tokens are removed per second. *SET* measures the semantic execution time, which is the time spent executing language-specific heuristics within the Seru framework, hence it is only available for Seru configurations. 37

6.1 Comparison of manual reduction rate (final version versus first) and the achieved reduction rates of all tested tools (reduction rate of first version) 43

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Algorithms

4.1 Main algorithm of the SeRu framework 18
4.2 An example algorithm for *SyntacticReducer* 19
4.3 *SemanticReducer* 20
4.4 Constant propagation 23
4.5 Loop unrolling 24

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Glossary

semantic reducer A group of program reducers which utilize the semantics of tokens and expressions in a program to perform transformations.. 15–17, 19, 20

SeRu Framework to combine semantic and syntactic reducers. Novel approach of this work.. 2, 5, 12, 15, 16, 21, 31–36, 38, 41–47, 49

syntactic reducer A group of program reducers which use the syntax of a language to perform transformations. Some use tokens directly while others utilize the grammar and the AST.. 15–21, 28, 29, 31, 34–36, 42–46

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acronyms

AGR language-agnostic program reducer. 1, 2, 6, 7, 11–13, 49, 50

APR Automatic Program Reduction. 1

AST Abstract Syntax Tree. 5, 6, 12, 15, 19, 23

BNF Backus-Naur form. 7

DD Delta Debugging. 6, 7

HDD Hierarchical Delta Debugging. 6, 7, 11, 15, 19, 25

LLM Large Language Model. 12, 47, 49

SPR language-specific program reducer. 1, 2, 7, 12

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[1] Abdulaziz Alaboudi and Thomas D. LaToza. An Exploratory Study of Debugging Episodes, May 2021. arXiv:2105.02162 [cs].

[2] ANTLR. `https://www.antlr.org/` (accessed 2024-12-06).

[3] Cyrille Artho. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer*, 13(3):223–246, June 2011.

[4] Max Bramer. *Logic Programming with Prolog*. Springer London, London, 2013.

[5] Creating test or performance reproducers · cue-lang/cue Wiki. `https://github.com/cue-lang/cue/wiki/Creating-test-or-performance-reproducers` (accessed 2025-01-17).

[6] A guide to testcase reduction - GCC Wiki. `https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction` (accessed 2025-01-17).

[7] Submit a Bug Report - Oracle Java. `https://docs.oracle.com/en/java/javase/21/troubleshoot/submit-bug-report.html#GUID-A2BFDCEC-9ABF-42C7-A337-B691E1E34C09` (accessed 2025-01-17).

[8] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic Testing: A New Approach for Generating Next Test Cases, February 2020. arXiv:2002.12543.

[9] CUE About page. `https://cuelang.org/docs/` (accessed 2024-11-20).

[10] Github cue-lang/cue. `https://github.com/cue-lang/cue` (accessed 2025-01-09).

[11] The CUE Language Specification. `https://cuelang.org/docs/reference/spec/` (accessed 2024-11-29).

[12] ECMAScript® 2025 Language Specification. `https://tc39.es/ecma262/` (accessed 2024-11-20).

[13] Go Modules Reference - The Go Programming Language. `https://go.dev/ref/mod` (accessed 2024-11-20).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[14] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 380–394, Toronto Canada, October 2018. ACM.

[15] Renáta Hodován. renatahodovan/picireny. `https://github.com/renatahodovan/picireny` (accessed 2024-11-20).

[16] Renáta Hodován and Ákos Kiss. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, A-TEST 2016, pages 31–37, New York, NY, USA, November 2016. Association for Computing Machinery.

[17] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Coarse Hierarchical Delta Debugging. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 194–203, September 2017.

[18] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.*, 33(8):220:1–220:79, 2024.

[19] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. HDDr: a recursive variant of the hierarchical Delta debugging algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, pages 16–22, New York, NY, USA, November 2018. Association for Computing Machinery.

[20] Robert Kowalski. Logic Programming. In *Handbook of the History of Logic*, volume 9, pages 523–569. Elsevier, 2014.

[21] Gereon Kremer, Aina Niemetz, and Mathias Preiner. ddSMT 2.0: Better Delta Debugging for the SMT-LIBv2 Language and Friends. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 231–242, Cham, 2021. Springer International Publishing.

[22] David Maier, K. Tuncay Tekle, Michael Kifer, and David S. Warren. Datalog: concepts, history, and outlook. In Michael Kifer and Yanhong Annie Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*, pages 3–100. ACM, September 2018.

[23] Ghassan Misherghi and Zhendong Su. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 142–151, New York, NY, USA, 2006. Association for Computing Machinery.

[24] Modules: Packages | Node.js v23.2.0 Documentation. `https://nodejs.org/api/packages.html` (accessed: 2024-11-20).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[25] uw-pluverse/perses. `https://github.com/uw-pluverse/perses` (accessed 2024-12-06).

[26] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 335–346, New York, NY, USA, June 2012. Association for Computing Machinery.

[27] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*, pages 361–371, Gothenburg Sweden, May 2018. ACM.

[28] Jia Le Tian, Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yiwen Dong, and Chengnian Sun. Ad Hoc Syntax-Guided Program Reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 2137–2141, San Francisco CA USA, November 2023. ACM.

[29] JavaScript With Syntax For Types. `https://www.typescriptlang.org/` (accessed 2024-11-28).

[30] Sten-Åke Tärnlund. Horn clause computability. *BIT Numerical Mathematics*, 17(2):215–226, June 1977.

[31] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. Probabilistic Delta debugging. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, pages 881–892, New York, NY, USA, August 2021. Association for Computing Machinery.

[32] Guancheng Wang, Yiqian Wu, Qihao Zhu, Yingfei Xiong, Xin Zhang, and Lu Zhang. A Probabilistic Delta Debugging Approach for Abstract Syntax Trees. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 763–773, October 2023. ISSN: 2332-6549.

[33] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Jiarui Zhang, Puzhuo Liu, Yu Jiang, and Chengnian Sun. T-Rec: Fine-Grained Language-Agnostic Program Reduction Guided by Lexical Syntax. *ACM Trans. Softw. Eng. Methodol.*, August 2024. Just Accepted.

[34] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, and Chengnian Sun. Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):97:636–97:664, April 2023.

[35] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[36] Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. LPR: Large Language Models-Aided Program Reduction, May 2024. arXiv:2312.13064 [cs].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub