Reusing Constraint Proofs in Symbolic Analysis

Doctoral Dissertation submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Meixian Chen

under the supervision of
Prof. Mauro Pezzè

May 2018

Dissertation Committee

Prof. Walter Binder	Università della Svizzera Italiana, Switzerland
Prof. Mehdi Jazayeri	Università della Svizzera Italiana, Switzerland
Prof. Andrea De Lucia	Università Degli Studi di Salerno, Italy
Prof. Gordon Fraser	University of Sheffield, United Kingdom

Dissertation accepted on 24 May 2018

Prof. Mauro Pezzè
Research Advisor
Università della Svizzera Italiana, Switzerland

Prof. Walter Binder
PhD Program Director

Prof. Michael Bronstein
PhD Program Director

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Meixian Chen
Lugano, 24 May 2018

To my family.

Abstract

Symbolic analysis is an important element of program verification and automatic testing. Symbolic analysis techniques abstract program properties as expressions of symbolic input values to characterise the program logical constraints, and rely on Satisfiability Modulo Theories (SMT) solvers to both validate the satisfiability of the constraint expression and verify the corresponding program properties.

Despite the impressive improvements of constraint solving and the availability of mature solvers, constraint solving still represents a main bottleneck towards efficient and scalable symbolic program analysis. The work on the SMT bottleneck proceeds along two main research lines: (i) optimisation approaches that assist and complement the solvers in the context of the program analysis in various ways, and (ii) reuse approaches that reduce the invocation of constraint solvers, by reusing proofs while solving constraints during symbolic analysis.

This thesis contributes to the research in reuse approaches, with *REusing-Constraint-proofs-in-symbolic-AnaLysis (*ReCal*)*, a new approach for reusing proofs across constraints that recur during analysis. *ReCal* advances over state-of-the-art approaches for reusing constraints by (i) proposing a novel canonical form to efficiently store and retrieve equivalent and related-by-implication constraints, and (ii) defining a parallel framework for GPU-based platforms to optimise the storage and retrieval of constraints and reusable proofs.

Equivalent constraints vary widely due to the program specific details. This thesis defines a canonical form of constraints in the context of symbolic analysis, and develops an original canonicalisation algorithm to generate the canonical form. The canonical form turns the complex problem of deciding the equivalence of two constraints to the simple problem of comparing for equality their canonical forms, thus enabling efficient reuse for recurring constraints during symbolic analysis.

Constraints can become extremely large when analysing complex systems, and handling large constraints may introduce a heavy overhead, thus harming the scalability of proof-reusing approaches. The *ReCal* parallel framework largely improves both the performance and scalability of reusing proofs by benefitting from Graphics Processing Units (GPU) platforms that provide thousands of computing units working in parallel. The parallel *ReCal* framework *ReCal-gpu* achieves a 10-times speeding up in constraint

solving during symbolic execution of various programs.

Contents

Figures

Tables

Chapter 1

Introduction

Program analysis is the automatic process of analysing the behaviour of programs and verifying the desired properties such as correctness and reliability [PY07]. Many program analysis methods, such as model checking, control-flow analysis, data-flow analysis, testing, have been developed to help programmers understand and evaluate complex systems. Nowadays, program analysis techniques are approaching to industrial maturity level and play an important role in software development [GLM12].

Symbolic analysis is one of the popular techniques in the field of program analysis and automatic testing. It uses symbolic expressions (logic formulas over algebraic expressions of typed variables) to describe the possible executions of a program, and relies on constraint solvers to validate the symbolic expressions and thus to verify the properties of interest. *Symbolic execution* is a type of symbolic analysis, whose goal is to determine the execution conditions of program paths [Kin76, Cla76]. A popular application of symbolic execution in industry is to generate test cases to achieve high code coverage of software systems [CDE08, TdH08, CS13]. When used to generate test cases, symbolic execution (i) traverses the program paths and simulates the execution of the statements in those paths by assuming symbolic values as program inputs, (ii) systematically records the conditions to execute each branch in each path, and thus builds a logic formula (called the path condition) that represents the execution conditions of the path with respect to the symbolic inputs, and (iii) attempts to solve the path conditions with an automatic constraint solver to find concrete input values to execute the corresponding program paths, that is, to find a test case for each path. When using constraint solvers to solve a path condition formula, we may (i) either obtain a satisfiability verdict for the formula, along with concrete values that make the formula hold true, and that can thus be used as actual inputs to cause the corresponding program path to execute, (ii) or an un-satisfiability verdict, that is, a proof that the path that corresponds to the path condition is infeasible.

Automatic constraint solving is known to be a very challenging task, both practically and theoretically, and it is a very active research field, with many important results

produced over the last years, but also many open problems yet to be solved [BHvM09, RVBW06, MHL+13]. A widely studied approach to address the problem is to embrace restrictions on the formulas that the constraint solver shall cope with, and thus design constraint solvers that can efficiently compute the solution for formulas that comply with those restrictions. In particular, many popular constraint solvers embrace the Satisfiability Modulo Theories (SMT) approach that consists in considering only formulas with variables that belong to a specified domain (called a *theory* in the terminology of SMT solvers) and within a limited set of algebraic and logic expressions, and thus achieve efficient constraint solvers for a relevant set of decidable theories [BFT16]. However, despite the many relevant results in the last years [DMB08, CGSS13, Dut14, BCD+11], the SMT problem is still a challenge in many cases. Theoretically, depending on the logic and the underlying theory, it could be NP-hard, and in general undecidable [BFT16]. Practically, modern solvers can very efficiently deal with many constraints within the boundaries of the considered logics and theories, but become increasingly less efficient for formulas of increasing size, and impractically slow to solve some constraints that belong to non-trivial logic classes, for instance non-linear constraints [DMB08, SBdP11].

These limitations crucially impact on the efficiency of the program analysis techniques that rely on constraint solvers. As a matter of facts, the symbolic analysis of complex software systems produces enormous path conditions that contain heterogeneous constraints, making constraint solving be the main bottleneck for symbolic analysis techniques to achieve industry scale application. Many previous empirical studies quantified this phenomenon, showing that already to accomplish the symbolic execution of programs where the path conditions consist of only constraints over linear expressions, the time spend on constraint solving accounts for more than 90% of the overall execution time [CDE08, VGD12, BDP16].

To mitigate the impact of constraint solving during symbolic analysis, different approaches have been proposed in the recent years. Other than the many improvements achieved from the scientific community that work on constraint solvers to efficiently target specific domains of logics [BDHJ14, CDW16, BHJ17a, BHJ+17b] several researchers have proposed optimisations that aim to assist and complement the solvers in the context of the program analysis techniques. For example, Palikareva et al. proposed to use multiple constraint solvers in parallel, exploiting constraint solvers optimised for constraints in different logics, aiming to benefit from the fastest answer from any solver [PC13]. Braione et al. proposed to apply rewriting rules to simplify non-linear constraints before querying solvers, thus increasing the number of non-linear formulas that can be successfully solved, and improving the response time of the solver [BDP13]. Erete et al. proposed to exploit information about the domain of the program under analysis to optimise the usage of solvers, rather than using them in black-box fashion [EO11]. Machine learning techniques and heuristic algorithms have also been applied to foster the ability of solving complex constraints [LLQ+16, DA14, TSBB17].

In this thesis, we focus on approaches for reusing proofs while solving constraints during symbolic analysis, moving on beyond recent work that caches and reuses solutions [VGD12, YPK12, CDE08]. In particular, this recent work starts from the observation that, both during the analysis of the same program and across programs, a large amount of the queries to the constraint solver concern equivalent and logical related formulas. This observation led to approaches ([VGD12, YPK12, CDE08]) that complement symbolic execution with caching frameworks that record the proofs of the solved constraints, to reuse these proofs to solve subsequent formulas that are equivalent or related by implication to some already solved formula. The available studies provide initial evidence that this type of approach, which we refer to as *proof caching and reuse*, can significantly reduce the number of queries to constraint solvers, and thus drastically reduce the time required to complete the symbolic analysis of programs.

The innovative idea of reusing proofs to improve the efficiency of symbolic program analysis opens a promising research direction, but also raises significant challenges. A core challenge in caching and reusing proofs is the ability of identifying cached proofs that can be reused to solve new formulas. The seminal work of Visser et al., Cadar et al., and Yang et al. focused on identifying whether a new formula is equivalent to or related by implication with an already solved formula [VGD12, YPK12, CDE08]. The equivalence of two formulas is a sufficient condition to reuse the proof that is available for either formula. The implication between formulas allows to infer the satisfiability of a new formula: if a new formula is implied by a formula already proven to be satisfiable, or implies a formula already proven to be unsatisfiable, the new formula is either satisfiable or unsatisfiable, respectively, and we can reuse the available proof. However, identifying the logical equivalence or the implication relations between constraints is a hard task, and ultimately can be as complex as solving the formulas in the general case.

The seminal approaches restrict their attention on identifying equivalence and implication relations that depend on the syntactic structure of the formulas, possibly after some equivalence-preserving transformations that normalise the structure of the formulas to unveil mutually equivalent parts. Cadar et al. [CDE08] optimise the *Klee* symbolic executor with a constraint caching framework that identifies equivalent and implication related constraints, by matching textually identical logical expressions across the constraints. They introduced a transformation that they refer to as *constraint slicing*, which consists in separating the constraints into logically independent sub-constraints that predicate on distinct set of variables. This reduces the size of the constraints that must be dealt in the caching framework, and thus increases the chances of identifying textual equivalences across the constraints. In the *Green* framework, Visser et al. [VGD12] normalise the constraints by abstracting away the names of the program variables, and exploiting arithmetic rules to reduce the set of comparison operators that appear in the constraints, thus favouring the chances to spot equivalent constraints across the analysis of different programs or different parts of a program.

The seminal work of *Klee* and *Green* indicates the feasibility of reusing proofs to mitigate the impact of constraint solving in symbolic analysis, but addresses the reuse of proofs across a limited class of equivalent constraints and constraints related by implication. As an example, *Green* would not identify as equivalent two constraints that consist of the same sub-formulas listed in different order in a conjunctive formula, because they would appear as different formulas after the renaming the inner variables. As another example, *Klee* would not recognise the implication between the formula $V_1 > 5$ and the formula $V_1 > 1$, because these two formulas do not textually match with each other.

A main contribution of this PhD work is to define a novel canonical form to represent constraints in linear integer arithmetics. Our canonical form allows us to identify equivalences and implications that could not be revealed with approaches that work on the original structure of the very same constraints. Our canonical form crucially extends the class of equivalent constraints and constraints related by implication that can be dealt with in the caching framework. In the thesis, we discuss the properties of our algorithm for computing the canonical form.

Another significant challenge in caching and reusing proofs is the cost of identifying the reusable proofs. Effective approaches must be faster in identifying reusable solutions than constraint solvers in computing the proofs. The experimental results reported in this thesis indicate serious scalability issues for the state-of-the-art proof reusing approaches, scalability issues that are shared by the serial computations of our canonical form algorithm. Indeed, when analysing complex programs with symbolic analysers, the constraints computed during the analysis can become extremely large, up to including hundred of variables and sub-formulas. Applying transformations like simplifications and normalisations to such large constraints inevitably causes heavy computation overhead. In this thesis, we observe that the typical conjunctive structure of the constraints produced in symbolic program analysis is suitable for parallelising the computation of our canonical form, and propose a parallel algorithm that we realise as an instance of General-Purpose computing on Graphics Processing Units (GP-GPU). A GP-GPU platform can provide thousands of computing units working in parallel, and allows us to efficiently compute the canonical form for constraints of arbitrary size.

The main contribution of this thesis is *ReCal*, an approach for *REusing-Constraint-proofs-in-symbolic-AnaLysis*. In details, *ReCal* contributes to the state of the art by:

(i) introducing logical simplification rules, such as eliminating redundant clauses and checking conflicting clauses pairs, to simplify constraints beyond the simple simplifications defined in *Klee* and *Green*,

(ii) defining a notion of equivalence of constraints that goes beyond the simple syntactic normalisation introduced in *Green*, and proposing an efficient canonicalisation algorithm to compute the canonical form of constraints according to our equivalence relation,

(iii) exploiting an innovative parallel algorithm on a GPU-based platform to efficiently compute the canonical form for constraints of arbitrary size,

(iv) introducing rules to identify the logical implications between constraints beyond the simple identification of contained sub-formulas of *Klee*, to enable the reuse of proofs across a large class of mutually stricter and mutually weaker constraints,

(v) developing an efficient caching framework that implements the *ReCal* approach to retrieve proofs over a large repository of constraints,

(vi) presenting a set of experimental results that provide compelling empirical evidence of the benefits of the *ReCal* approach both in absolute terms and in comparison with state-of-the-art approaches for reusing proofs of constraints.

In the last four years, during the development of this PhD work, two new approaches to reusing proofs across constraints were independently proposed by other researchers. In 2015, Jia et al. [JGY15] proposed *GreenTrie*, an approach that extends the *Green* framework of Visser and colleagues with mathematic inference rules to reuse proofs across constraints related by implication. With these rules, *GreenTrie* captures a set of implication-related constraints that largely overlap with the constraints that can be captured with our *ReCal* approach. Both *GreenTrie* and *ReCal* were independently presented at the International Symposium on Software Testing and Analysis (ISSTA) in 2015, where we discussed the synergies of thew two approaches. In the experiments reported in this thesis, we report the results of our empirical comparison of *ReCal* with *GreenTrie* that show that *ReCal* outperforms *GreenTrie* in both effectiveness and efficiency. In 2017, Aquino et al. [ADP17] introduced a new framework (*UtoPia*) for caching and reusing constraints, in which they heuristically identify constraints that share available proofs even without being necessarily mutually equivalent or related by implication. The work of Aquino et al. comes from the same research group, and is seeded in the initial work on *ReCal* that continued with two complementary research lines, the work on *UtoPia* on one side, and the refinement and parallel implementation of *ReCal* on the other side. In the experiments reported in this thesis we report the results of our empirical comparison of *ReCal* with *UtoPia* that show the intrinsic complementarity of these two approaches.

1.1 Research Hypothesis and Contributions

The main research hypothesis of this PhD work is:

> *Constraint proofs computed during symbolic analysis can be effectively reused to incrementally prove new constraints more efficiently than invoking a solver for proving each constrains, thus improving the overall performance of symbolic analysis.*

Previous studies have observed that both equivalent and related-by-implication constraints recur during symbolic analysis and have proposed different approaches to reuse proofs [CDE08, VGD12, BDP16]. Our studies confirm the recurrence of equivalent and related-by-implication constraints, indicate that current approaches can effectively reuse an interesting subset of proofs, and reveal several intrinsic limitations of state-of-approaches in proof reuse, leaving a large space for improvements.

In this PhD work, we contribute to the reuse of constraint proofs during symbolic analysis by

(i) defining a new concept of equivalence of constraints in the context of program analysis,

(i) proposing *ReCal*, an approach to reuse proofs based on the new concept of equivalence,

(i) confirming the ability of *ReCal* to reuse a larger set of proofs than current approaches, by reporting the results of our experimental study on a large set of constraints generated from diverse programs,

(i) defining a parallel algorithm that implements the *ReCal* approach, and largely improves the overall performance of the *ReCal* approach,

(i) presenting the GPU-based proof-reusing framework *ReCal-gpu*, that outperforms all the current approaches in terms of saving time, and

(i) discussing a set of experimental results that confirm the effectiveness and efficiency of *ReCal-gpu*.

We presented the initial results of this PhD work at the Doctoral Symposium and at the main research track of of the ACM International Symposium on SoftwareTesting and Analysis (ISSTA) in 2014 [Mei14] and 2015 [ABC+15], respectively.

1.2 Structure of the Dissertation

This document is organised as follows.

- Chapter 2 discusses the main research directions towards mitigating the impact of constraint solving in symbolic analysis: improving constraint solvers, reducing the amount of symbolic states to analyse, and reusing proofs. The chapter discusses in details the approaches for reusing proofs, which are the approaches most closely related to the research work presented in this dissertation.
- Chapter 3 overviews the *ReCal* approach, discusses the abstraction and simplification steps that all together identify equivalent constraints, and presents the search engine, which extends the reuse of proofs of formulas related by logical implication.

- Chapter 4 introduces the *Canonicalisation* form, a key element for efficiently retrieving reusable proofs, and presents the *Canonicalisation* Algorithm.
- Chapter 5 presents the GPU-based reusing framework *ReCal-gpu*, the parallel computation of the *ReCal* that largely reduces the computational costs.
- Chapter 6 presents the experimental setup, and discusses the experimental results that show the effectiveness of *ReCal* in the comparison with the state-of-the-art.
- Chapter 7 summarises the main contributions of the PhD work and indicates the new research directions that this work opens.

Chapter 2

Constraint Solving in Program Analysis

Symbolic analysis techniques rely on constraint solvers to prove path condition constraints. Despite the recent progress in theory and the maturity of the corresponding tools constraint solvers still represent a main bottleneck for symbolic analysis. This Chapter reviews the achievement of symbolic analysis techniques and the obstacles to achieve scalability, and overviews the main approaches to mitigate the impact of constraint solving during symbolic analysis. It identifies four main research lines, presents in details the work about reuse of proofs, discusses the core limitations of the current approaches, and identifies the opportunities of further improving the reuse of proofs, which inspired the work documented in this dissertation.

Symbolic analysis techniques, such as symbolic execution, are becoming more and more popular in automatic testing and program analysis, and are approaching an industrial maturity level.

For example, modern tools based on symbolic execution exploit many different exploration strategies and path selection methods, to effectively and efficiently explore programs. The most common strategies are *Depth-First Search (DFS)*, which analyses program paths till the end before backtracking to the deepest and nearest unexplored branches, and *Breadth-First Search (BFS)*, which analyses all branches at each decision point before progressing further through the possible program paths. Other approaches investigate randomised path selection heuristics. KLEE [CDE08], for instance, refines the path selection strategy by assigning probability factors to program paths, to favor the paths that were explored fewer times. Other symbolic execution tools, e.g., EXE [CGP+06], KLEE [CDE08], MAYHEM [CARB12], and S^2E [CKC12] introduced heuristics to maximise the code coverage for different classes of programs.

Symbolic analysis relies on *Satisfiability Modulo Theories (SMT) solvers* [DMB08, CGSS13, Dut14, BCD+11] to validate the constraints during exploring program paths. Modern SMT solvers can efficiently handle many complex constraints, but they still represent a main bottleneck to the scalability of symbolic analysis, due to the intrin-

sic complexity of the constraint solving problem. In fact, solving SMT problem is in general very hard: it is NP-complete for classical propositional logic, PSPACE complete for more complex logics such as the quantifier-free non-linear real arithmetic logic, and even undecidable for most quantified logics. Moreover, the need to solve many large constraints, characterised by hundred of variables and hundreds of clauses, exacerbates the problem. Several empirical studies indicate that the time spend in solving constraints accounts for more than 90% of the overall execution time of symbolic analysis [CDE08, VGD12, BDP16].

For example, a constraint that makes constraint solvers struggle, despite being easy to solve intuitively, is the quartic Diophantine equation:

$$A^4 + B^4 + C^4 = D^4.$$

This constraint is known to have infinitely many solutions in the domain of positive integers, but yet Z3 [DMB08] fails to figure out any of such solutions after running for 24 hours.

Many approaches address the constraint solving bottleneck problem, by exploring four main research lines: (i) invoking SMT solvers in parallel, (ii) compensating SMT solvers with external optimisation tailed for specific domains, (iii) reducing the number of program states to explore by identifying and removing redundant states, (iv) reusing constraint proofs to reducing the frequency of querying SMT solvers. This thesis is led by the last thread of approach in proof reusing.

We discuss the first three threads of work in section 2.1. In section 2.2, we investigate the recent approaches for speeding up symbolic analysis by proof reusing. For each approach, we discuss about the scope of application, details of the techniques, and innovative contribution comparing to the other related work.

2.1 Reducing the Impact of Constraint Solving

In this section, we survey the approaches to improve the performance of symbolic analysis that rely on (i) parallel solvers, (ii) external optimisations, (iii) heuristics and machine Learning, and (iv) reducing redundant states.

2.1.1 Parallel Solvers

Modern SMT solvers implement various strategies to decide the satisfiability of constraint formulas, and perform differently when dealing with different kinds of constraints. Thus some solvers may efficiently solve some kinds of formulas, and be much less efficient in solving other kinds of formulas. Palikareva and Cadar observe that for most constraint queries, it is impossible to decide in advance which solvers will perform better, and propose a framework which parallel queries different solvers when solving a target constraint and getting proof from the fastest one [PC13]. Palikareva and Cadar's

framework is integrated in the KLEE symbolic executor [CDE08]. To efficiently invoke different solvers, Palikareva and Cadar observe that formulas can be expressed in the SMT-LIB standard format, which is currently supported by most constraint solvers, but advice against this practice, due to the verbose format of SMT-LIB, which reduces the performance of serializing formulas. Palikareva and Cadar use macros and templates to automatically assert a formula to many solvers by means of their APIs, with a limited overhead.

2.1.2 External Optimisations

SMT solvers are often used in a black-box fashion during symbolic analysis, without integrating available contextual and domain information of the programs, thus leaving many opportunities to optimise the usage of SMT solvers.

Braione et al. propose mathematic rewriting rules to simplify complex constraints, and suggest a combination of lazy initialisation and rewriting rules to exclude invalid inputs, thus simplifying path condition constraints by exploiting either program-specific calculations or the logic of the verification tasks [BDP13].

Erete and Orso optimise the solving procedure during dynamic symbolic execution [EO11]. Dynamic symbolic execution (DSE) runs a program with both concrete and symbolic inputs, and uses concrete execution to drive symbolic execution along a specific path that is guaranteed to be feasible. DSE tracks a path condition for the explored paths and negates the last constraint in the path condition that corresponds to a not-yet-covered branch, to execute new paths [CTS08].

Erete and Orso exploit the contextual information of the concrete inputs to help the solver find a solution to the path conditions corresponding to variations of that path. They suggest to systematically substitute the variables in a target path condition with their associated concrete values to reduce the path conditions, and incrementally use the models built during the evaluation to simplify the constraint solving task.

2.1.3 Heuristics and Machine Learning

Dinges et al. combine linear constraint solving with heuristic search to address nonlinear path conditions [DA14]. Solving nonlinear constraints is in general undecidable. Some solvers, for instance Z3, Coral, and Dreal [DMB08, SBdP11, GAC12] generate proofs for some usually small subset of nonlinear constraints within a reasonable time. Dinges et al. separate complex paths into linear and nonlinear subset, use solvers to generate proofs for the linear sub-constraint, and use heuristic approaches to search for solutions that satisfy the nonlinear sub-constraints, starting from the results of the linear sub-constraints.

Li et al. integrate machine learning technique with constraint solving to tackle complex path condition [LLQ+16]. Li et al.'s approach translates constraint solving

into optimisation problems that they solve through machine learning guided samplings and validations.

2.1.4 Reducing Redundant States

Both Person et al. and Bugrara and Engler reduce the number of states to be explored by identifying redundant program states that include (i) states unaffected by the modification during regression analysis [PYRK11], and (ii) new states that do not improve program coverage during dynamic program analysis [BE13] .

The DiSE framework of Person et al. [PYRK11] identifies the states that differ across incremental versions of the program with a combination of forward and backward static data flow analysis, and symbolically analyse only the new states. The DiSE framework improves significantly the performance of symbolic execution, but it is limited to regression analysis.

Bugrara and Engler [BE13] observe that not all the paths explored during symbolic analysis reach new areas of the code, and exclude the states that do not improve test coverage from analysis, thus speeding up symbolic execution without reducing test coverage. Bugrara and Engler's approach records the explored states and the constraints in the corresponding path conditions, and show that new states whose constraint can be satisfied with a valid combination of solutions (variable assignments) of constraints of known states, will not reach new code regions, and thus may not be explored.

2.2 Reusing Proofs to Speed Up Symbolic Execution

A recent trend of research has studied techniques for reducing the number of queries to constraint solvers by caching and reusing results from previous analysis sessions. The work by Yang et al. on memoized symbolic execution merges states that do not advance towards the coverage [YPK12]. The *Klee* caching frameworks limit the number of queries issued to the constraint solver [CDE08]. Visser et as.'s *Green* approach [VGD12] and the Jia et al.'s *GreenTrie* extension [JGY15] cache formulas together with their solutions in a distributed key-value in-memory database suited for extremely fast local lookups. Aquino et al.'s [ADP17] *Utopia* stores formulas with data that characterise their solution space for retrieving solutions reusable across formulas that share solutions despite their differences.

In the following we provide additional details about *Klee*, *Green*, *GreenTrie* and *Utopia*, the approaches most close to *ReCal*.

2.2.1 KLEE

The *KLEE* symbolic executor targets LLVM (Low Level Virtual Machine) bytecode. It automatically generates test cases that maximize code coverage [CDE08]. *KLEE* reduces

the calls to the constrain solver by incrementally cacheing formulas to reuse their solutions for equivalent and implied formulas.

KLEE implements a *slicing* technique that slices constraints into mutually independent and simpler constraints, that is, constraints whose clauses do not share any variables, as illustrated in the following example.

$$\text{original constraint } F$$
$$x + 2y < 0 \wedge a + b = 0 \wedge x - y > 0$$

$$\text{independent slices}$$
$$\begin{array}{c|c} F_1' & F_2' \\ x + 2y < 0 \wedge x - y > 0 & a + b = 0 \end{array}$$

Slicing may largely reduce the size of the constraints to be solved, due to the way symbolic execution generates and updates path conditions.

KLEE maintains two caching frameworks the *branch cache* that stores the results of individual queries to the constraint solver, and the *counter-example cache* that store unsatisfiable constraints with their unsatisfiability proof. *KLEE* searches the *counter-example cache* by exploiting Hoffmann and Hoehler's UBTree data structure [Com79], which allows efficient searching for both subsets and supersets of constraint sets. *KLEE* uses the counter-example cache to efficiently determine if a new formula either is contained in a satisfiable constraint set or contains any unsatisfiable constraint set solved in the past, to determine that the path condition is satisfiable or not, respectively. Proofs between sub/super constraints can be reused: (i) if a satisfiable super constraint of the target constraint is found, the target constraint can be solved by reusing the proof, (ii) if an unsatisfiable sub constraint of the target constraint is found, we can infer the target constraint is unsatisfiable as well, since the target constraint is equivalent or stricter than any of its sub constraints, (iii) if a satisfiable sub constraint of the target constraint is found, we can try to reuse the proof to solve part of the target constraint. *KLEE* has been successfully experimented to the 89 stand-alone programs in the *GNU coreutils* utility suite, which are becoming a common benchmark in the field.

Looking up for a sub/super set is less straightforward than searching for the exact constraint. *KLEE* propose to use the UBTree data structure [Com79] to store the constraint clauses and retrieval for a sub/super set of clauses. In both reusing strategies, constraints are stored in memory for fast access.

Experiments with over 89 stand-alone programs in the GNU coreutils utility indicate that the *KLEE* cache framework reduces the solving time by 41%.

2.2.2 Green

Visser et al.'s *Green* framework reuses solutions of integer linear path condition constraints [VGD12].

Green identifies equivalent constraints generated also across different programs, by comparing constraints after normalisation. The *Green* normalisation precess of constraints is composed of three steps:

(i) *Slicing* which implement *KLEE*'s slicing;

(ii) *Mathematic simplification*, which implements a set of simple mathematic transformation rules, for instance the term rewriting $x+y=2y \rightarrow x-y=0$, and the division of coefficients by the greatest common divisor: $4x+8y=12 \rightarrow x+2y=3$.

(iii) *Normalisation*. *Green* eliminates program specific details by renaming and reordering variables and terms. After mathematically simplifying the formula, *Green* reorders first the terms in a clause by the values of the coefficients, for example $3x+y=2 \rightarrow y+3x=2$, and then the clauses by lexical order of the coefficients appearing in each clauses. Finally, *Green* rewrites the variable names according to their occurrence in the sorted formulas, for example $y+3x=2 \rightarrow v_1+3v_2=2$.

The normalised constraints as well as the proofs are stored in *Redis* database [Red21], which is a key-value in-memory database suitable for fast local lookups. The *Green* framework does not support proof reusing by logical implication.

In the empirical study of *Green*, the author observe that, equivalent constraints recur not only within the same program, and also from different programs that perform similar function [VGD12].

2.2.3 GreenTrie

Jia et al.'s *GreenTrie* approach extend the *Green* framework with logical implication rules [JGY15]. Being based on *Green* caching framework, *GreenTrie* also targets formulas in QFLIA logic. *GreenTrie* extends *Green* based on satisfiability relations that depend on formula implications: if a formula F is satisfied by a model m, and F implies a formula G, then also the formula G is satisfied by the model m, and if a formula F is unsatisfiable and G implies F, then also G is unsatisfiable.

GreenTrie determines implications between formulas by comparing the clauses that comprise the two formulas using the following syntactical rules:

$$(R1)\ \frac{}{C \implies C} \qquad (R2)\ \frac{n \neq n'}{P+n=0 \implies P+n' \neq 0}$$

$$(R3)\ \frac{n \geq n'}{P+n=0 \implies P+n' \leq 0} \qquad (R4)\ \frac{n \leq n'}{P+n=0 \implies P+n' \geq 0}$$

$$(R5)\ \frac{n > n'}{P+n \leq 0 \implies P+n' \neq 0} \qquad (R6)\ \frac{n > n'}{P+n \leq 0 \implies P+n' \leq 0}$$

$$(R7)\ \frac{n < n'}{P+n \geq 0 \implies P+n' \neq 0} \qquad (R8)\ \frac{n < n'}{P+n \geq 0 \implies P+n' \geq 0}$$

GreenTrie improves *KLEE* strategy of reusing super/sub constraints by storing the constraints in Implication Partial Order Graph (IPOG). In the *GreenTrie* IPOG, the clauses of constraints are associated to nodes and are ordered by the clause order appearing in the normal form.

The type of constraint normalisation exploited in *GreenTrie* (and in *Green*) is however limited for constraints in which the corresponding clauses and variables appear in different orders. For example, both *GreenTrie* and Green, would fail to identify the equivalence of the following constraints:

$$\begin{array}{ll} F1 & 3x + 2y + z \leq 0 \wedge 2x + 3y + z \leq 0 \wedge y \leq 0 \\ F2 & 3a + 2b + c \leq 0 \wedge 2a + 3b + c \leq 0 \wedge a \leq 0 \end{array}$$

$F1$ and $F2$ can indeed be shown to be equivalent by renaming x to b and y to a, but the simplification and normalisation rules of *GreenTrie* and *Green* produce different normal forms, that is,

$$\begin{array}{ll} NF1 & 3V_1 + 2V_2 + V_3 \leq 0 \wedge 2V_1 + 3V_2 + V_3 \leq 0 \wedge V_2 \leq 0 \\ NF2 & 3V_1 + 2V_2 + V_3 \leq 0 \wedge V_1 + 3V_2 + V_3 \leq 0 \wedge V_1 \leq 0 \end{array}$$

hence failing to identify that the two constraints are equivalent.

The limitations of syntactic normalisation, open the study of new normal forms, that we propose in Chapter 3.

2.2.4 Utopia

Aquino et al.'s *UtoPia* framework reuses solutions by heuristically identifying constraints that share available proofs even without being necessarily mutually equivalent or related by implication [ADP17]. To this end, *UtoPia* proposes an original metric, referred to as *SatDelta*, to estimate the distance of the solution space of a target constraint (yet to be solved) from the solution spaces of the constraints with available solutions, and selects candidate reusable solutions from constraints with close solutions spaces. The metric SatDelta evaluates the constraints on a set predefined solutions, computes the amount by which the predefined solutions miss the satisfaction of the atomic predicates in the constraints, aggregates these missing amounts based on the logical structure of the constraints, and finally estimates the distance of the solution spaces of constraints as the absolute difference of the respective SatDelta-values.

To find a reusable solution for a target constraint, *Utopia* works as follows:

(i) it computes the SatDelta-value of the target constraint,

(ii) it retrieves a small set of candidate reusable solutions, which it takes from the cached constraints with SatDelta-value closest to the one of the target constraint,

(iii) it tests the candidate solutions against the target constraint and reports the first satisfying solution, if any, or no reusable solution otherwise.

ReCal and UtoPia are complementary to each other. Being a heuristic approach, UtoPia can experience false negatives, that is, it may fail to identify a reusable solution that there exists in the repository, because the computation of SatDelta may yield different results for the target and the cached constraint, respectively. In UtoPia, this can happen regardless of whether the two constraints are equivalent or related by implication. Conversely, *ReCal* precisely identifies a large class of equivalent and implication-related constraints. Utopia has been developed in the same research group in which we developed this PhD Work, and is seeded in the initial work on *ReCal* that continued with two complementary research lines, that is, the work on *UtoPia* on one side, and the refinement and parallel implementation of *ReCal* on the other side. The experiments reported in this thesis provide empirical evidence of the complementarity of these two approaches.

Chapter 3

Reusing Proof in Symbolic Analysis

In this chapter, we propose the new ReCal *reusing framework, a contribution of this PhD thesis. The core contribution of the* ReCal *framework is a novel canonical form to efficiently identify a large class of equivalent constraints, a new algorithm to identify a large class of constraints related by logical implication, and logical simplification rules to increase the likelihood of succeeding in identifying both types of constraints. The chapter presents the reference logic of the constraints that can be addressed with* ReCal*, and a set of sample constraints collected from symbolic analysis that exemplify the novel characteristics of* ReCal *with respect to the competitor proof caching and reusing frameworks at the state of the art. It then presents the logical components of the* ReCal *framework, preprocessing, logical simplification, canonicalisation and search.*

This chapter presents the *REusing-Constraint-proofs-in-symbolic-AnaLysis* (*ReCal*) approach. We frame the class of constraints that *ReCal* addresses, introduce a set of examples that highlight the distinctive characteristics of the *ReCal* framework with respect to the competing approaches at the state of the art, and describe the internals of the *ReCal* framework.

3.1 Reference Logic

ReCal addresses constraints expressed as conjunctive formulas in the quantifier-free linear integer arithmetic (QF_LIA) logic. *ReCal* focuses primarily on constraints produced during symbolic execution that consist in quantifier-free formulas in conjunctive form. Among those, the constraints composed of linear equalities and inequalities over integers represent a large class of constraints that occur in many practical settings of symbolic execution. Most popular constraint solvers [DMB08, CGSS13, Dut14] address constraints in QF_LIA logic, confirming the general interest in coping with constraint

solving problems in this logic. Similarly to *ReCal*, also the related approaches *Green* and *GreenTrie* focus on QF_LIA logic constraints.

3.2 Motivating Examples

In this section, we discuss sample constraints collected from practical applications of symbolic program analysis, and we use them to exemplify the novelty of *ReCal*. The examples highlight (i) the type of equivalent constraints that *ReCal* can identify and the state-of-the-art approaches cannot, (ii) the type of constraints related by implication that *ReCal* can identify and the competing approaches cannot, and (iii) the type of logical simplifications that *ReCal* distinctively exploits to increase the likelihood of succeeding in identifying both equivalent constraints and constraints related by implication.

3.2.1 Reusing Proofs across Equivalent Constraints

Equivalent constraints may occur with different concrete representations due to program specific information and context dependencies. For example, two equivalent constraints can refer to different variable names, include the same terms and clauses in different order of occurrence, or even include different arithmetic expressions. Differences in the representation of the constraints may preclude the ability of a proof reusing framework to identify the equivalence of the constraints.

Let us for instance consider the constraint C_1

$$balance \geq 0 \wedge balance - withdraw\_value \geq 0$$

that we collected while symbolically analysing a program for computing bank transactions, and the constraint C_2

$$measure\_value \geq alarm\_value \wedge measure\_value \geq 0$$

that we collected from a program that works in the context of alarm signalling system. Given the simplicity of these two constraints, people can straightforward identify that they are equivalent: In fact, C_1 and C_2 map to the same constraint if we both swap the two clauses of C_2, and rename the variables *balance* and *measure_value* as V_1, and the variables *withdraw_value* and *alarm_value* as V_2. These transformations turn the two constraints in the same formula:

$$C_1 \equiv C_2 \equiv V_1 \geq 0 \wedge V_1 - V_2 \geq 0.$$

Clause swapping and variable renaming are equivalence-preserving transformations, that is, these transformations do not alter the equivalence of the constraints to which they are applied. Thus, if we can transform two constraints into exactly the same

constraint with these transformations, we can identify that they are indeed equivalent. Thus, with reference to the above example, if we know a solution for the constraint C_1, say because we already solved C_1 at some point of a symbolic analysis session, we can reuse the solution of C_1 to solve C_2.

In this example, C_1 and C_2 are two simple constraints, and thus, as we commented above, in this case it is easy for people to spot the equivalence preserving transformations needed to identify the equivalence of these two constraints. However, to automatically reuse proofs, we need an efficient algorithm for deciding the equivalence of constraints, even when they appear with different representations, and for constraints of arbitrary size. Furthermore, the perspective algorithm shall be able to efficiently look up the equivalent constraints out of large repositories of constraints.

The *ReCal* approach defined in this thesis is grounded on the observation that we can unveil the equivalence of constraints regardless of whether they may arise with different representations during symbolic analysis, by transforming the constraints into a suitable abstract canonical form. A core technical contribution of *ReCal* is the definition of a canonical form of conjunctive QF_LIA constraints that unveils the equivalence between constraints that are equivalent modulo a suitable renaming of the variables and a suitable reordering of the clauses in the constraints. We remark that the constraint caching framework implemented in *Klee* maintains the constraints with the original names of the variables, thus it cannot address equivalent constraints that differ in the names of the variables, such as the two constraints C_1 and C_2 in the example above. The *Green* constraint caching framework acknowledges the need of normalising the variables names and the comparison operators in the constraints, to increase the chances of identifying equivalence of constraints that arise with different representations, but relies on a very limited normal form based on a positional rewriting of the constraints that sorts the monomial terms based on their coefficients in descending order. For example, with reference to the constraints C_1 and C_2, since all variables have a unitary coefficient, *Green* would simply normalise the names of the variables without changing their positions in the formulas, thus transforming the the two formulas into

$$C_1 \equiv V_1 \geq 0 \ \wedge V_1 - V_2 \geq 0$$

$$C_2 \equiv V_1 - V_2 \geq 0 \wedge V_1 \geq 0$$

that do not suffice to identify the equivalence of C_1 and C_2 by textual match. The missing transformation, that is, swapping the two conjuncts in either formula, which may seem straightforward in this simple example, is far from trivial for constraints with arbitrary sets of both variables and conjuncts, as the ones that can be dealt with *ReCal*.

ReCal addresses the problem of identifying equivalent formulas by defining an abstract canonical form of conjunctive QF_LIA constraints that satisfies three essential properties:

(i) semantically equivalent constraints transform into the same constraint once converted into the *ReCal* canonical form,

(ii) constraints that correspond to the same *ReCal* canonical form are equivalent to each other,

(iii) different canonical forms can be strictly ordered.

A canonical form that satisfies the first property allows us to reduce the complex problem of deciding the equivalence of constraints to the simpler problem of checking their canonical forms for equality. The second property guarantees the soundness when deciding the constraint equivalence as above. A canonical form that satisfies the strict ordering property can be used as an index for fast searching through a large repository of constraints.

3.2.2 Reusing Proofs across Constraints Related by Implication

A proofs derived for a constraint can be re-used for equivalent constraints, as exemplified in the previous section, and also as a proof of satisfiable stricter constraints or of unsatisfiable weaker constraints. For instance, a solution of the satisfiable constraint $x < 0 \wedge x + y = 0$ is also a solution of the constraint $x < 1$, since the former constraint is stricter than (that is, it logically implies) the latter one. Similarly, if we have a proof that the constraint $x < 0 \wedge x > 0$ is unsatisfiable, then we can infer that the constraint $x < 0 \wedge x > 1 \wedge x \neq 2$ is also unsatisfiable, since the former constraint is weaker (is logically implied from) the latter one.

The *ReCal* approach that we define in this chapter embraces the following definition of stricter-ness between constraints: Given two constraints C_1 and C_2 in conjunctive forms, C_1 is stricter than C_2 (dually C_2 is weaker than C_1) if for each clause l_2 of C_2 there is a corresponding clause l_1 in C_1, such that l_1 is either equal to or stricter than l_2. In this case, if C_1 is satisfiable, therefore C_2 is also satisfiable and any solution for C_1 is a solution for C_2; If C_2 is unsatisfiable, C_1 is also unsatisfiable.

Identifying logically stricter or weaker constraints can be computationally expensive. In *ReCal*, we propose an efficient way to identify stricter and weaker constraints expressed in conjunctive form, based on whether we can determine an inclusion of all clauses of a constraint with respect to another constraint, by inspecting the relative order of the constant terms in the clauses of the two constraints, respectively. Furthermore, to efficiently retrieve stricter and weaker constraints through large repositories of constraints, we define an inverted index that associates each clause that appears in at least a constraint in the repository to all constraints in the repository that include that clause. Inverted indexes are a widely used technique for fast information retrieval [YDS09].

3.2.3 Improving Proof Reusability by Logical Simplifications

In *ReCal* we introduce two logical simplification rules, namely redundant clause elimination and conflicting clause detection.

Redundant clause elimination simplifies redundant information within a single constraint by removing clauses that are implied by other clauses included in the constraint. For example, let us consider the constraint L_1

$$V_1 - V_2 \geq 0 \wedge V_1 - V_2 \geq 1 \wedge V_1 - V_2 \geq 2 \wedge V_1 - V_2 \geq 3$$

that resembles a typical constraint generated with symbolic execution during the analysis of a program that iteratively enters a loop condition multiple times. The first three clauses in this constraint are implied by the last one, thus L_1 can be simplified as just $V_1 - V_2 \geq 3$, by removing these three redundant clauses. By reducing the amount of redundant information in the constraints maintained in the proof caching and reusing framework, redundant clause elimination increases the chances of identifying equivalent constraints.

Conflicting clause detection checks whether a constraint includes any pair of clauses that are mutually contradictory, thus entailing the unsatisfiability of the constraint. If so, the conflicting clauses represent themselves a solution, that is, an unsat-core that proves the unsatisfiability of the constraint. In this case, *ReCal* can return the identified solution directly.

3.3 The ReCal Proof Caching and Reusing Framework

In this section, we define the *ReCal* framework for searching for reusable proofs of constraints out of a large repository of cached ⟨constraint, proof⟩ pairs. When queried for the satisfiability of a target constraint, *ReCal* searches whether the repository contains a reusable proof for that constraint in the four phases shown in figure 3.1:

(i) *preprocessing*, *ReCal* normalises the target constraint by formula slicing, algebraic transformations, and variable name abstraction;

(ii) *logical simplification*, *ReCal* eliminates redundant clauses, and detects conflicting ones;

(iii) *canonicalisation*, *ReCal* computes the canonical form of the constraint;

(iv) *search*, *ReCal* exploits the canonical form of the constraint for quickly searching for either an equivalent or a stricter/weaker constraint in the repository.

The *preprocessing* phase exploits the ideas originally proposes in *Klee* (namely, formula slicing) and *Green* (namely, algebraic transformations and variable name abstraction), thus alines *ReCal* to the state of the art. The *logical simplification*, *canonicalisation* and *search* phases are an original contributions of *ReCal*.

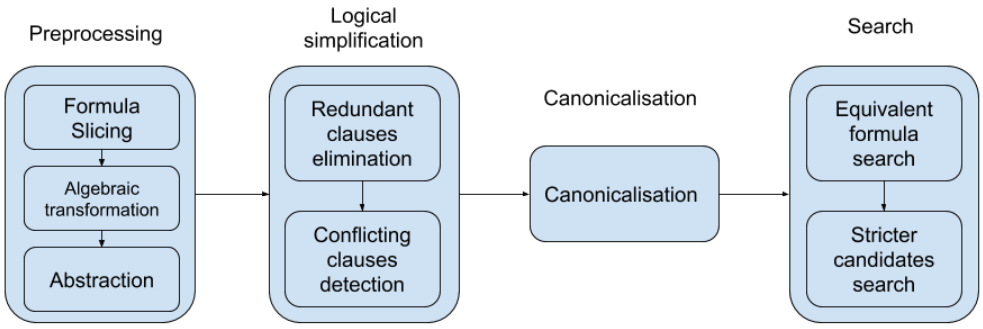


Figure 3.1. The ReCal process

3.3.1 Preprocessing

The preprocessing phase of *ReCal* transforms a constraint formula into a simplified and context independent abstract form, by exploiting ideas inherited from the state-of-the-art approaches *Klee* and *Green*. As shown in Figure 3.1, the preprocessing step includes

(i) *(Slicing)* a formula slicing step that decomposes the constraint into independent constraints,

(ii) *(Algebraic Transformation)* an algebraic transformation step that simplifies variable terms and normalises the structure of each linear inequality clause of a constraint,

(iii) *(Abstraction)* an abstraction step that transforms a constraint into a matrix-based representation that abstracts away the program-dependent names of the variables in the constraint.

Slicing

The satisfiability of a conjunctive QF_LIA constraint can be derived from the satisfiable results of its independent sub-constraints. Two constraints are independent if and only if they do not share any variables. *Slicing*, first proposed in *KLEE* by Cadar et al. [CDE08], consists in dividing a conjunctive constraint into independent sub-constraints, such that the independent sub-constraints do not share any variables. The *sliced* sub-constraints can be solved independently from each other, and the satisfiability of a complex constraint can be determined from the satisfiability of its (smaller and simpler) sub-constraints: if all the independent sub-constraints are satisfiable, the original constraint is satisfiable too, otherwise, if any sub-constraint is unsatisfiable, the original constraint is unsatisfiable. *Slicing* makes the constraints smaller and easier to handle, ultimately increasing the probability of matching the sub-constraints with some cached constraints.

We implement the formula slicing algorithm in four steps: (i) we initialise an undirected graph with a node for each clause and each variable in the constraint, (ii) we connect the nodes that correspond to each clause to the nodes that correspond to the variables referred in that clause, (iii) we iteratively visit the graph in depth first order, to identify the connected components, (iv) we extract the clauses that correspond to each connect component to build a sub-constraint.

Below, we show a simple example of formula slicing:

Constraint before slicing

$$C_0 :: a + 3b < 2 \wedge c < 0 \wedge x - 1 < 6 - 2x \wedge 2a + c \neq 1 \; x \neq 4$$

Constraint after slicing

C_1:	C_2
$a + 3b < 2 \wedge c < 0 \wedge 2a + c \neq 1$	$x - 1 < 6 - 2x \wedge x \neq 4$

ReCal handles the sub-constraints produced in the slicing step independently from each other in the next phases, and aggregates the corresponding satisfiability results at the end of the process, to infer the satisfiability value of the original constraint.

Algebraic Transformation

The *algebraic transformation* step simplifies each inequality clause of a constraint by applying three mathematic rewriting rules:

(i) summing all the terms that refer to the same variable, and the constant terms,

(ii) replacing the comparison operator with either $\leq$ or $\neq$ according to the rules described in Figure 3.2,

(iii) dividing all the coefficients by their greatest common divisor.

The rules in Figure 3.2 exploit the property that all integer algebraic constraints are equivalent to constraints expresses only with the comparison operators $\leq$ and $\neq$. This approach could be extended to handle formulas with real operands by including also the operator $=$.

Original (in)equality	Simplified (in)equality
$terms \leq k$	$terms \leq k$
$terms \neq k$	$terms \neq k$
$terms < k$	$terms \leq k - 1$
$terms \geq k$	$-1 * (terms) \leq -k$
$terms > k$	$-1 * (terms) \leq -k - 1$
$terms = k$	$terms \leq k \wedge -1 * (terms) \leq -k$

Figure 3.2. Comparison simplifications

Abstraction

The *abstraction* step eliminates the program-specific variable names from constraints. To this end, we turn the constraints into a matrix-wise representation as follows. Given a constraint with n clauses and m variables, we turn it in a matrix that associates the clauses in the constraint with the rows of indexes from 1 to n in the matrix, according to their order of occurrence of the clauses in the constraint, and the variables in the constraint with the columns of indexes from 1 to m in the matrix, according to the lexicographic ordering of the names of the variables. Then, we transform the constraint formula into a matrix M of size $n \times (m+2)$ such that:

- $M[i,j] = c$ iff the i-th clause of the formula contains term $c * v_j$ and $1 \leq j \leq m$,
- $M[i,m+1] =$ constant term of the i-th clause,
- $M[i,m+2] =$ comparison operator of the i-th clause (either $\leq$ or $\neq$).

We call the $(m+1)$th column of the resulting matrix the *constant term column*, the $(m+2)$th column the *comparison operator column*, and the rest of the columns the *monomial term columns*.

Figure 3.3 illustrates the *preprocessing* phase through a simple example, spelling out the three elements of the algebraic transformation step.

Input formula	$x-1<6-2x \wedge x \neq 1$
Summing up terms	$3x<7 \wedge x \neq 1$
Transforming comparison operators	$3x \leq 6 \wedge x \neq 1$
Dividing clauses by their gcd	$x \leq 2 \wedge x \neq 1$
Abstraction	$\left\vert \begin{array}{ccc} 1 & 2 & \leq \\ 1 & 1 & \neq \end{array} \right\vert$

Figure 3.3. Example of Preprocessing

3.3.2 Logical simplification

The logical simplification step simplifies a constraint by identifying both (i) redundant clauses, that is, clauses that are subsumed by other clauses in the constraint, and that thus can simply be pruned away without changing the semantic of the constraint, and (ii) conflicting clauses, that is, a pair of clauses that are mutually contradictory, and that thus entail that the whole conjunctive constraint is indeed unsatisfiable.

Redundant Clause Elimination

Constraints generated from symbolic analysis might include redundant clauses. A *redundant clause* is a clause of a constraint that is logically implied by other clauses in the same constraint, and can thus be removed without affecting the satisfiability value

of the constraint. *ReCal* identifies and removes redundant clauses from both the stored and the target constraints, since considering constraints with no (or minimal) clause redundancy increases the chance of spotting reusable proofs. Moreover, reducing the size of constraints can speed up the search process.

Since identifying redundant clauses according to the most general interpretation of logic implication is a hard task, we consider the tradeoff between the amount of reusable proofs that we can identify and the cost of processing the constraints, and we deal with the subset of redundant clauses that can be identified by evaluating the subsumption between pairs of clauses within the same constraint. In details, *ReCal* addresses the class of redundant clauses that correspond to inequalities comprised of the same linear combinations of variables, possibly with the same or related constant terms and comparison operators. Formally:

Definition 3.1. ***Redundant clauses:***
Given two clauses c_1 and c_2 of a constraint F, c_1 is redundant with respect to c_2 (that is, $c_2 \implies c_1$) if both c_1 and c_2 include the same set of variables with equal coefficients, respectively, and either (i) both c_1 and c_2 have the same comparison operator and the same constant term (that is, c_1 and c_2 represent exactly the same inequality), or (ii) c_2 has the comparison operation $\leq$ and a constant term that is less than the constant term of c_1.

For example, the following constraints contain the redundant clauses marked in blue:

$x + y \leq 1 \wedge x + y \leq 1$,
$x + y \neq 1 \wedge x + y \leq 0$.

Eliminating those redundant clauses does not affect the satisfiability value of the two constraints.

ReCal eliminates redundant clauses from constraints in matrix form as follows:

(i) it scans the rows in the constraint matrix, and looks for rows that consist of exactly the same coefficients in the monomial term columns, that is, without considering the coefficients that represent the constant term and the comparison operator of the constraint clauses,

(ii) upon finding any pair of matching rows, it compares the values of comparison operator and constant term to decide if either or the two rows represent a redundant clause with respect to the other, by referring to the definition above.

At the end of the process, *ReCal* prunes the constraint matrix by removing the rows that correspond to redundant clauses.

Conflicting Clause Detection

Conflicting clauses are groups of clauses that stand by themselves as a logical contradiction in a conjunctive constraint, thus implying that the constraint as a whole is itself unsatisfiable. Detecting conflicting clause in general is as hard as inferring the unsatisfiability of a constraints. *ReCal* focuses only on pair-wise conflicts, that is, conflicts between pairs of clauses, aiming to speed up the identification of unsatisfiable constraints, while keeping the checking time limited within reasonable budget. *ReCal* identifies pair-wise conflicting clauses as early as possible, to avoid wasting time in further processing and searching for reusable proofs for those unsatisfiable constraints, which would ineffectively degrade its performance.

Formally, a conflicting clause pair is defined as follows:

Definition 3.2. ***Conflicting clause pair:***
Given a clause c_1 in a constraint F, if there exist another clause c_2, such that $c_2 \wedge c_1 \implies false$, then c_1 and c_2 cause a conflict.

To identify pair-wise conflicting clauses, *ReCal* scans the possible pairs of clauses that correspond to inequalities of the original constraint, sums the coefficients for the homologous variables across the two inequalities, and verifies if the resulting inequality is a purely numeric contradiction.

For example, the constraint below includes a pair of conflicting clauses marked in red:

$$2x + 3y \leq 1 \wedge -2x - 3y \leq -5 \wedge 2x + y \leq 0,$$

since, by summing the homologous coefficients across the first two clauses of the constraint, we obtain

$$(2x + 3y) + (-2x - 3y) \leq 1 + (-5) \implies 0 \leq -4,$$

which is a numeric contradiction.

When detecting a conflicting clause pair, *ReCal* returns an unsatisfiability verdict for the considered constraint, skipping any further processing of the constraint.

3.3.3 Canonicalisation

The core step of *ReCal* is *canonicalisation*, which transforms the matrix representation of the constraint in canonical form, that is, a matrix representation unique for all equivalent constraints, thus enabling fast lookup over a large repository of constraints. The *ReCal* canonical form of constraints satisfies two main properties,

(i) if two constraints are equivalent, they correspond to the same canonical form;

(ii) if two constraints have the same canonical form, then they are equivalent.

The first property allows us to use the canonical form to efficiently retrieve equivalent constraints by comparing their canonical forms for equality. The second property guarantees that, by doing so, we soundly identify only equivalent constraints and no false positive.

ReCal defines the canonical form of the constraints by defining an algorithm, called the *canonicalisation* algorithm, that iteratively reorders the rows and the columns of the constraint matrixes, and is proved to converge to exactly the same matrix if and only if the corresponding constraints are equivalent. The definition of the *canonicalisation* algorithm is a core contribution of this thesis, and we extensively discuss it in Chapter 4. Below, we explain the search phase of *ReCal* under the assumption that the *canonicalisation* algorithm guarantees the equality of the matrix representations of the equivalent constraints.

3.3.4 Efficient Retrieval of Reusable Proofs

The *Search* phase of *ReCal* searches for equivalent constraints, and searching for constraints related by implication, exploiting the canonic form of the constraints as the core element for efficiently searching reusable constraint proofs.

Search for Reusable Proofs from Equivalent Constraints

To retrieve equivalent constraints, *ReCal* exploits the canonic form to build an efficient search index. The canonic form reduces the complex problem of comparing constraints for equivalence to the simpler problem of comparing their canonic forms for equality. Thus, for all constraints stored in its cache, *ReCal* uses the canonic form of those constraint as the entry of a (hashed) map $\langle$*canonic_form*, *available_proof*$\rangle$. Upon receiving a new constraint to solve, *ReCal* computes the canonical form of that constraint, and uses the canonical form to lookup a corresponding key in the map. If it finds a correspondence in the map, it returns the proof as a a (reusable) proof for the target constraint.

Search for Reusable Proofs From Constraints Related by Implication

If the search for equivalent constraints does not return any entry, *ReCal* continues looks for proofs from constraints related by implication. In general, if a constraint F_1 implies a constraint F_2, the satisfiability result of F_1 implies the satisfiability of F_2 (since the solution space of F_1 is a subset of the one of F_2). Vice-versa, the unsatisfiability of F_2 implies the unsatisfiability of F_1.

KLEE [CDE08] reuses proofs from super- and sub-constraints identified as constraints that include all the clauses of the target constraint, and constraints comprised of a set of clauses that are all included in the target constraint, respectively. *GreenTrie* [JGY15] introduces implication rules that allow proof reusing from constraints

with different restriction on the constant terms. The class of constraints related by implication that we consider in *ReCal* goes beyond the ones considered in both these approaches.

ReCal automatically identifies implication relations that can be deduced from comparing the clauses of two constraints F and G as follows: a constraint F is stricter than a constraint G if and only if for each clause $c_G \in G$ there exists a clause $c_F \in F$ that implies c_G. Notice that F can contain additional other than the clause c_F that correspond to clauses c_G. We also say that the constraint G is weaker than F. For example, the constraint $F1$

$$x \leq 2 \wedge\ x + y \leq -1\ y \leq 0$$

is stricter than the constraint $G1$

$$a \leq 3 \wedge\ a + b \neq 0$$

In this example the correspondence between the first two clauses of the constraints ($x \leq 2 \Rightarrow a \leq 3$ and $x + y \leq -1 \Rightarrow a + b \neq 0$) is straightforward once defined the mapping between x, y and a, b ($x = a$ and $y = b$).

To identify a constraint that is stricter or weaker than a target constraint it is necessary to compare the latter with all the constraints in the repository. To accomplish this task efficiently, we created an inverted index that associates the clauses of the constraints in the repository with all the constraints that contain such clauses, drawing inspiration from the Google Search term-to-pages inverted index [BDH03, YDS09]. The inverted index stores pairs of the form
$\langle$*clause_entry*, $\{\langle$*constraint_reference*, *comparison_operator*, *free_coefficient*$\rangle, \ldots\}\rangle$,
one for each clause that appears in a constraint in the repository. The *clause_entries* depend on the coefficients of the variables as computed by the *canonicalization* algorithm, with the exception of the constant term and the comparison operator. Thus, clauses with the same sets of variable coefficients map to the same *clause_entry* in the inverted index, even if they may differ in the respective constant terms and comparison operators. The inverted index associates each clause_entry with the set of constraints that contain at least a clause that corresponds to the clause_entry, that is, a clause with the same variable coefficients of the clause_entry. For each constraint associated with a clause_entry, the index keeps also track of the comparison operator and the constant term of the corresponding clause within the constraint.

To exploit constraints related by implication, *ReCal* stores the satisfiable constraints and un-satisfiable constraints in two different repositories. Given a constraint C that contains the clauses $c_1, \ldots, c_n$, *ReCal* searches for reusable proofs from constraints related by implication as follows.

First, *ReCal* checks if it can find a satisfiable stricter constraint in the repository of satisfiable constraints. For each clause $c_i \in C$, it accesses the inverted index to identify the set $S_i = \{C_1^i, C_2^i, ...\}$ of constraints that contain at least a clause with the same

clause_entry of c_i. By definition, a satisfiable stricter constraint of C shall contain all the clause_entries of C, hence *ReCal* computes the intersection of the sets S_i, to get the set S of constraints that can be stricter than C. We call the constraints in S the candidate stricter constraints. If the intersection set S is empty, there is no stricter constraint in the repository. Otherwise, for each collected stricter candidate in S, *ReCal* pairs the clauses of the stricter candidate and the ones of C according to results the inverted index, and compares the comparison operator and the constant term of the clauses in the stricter candidate with the comparison operator and the constant term of the clauses in C, respectively. If this check reveals that all clauses in the stricter candidate are truly stricter than the corresponding ones of C, then *ReCal* infers that C is indeed satisfiable, and can reuse the proof associated with the stricter candidate.

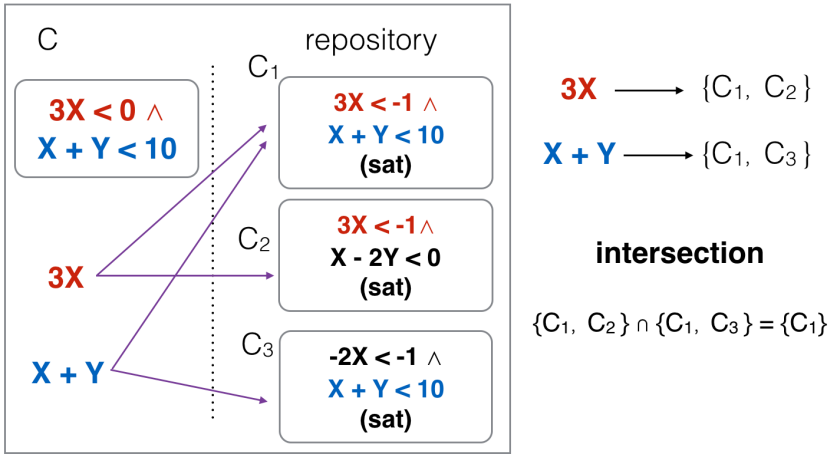


Figure 3.4. Example of retrieving satisfiable stricter candidates

Figure 3.4 shows an example of retrieving satisfiable constraint candidates. In the example, the simplification and canonicalisation steps return a target constraint C with two clauses (In the figure, we use the variable and clause representation instead of matrix for simplicity of presentation). If the search with the inverted index finds $\{C_1, C_2\}$ by looking up with the first clause_entry $3X$, and $\{C_1, C_3\}$ by looking up the second clause_entry $X + Y$, the intersection of the two sets is ,$\{C_1\}$, that is, C_1 is the only stricter candidate constraint. *ReCal* finalises the search by compare the operators and constant terms of each pair of clauses of C and $C1$. In this case, *ReCal* concludes that C_1 is indeed stricter than C, and can reuse the proof of C_1 to solve C.

If *ReCal* does not find any stricter constraint in the repository of satisfiable constraints, it searches for unsatisfiable weaker constraints in the other repository. *ReCal* identifies the set of constraints with clauses that can be logically implied by any c_i in C using the inverted index.

In a nutshell, the *ReCal* approach reduces the problem of comparing a target con-

straint for stricter- or weaker-ness with respect to the cached constraints, to comparing the operators and the constant terms of the target constraint with the ones of a small subsets of constraints identified with the inverted index. *ReCal* updates the inverted index offline, when it adds new constraints and proofs to the repository, without affecting the performance of the search phase of *ReCal*.

Chapter 4

The Canonicalisation Algorithm

The core of the ReCal *approach proposed in this thesis is an effective method to determine whether two conjunctive QF_LIA constraints are equivalent, abstracting from differences in their representations, that is, they may refer to variables denoted with different names, and may list corresponding terms and clauses in different orders.* ReCal *defines an original canonical form of the conjunctive QF_LIA constraints, and exploits such canonical form to recast the difficult problem of deciding the equivalence of constraints to the simple problem of comparing the equality of the canonical forms of the constraints, thus enabling an efficient search procedure to find equivalent constraints across large repositories of constraints. This chapter defines the* ReCal *canonical form both declaratively, by specifying the class of equivalent constraints that it identifies, and operationally, by presenting the* Canonicalisation *algorithm that computes the canonical form of a conjunctive QF_LIA constraint.*

4.1 The ReCal Canonical Form

The previous chapter presents the core characteristic of *ReCal* of revealing the equivalence of constraints that may differ in their mutual representations due to the specific context of symbolic analysis in which they originally arose. In particular, *ReCal* defines a canonical form of conjunctive QF_LIA constraints that determines the equivalence of structurally different constraints, by turning them into the same representation with equivalence-preserving transformations.

For instance, the following constraints

$$C_1 : 3x + y \leq 0 \wedge x + 2y \leq 0$$
$$C_2 : 2a + b \leq 0 \wedge a + 3b \leq 0$$

can be turned into equal representations by swapping the two clauses of the second constraint, renaming the variables x and b as v_1 and the variables y and a as v_2, re-

spectively, and reordering the terms that refer to variable v_1 before the ones that refer to variable v_2 in each clause. These transformations do not alter the equivalence characteristics of the constraints and produce

$$C_1 \equiv C_2 \equiv\ 3v_1 + v_2 \leq 0 \wedge v_1 + 2v_2 \leq 0$$

This rewriting of the two constraints indicates a straightforward mapping between the solutions of C_1 and C_2.

Below, we formalise the intuitive notion of equivalence that underlies this example as an equivalence relation among conjunctive QF_LIA constraints that can be rewritten as the same constraint by suitably permuting clauses and variables, and abstracting the names of the variables in the constraints. The *ReCal* canonical form identifies this equivalence relation.

We define the *ReCal* canonical form conjunctive QF_LIA constraints as follows. Let us assume C is a conjunctive QF_LIA constraint with n variables and m clauses, and $permute(C) = \{C_1, C_2, ...C_{n!*m!}\}$ is the set containing all the constraints generated by arbitrarily permuting the terms and clauses in C, and arbitrarily renaming distinct variables with distinct names. Then, the *ReCal* canonical form is the canonical form of the equivalence relation such that

$$C \equiv C' \iff C' \in permute(C)$$

The readers should notice that this equivalence relation identifies a class of logically equivalent constraints, because renaming the variables, permuting the terms and permuting the clauses of a constraints are equivalence-preserving transformations.

The *ReCal* canonical form of a constraint C, $Canonicalisation(C)$, is a permutation of the constraint C ($Canonicalisation(C) \in permute(C)$) that satisfies the following property:

$$C \equiv C' \iff Canonicalisation(C) = Canonicalisation(C') \tag{4.1}$$

The right arrow part of the co-implication, that is,

$$C \equiv C' \implies Canonicalisation(C) = Canonicalisation(C')$$

guarantees that two *ReCal*-equivalent constraints share the same canonical form, thus allowing *ReCal* to turn the difficult problem of deciding equivalence of constraints into the simple comparison of the equality of their corresponding canonical forms. The left arrow part of the co-implication, that is,

$$C \equiv C' \impliedby Canonicalisation(C) = Canonicalisation(C')$$

guarantees the soundness of the result when *ReCal* matches the canonical forms to determine the equivalence of two constraints.

We define the *ReCal* canonical form operationally, by designing a deterministic and terminating algorithm *Canonicalisation* that is guaranteed to compute a representation of the constraints that is unique for all constraints that belong to the same equivalence class. Conversely, the output of algorithm *Canonicalisation* differs for constraints that belong to different equivalence classes.

Definition 4.1. *Let $LC_\wedge$ be the set of all conjunctive QF_LIA constraints.*
Let $permute(C) \in 2^{LC_\wedge}$ denote the set of constraints that can be obtained by permuting the clauses and the variables of a constraint $C \in LC_\wedge$.
Let $\equiv$ be an equivalence relation over $LC_\wedge$ such that $C_1 \equiv C_2 \iff C_1 \in permute(C_2)$, that is, C_1 and C_2 are equivalent if it is possible to permute properly the clauses and rename the variables of C_2 to obtain C_1. It is easy to verify that this relation is reflexive, symmetric and transitive.
Then, Canonicalisation : $LC_\wedge \to LC_\wedge$ *is a deterministic and always terminating algorithm such that:*

$\forall C \in LC_\wedge$, Canonicalisation$(C) \in permute(C)$

$\forall C_1, C_2 \in LC_\wedge, C_1 \equiv C_2 \iff$ Canonicalisation$(C_1) =$ Canonicalisation(C_2).

4.2 The Canonicalisation Algorithm

Canonicalisation is a deterministic algorithm that iteratively permutes the clauses and the variables of constraints. *Canonicalisation* always converges to a fixed point, and guarantees that the constraint at the fixed point is unique for all input constraints that belong to the same equivalence class.

Algorithm 1 specifies *Canonicalisation*. The algorithm processes an input conjunctive QF_LIA constraint, and starts by producing a classic matrix of coefficient representation of the input formula, abstracting away the names of the variables (line 4). In *ReCal*, turning a constraint into a matrix corresponds to the *abstraction* step that we presented in chapter 3, that is: Given a formula with n clauses and m variables, the algorithm numbers progressively the variables that appear in the formula, builds an $n \times m$ matrix, and sets the value of the (i,j)th element of the matrix to the coefficient of the j-th variable in the i-th clause. If a variable does not appear in a clause, the algorithm sets the corresponding coefficient to 0. The algorithm augments the matrix with two columns $m+1$ and $m+2$ that represent the constant term and the comparison operator in each clause, respectively. Each row of the resulting matrix represents a clause in the original formula. The top part of Figure 4.1 exemplifies some sample constraints (*Input constraints*) and the corresponding matrix representations (*Matrixes at the beginning of* Canonicalisation).

Algorithm 1 *Canonicalisation* (C)

Require:

1: $C \in LC_{\wedge}$, a conjunctive QF_LIA constraint

Ensure:

2: Returns a matrix representing the canonical form of C

3: /* Algorithm begins */
4: $M \leftarrow constraintAsMatrix(C)$
5: /* Phase 1 begins */
6: $M \leftarrow orderRowsByComparisonOp(M)$
7: $M \leftarrow orderRowsByConstantTerm(M)$
8: **if** $converged(M)$ **then return** M
9: **end if**
10: /* Phase 2 begins */
11: $M \leftarrow orderRowsByGreatestValues(M)$
12: $M \leftarrow orderColsByGreatestValues(M)$
13: **if** $converged(M)$ **then return** M
14: **end if**
15: /* Phase 3 begins */
16: **repeat**
17: $M' \leftarrow M$
18: $subM \leftarrow extractItemsWithStableCols(M)$
19: $M \leftarrow orderRowsLexicographicallyBySubM(M, subM)$
20: $subM \leftarrow extractItemsWithStableRows(M)$
21: $M \leftarrow orderColsLexicographicallyBySubM(M, subM)$
22: **until** $not\ changed(M, M')\ or\ converged(M)$
23: **if** $converged(M)$ **then return** M
24: **end if**
25: /* Phase 4 begins */
26: $subM \leftarrow extractRowsAndColsWithUnstableOrder(M)$
27: $permutations \leftarrow enumerateAllValidPermutations(subM)$
28: $p \leftarrow lexicographicMax(permutations)$
29: $M \leftarrow applyPermutation(M, p)$ **return** M

Algorithm *Canonicalisation* processes the matrix representation of the input constraint in four phases (starting at lines 5, 10, 15 and 25, respectively). Each phase sorts the rows and the columns of the matrix according to an ordering relation, using different ordering relations across the four phases. Each phase preserves the order established by the previous phases and strengthens the order between the rows and the

columns that were not assigned a relative order in the previous phases. Upon convergence, the algorithm determines the same order of the rows and the columns for all constraints in the same equivalence class (lines 8, 13, 23 or 29). We discuss the four phases of the algorithm below, and present the proof of convergence and correctness of the algorithm in Appendix A.

- Input constraints:

$$C_1 \begin{array}{l} 3x+y \leq 0 \\ \wedge\ y \neq 0 \\ \wedge\ y - 1 \leq 0 \\ \wedge\ x+2y \leq 0 \end{array} \qquad C_2 \begin{array}{l} 2a+b \leq 0 \\ \wedge\ a \neq 0 \\ \wedge\ a+3b \leq 0 \\ \wedge\ a - 1 \leq 0 \end{array} \qquad C_3 \begin{array}{l} 2i+j \leq 0 \\ \wedge\ i + 2j \leq 0 \\ \wedge\ i \neq 0 \\ \wedge\ i+3j \leq 0 \\ \wedge\ i - 1 \leq 0 \end{array}$$

- Matrixes at the beginning of *Canonicalisation*:

$$C_1 \left|\begin{array}{cccc} 3 & 1 & 0 & \leq \\ 0 & 1 & 0 & \neq \\ 0 & 1 & -1 & \leq \\ 1 & 2 & 0 & \leq \end{array}\right| \quad C_2 \left|\begin{array}{cccc} 2 & 1 & 0 & \leq \\ 1 & 0 & 0 & \neq \\ 1 & 3 & 0 & \leq \\ 1 & 0 & -1 & \leq \end{array}\right| \quad C_3 \left|\begin{array}{cccc} 2 & 1 & 0 & \leq \\ 1 & 2 & 0 & \leq \\ 1 & 0 & 0 & \neq \\ 1 & 3 & 0 & \leq \\ 1 & 0 & -1 & \leq \end{array}\right|$$

- Matrixes after phase 1 of *Canonicalisation*:

$$C_1 \left|\begin{array}{cccc} 3 & 1 & \mathbf{0} & \leq \\ 1 & 2 & \mathbf{0} & \leq \\ \mathbf{0} & \mathbf{1} & \underline{\mathbf{-1}} & \leq \\ \mathbf{0} & \mathbf{1} & \underline{\mathbf{0}} & \neq \end{array}\right| \quad C_2 \left|\begin{array}{cccc} 2 & 1 & \mathbf{0} & \leq \\ 1 & 3 & \mathbf{0} & \leq \\ \mathbf{1} & \mathbf{0} & \underline{\mathbf{-1}} & \leq \\ \mathbf{1} & \mathbf{0} & \underline{\mathbf{0}} & \neq \end{array}\right| \quad C_3 \left|\begin{array}{cccc} 2 & 1 & \mathbf{0} & \leq \\ 1 & 2 & \mathbf{0} & \leq \\ 1 & 3 & \mathbf{0} & \leq \\ \mathbf{1} & \mathbf{0} & \underline{\mathbf{-1}} & \leq \\ \mathbf{1} & \mathbf{0} & \underline{\mathbf{0}} & \neq \end{array}\right|$$

- Matrixes after phase 2 of *Canonicalisation*:

$$C_1 \left|\begin{array}{cccc} \underline{\mathbf{3}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{1}} & \underline{\mathbf{2}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{-1}} & \leq \\ \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \neq \end{array}\right| \quad C_2 \left|\begin{array}{cccc} \underline{\mathbf{3}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{1}} & \underline{\mathbf{2}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{-1}} & \leq \\ \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \neq \end{array}\right| \quad C_3 \left|\begin{array}{cccc} \underline{\mathbf{3}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \leq \\ \mathbf{1} & \mathbf{2} & \underline{\mathbf{0}} & \leq \\ \mathbf{2} & \mathbf{1} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{-1}} & \leq \\ \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \neq \end{array}\right|$$

- Matrixes after phase 3 of *Canonicalisation*:

$$C_3 \left|\begin{array}{cccc} \underline{\mathbf{3}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{2}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{1}} & \underline{\mathbf{2}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{-1}} & \leq \\ \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \neq \end{array}\right|$$

Figure 4.1. Intermediate and final results of Canonicalisation on sample constraints

In the first phase (lines 6—7), algorithm *Canonicalisation* sorts the rows of the matrix in decreasing order considering only their comparison operators (line 6) and constant terms (line 7). We exemplify the result of this phase in Figure 4.1 (*Matrixes after phase 1 of* Canonicalisation). In the examples, we assume that the comparison $\leq$ is always greater than $\neq$ (we refer to an enumerative encoding of the comparison operators).

In Figure 4.1 we observe that, for the constraint C_1 the first phase of *Canonicalisation* orders the row with comparison operator $\neq$ at the end of the matrix and, out of the three rows with comparison operator $\leq$, it orders the one with constant term -1 as last. The relative order of the first two rows and the first two columns is unknown after this phase. We represent in bold the items that converge to a stable row or column position after each phase, and underline the items that converge to a stable position on both dimensions. For example, after phase 1, the relative order of the last two rows of the matrix of C_1 are fixed, and thus it will not change until the end of the algorithm, while the relative order of the first two rows may change.

In the second phase (lines 11—12), algorithm *Canonicalisation* sorts the rows and the columns of the matrix computed in the first phase. When sorting the columns, the algorithm leaves untouched the last two columns, that is, the columns of constant terms and comparison operators. In this phase, the ordering criterion depends on comparing the coefficients of the coefficient vectors in either the rows or the columns, respectively, and sorting them in decreasing order. Specifically, given two coefficient vectors, the vector whose greatest coefficient is greater than the greatest coefficient of the other one will precede the other vector in the sorted matrix. If the greatest coefficients of two vectors are equal, the order depends on the second greatest coefficients, and so forth. For example, with reference to Figure 4.1 (*Matrixes after phase 2 of* Canonicalisation), comparing according to this criterion the first and second row of the matrix of C_2 after the first phase, we determine that the vector $1, 3$ is greater than $2, 1$; thus the algorithm swaps the first two rows. Similarly, comparing the first and second column of the same matrix, we determine that $3, 1, 0, 0$ is greater than $1, 2, 1, 1$, and thus swap the first two columns.

In the second phase, algorithm *Canonicalisation* cannot discriminate the order of the rows and the columns that contain identical sets of coefficients, though possibly in different positions in the respective vectors. In the example of Figure 4.1, the second and third rows of C_3 contain the same elements, though in different positions. In the example, the first two phases of the algorithm produce the canonical forms for C_1 and C_2 thus proving the equivalence of the two constraints, but the algorithm needs the next phase to produce the canonical form of C_3.

In the third phase (lines 16—22), algorithm *Canonicalisation* orders the yet-not-fully-sorted rows and columns, by leveraging a positional ordering based of the row and column items that were already assigned an already-sorted position along the other dimension. In particular, algorithm *Canonicalisation* orders the rows according to the

decreasing lexicographic order of the items of already-sorted column (lines 18 and 19), and the columns according to the decreasing lexicographic order of the items of already-sorted row (lines 20 and 21). For example, for the constraint C_3 in Figure 4.1, since all columns were already assigned a fixed order after the second phase, the third phase sorts the third and the second row in lexicographic order, thus converging to the final canonical form of C_3.

Algorithm *Canonicalisation* iterates the third phase until it can determine incrementally stricter orders of the rows and the columns, and stops when it converges to a fully ordered canonical form, or when further iterations do not produce new ordering opportunities. Figure 4.2 shows an example that needs several iterations of the third phase to converge.

As shown in Figure 4.2 that illustrate the evolution of the matrix representation of constraint C_4, after the first and the second phase of the *Canonicalisation* algorithm, the last two rows and the last four columns (including the comparison operators) of constraint C_4 are already sorted. In the first iteration of the third phase, the algorithm decides the order of the first three unstable rows, by extracting the sub-vectors of the items that correspond to stable columns (marked in bold in the figure). Since $(1, 2, 0, \leq)$ $\leq (2, 1, 0, \leq) = (2, 1, 0, \leq)$, the first row can be ordered with respect to the other two. Similarly, when sorting the columns, we can order them based on the sub-vectors that correspond to already-sorted rows, that is, the sub-vectors $(1, 2) \leq (2, 1) = (2, 1)$. At this point, the order of the first two rows as well as the first two columns are not yet finalised, hence the algorithm iterates again the third phase. In the new iteration, the first two rows include four items in sorted columns, thus the algorithm considers the sub-vectors $(1, 2, 1, 0, \leq) \leq (3, 2, 1, 0, \leq)$, that lead to swap the two rows. Similarly, for the columns, the algorithm considers the sub-vectors $(1, 2, 1) \leq (3, 2, 1)$, and swaps the first two columns, thus converging to the canonical form of the constraint C_4.

In the third phase, algorithm *Canonicalisation* cannot discriminate the order of the rows (columns) that contain identical sets of coefficients on all columns (rows) with final positions, but differ only in the coefficients of columns (rows) with yet unspecified relative order. As shown in Figure 4.3, matrices with few values repeated many times may still be different after the third step of the algorithm, despite corresponding to equivalent constraints.

In the fourth phase (lines 26—29), the algorithm sorts the rows and the columns that have not been ordered yet, by exhaustively enumerating all possible permutations of rows and columns. It extracts the sub-matrix formed of the still unsorted rows and columns (line 26), enumerates the matrices that correspond to permutations of rows and columns that do not violate the relative order established in the previous phases (line 27), identifies the maximum permutation according to the lexicographic order of all matrices flattened as sequences of numbers (line 28), and applies this permutation to the original matrix (line 29). Since the possible permutations are always a finite set, this phase is guaranteed to always terminate, and also guarantees the termination of

- Input constraints:

$$C_4 \quad \begin{array}{l} 2V_1 + 2V_2 + V_3 + V_4 + V_5 \leq 0 \\ \wedge V_1 + 4V_2 + V_3 + 3V_4 + 2V_5 \leq 0 \\ \wedge V_1 + V_2 + 2V_3 + 2V_4 + 2V_5 \leq 0 \\ \wedge 2V_1 + 1V_2 + 3V_3 + 4V_4 + V_5 \leq 0 \\ \wedge 2V_1 + 3V_2 + 4V_3 + V_4 + V_5 \leq 0 \end{array}$$

- Matrix at the beginning of *Canonicalisation*:

$$\left|\begin{array}{cccccc|c} 2 & 2 & 1 & 1 & 1 & 0 & \leq \\ 1 & 4 & 1 & 3 & 2 & 0 & \leq \\ 1 & 1 & 2 & 2 & 2 & 0 & \leq \\ 2 & 1 & 3 & 4 & 1 & 0 & \leq \\ 2 & 3 & 4 & 1 & 1 & 0 & \leq \end{array}\right|$$

- Matrix after phase 1 and 2 of *Canonicalisation*:

$$\left|\begin{array}{cccccc|c} 4 & 1 & 3 & \mathbf{1} & \mathbf{2} & \mathbf{0} & \leq \\ 1 & 2 & 4 & \mathbf{2} & \mathbf{1} & \mathbf{0} & \leq \\ 3 & 4 & 1 & \mathbf{2} & \mathbf{1} & \mathbf{0} & \leq \\ \mathbf{1} & \mathbf{2} & \mathbf{2} & \underline{\mathbf{1}} & \underline{\mathbf{2}} & \underline{\mathbf{0}} & \leq \\ \mathbf{2} & \mathbf{1} & \mathbf{1} & \underline{\mathbf{2}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \leq \end{array}\right|$$

- Matrix after the 1st iteration of phase 3 of *Canonicalisation*:

$$\left|\begin{array}{cccccc|c} 2 & 4 & \mathbf{1} & \mathbf{2} & \mathbf{1} & \mathbf{0} & \leq \\ 4 & 1 & \mathbf{3} & \mathbf{2} & \mathbf{1} & \mathbf{0} & \leq \\ \mathbf{1} & \mathbf{3} & \underline{\mathbf{4}} & \underline{\mathbf{1}} & \underline{\mathbf{2}} & \underline{\mathbf{0}} & \leq \\ \mathbf{2} & \mathbf{2} & \underline{\mathbf{1}} & \underline{\mathbf{1}} & \underline{\mathbf{2}} & \underline{\mathbf{0}} & \leq \\ \mathbf{1} & \mathbf{1} & \underline{\mathbf{2}} & \underline{\mathbf{2}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \leq \end{array}\right|$$

- Matrix after the 2nd iteration of phase 3 of *Canonicalisation*:

$$\left|\begin{array}{cccccc|c} \underline{\mathbf{1}} & \underline{\mathbf{4}} & \underline{\mathbf{3}} & \underline{\mathbf{2}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{4}} & \underline{\mathbf{2}} & \underline{\mathbf{1}} & \underline{\mathbf{2}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{3}} & \underline{\mathbf{1}} & \underline{\mathbf{4}} & \underline{\mathbf{1}} & \underline{\mathbf{2}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{2}} & \underline{\mathbf{2}} & \underline{\mathbf{1}} & \underline{\mathbf{1}} & \underline{\mathbf{2}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{1}} & \underline{\mathbf{1}} & \underline{\mathbf{2}} & \underline{\mathbf{2}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \leq \end{array}\right|$$

Figure 4.2. A sample constraint for which Canonicalisation converges with multiple iterations of the third phase

$$C_1 \left|\begin{array}{ccccc} 0 & 0 & 1 & 0 & \leq \\ 0 & 1 & 0 & 0 & \leq \\ 1 & 0 & 0 & 0 & \leq \end{array}\right| \quad C_2 \left|\begin{array}{ccccc} 0 & 0 & 1 & 0 & \leq \\ 1 & 0 & 0 & 0 & \leq \\ 0 & 1 & 0 & 0 & \leq \end{array}\right| \quad C_3 \left|\begin{array}{ccccc} 1 & 0 & 0 & 0 & \leq \\ 0 & 1 & 0 & 0 & \leq \\ 0 & 0 & 1 & 0 & \leq \end{array}\right|$$

Figure 4.3. Sample constraints for which Canonicalisation converges in the fourth phase

algorithm *Canonicalisation* in general. In the example of Figure 4.3 the fourth phase of the algorithm transforms matrices C_1 and C_2 into C_3, which is their canonical form.

While the first three phases are computationally inexpensive, the fourth phase of *Canonicalisation* is expensive, and can thus affect the overall performance of the algorithm. In general, the problem of determining the equivalence of constraints is as hard as the Graph Isomorphism problem, for which no polynomial-time algorithm is known yet [KST12]. The complexity of canonicalisation algorithm is polynomial up to the third phase, which is usually enough to determine the equivalence between constraints in many practical cases. Specifically, the complexity of the first three steps is $O(min(n,m) * n * m * log(max(n,m)))$. The fourth phase applies only to matrices containing sub-matrices with the same comparison operator for all rows, equal constant terms, equal sets of variable coefficients, and exactly equal sequences of the coefficients of those variables for which the first three phases succeeded to establish a stable relative order. Intuitively, this is a rare case: in the experiments reported in Chapter 6, the algorithm *Canonicalisation* converges within phase three on more than 98% of the constraints that we analysed.

In Appendix A, we present the proof that *Canonicalisation* algorithm converges on any input constraint, and the canonical form it generates satisfies Definition 4.1.

4.3 Complexity of Computing the ReCal Canonical Form

In this section we briefly discuss the possibility of using the *Canonicalisation* algorithm to address the Graph Isomorphism (GI) problem.

In general, the matrix equivalence problem is as hard as the Graph Isomorphism (GI) problem, which for the moment is not known to be solvable in polynomial time, and many researchers believe it is NP-hard [KST12]. Two graphs are isomorphic if they contain the same number of graph vertices connected in the same way. Formally, two graphs G and H with graph vertices $V = \{v_1, v_2, ..v_n\}$ are isomorphic if there exists a permutation p of V, such that (v_i, v_j) is in the set of graph edges E_G if and only if $(p(v_i), p(v_j))$ is in the set of graph edges E_H.

In particular, the GI problem can be reduced to the matrix equivalence problem in two steps:

(i) Reduce the GI problem to the Bipartite Graph Isomorphism problem.
Given an undirected graph $G = (V_G, E_G)$, where V is the vertex set and E is the edge set, G can be transformed into a bipartite graph $G_B = (V_1, V_2, E_{G_B})$ by mapping BG:

$$V_G \rightarrow V_1$$
$$E_G \rightarrow V_2$$
$$\{(v_1, v_2) | v_1 \in V_G, v_2 \in E_G, v_2 = (v_1, \_) or (\_, v1)\} \rightarrow E_{G_B}.$$

Two graphs G_1 and G_2 are isomorphic, if and only if their bipartite graphs $BG(G_1)$ and $BG(G_2)$ are isomorphic.

(ii) Reduce the Bipartite Graph Isomorphism problem to Matrix Equivalent problem . A bipartite graph $G_B = (V_1, V_2, E_{G_B})$ can be represented in adjacency matrix M, with the rows for vertices of V_1, and columns for vertices of V_2, and cell $M[i, j]$ for edge $e = (v_{1_i}, v_{2_j} \in E_{G_B})$. Two bipartite graphs G_{B_1} and G_{B_2} are isomorphic, their adjacency matrices M_1 and M_2 can be transformed to each other by permutation on rows and columns.

By reducing GI problem to the matrix equivalence problem (and the reduction as above can be done in polynomial time), any algorithm that solves the matrix equivalence problem can be adopted to solve the GI problem. Thus, although theoretically there not exist a polynomial algorithm to solve GI problem, our *Canonicalisation* algorithm can be used to efficiently solve the problem in many practical cases.

Chapter 5

The GPU-based parallel approach to proof reusing

Extending symbolic program analysis with a proof reusing approach like the ReCal *approach proposed in this thesis, aims to improve the efficiency of the analysis, under the hypothesis that retrieving and reusing a proof should be significantly faster than solving the constraint a constraint solver. However, effective proof reusing approaches rely on sophisticated simplifications and normalisations of the constraints that may incur in critical overheads, up to even penalising the overall performance of the analysis. In this chapter, we present* ReCal-gpu*, a parallel proof reusing framework that implements the* ReCal *approach described in the previous chapters by exploiting a parallel GPU computing capability to achieve very high efficiency. The chapter describes the overall structure of* ReCal-gpu *with respect to the structure of* ReCal*, and presents the parallel version of both the* logical simplification *and the* Canonicalisation *steps.*

5.1 Parallel Deployment of ReCal

The parallel deployment of the *ReCal* approach is an instance of general purpose computing on graphics process units (GP-GPU) that consists in applying Graphics processing units (GPU) to implement highly efficient applications out of the domain of computer graphics. GPUs are parallel computation platforms originally designed to rapidly manipulating computer graphics and image processing. Thanks to the massive-parallel architecture, GPUs can execute computationally intensive tasks very much faster than conventional CPUs. In this chapter, we present the GP-GPU deployment of *ReCal*.

In *ReCal*, simplification and normalisation of the constraints by means of the *Canonicalisation* algorithm are essential steps to identify many equivalent constraints and constraints related by implication. However, when dealing with large constraints char-

acterised by hundred of variables and clauses, processing these simplifications and normalisations can incur in high overheads. In this chapter, we propose a parallel GP-GPU version of *ReCal*, which efficiently simplifies and normalises constraints, thus overcome the scalability issues of the serial algorithms.

The basic intuition of the GP-GPU deployment of *ReCal* is to exploit the conjunctive and linear structure of the constraints, the elaborate in separate parallel threads the computations that relate to distinct clauses and distinct variables of the constraints, which in turn correspond to distinct rows and columns, respectively, in the matrix representation of the constraints. Based on this intuition, we design *ReCal-gpu*, a GPU-based parallel proof reusing framework.

The logical architecture of *ReCal-gpu* shares the work flow of *ReCal* that we present in Figure 3.1 at page 22. After *preprocessing*, *ReCal-gpu* abstracts the input constraint as matrixes, and exploits the matrix representation to parallelise both the *logical simplification* and *Canonicalisation* phases, which account for the large majority of the processing time of *ReCal*. *ReCal-gpu* executes the *logical simplification* step by allocating a computation unit for each pair of rows of the matrix (that is, pairs of clauses of the constraint), and executes each phase or iteration of the *Canonicalisation* algorithm by allocating a computation unit to reason on each row and column of the matrix. Notice that, to achieve high efficiency, the parallel version of *Canonicalisation* implements up to the first three steps of the serial *Canonicalisation* (Algorithm 1).

We present the parallel algorithms of the *logical simplification* and *Canonicalisation* phases in the next sections.

5.2 Parallel Logical Simplification

The *ReCal-gpu logical simplification* phase is composed of a *redundant clauses elimination* and a *conflicting clauses detection* step: *ReCal-gpu* checks (i) each pair of clauses in parallel to evaluate which pairs include redundant clauses (Definition 3.1) and (ii) whether there exists any pair that corresponds to conflicting clauses (Definition 3.2).

The *logical simplification* phase *logically simplifies* the input matrix representation of the constraint, by forking separate threads that evaluate redundancy and conflicts with respect to each possible pairs of clauses of the input constraint. For a formula with n clauses, The *logical simplification* phase elaborates $n(n-1)/2$ pairs, which for a constraint with 100 clauses amount to 4,950 pairs. Thus, the computation can be executed very efficiently on a standard GPU processor that can host the execution of thousands of threads, and indeed can execute thousands of those threads in parallel. For example, a GeForce GTX 580 GPU Unit can host 24,576 threads, and optimises their execution with very high degree of parallelism.

Algorithm 2 describes the parallel logical simplification phase in pseudocode. It instantiates two global variables, a boolean variable *conflict* with initial value set to *false*, and an array *redundant*, with a cell for each row in the input matrix, with all

the values initialised as 0 (false). Function *Simplification* starts by calling function *parallel_simpl*, which in turn spawns two sets of parallel threads that check conflicting clauses according to function $check\_conflict$ and redundant clauses according to function $eliminate\_redundant\_clauses$, respectively. Each of these thread sets includes a thread for each possible pair $((M[i,*],M[j,*]))$ of rows of the constraint matrix M. When all the threads terminate, if $conflict$ was marked as $true$ in any of the conflict detection threads, *Simplification* returns $UNSAT$, otherwise, it returns the matrix M' obtained by pruning the rows of M that positionally correspond to the items of $redundant$ that were marked as $true$ in any of the redundancy detection threads.

Function $check\_conflict$ checks whether a pair of clauses correspond to conflicting clauses. It evaluates if a pair $((M[a,*],M[b,*]))$ of clauses satisfies the Definition 3.2, and if it is the case, it marks the global variable *conflict* as $true$. Function $eliminate\_redundant\_clauses$ evaluates if a pair of clauses includes a redundant clause. It checks if the clause $M[a,*]$ implies the clause $M[b,*]$ (or vice-versa clause $M[b,*]$ implies $M[a,*]$) according the Definition 3.1, if so, it marks the b-th value (resp. a-th) in array $redundant$ as 1 (true). To avoid critical races, all parallel threads access the matrix as read-only, and update the values of the global variables *conflict* and *redundant* only once a pair of conflicting clauses or a redundant clause is confirmed.

For an input constraint matrix with n rows and m columns, the asymptotic complexity of parallel *logical simplification* is $O(m)$, because the algorithm processes the n rows in parallel, with each thread simply scanning the m coefficients in the corresponding row. The non parallel version has complexity $O(n*m)$.

5.3 Parallel Canonicalisation

Algorithm *Canonicalisation* is a core component of *ReCal*: it identifies the class of equivalent constraints that can be turned into exactly the same canonical form by suitably renaming the variables, and suitably permuting the order of the clauses and the variables in the constraints. The iterative version of algorithm *Canonicalisation* that we introduced in the previous chapter to operationally define the *ReCal* canonical form incurs crucial scalability issues, since it goes through several expensive permutations of the input constraint matrix. Our experiments indicate that implementing the *Canonicalisation* algorithm as a direct transposal of the corresponding definition algorithm leads to crucial performance penalties, when dealing with constraints with hundreds of clauses over large amounts of variables.

This section presents a parallel version of the *ReCal Canonicalisation* algorithm, which we implement in *ReCal-gpu*, and which dramatically improve the performance of *ReCal*.

The parallel *Canonicalisation* algorithm sorts rows and columns of the constraint matrix to obtains the same matrix for all constraints belonging to the same equivalence class. The parallel *Canonicalisation* exploits the values in each row and column (first

Algorithm 2 Parallel logical simplification

```
    /* Global variables (in the GPU memory): */
 1: conflict ← false /* True as soon as a conflict is detected */
 2: redundant ← {0,0,...} /* redundant marks for each clause, initialised as non-redundant */

 3: function Simplification(M)
    /* M: a matrix (n rows, m columns) representing a constraint */
 4:     parallel_simpl(M)
 5:     if conflict == True then
          return UNSAT /* Implemented in GPU-opt */
 6:     end if
 7:     for i=1..n do
 8:         if redundant[i] ==0 then add M[i,*] to M'
 9:         end if
10:     end for
        return M'
11: end function

12: function parallel_simpl(M)
    /* M: a matrix (n rows, m columns) representing a constraint */
13:     Spawn thread (∀i = 1..n, j = (i+1)..n) :
        check_conflict(M[i,∗],M[j,∗])
14:     Spawn thread (∀i = 1..n, j = (i+1)..n) :
        eliminate_redundant_clauses(M[i,∗],M[j,∗])
15:     Start all threads and wait all terminate
16: end function

17: function check_conflict(M[a,∗],M[b,∗])
18:     if M[a,m−1] == " ≤ " && M[b,m−1] == ' ≤ " then
19:         allZero ← ∀j = 1..m−2 : M[a, j]+M[b, j] == 0
20:         if allZero && M[a,m]+M[b,m] < 0 then
21:             conflict = true
22:         end if
23:     end if
24: end function

25: function eliminate_redundant_clauses(M[a,∗],M[b,∗])
26:     allSame ← ∀j = 1..m−2 : M[a, j] == M[b, j]
27:     if allSame == True then
28:         if M[b,m−1] = " ≤ " && M[a,m] > M[b,m] then
29:             redundant[a] =1
30:         end if
31:         if M[a,m−1] = " ≤ " && M[a,m] < M[b,m] then
32:             redundant[b] =1
33:         end if
34:     end if
35: end function
```

two phases of the algorithm), as well as the incrementally identified rows and columns with stable mutual ordering (third phase of the algorithm). It works by allocating separate threads to process each row and each column of the constraint matrix, to compute a hash code that uniquely identifies each row and each column, and exploits these hash codes to identify the unique mutual ordering of rows and columns according to the order of the corresponding hash codes.

Parallel *Canonicalisation* generates the hash codes iteratively, following the phases of the sequential algorithm. It computes the hash codes of rows and columns with respect to the sorted values in the rows and columns, and then exploits the values in the incrementally identified columns (rows) with finalized mutual ordering, until either obtaining hash codes different for all raws and columns, thus uniquely identifying rows and columns, or or reaching a fixed-point.

Below, we present the parallel *Canonicalisation* algorithm in detail, discuss the consistence between the sequential and the parallel version of *Canonicalisation*, and compare their complexity.

5.3.1 The $Canonicalisation_{par}$ algorithm

$Canonicalisation_{par}$, the parallel version of the *Canonicalisation* algorithm, is designed to to parallelise the computation, and minimise memory copying when executed on a cluster of GPU processing units.

Algorithm 3 presents $Canonicalisation_{par}$ in pseudocode. The algorithm starts executing function *parallel_canonicalise*, which takes in input a constraint represented in matrix form, and suitably permutes the rows and columns of the matrix. The permutation depends on the row and column *hashcodes* that is computed with function *compute_hashcodes*, and applies to all columns but the last two that represent constant terms and comparison operators. Function *parallel_canonicalise* returns the re-ordered matrix as the canonical form of the input constraint.

Figure 5.1 illustrates the algorithm through the example of the sample constraint that we presented in Chapter 4 (Figure 4.2). The figure shows (a) a sample constraint with 5 variables and 5 clauses, (b) its matrix representation (5 rows and 7 columns), (c) the 32-bit hashcodes (in hexadecimal format) computed for both the rows and the columns of this matrix when executing function *compute_hashcodes*, and (d) the canonical form that function *parallel_canonicalise* returns after permuting rows and columns of the matrix according to their corresponding hashcodes.

The core characteristic of Algorithm 3 is the parallel computation of the hashcodes of rows and columns: Function *compute_hashcodes* spawns a thread for computing the hashcode of each row and each column. Each thread receives an index that identifies either a row or a column of the input matrix, and a vector entry to the values of the row or column, respectively. All threads access the constraint matrix in parallel as read-only, and write their results (the hashcodes) in the global arrays *hrows* and *hcols* at mutually exclusive positions, thus avoiding critical races by construction. When all threads terminate, the values of row and column hashcodes are available in the global vectors $hrows$ and $hcols$, respectively, and function *compute_hashcodes* returns the results to function *parallel_canonicalise*.

The parallel threads coordinate each other through a set of shared global vectors (Algorithm 3, lines 13—18): The global vectors $hrows$ and $hcols$, initially set to zero,

Algorithm 3 $Canonicalisation_{par}$

```
1: function parallel_canonicalise(M)
   /* M: a matrix (n rows, m columns) representing a constraint */
2:     hrows, hcols ← compute_hashcodes(M)
3:     M' ← sort the rows of M by decreasing hrows
4:     M'' ← sort the first m−2 columns of M' by decreasing hcols
5:     return M''
6: end function

7: function compute_hashcodes(M)
   /* M: a matrix (n rows, m columns) representing a constraint */
8:     Spawn thread (∀i = 1..n): row_hashcode(i, M[i, ∗])
9:     Spawn thread (∀j = 1..m): col_hashcode(j, M[∗, j])
10:    Start all threads and wait all terminate
11:    return hrows, hcols
12: end function

   /* Global variables (in the GPU memory): */
13: hrows ← {0, 0, ...} /* Row hashcodes (n items, initially zero) */
14: hcols ← {0, 0, ...} /* Column hashcodes (m items, initially zero) */
15: hrows' /* Additional locations for the row hashcodes */
16: hrows'' /* Further additional locations for the row hashcodes */
17: hcols' /* Additional locations for the column hashcodes */
18: hcols'' /* Further additional locations for the column hashcodes */

19: function row_hashcode(i, vals)
   /* i: row index */
   /* vals: row values */
20:    unique ← false
21:    fixpoint ← false
22:    while ¬unique ∧ ¬fixpoint do

23:        hrows''[i] ← hash(sort(pairs(vals, hcols)))
24:        hrows'[i] ← hrows[i]

25:        synch_all_threads

26:        unique ← ⋀_{j≠i} hrows''[i] ≠ hrows''[j]
27:        fixpoint ← |{distinct(hrows'')}| = |{distinct(hrows')}|
28:                 ∧ |{distinct(hcols'')}| = |{distinct(hcols')}|
29:        hrows[i] ← hrows''[i]

30:        synch_all_threads
31:    end while

32:    hrows'[i] ← hrows[i]
33:    return
34: end function

35: function col_hashcode(j, vals)
   /* ...Omitted: Dual case with respect to function row_hashcode... */
36: end function
```

store the row and column hashcodes incrementally computed by each thread, thus allowing any thread to inspect the hashcodes computed by other threads. The vectors $hrows'$, $hrows''$, $hcols'$ and $hcols''$ provide additional locations to redundantly store

$$
\begin{aligned}
&2V_1 + 2V_2 + V_3 + V_4 + V_5 \leq 0 \\
\wedge &V_1 + 4V_2 + V_3 + 3V_4 + 2V_5 \leq 0 \\
\wedge &V_1 + V_2 + 2V_3 + 2V_4 + 2V_5 \leq 0 \\
\wedge &2V_1 + 1V_2 + 3V_3 + 4V_4 + V_5 \leq 0 \\
\wedge &2V_1 + 3V_2 + 4V_3 + V_4 + V_5 \leq 0
\end{aligned}
$$

(a) Input constraint

2	2	1	1	1	const(0)	$\leq$
1	4	1	3	2	const(0)	$\leq$
1	1	2	2	2	const(0)	$\leq$
2	1	3	4	1	const(0)	$\leq$
2	3	4	1	1	const(0)	$\leq$

(b) A constraint matrix M
(cfr. Algorithm 3, line 1)

$$
\begin{aligned}
hrows &= [2956225, 5e554618, c5729135, d0253753, 70e97b53] \\
hcols &= [e19e72a5, 1efd529e, 832f0df7, acb9d1f7, e0f84795, -, -]
\end{aligned}
$$

(c) The row/column hashcodes
(cfr. Algorithm 3, line 2)

1	2	2	1	1	const(0)	$\leq$
2	1	4	1	3	const(0)	$\leq$
1	2	3	4	1	const(0)	$\leq$
2	1	1	2	2	const(0)	$\leq$
1	2	1	3	4	const(0)	$\leq$

(d) M with permuted rows and columns
(cfr. Algorithm 3, lines 3 and 4)

Figure 5.1. Result of parallel Canonicalisation on a sample constraint

temporary values of the hashcodes, to avoid critical read/write races during the parallel computations.

Function *row_hashcode* (Function *col_hashcode*), the core part of the $Canonicalisation_{par}$ algorithm, computes a row (column) hashcode. Here we decribe in details Function *row_hashcode*.

Function *row_hashcode* iterates through a loop (lines 22—31) that computes the value of the hashcode multiple times, until either computing a hashcode value that uniquely identifies the current row with respect to the hashcodes of the other rows (condition $unique$ at line 22), or reaching a fixed-point at which any additional iteration would not add new information on the relative positions of the rows (condition $fixpoint$ at line 22). Below we explain the core steps of each iteration: the computation of the hashcode value (line 23), the check of uniqueness of the computed hashcode (line 26), and the check of fixed-point convergence (line 27).

The algorithm computes the value of the hashcode (line 23) by (i) pairing each row coefficient with the current hashcode of the column that correspond to the coefficient, (ii) sorting the pairs in descending order, and (iii) hashing the sorted list of pairs to a 32-bit hashcode.[1] This step computes distinct hashcodes for rows that either differ in the (sorted) list of their coefficients, or include the same coefficients in columns that are identified with distinct hashcodes, since the sorted list of ⟨row coefficient, column hashcode⟩ pairs will differ in these cases, up to the rare case hashing collisions.

Figure 5.2 illustrates the first iteration of the computation of the row hashcodes (line 23) through the example of the matrix of Figure 5.1 (a). The figure shows (a) the input constraint matrix, (b) the vectors of pairs ⟨value, column_hashcode⟩ that the *row_hashcode* threads build at the beginning of the first iteration of the loop, by pairing the values in the considered row with the hashcodes of their corresponding columns (the readers should notice that all column hahscodes are initially set to zero because we are showing the first iteration of the loop), (c) the sorted vectors obtained by sorting the pairs in descending order, and (d) the new row hascodes computed by applying the hash function to the sorted vectors.

In Figure 5.2 we observe that the computations associate the same hashcode value ($e1a3c9d5$) to the second, fourth and fifth rows of the input matrix, as expected, since these rows are identical after sorting the ⟨row coefficient, column hashcode⟩ pairs. The first and third rows correspond to distinct sorted vectors, and thus result in computing distinct hashcodes ($e19e72a5$ and $e0f84795$, respectively, marked in blue). At the first iteration of the loop, which is the case illustrated in the figure, the distinctness of the hashcodes depends only on the distinctness of the sorted lists of row coefficients, since all column hashcodes are initially zero. (The result is consistent with the example shown in Figure 4.2, Chapter 4: after the first and the second phrase of the *Canonicalisation* algorithm of sorting by constant, comparison operator and coefficient values, the first and third rows can be distinguished from the other row.) In the following iterations, *col_hashcode* threads compute different hashcodes, which distinguish the three rows with equal hashcodes after the first iteration.

The threads of function *row_hashcode* compute the hashcodes in parallel, and no thread overrides the previous value of the hashcode with the new one before synchro-

[1] Our prototype implementation of the algorithm refers to the popular hash function *djb*: http://www.cse.yorku.ca/ oz/hash.html.

2	2	1	1	1	const(0)	$\leq$
1	4	1	3	2	const(0)	$\leq$
1	1	2	2	2	const(0)	$\leq$
2	1	3	4	1	const(0)	$\leq$
2	3	4	1	1	const(0)	$\leq$

(a) The input matrix

$\langle 2,0\rangle$	$\langle 2,0\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle const(0),0\rangle$	$\langle \leq,0\rangle$
$\langle 1,0\rangle$	$\langle 4,0\rangle$	$\langle 1,0\rangle$	$\langle 3,0\rangle$	$\langle 2,0\rangle$	$\langle const(0),0\rangle$	$\langle \leq,0\rangle$
$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle 2,0\rangle$	$\langle 2,0\rangle$	$\langle 2,0\rangle$	$\langle const(0),0\rangle$	$\langle \leq,0\rangle$
$\langle 2,0\rangle$	$\langle 1,0\rangle$	$\langle 3,0\rangle$	$\langle 4,0\rangle$	$\langle 1,0\rangle$	$\langle const(0),0\rangle$	$\langle \leq,0\rangle$
$\langle 2,0\rangle$	$\langle 3,0\rangle$	$\langle 4,0\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle const(0),0\rangle$	$\langle \leq,0\rangle$

(b) The ⟨coefficient, hashcode⟩ pairs

$\langle 2,0\rangle$	$\langle 2,0\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle const(0),0\rangle$	$\langle \leq,0\rangle$
$\langle 4,0\rangle$	$\langle 3,0\rangle$	$\langle 2,0\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle const(0),0\rangle$	$\langle \leq,0\rangle$
$\langle 2,0\rangle$	$\langle 2,0\rangle$	$\langle 2,0\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle const(0),0\rangle$	$\langle \leq,0\rangle$
$\langle 4,0\rangle$	$\langle 3,0\rangle$	$\langle 2,0\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle const(0),0\rangle$	$\langle \leq,0\rangle$
$\langle 4,0\rangle$	$\langle 3,0\rangle$	$\langle 2,0\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle const(0),0\rangle$	$\langle \leq,0\rangle$

(c) The sorted pairs

2956225
$dc2a7465$
$c5729135$
$dc2a7465$
$dc2a7465$

(d) The hashcodes

Figure 5.2. Computation of row hashcodes at the first iteration of function row_hashcode (cfr. Algorithm 3, line 23)

nising (line 25), since threads can only read value of the hashcode. Each thread temporarily stores the newly computed hashcode value in vector $hrows''$ (line 23), and the previous value of the hashcode in vector $hrows'$ (line 24) to make values available to the next step. After threads synchronization (line 25), vectors $hrows'$ and $hrows''$ contain the values of all hashcodes before and after the computation of this step, respectively.

At the next step, all threads compute the uniqueness (line 26) and fixed-point (line 27) checks, and update the main hashcode vector (line 29) in parallel, up to the next synchronization point (line 30). Both the uniqueness and fixed-point checks refer to the values in the vectors $hrows'$ and $hrows''$ that are only read and never modified at this step.

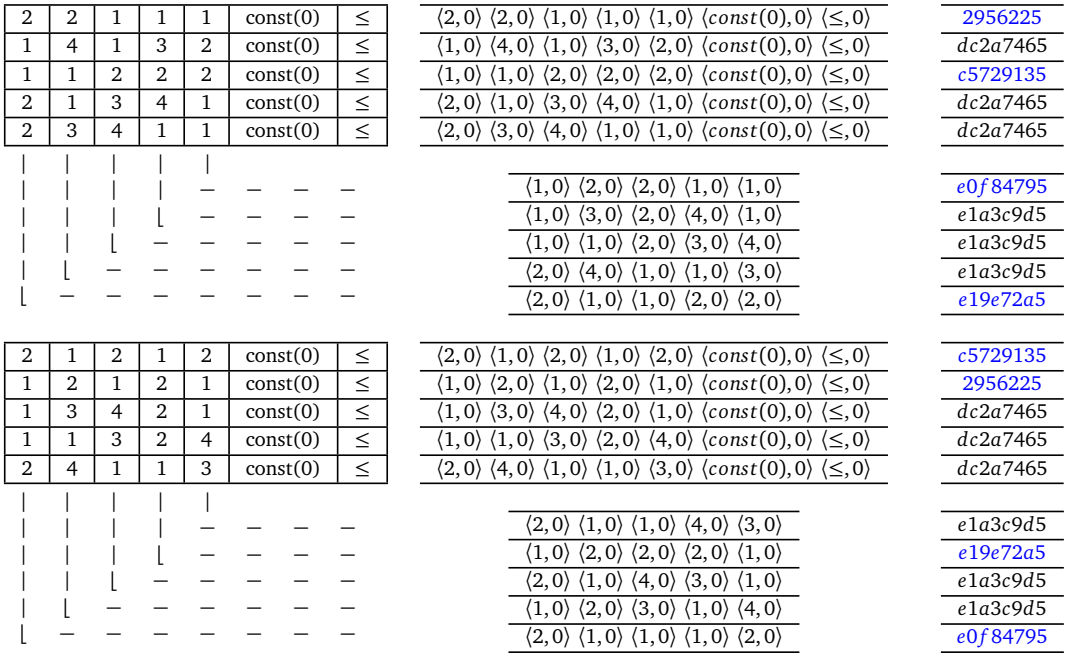
The uniqueness of the computed hashcode (line 26) is determined by comparing the newly computed hashcode with the hashcodes computed by the other *row_hashcode* threads. For example, with reference to Figure 5.2(d), after the first iteration of all *row_hashcode* threads, the uniqueness check at line 26 succeeds for the threads that are computing the hashcodes of either the first and the third rows of the input matrix, since both these threads resulted in computing hashcode values (2956225 and c5729135, respectively) that are distinct from the ones of all other rows. Conversely, the uniqueness check fails for the threads that are computing the hashcodes of the first, the third and the fifth row, since all these threads computed the same hashcode value ($dc2a7465$) and then this hashcode value does not univocally identify these rows.

Once generated unique hashcodes for all rows and columns, we can uniquely order rows and columns. For any thread that terminates due to the success of the uniqueness check, the algorithm forces the three vectors $hrows$, $hrows'$ and $hrows''$ to hold exactly the same hashcode value at the corresponding locations (line 29 and line 32), thus guaranteeing that all other running threads will read consistent values when referring to any of these vectors at any later iteration for computing the hashcodes of the other dimension. The uniqueness check is equivalent to the *converge* concept in the sequential *Canonicalisation* algorithm in phase 3.

Function *row_hashcode* evaluates fixed-point convergence (line 27) by checking the absence of changes between consecutive iterations. The hashcodes that reach a fixed-point, leave undecided the ordering of the rows or columns (end of phase 3), and are disambiguated with phase 4 of the algorithm. $Canonicalisation_{par}$ refers to the sequential implementation for computing this phase.

In Chapter 6, we report the empirical results of the experiments about both the convergence of $Canonicalisation_{par}$ on constraints generated in symbolic analysis and the impact of the non-convergence cases on the ability of *ReCal* to find equivalent constraints.

In the next section, we exemplify the execution of algorithm $Canonicalisation_{par}$ on two sample equivalent constraints, and compare $Canonicalisation_{par}$ with *Canonicalisation*.



(a) The input matrices (b) The ⟨coefficient, hashcode⟩ pairs (d) The hashcodes

Figure 5.3. First iteration of algorithm $Canonicalisation_{par}$ on two sample equivalent constraints

5.3.2 Executing $Canonicalisation_{par}$

This section discusses the execution algorithm $Canonicalisation_{par}$ by example, referring to two sample constraints. Figure 5.3 illustrates the first iteration of the algorithm, by showing

(a) the two input matrices that represent the sample constraints that are equivalent, but require non trivial permutations of rows and columns to be turned into canonical form in order to reveal their equivalence,

(b) the ⟨coefficient, hashcode⟩ pair vectors built at the beginning of the first iteration of the *row_hashcode* (resp.*col_hashcode*) threads for each row (resp. column) of both these matrices (as we commented above, at the beginning of the first iteration, all hashcodes hold the initial value, that is, zero),

(c) the new hascodes computed by the *row_hashcode* and *col_hashcode* threads for all rows and columns.

We already presented the computation of the *row_hashcode* threads for the first of the two matrixes in Figure 5.2. We report it here to facilitate comparing the results of the algorithm on the two constraints.

In the figure, we mark in blue the row and column hashcodes that are unique with respect to the ones of all other rows and columns, respectively. The threads that computed these marked hashcodes terminate immediately after the first iteration, due to the success of the uniqueness check. The readers should notice that rows and columns characterised by different sets of coefficients result in distinguishable hashcodes already after the first iteration.

After the first iteration, the state of the hashcode vectors does not satisfy the fixed-point convergence check, thus the algorithm executes further iterations of the threads *row_hashcode* and *col_hashcode* that are still alive. Figure 5.4 and Figure 5.5 illustrate the second and third iteration of the threads, where the coefficients in the rows and columns are paired with the (non-zero) hashcodes of the crossing columns and rows, respectively, and then a new hashcode value is computed based on these pair vectors. The second iteration produces 3 new rows and 3 new columns with unique hashcodes, and the third iteration converges because all rows and columns have unique hashcodes for both constraints. By sorting the rows and columns of the two input matrixes according to the final hashcodes, the two constraints are both transformed to the same canonical form, the one that was already shown in Figure 5.1(d).

By comparing the above examples with the example in Figure 4.2 (previous chapter), we observe that the second and third iterations of $Canonicalisation_{par}$ and *Canonicalisation* yield the same ordering of the two matrixes. In Appendix B, we present the proof that $Canonicalisation_{par}$ is consistent with the sequential *Canonicalisation* algorithm, thus confirming its correctness with respect to the definition of the *ReCal* canonical forms of constraints.

5.3.3 Computational Complexity of $Canonicalisation_{par}$

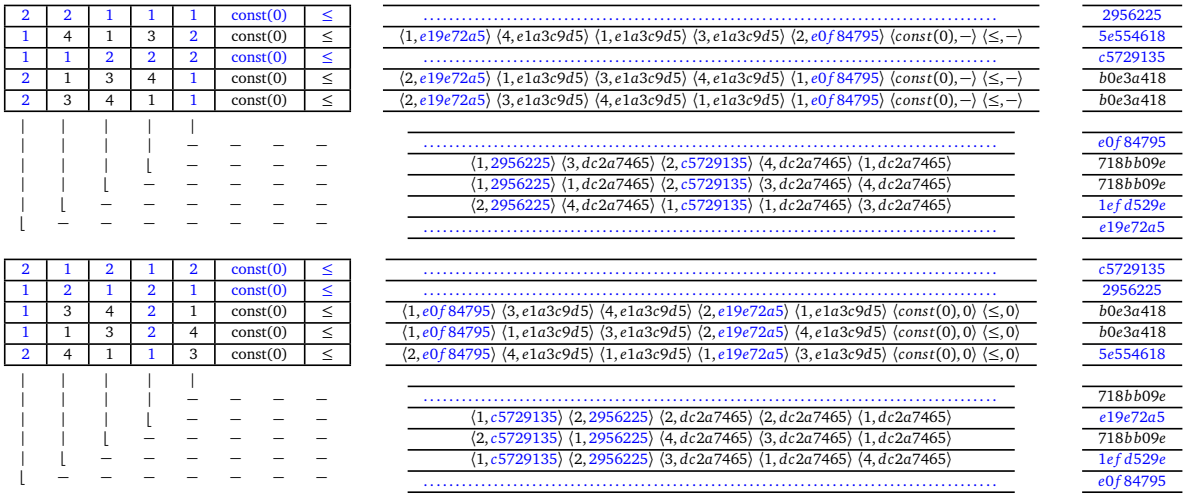
This section discusses the computational complexity of algorithm $Canonicalisation_{par}$ when executed on an input constraint matrix with n rows and m columns.

The complexity of a single iteration of a *row_hashcode* thread is $O(m * log(m))$, because the complexity is dominated by the step that sorts the ⟨row coefficient, column hashcode⟩ pair vector, and this vector contains m pairs. Dually, the complexity of a single iteration of a *col_hashcode* is $O(n * log(n))$.

Since each iteration increases the convergence of rows and columns by at least one, otherwise the algorithm reaches the fixed-point, the upper bound to the number of iterations of the *row_hashcode* and *col_hashcode* threads is n and m, respectively. Thus, the worst case complexity of executing all threads is $O(n * m * log(max(n, m)))$.

Our current data indicate that the convergence is usually much faster than n or m iterations: In our empirical studies, the algorithm converges within 4 iterations for the 99% of the constraints).

All other steps of the algorithm are linear in either n or m, thus the overall complexity is dominated by the execution of the threads: $O(n * m * log(max(n, m)))$.

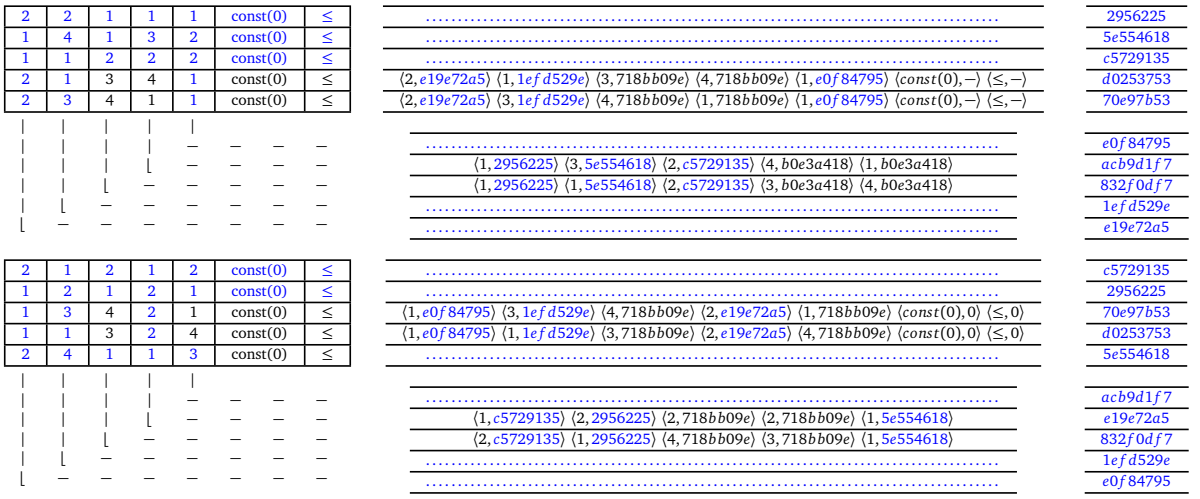


(a) The input matrices

(b) The ⟨coefficient, hashcode⟩ pairs

(c) The hashcodes

Figure 5.4. Second iteration of algorithm $Canonicalisation_{par}$ on the two sample equivalent constraints



(a) The input matrices

(b) The ⟨coefficient, hashcode⟩ pairs

(c) The hashcodes

Figure 5.5. Third iteration of algorithm $Canonicalisation_{par}$ on the two sample equivalent constraints

Thus, algorithm $Canonicalisation_{par}$ is an order of magnitude faster than the sequential version of the *Canonicalisation* algorithm presented in Algorithm 1, which is $O(min(n,m) * n * m * log(max(n,m)))$.

5.4 CUDA ReCal-gpu implementation

We implemented a prototype of *ReCal-gpu* on the CUDA platform, a parallel computing platform introduced by Nvidia, which allows software engineers to use a CUDA-enabled GPUs for general purpose processing. CUDA consists of a software layer that enables direct access to the GPU's virtual instructions and parallel computational elements, making it easier for specialists in parallel programming to use GPU resources [SK10]. The *ReCal-gpu* prototype is implemented in C++ programming language.

In this section, we present the main structure of the implementation of the parallel algorithm of *logical simplication* and *canoncalisation*. The implementation includes two parts, the sequential part running on CPU and the parallel part running on GPU. CUDA provides memory copy functions such as *cudaMemcpy* for communication between the sequential and the parallel functions.

The parallel processing of a constraint (function *normal*) starts from mapping the constraint matrix into GPU memory (line 4 in the code shown below), and then calls function *normalisation* to compute the canonical form on the GPU units. Function *normalisation* is declared in file *Kernel.cu*, where we define the parallel functions to execute on the GPU units.

```
1 // Normalise.cpp
2 int* normal(int* matrix){
3   ....
4   cudaMemcpy(gpu_matrix,matrix, size,cudaMemcpyHostToDevice);
5   int* canon_form = normalisation(gpu_matrix, ...);
6   return canon_form;
7 }
```

Kernel.cu declares two parallel functions: function *kernel_logical_simplification* and function *kernel_compute_hashcodes*. The former function realises the logical simplification steps *check_conflict* and *eliminate_redundant_clauses* (Algorithm 2). The latter function realises the $Canonicalisation_{par}$ algorithm (Algorithm 3).

Function *normalisation* invokes *kernel_logical_simplification* by passing the parameters of thread-grid size, the address of the constraint matrix, and the address of the global flags to mark conflicts and redundant clauses, while it refers to the thread *ID* to select the matrix rows to be inspected for conflict and redundancy. The API *cudaDeviceSynchronize* allows for waiting the termination of all logical simplification threads. Then, function *normalisation* iteratively executes function *kernel_compute_hashcodes* to compute the hashcodes, until converging or reaching the fixed-point. At the end, function *normalisation* copies the hashcodes in the CPU memory, permutes the matrix according to the hashcodes, and returns the permuted matrix as the canonical form.

```
1  // Kernel.cu
2  __global__ void kernel_logical_simplification(int* gpu_mat, boolean* conflict, ↩
       int* redundant,...);
3  __global__ void kernel_compute_hashcodes(int* gpu_mat,signed long* hashcodes, ↩
       ...);
4
5  int* normalisation(int* gpu_matrix, ...){
6  ...
7   // logical simplification
8   kernel_logical_simplification<<<threadsPerBlock,blocksPerGrid>>>(gpu_mat, ↩
        conflict, redundant,...);
9   cudaDeviceSynchronize();
10
11  //canonicalisation
12  while(converge == false or fixed_point == false){
13    kernel_compute_hashcodes<<<threadsPerBlock,blocksPerGrid>>>(gpu_mat, ↩
          hashcodes,...);
14    cudaDeviceSynchronize();
15  }
16
17  cudaMemcpy(hashcodes_c, hashcodes, size,cudaMemcpyDeviceToHost);
18  permutation(matrix, hashcodes_c);
19  return matrix;
20 }
```

Chapter 6

Evaluation

In this chapter we presents the experimental evaluation of the ReCal *proof-reusing approach. The experiments indicate both to what extent the approach fosters the reuse of available proofs across, by referring to a large benchmark of constraints generated during the symbolic analysis of a set of programs, and to what extent* ReCal *improves the performance of constraint solving with respect to the constraints in the benchmark. We replicate the experiments with the state-of-the-art approaches Green, GreenTrie and Utopia, to better interpret the experimental data about* ReCal*. The results of our experiments complement previous experiments in the field by providing further evidence of the occurrence of equivalent and implication-related constraints during symbolic analysis, and testify the crucial contribution of* ReCal *to improve the efficiency of symbolic program analysis: In our experiments* ReCal *improves the constraint solving speed of a factor of 10 with respect to solving all constraints with the cutting-edge solver Microsoft Z3, and outperforms all competing proof-reusing approaches.*

In the previous chapters of this thesis we presented the *ReCal* proof-reusing approach that aims to improve the performance of symbolic analysis by reusing proofs that recur across multiple constraints. In this chapter, we provide empirical evidence that *ReCal* meets its goal. We collected a large benchmark of constraints generated by using the symbolic executors JBSE [BDP16] and Crest [BS08] to analyse many programs, and we experimented with *ReCal* to improve the performance of solving these constraints.

In the experiments, we quantify both the ability of *ReCal* to reuse proofs for the considered constraints and the performance improvement that derives from using the *ReCal* proof-reusing framework. The overall goal of proof-reusing frameworks is to reduce the constraint solving time, and thus mitigate the bottleneck of constraint solvers. This goal competes with the costs of the constraint processing steps that the proof-reusing frameworks exploit to increase their effectiveness in identifying many reusable

proofs, that is, simplification and normalisation of the constraints. In *ReCal* such costs may cause critical overheads, especially for arbitrarily large constraints. In this thesis, we propose a parallel algorithm for GP-GPU to reduce the constraint solving costs.

The experiments discussed in this chapter are designed to answer the following research questions:

Q1 How frequently do equivalent and implication-related constraints occur during symbolic program analysis, and how effective is *ReCal* in reusing proofs by identifying equivalent and implication-related constraints?

Q2 To what extent does *ReCal* improve the efficiency of symbolic program analysis?

Q3 What is the impact of the *ReCal* canonical form (computed with the *Canonicalisation* algorithm) on the effectiveness of the proof-reusing framework?

Research question Q1 focus on the feasibility of using the *ReCal* proof-reusing approach to improve the efficiency of symbolic program analysis. The usefulness of *ReCal* depends on both the frequency of equivalent and implication-related constraints and the effectiveness of *ReCal* in identifying reusable proofs by exploiting these relations across the constraints.

Research question Q2 focus on the end-to-end efficiency of *ReCal*, since processing the constraints with simplification and normalisation algorithms can cause critical overheads. We address Q2 by comparatively evaluating the efficiency of solving constraints with Z3, *ReCal* (with both its parallel and sequential version) and the competing state-of-the-art proof-reusing approaches, *Green*, *Greentrie* and *Utopia*.

The research question Q3 focus on the contribution of the *Canonicalisation* algorithm, which is a distinctive characteristic of *ReCal* with respect to the other proof-reusing approaches.

6.1 ReCal Prototype(s)

We experiment with three prototypes implementations of *ReCal*, which implement the different aspects of the approach: $ReCal_{seq}$, $ReCal_{gpu}$ and $ReCal_{gpu+}$.

$ReCal_{seq}$ implements the sequential *ReCal*, and is limited to reusing proofs from equivalent constraints. It is implemented in Python. It includes the sequential implementation of the *Canonicalisation* algorithm, and uses the canonical form of the constraints as search index for efficiently searching equivalent constraints. With $ReCal_{seq}$, we measure the amount of constraints that can be reused by equivalence checking, which we use as a baseline to quantify both the improvement of identifying constraints related by implication and the improvements in execution cost of the *Canonicalisation* algorithm.

$ReCal_{gpu}$ implements the parallel *ReCal*, with the same characteristics of $ReCal_{seq}$. It is implemented in C++ on the CUDA platform [SK10]. With $ReCal_{gpu}$, we mea-

sure the efficiency gain that we obtain with the parallel algorithm with respect to the sequential one.

$ReCal_{gpu+}$ extends the parallel implementation of *ReCal* with the identification of constraints related by implication, and the detection of internal conflicts between pairs of clauses in the constraints. With $ReCal_{gpu+}$, we measure the effectiveness and efficiency gain of the new these new characteristics of *ReCal*.

Table 6.1 summarises the differences between the prototypes.

	Canonicalisation	*Parallelism*	*Conflict Detection*	*Logical Implication*
$ReCal_{seq}$	✓			
$ReCal_{gpu}$	✓	✓		
$ReCal_{gpu+}$	✓	✓	✓	✓

Table 6.1. Features of the ReCal prototypes

6.2 Experimental Setting and Design

We evaluate the effectiveness and efficiency of *ReCal* by experimenting with a dataset of constraints that we generated by analysing a set of 22 third party subject programs with symbolic execution. The program set consists of both C and Java programs, and we use the symbolic executors Crest [BS08] and JBSE [BDP16] to analyse the C and the Java programs, respectively. Table 6.2 lists the subject programs (column Program), the sizes in lines of code (column LOC), the programming language (column Language) and the number of constraints (uniquely different from each other) (column Constraints) that we collected during the analysis of each program. The programs are taken from different repositories and are used as case studies in many scientific papers.[1]

When experimenting with the constraints from a program, *ReCal* starts with an empty repository, and considers the constraints incrementally in the order in which they are generated during symbolic execution. For each constraint, *ReCal* tries to find an equivalent constraint or a suitable implication-related constraint in the repository. When *ReCal* does not find any suitable constraint in the repository (we say that the current constraint results in a cache miss), *ReCal* generates a proof with the constraint solver Z3, and incrementally populates the repository with the constraint and the proof computed with Z3. After processing all the constraints from a program, *ReCal* reports the overall solving time, that is, the time spent to process the constraint and searching

[1]The interested readers can find the references to the subject programs at http://star.inf.usi.ch/recal/refs

Program	LOC	Language	#Constraints
afs	75	Java	203
avl	519	Java	11,161
ball	271	Java	210
block	79	Java	505
cdaudio	2171	C	55,329
collision	127	Java	6,812
dijkstra	142	Java	85
diskperf	1104	C	103,505
division	87	Java	1,257
floppy	1137	C	100,006
grep	10068	C	100,126
kbfiltr	599	C	188
knapsack	120	Java	7651
doubly-linked-list	806	Java	876
multiplication	277	Java	25,217
old-tax	78	Java	43
new-tax	78	Java	55
reverseword	32	Java	173
swapwords	181	Java	38,104
tcas	200	Java	13,476
treemap	806	Java	332,950
wbs	297	Java	239

Table 6.2. Subject programs

for proofs plus the solving time spent in case of cache misses, and the *reuse-rate*, that is, the percentage of constraints for which it succeeded in identifying a reusable proofs (we say that these constraints resulted in cache hits). In detail *ReCal* computes the *reuse-rate* as:

$$\textit{reuse-rate} = \frac{H}{H + M} \tag{6.1}$$

where H is the number of constraints for which *ReCal* identified a proof from the repository (cache hit), and M is the number of constraints for which *ReCal* fails to identify a reusable proof (cache miss).

We conducted comparative experiments with the same settings by using the state-of-the-art approaches Green, GreenTrie and Utopia, and measured the overall solving time and the proof reuse-rate across the same constraint datasets with these approaches.

We stored the constraint repository during the execution of *ReCal* with *Redis* [Red21],

program	Green	GreenTrie	Utopia	$ReCal_{seq}$	$ReCal_{gpu+}$
afs	76%	97%	**99%**	98%	**99%**
avl	73%	84%	91%	94%	**95%**
ball	6%	55%	**94%**	83%	84%
block	33%	56%	**49%**	35%	36%
cdaudio	16%	65%	**99%**	**99%**	**99%**
collision	76%	97%	**99%**	**99%**	**99%**
dijkstra	0%	48%	68%	96%	**99%**
diskperf	44%	87%	**99%**	**99%**	**99%**
division	0%	**99%**	**99%**	14%	**99%**
floppy	46%	93%	**99%**	**99%**	**99%**
grep	0%	**99%**	**99%**	**99%**	**99%**
kbfiltr	11%	51%	95%	92%	**99%**
knapsack	57%	**99%**	**99%**	68%	**99%**
doubly-linked-list	87%	96%	**96%**	91%	94%
multiplication	0%	**99%**	**99%**	**99%**	**99%**
new-tax	36%	82%	65%	43%	**89%**
old-tax	37%	86%	65%	44%	**90%**
reverseword	**99%**	**99%**	**99%**	**99%**	**99%**
swapwords	0 %	**99%**	**99%**	49%	**99%**
tcas	39%	98%	**99%**	**99%**	**99%**
treemap	96%	**99%**	**99%**	**99%**	**99%**
wbs	20%	66%	97%	**99%**	**99%**

Table 6.3. Reuse-rates of the different approaches

an in-memory database store that supports efficient data retrieval. We executed the experiments on a machine with 2.3 GHz Intel Core i7 processor and 16 GB of RAM, and a NVIDIA GeForce GT 650M 1024 MB GPU processor. The benchmark data and the prototypes are available on GitHub: https://github.com/meixianchen/ReCal-sym.

6.3 Experiment Results

In this section, we report the experiment results to answer the research questions outlined at the beginning of this chapter.

6.3.1 Effectiveness of ReCal

We evaluate the effectiveness of *ReCal* in fostering the reuse of constraint proofs, as the *reuse-rate* of *ReCal* obtained with the prototypes $ReCal_{seq}$, $ReCal_{gpu}$ and $ReCal_{gpu+}$. We recall that in each experiment, *ReCal* starts with an empty repository, and solves the constraints incrementally, in the order in which they are generated during symbolic execution. For each target constraint, *ReCal* slices it into independent sub-constraints, applies preprocessing, simplification and canonicalisation to compute the canonical form of the sub-constraints, and uses the canonical forms as search index to identify equivalent ($ReCal_{seq}$, $ReCal_{gpu}$ and $ReCal_{gpu+}$) or implication-related ($ReCal_{gpu+}$) constraints that were solved in the the analysis session up to that point.

Table 6.3 reports the reuse-rates measured for the constraints of the programs considered in the experiments, with Green, GreenTrie, Utopia, $ReCal_{seq}$ and $ReCal_{gpu+}$. The reuse-rate of $ReCal_{gpu}$ is the same as the reuse-rates of $ReCal_{seq}$, by construction. We highlight the best results for each program in green colour. We visualise the comparison between the approaches in the box-plot of the data in Figure 6.1. The experimental results shown in Figure 6.1 suggest the following observations:

(i) $ReCal_{seq}$ yields a *reuse-rate* higher than 90% for 15 (out of 22) programs, and higher than 95% for 12 programs. The first quartile, median, and third quartile are 68%, 98%, and 99%, respectively. The results confirm that equivalent constraints occur frequently during symbolic analysis.

(ii) $ReCal_{gpu+}$ improves the proof-reusing power by identifying constraints related by implication, further increasing the *reuse-rate*. The first quartile, median, and third quartile increase to 95%, 99%, and 99%, respectively. $ReCal_{gpu+}$ steadily achieves high *reuse-rates* on most programs.

(iii) Green [VGD12], which identifies equivalent constraints by clause normalisation, results in a lower *reuse-rate* than *ReCal* on the average, with a median value of 37%.

(iv) GreenTrie [JGY15], which identifies constraints related by implication with induction rules, and Utopia [ADP17], which heuristically searches for candidates solutions from similar constraints, result in *reuse-rate* similar to $ReCal_{gpu+}$. $ReCal_{gpu+}$ and Utopia reach the highest rate ($> 95\%$) at the first quartile.

In summary, our results suggest that Utopia and $ReCal_{gpu+}$ steadily achieve high *reuse-rate*, and are effective in identifying reusable proofs. The good results of GreenTrie are less steady than Utopia and $ReCal_{gpu+}$. Since Utopia considers a wider scope of proofs than *ReCal*, we are not surprised to find out in some cases it outperforms *ReCal* in finding more proofs. However, $ReCal_{gpu+}$ achives a better reuse-rate than Utopia for program avl, dijkstra, kbfiltr, new-tax and old-tax, as shown in Table 6.3. This suggest a good degree of complementarity between Utopia and $ReCal_{gpu+}$.

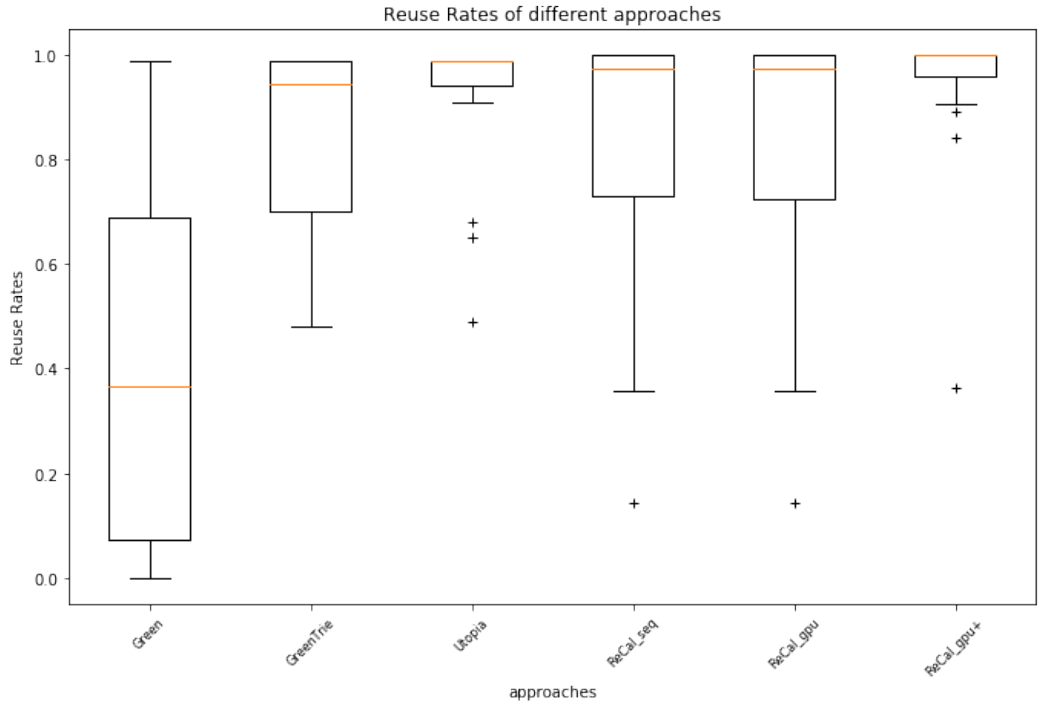


Figure 6.1. Summary statistics on the reuse-rates of the different approaches

6.3.2 Efficiency of ReCal

Proof-reusing approaches aim to reduce the constraints solving time in order to foster efficient program analysis. In the experiments we quantified the efficiency gain that can be obtained with *ReCal*, by comparing the time spent to solve all constraints in our experiments with the solver Z3, with the time spent when using Green, GreenTrie, Utopia, $ReCal_{seq}$, $ReCal_{gpu}$ and $ReCal_{gpu+}$, as illustrated in Figure 6.2. The experimental results shown in Figure 6.2 suggest the following observations:

(i) $ReCal_{seq}$ improves only a 14% of the running time with respect to using the solver Z3 without reusing any proof, and is very much slower than Green, GreenTrie, and Utopia. For example, comparing $ReCal_{seq}$ and Green, we see that $ReCal_{seq}$ takes approximatively twice the time of Green, even though in the previous section we observed that $ReCal_{seq}$ obtains higher reuse-rates than Green.

(ii) $ReCal_{gpu}$ significantly reduces the running time of *ReCal* from 6,662 seconds to 521 seconds.

(iii) $ReCal_{gpu+}$ further reduces the running time to 330 second, which is less than 5% of the running time of solver Z3.

The results suggest that the relatively low performance of $ReCal_{seq}$ depends mainly on the overhead of the simplification and canonicalisation steps, which the the parallel GPU computing of $ReCal_{gpu}$ and the reuse of proofs from logical implication of $ReCal_{gpu+}$ largely improve.

In summary, $ReCal_{gpu+}$ outperforms all other approaches in our experiments. As discussed above, the excellent performance of $ReCal_{gpu+}$ when considering constraint related by implication and conflicting clauses derives from conflict detection and identification of implication-related constraints. With conflict detection, $ReCal_{gpu+}$ can straightforwardly decide the unsatisfiability of some constraints without further processing, thus avoiding unnecessary computation. The time of each step in $ReCal_{gpu}$ and $ReCal_{gpu+}$ is presented in Figure 6.3: by considering conflicting clauses, $ReCal_{gpu+}$ takes more time than $ReCal_{gpu}$ in the simplification step, but saves 75% of the time spend in canonicalisation, which accounts for most of the execution time of $ReCal_{gpu}$. We run the experiment for 10 times to estimate the variance of the execution time: the variances of time of the $ReCal_{gpu}$ and $ReCal_{gpu+}$ are within 10%.

Figure 6.4 refines the reported data, by providing the details on the running time of $ReCal_{gpu+}$ and the other approaches on a per-program basis. The figure indicates the execution time in log scale, and highlights the time-axis at the time of 1 seconds. We observe that the most competitive approaches are $ReCal_{gpu+}$ and Utopia. Utopia is often faster than $ReCal_{gpu+}$ in the context of the programs for which the overall running time is less than 1 second, but $ReCal_{gpu+}$ outperforms Utopia in most of the cases that demand high solving effort.

We further investigated the type of constraints for which Utopia outperforms $ReCal_{gpu+}$. A representative case is the one of constraints that require many iterations for the *Canonicalisation* algorithm to converge. For example, for program AVL, Utopia takes 9.14 seconds to solve all the constraints, while $ReCal_{gpu+}$ takes 13.6 seconds, the size of constraints from AVL is up to 7 clauses with 28 variables, but the *Canonicalisation* algorithm takes up to 9 iterations (with an average of 4 iterations). In many other cases, the *Canonicalisation* algorithm converges quickly in 3 or less than 3 iterations. This indicates the complementarity of $ReCal_{gpu+}$ and Utopia.

6.3.3 Effectiveness of Canonicalisation

In this section, we report the results of a set of experiments designed to evaluate the impact of the *Canonicalisation* step on the effectiveness of *ReCal*, and provide data on how often the algorithm $Canonicalisation_{par}$ converges to computing the complete canonical form of the constraints in our benchmark.

We evaluate the impact of algorithm *Canonicalisation* on the effectiveness of *ReCal*, by measuring the reuse-rate of *ReCal* both with and without activating the *Canonicalisation* step across our experiments. Without the *Canonicalisation* step, *ReCal* achieves results similar to the Green approach, that is, it can identify the equivalent constraints that turn into equal formulas after normalising and simplifying their clauses, but cannot

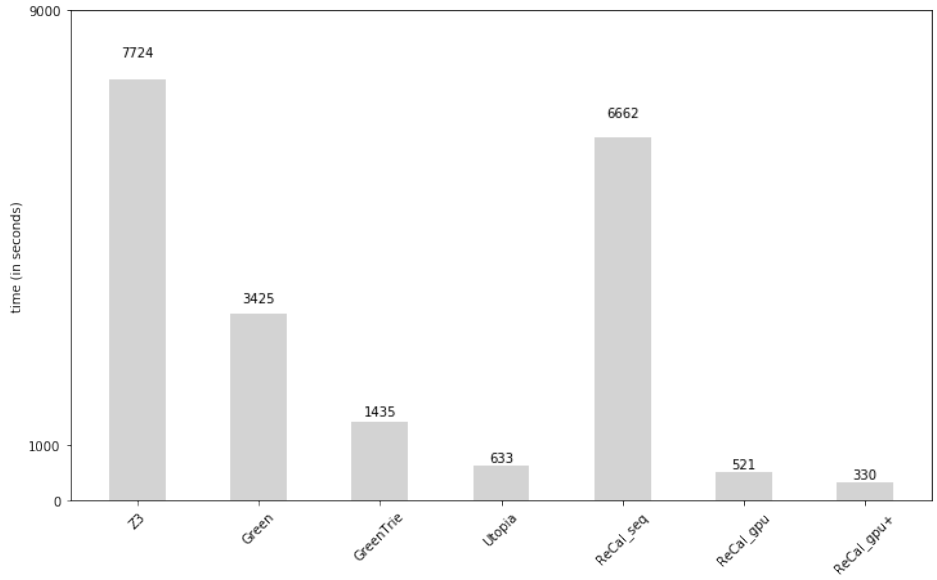


Figure 6.2. Time to solve all constraints with the different proof-reusing approaches

identify the equivalent constraints in which the corresponding clauses and the corresponding terms are listed in different orders. Indeed, for the constraints of the programs considered our experiments, the reuse-rates of *ReCal* with and without the *Canonicalisation* step correspond to the reuse-rates of $ReCal_{gpu}$ and Green that we already summarised in Figure 6.1. These data confirm the significant impact of the *Canonicalisation* step that allows $ReCal_{gpu}$ to achieve a median reuse-rate of 98%, while Green achieves a median reuse-rate of only 37%.

To further strengthen the evidence of the impact of the *Canonicalisation* step, we investigate the effectiveness of *ReCal* across the constraints of different programs, both with and without the *Canonicalisation* step, considering both the cases of inter- and intra-program constraints. We experimented *ReCal* on inter-program constraints as follows: For each pair of programs, we use one program to populate the repository and the other program to measure the reuse-rate with respect to these latter constraints only. Figure 6.5 illustrates the relative increase in the reuse-rate measured for each pair of programs when we repeated the related experiment either with and without the *Canonicalisation* step. The figure shows only the programs that share at least a constraint with at least another program, and marks with colours of increasing intensity the program pairs depending on the increment of reuse rate (from 0 to 40%, with a grey mark indicating no reuse). Figure 6.5 confirms the positive impact of *Canonicalisation* that improves the inter-program effectiveness of *ReCal* up to 40% extra reuse.

Algorithm $Canonicalisation_{par}$ implements the first three phases of the definition of *Canonicalisation* given in Chapter 4, dismissing the fourth phase that has exponential complexity. Thus, $Canonicalisation_{par}$ trades precision for efficiency, and may not

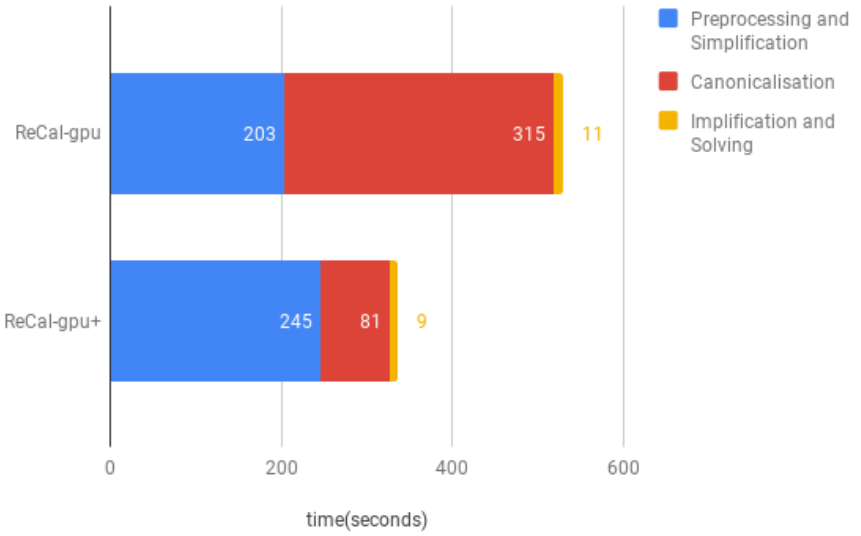


Figure 6.3. Execution time of each step of $ReCal_{gpu}$ and $ReCal_{gpu+}$

converge for constraints that require the fourth phase, a rare case in practice. We evaluate the impact of the fourth step to identify reusable proofs as the ration between the constraints for which $Canonicalisation_{par}$ converges to the complete canonical, and the constraints for which $Canonicalisation_{par}$ reaches a fixed-point the third phase without converging. In Table 6.4, we list the programs considered in our experiments (column program), the total number of constraints for each program (column total), and the number of these constraints for which $Canonicalisation_{par}$ does not converge (column unconverge). These data indeed confirm that the non-convergence cases are a very limited portion of the total in each program, and overall 1.23% of the constraints in our benchmark.

The constraints that do not converge to the complete canonical form may nonetheless succeed in revealing equivalent constraints, because the incomplete canonical forms are often very close (or sometimes even equal) to the complete ones. Thus, to better quantify the impact of the non-convergence cases of algorithm $Canonicalisation_{par}$, we measured (column hit-over-unconverge) for how many of these constraints *ReCal* was able to identify a matching equivalent constraint, regardless of computing the incomplete canonical form. The results indicate that *ReCal* can find equivalent constraints for 97.94% of the constraints with incomplete canonical form.

The few cache-miss of the constraints with incomplete canonical form are constraint for which either there is at leat one equivalent constraint, and *ReCal* fails to find the match due to the incomplete canonical form, or there exist no equivalent constraints of the target constraint in the repository. We measure these effects by executing the the fourth phase of the algorithm on the small set of cache-miss constraints with incom-

plete canonical form. Column miss-over-unconverge of Table 6.4 reports the number of cache-miss constraints that would have been identified if we had computed a complete canonical form (including the fourth phase). The results indicate that cache-misses due to un-convergence of algorithm $Canonicalisation_{par}$ are indeed rare, occurring for only 0.003% of the (about 800,000) constraints considered in our experiments.

In summary, the data reported in this section confirm that the *Canonicalisation* step of *ReCal* is effective and precise for identifying equivalent constraints.

program	total	unconverge	hit-over-unconverge	miss-over-unconverge
afs	203	0	0	0
avl	11161	3034 (27.18%)	2867 (94.50%)	26(0.23%)
ball	210	0	0	0
block	505	2 (0.4%)	1(50%)	1(50%)
cdaudio	55329	0	0	0
collision	6812	40 (0.59%)	38 (95%)	1 (0.2%)
dijkstra	85	0	0	0
diskperf	103505	0	0	0
division	1257	0	0	0
floppy	100006	0	0	0
grep	100126	0	0	0
kbfiltr	188	0	0	0
knapsack	7651	0	0	0
list	876	0	0	0
multiplication	25217	0	0	0
new-tax	55	0	0	0
old-tax	43	0	0	0
reverseword	38104	0	0	0
swapwords	173	0	0	0
tcas	13476	54 (0.4%)	52 (96.3%)	1 (0.01%)
treemap	332950	6712 (2.02%)	6681 (96.3%)	4 (0.00%)
wbs	239	0	0	0
total	**798171**	**9842 (1.23%)**	**9639 (97.94%)**	**33 (0.003%)**

Table 6.4. Unconvergence of Canonicalization algorithm

6.4 Threads to Validity

The main threat to the validity of our results concerns the generality of our proof-reusing approach in symbolic analysis. The specificity of the considered subject pro-

grams and program analysis tools can impact the construct validity of the experiments. We selected the subject programs from popular online software repositories, including repositories used for other scientific testing and analysis experiments, covering different kind of programs. In this dissertation, we focus on constraints produced with symbolic execution that represents a widely studied analysis technique that integrates with SMT solvers. To test *ReCal* beyond symbolic execution constraints, we executed a preliminary experiment on constraints generated by the invariant synthesiser GK-tail [MPS17] for a set of programs of the Guava library. The constraints from GK-tail are in different logic and not in conjunctive form, and thus we applied transformations to turn them into disjunctive normal form, and then applied our approach to each of the conjunctive sub-constraints. Our proof reusing approach achieves very higher score of reuse-rates on these sub-constraints. Due to the different logic of the constraints of GK-tail, these preliminary results are not included in this dissertation, but they suggest the generality of the proof reusing approaches for more analysis tools.

In the context of our approach, parallelisation is the key element to improve the efficiency of proof reusing. Conceptually, it would be possible to exploit parallel algorithms in other proof reusing approaches by defining proper data structures to represent constraints, and normalisation algorithms that allow parallel decomposition of the computation. However, none of these requirements is directly satisfiable in the existing approaches, which is why we could not compare our approach with the parallel version of the other approaches.

As mentioned in Chapter 2, there exist proposals of different path selection strategies in symbolic execution, which would probably affect proof reusing. Thus, it would be an interesting future work to further investigate the improvement of proof reusing under different path selection strategies.

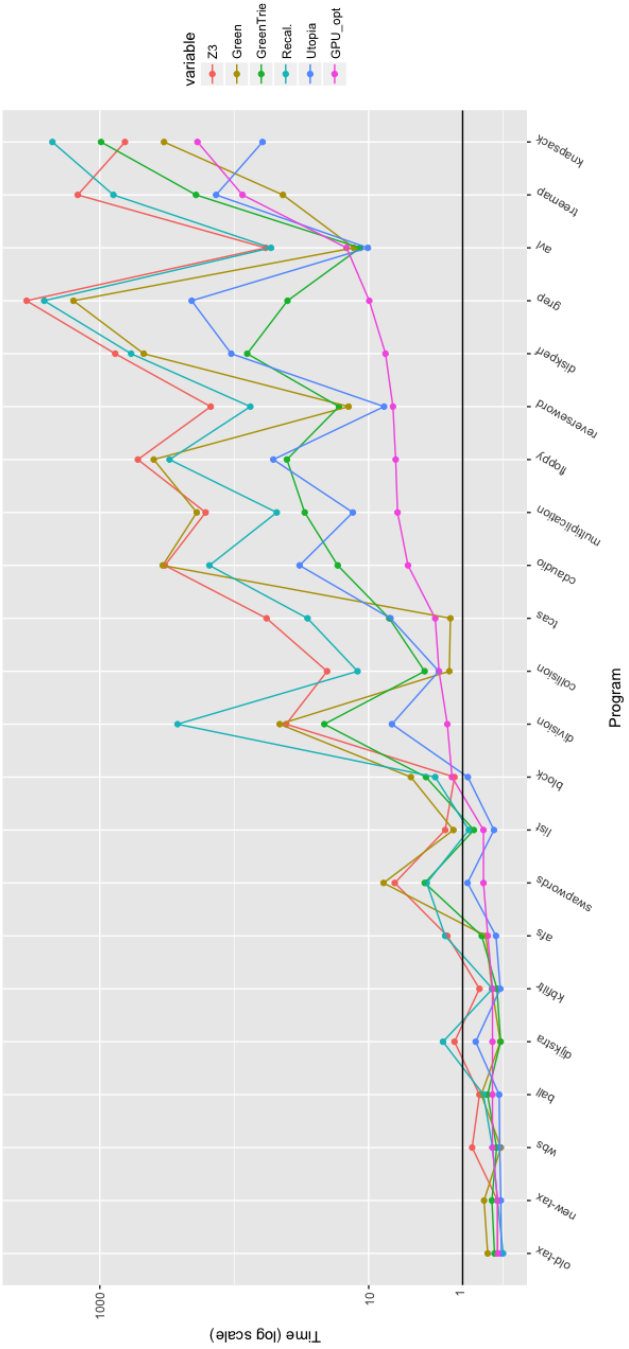


Figure 6.4. Execution time (log scale) to solve the constraints of each program with the different proof-reusing approaches

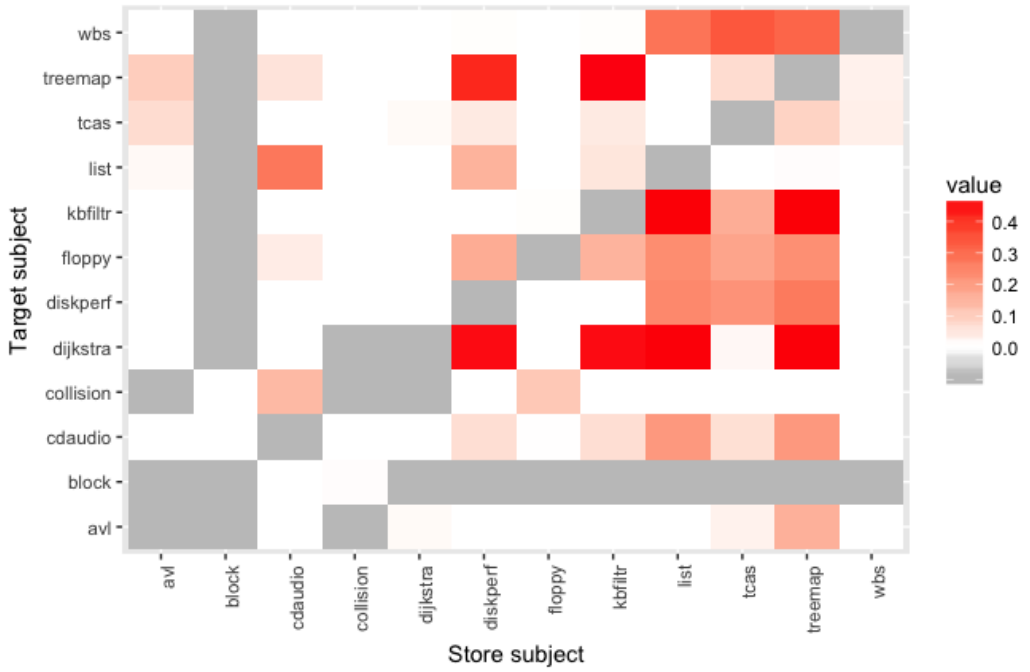


Figure 6.5. Incremental reuse-rate for Inter-program reuse-rates with Canonicalisation

Chapter 7

Conclusion

Symbolic analysis is a popular program analysis techniques. Symbolic analysis describes program properties with symbolic expressions, and relies on constraint solvers to validate the symbolic expressions and verify the properties of interest. Constraint solving accounts for a large part of the overall execution time, and still represents a main bottleneck towards the applicability and scalability of symbolic analysis techniques, in spite of the advanced development of solvers in the recent decades.

In this dissertation, we explore the feasibility of proof reusing to mitigate the impact of constraint solvers during the process of symbolic analysis. Previous studies show that constraints re-occur when analysing a program, and that it is possible to identify and reuse recurrent constraints, thus reducing the calls to constraint solvers. Previous work on reusing recurrent constraints achieves a significant, but still limited reuse due to some intrinsic limitations of the approaches.

In this thesis, we propose *ReCal*, *REusing-Constraint-proofs-in-symbolic-AnaLysis*, a novel approach for reusing proofs by exploiting the relations among constraints. *ReCal* improves over the state-of-the-art proof reusing techniques by both re-defining the concept of equivalence of constraint of the QF_LIA logic, and proposing a parallel *Canonicalisation* algorithm to effective identify reusable proofs during symbolic analysis. Proof reusing approaches propose some simplifications and normalisations to quickly search for equivalent constraints. Simplifications miss many reuse opportunities, while current normalisation approaches results in significant overhead, thus reducing the effectiveness of proof reuse, especially for constraints that consist of many clauses and variables.

With *ReCal*, we propose a new canonical form that identifies a larger set of reusable proofs than previous approaches, and a parallel framework *ReCal-gpu*, which significantly speeds up the constraint solving processes. We report the results of a set of experiments on a large set of constraints generated from a variety of real-world programs by symbolic execution tools. The empirical results indicate that *ReCal* can both effectively identify reusable proofs and reuse them to solve more than 90% of constraints during symbolic execution and significantly reduce to constraint solving time

than calling solver directly, largely improving over state-of-the-art approaches.

The current *ReCal* approach focuses on conjunctive QF_LIA constraints, and we mainly investigated the application of reusing constraint proofs during symbolic execution. Thus, an interesting future work of this PhD dissertation would be to extend the reusing approach to other logics and theories, such as, non-linear logics, logics based on real number arithmetics, and logics that include operators other than conjunction. Moreover, given that this PhD work has demonstrated the effectiveness of improving symbolic execution with proofs reusing capabilities, another interesting research direction would be to investigate the improvement of proof reusing in other static analysis techniques with heavy constraint solving requirements.

Contributions

The main contribution of this thesis is a parallel proof reusing approach that exploits relations of constraints to mitigate the bottleneck of constraint solving in symbolic analysis. In details, this thesis contributes to the state of the art in reusing proofs during symbolic analysis as follows:

Canonical form: We define a new concept of equivalence of constraints, and propose a *Canonicalisation* algorithm to generate the canonical form for efficiently searching for equivalent constraints with reusable proofs. The canonical form fulfils two purposes: deciding constraint equivalence, and serving as indices for fast look-up over a large repository of constraints.

Logical implication rules: We extend the current existing normalisation tactics of constraints by introducing logical simplification rules to eliminate redundant clauses and check conflicting clauses, thus further simplifying constraints by self-implication. We propose proof reusing by logical implication, which goes beyond reusing proofs from equivalent constraints. By introducing logical implication rules, *ReCal* expand the scope of reusable proofs.

Parallel GPU computing for proof reusing: We explore the feasibility of applying parallel computing in proof reusing approaches, and propose the first parallel GPU deployment of *ReCal*.

Prototype implementation: We implement both the sequential and the parallel framework of the *ReCal* approach. We implement the parallel framework on CUDA platform, which invokes GPU processors for general purpose computation requiring GPU computing resources, and use the prototypes to experimentally evaluate the effectiveness and efficiency of different versions of *ReCal*.

Evaluation of the technique: We evaluate the *ReCal* approaches on eight hundred thousand constraints generated by symbolic analysis on real-world programs,

and compare the performance of *ReCal* to cutting-edge solver and state-of-the-art approaches. Our results indicates that, the *ReCal* approach is very effective and efficient in identifying reusing proofs, it identifies more reusable proofs in most of cases than the existing approaches, and outperforms all of them in term of performance. *ReCal* speeds up the process of constraint solving by an order of magnitude, thus confirming the feasibility of reusing proofs to improve the performance of symbolic analysis.

Appendices

Appendix A

Proofs and analysis of the Canonicalisation algorithm

This chapter demonstrates that *Canonicalisation* converges to the canonical form stated in Definition 4.1.

A.1 Termination of Canonicalisation

Definition A.1. *Let $LC_\wedge$ be the set of all conjunctive linear formulas.*

Definition A.2. *Let $permute(C) \in 2^{LC_\wedge}$ denote the set of formulas that can be obtained by permuting the clauses and the terms of a formula $C \in LC_\wedge$.*

Definition A.3. *Let $\equiv$ be the equivalence relation over $LC_\wedge$ such that, $\forall C_1, C_2 \in LC_\wedge$ $C_1 \equiv C_2 \iff C_1 \in permute(C_2)$.*

It is easy to verify that the $\equiv$ relation is indeed reflexive, symmetric, and transitive.

Definition A.4. *Let* Canonicalisation $: LC_\wedge \to LC_\wedge$ *be the algorithm defined in Algorithm 1 in Chapter 4.*

To prove the correctness of the *Canonicalisation* we first prove two simple lemmas.

Lemma A.5. *The algorithm* Canonicalisation *terminates for any input formula.*

Proof. To prove that the *Canonicalisation* algorithm terminates for any given input formula it is sufficient to prove that each phase terminates independently from its input (i.e., a formula for the preliminary phase, a matrix for all the other phases).

Convergence of the preliminary phase: The transformation of any given conjunctive formula into a matrix is a deterministic always terminating process. Each inequality is transformed in at most 2 rows of the resulting matrix, since all the iterations required to achieve this transformations can be executed with bounded for loops this preliminary phase is always terminating.

Convergence of phase 1: phase 1 sorts the rows of the input matrix once (according to the comparisons and free coefficients contained in the last two columns). The sorting of a set of elements is an always terminating process, thus this phase is always terminating.

Convergence of phase 2: phase 2 sorts the rows and columns of the matrix once (according to the terms contained in the terms sub-matrix). The same considerations on the previous phase hold, thus this phase is always terminating.

Convergence of phase 3: phase 3 iterates on the matrix reordering the rows and columns that have not being assigned a stable position yet according to the elements fixed during the previous phases. This phases is repeated until a fixpoint is reached and is thus potentially non-determinating. Notice that whenever a row or a column is moved during any iteration it will never move again on the next iterations; in fact, a necessary condition to move a given row (or column) is that the fixed elements on that row (or column) are strictly smaller or larger than others in another row (or column) thus enforcing a stable position for that row (or column). Because of this at any iteration there are only two possibilities:

1. at least one row or column (so far not in a stable position) is assigned a stable position. But then the total number of rows and columns yet to be assigned stable positions diminishes. Since this number is finite (roughly accounting to the sum of the number of rows and columns of the matrix) it will eventually turn to zero terminating this phase (since the matrix converged, i.e., all rows and columns are in stable positions).
2. No row or column is assigned a stable position. But then no row nor column moved with respect to the previous phase thus producing a fix point and terminating the algorithm.

In other words, at any iteration either a fix point is reached or new rows and/or columns are assigned a stable position. Since there are a finite number of rows and columns, after a finite number of steps all the rows and columns of the matrix are assigned a stable position and the phase terminates (since the matrix converged).

Convergence of phase 4: In the last phase the sub-matrix composed of the elements that have not being assigned stable positions in the past is considered. Since the original matrix is finite, this is a finite rectangular matrix. During this phase all possible permutations of the rows and columns of this sub-matrix are considered. If the extracted sub-matrix has n rows and m columns this accounts to consider $n! \times m!$ permutations. Although potentially huge this number is finite. Phase 4 considers all this permutations one by one, applies them to the extracted sub-matrix and identifies the one that produces the maximum matrix (according to

the lexicographical order of the matrices flattened as lists of numbers). The maximal permutation is then mapped back and applied to the original matrix once. Since the algorithm checks a finite (even if potentially huge) number of permutations and since the comparison of a pairs of permuted sub-matrices can be easily performed in linear time this phase always terminates.

□

A.2 Correctness of Canonicalisation

In this section, we prove the correctness of the *Canonicalisation* algorithm. First we proof that the canonical form computed by *Canonicalisation* is a permutation of the target constraint.

Lemma A.6. $\forall C \in LC_{\wedge}$, Canonicalisation$(C) \in permute(C)$.
That is, the algorithm Canonicalisation *computes a permutation of the input formula.*

Proof. The proof directly follows from the observation that each step of the algorithm produces a permutation of the matrix that represents the input constraint. Thus the lemma is trivially true by the construction of the algorithm. □

Now we prove the correctness of *Canonicalisation*.

Theorem A.7. *For any given pair of formulas $C_1, C_2 \in LC_{\wedge}$ it holds that $C_1 \equiv C_2 \iff$* Canonicalisation$(C_1) =$ Canonicalisation(C_2).

Proof. We first prove the right-to-left implication of the theorem. That is, $\forall C_1, C_2 \in LC_{\wedge}$ $Canonicalisation(C_1) = Canonicalisation(C_2) \implies C_1 \equiv C_2$.

This can be proved by combining Lemma A.6 (*Canonicalisation* permutes the input constraints) and the definition of the equivalence relation $\equiv$ (Definition A.3). In fact:

- $Canonicalisation(C_1) \in permute(C_1) \implies C_1 \equiv Canonicalisation(C_1)$;
- $Canonicalisation(C_2) \in permute(C_2) \implies C_2 \equiv Canonicalisation(C_2)$;
- $C_1 \equiv Canonicalisation(C_1) = Canonicalisation(C_2) \equiv C_2$;

and thus $C_1 \equiv C_2$;

We now prove the left-to-right implication of the theorem. That is, $\forall C_1, C_2 \in LC_{\wedge}$ $C_1 \equiv C_2 \implies Canonicalisation(C_1) = Canonicalisation(C_2)$.

First, since C_1 and C_2 are equivalent, that is $C_1 \in permute(C_2)$, there exists a bijective correspondence between the clauses and the variables of the two formulas that are in the same positions after a permutation that makes C_2 equal to C_1. Let us consider this bijection and denote it as $\sim correspondence$.

Next, we demonstrate that the four phases of *Canonicalisation* yield the same decisions on the relative order between any ∼corresponding pairs of clauses and variables of C_1 and C_2, that is, the ∼corresponding pairs of clauses (variables) have the same relative order or unknown relative order when considered in the scope of either formula.

Algorithm *Canonicalisation* – phase 1 The decisions on the relative order of the clauses depend on the comparison operator and the constant term in the clauses; notice that ∼corresponding clauses have the same comparison operator and the same constant term.

Thus, if two given clauses in C_1 have different comparison operators or different constant terms they will be reordered by this phase, and their ∼corresponding clauses in C_2 will be reordered in the same way. Otherwise, both the clauses in C_1 and their ∼corresponding clauses in C_2 will not be assigned any order.

Algorithm *Canonicalisation* – phase 2 The decisions on the relative order of the clauses (variables) depend on the set of coefficients associated with each clause (variable); notice that ∼corresponding clauses (variables) share the same sets of coefficients.

Thus, if two clauses (variables) of C_1 result in either unknown order because they are associated with identical sets of coefficients or in a specified order because the sets of coefficients differ, then the two ∼corresponding clauses in C_2 will result in exactly the same unknown or specified order.

Algorithm *Canonicalisation* – phase 3 All iterations of the third phase take order decisions based on the clauses and the variables with stable position after the first two phases and the previous iterations. A clause or variable has a stable position if its relative order is strictly specified with respect to all other clauses or variables, that is, its position will not change any further though the next steps of the algorithm. We develop the proof by showing that the following statements hold:

1. At the beginning of phase 3, the clauses and the variables with stable position in C_1 ∼correspond to clauses and variables with stable position in C_2.

2. Any iteration of phase 3 maintains that, after the iteration, the clauses and the variables with stable position in C_1 ∼correspond to clauses and variables with stable position in C_2.

3. Any iteration of phase 3 yields the same decisions on the relative order between any ∼corresponding pairs of clauses and variables of C_1 and C_2.

The first statement provides the base case for an inductive proof of the second statement that, in turn, is used in the proof of the third statement. The third statement guarantees that, regardless of the number of iterations, phase 3 will yield always the same decisions on the relative order between any ∼corresponding pairs of clauses and variables of C_1 and C_2.

Base of the induction: The first two phases of the algorithm guarantee the same relative order of any ∼corresponding clauses and variables in both C_1 and C_2. As a consequence, if after the first two phases a clause or a variable has a stable position in C_1, the ∼corresponding clause or variable of C_2 must have stable position too. This proves statement 1.

Inductive step: Let us consider that all clauses and variables with stable position in C_1 ∼correspond to clauses and variables with stable position in C_2 at the beginning of an iteration of phase 3. Next, we analyse the effect of the iteration on the order of the clauses of C_1 and C_2. Let c_{1a} be a clause in C_1 and let c_{2a} be its ∼corresponding clause in C_2, and let $\langle c^1_{1a}, c^2_{1a}, ...\rangle$ and $\langle c^1_{2a}, c^2_{2a}, ...\rangle$ be the vectors of the coefficients of the variables with stable position in these clauses. We observe that, since the variables with stable positions are in $\sim\ correspondence$ between C_1 and C_2, the coefficients $c^1_{1a}, c^2_{1a}, ...$ and in $c^1_{2a}, c^2_{2a}, ...$ stand at the cross between the ∼corresponding clauses c_{1a} and c_{2a} and the ∼corresponding variables in the two constraints. This implies that c^1_{1a} will be necessarily equal to c^1_{2a}, c^2_{1a} will be necessarily equal to c^2_{2a}, and so forth. Thus, in the current iteration, the ∼corresponding clauses of C_1 and C_2 will be ordered based on identical vectors of coefficients and will result in the same relative order within the two constraints. All above considerations can be repeated with reference to the ordering of the variables in the constraints. This proves statement 2 by induction because the statement holds at the beginning of the first instruction (statement 1) and continues to hold after each iteration, and proves that statement 3 holds for any iteration of phase 3.

Algorithm *Canonicalisation* – phase 4 The fourth phase orders the clauses and the variables of C_1 and C_2 with yet unstable positions after the first three phases. The algorithm consists of enumerating the set of the possible permutations of these clauses and variables, and selecting the maximum element after ordering the set lexicographically. We develop the proof by observing that:

1. Enumerating the possible permutations of the clauses and the variables with yet unstable positions always produces a finite set of constraints, since the possible permutations of a finite set of clauses (variables) are a finite number.

2. Enumerating the possible permutations of the clauses and the variables with yet unstable positions out of C_1 and C_2 always results in exactly the same set of constraints. This descends from the assumption that C_1 and C_2 are $\equiv$-equivalent, and thus the transitivity of the relation guarantees that any permutation of C_1 is equivalent to C_2 and the vice versa.

3. Since C_1 and C_2 share the same set of possible permutations, the maximum element of the set correspond to the same constraint for both C_1 and C_2.

In the end, if the algorithm converges in any of the first three phases, then it guar-

antees that all clauses and variables have been ordered to stable relative positions with complete $\sim$ $correspondence$ between C_1 and C_2, and thus, since the crosses between $\sim$corresponding clauses and variables correspond to equal coefficients in C_1 and C_2, we have that $\mathit{Canonicalisation}(C_1) = \mathit{Canonicalisation}(C_2)$. Otherwise the algorithm converges in the fourth phase that always yields exactly the same constraint for both C_1 and C_2. □

Appendix B

Proofs of the parallel canonicalisation algorithms

This chapter demonstrates the formal proof of the relation of the $Canonicalisation_{par}$ algorithm described in Algorithm 3 in Chapter 5 regarding to the sequential version *Canonicalisation* presented in Algorithm 1 in Chapter 4, refered as *sequential-canonicalise*. Specifically, we prove that algorithm $Canonicalisation_{par}$ subsumes *sequential-canonicalise* up to phase 3.

Theorem B.1. ***Algorithm*** $Canonicalisation_{par}$ ***subsumes sequential-canonicalise up to phase 3****: Given a constraint C, by applying* sequential-canonicalise *up to phase 3, if the order a row (or column without losing the generality) k is stable, then by applying the algorithm* $Canonicalisation_{par}$*, the hashcode of row k is distinct from other rows.*

To prove this theorem, we separately discuss the completeness of the algorithm $Canonicalisation_{par}$ in comparison with to *sequential-canonicalise* phase 1-2 and phase 3.

Lemma B.2. ***Phase 1-2 on position-free values****: Given a constraint C, by applying ReCal* Canonicalisation *phase 1-2, if the order a row k is stable, then by applying algorithm* $Canonicalisation_{par}$*, the hashcode generated after the* $1st$ *iteration of row k is distinct from other rows.*

Proof. ReCal *Canonicalisation* phase 1 is to reorder the matrix by constant terms and comparison operators, and the phase 2 is to reorder by the position-free coefficient values of rows or columns. If the order row k is decidable after phase 1-2, then row k consist of a different set of coefficient values than other rows. Applying algorithm $Canonicalisation_{par}$, in the $1st$ iteration, the hashcode $hrow_k^1$ computed in line 21—22 (now $hcols$ are all initialised as 0) is to hash the set of $(coefficient, 0)$ pairs. Since row k consists of different coefficient values than other rows, its hashcode $hrow[k]^1$ is distinct.

□

Lemma B.3. ***Phase 3 of iteration****: Given a constraint C, by applying ReCal* Canonicalisation *phase 3, if the order a row k is stable, then by applying algorithm* $Canonicalisation_{par}$*, the hashcode generated of row k is distinct from other rows.*

We provide the proof of the lemma by induction.

Proposition B.4. ***Base of the induction****: Lemma B.3 holds after the* 1*nd iteration of* Canonicalisation *phase 3. I.e., given a constraint C, by applying ReCal* Canonicalisation *phase* 3 *iteration* 1*, if the order a row k is stable, then by applying algorithm 3, the hashcode after iteration* 2 *generated of row k is distinct from other rows.*

Proof. Canonicalisation phase 3 decides the new order of rows based on the lexicographic order of the partial stable values. If row k becomes stable after the phase 3 iteration 1, let $M_{ki_1}, M_{ki_2}, .., M_{ki_t}$ be the sequence of partial stable values in row k, then there is no such another row j consists of the same sequence in the same position (otherwise the order of row k is undecided). By definition, the order of the corresponding columns $i_1, i_2, ..i_t$ are decided after ReCal phase 2, and based on lemma B.2 the hashcode of columns $i_1, i_2, ..i_t$ computed by the 1st iteration of algorithm $Canonicalisation_{par}$ are distinct. Thus, applying algorithm $Canonicalisation_{par}$ iteration 2, the set of pairs to compute $hrows[k]^2$ contains unique combination of $(M_{ki_1}, hcols[i_1]^1)$,$(M_{ki_2}, hcols[i_2]^1)$, ...,$(M_{ki_t}, hcols[i_t]^1)$, then $hrows[k]^2$ is distinct after algorithm $Canonicalisation_{par}$ iteration 2. □

Proposition B.5. ***Inductive step****: if Lemma B.3 holds after the n-th iteration, the Lemma B.3 also holds after* $n+1$*-th iteration.*

The proof of the inductive step is similar to the base step.

Proof. Suppose after *Canonicalisation* phase 3, iteration $n-1$-th , if the order of a row (or column) j is stable, applying n iteration of algorithm $Canonicalisation_{par}$ on the same constraint matrix, the hashcode of corresponding rows are distinct. Assume the order of row k become stable in *Canonicalisation* phase 3, iteration n-th, let $M_{ki_1}, M_{ki_2}, .., M_{ki_t}$ be the sequence of partial stable values in row k, then there is no such another row j consist the same sequence in the same position (otherwise the order of row k is undecided). By definition, the order of the corresponding columns $i_1, i_2, ..i_t$ are decided after *Canonicalisation* phase 3, iteration $n-1$-th, and the hashcode of columns $i_1, i_2, ..i_t$ computed by the n-th iteration of algorithm $Canonicalisation_{par}$ are distinct. Thus, applying algorithm $Canonicalisation_{par}$ iteration $n+1$, the set of pairs to compute $hrows[k]^{n+1s}$ contains unique combination of $(M_{ki_1}, hcols[i_1]^n)$,$(M_{ki_2}, hcols[i_2]^n)$, ...,$(M_{ki_t}, hcols[i_t]^n)$, then $hrows[k]^{n+1}$ is distinct after algorithm $Canonicalisation_{par}$ iteration $n+1$. □

So far we finish the proof of Theorem B.1. In practical setting, we apply *sequential-canonicalise* up to phase 3 to generate canonical form for proof reusing (the enumeration phase 4 is very costly). Theorem B.1 suggests that the canonical forms generated by

$Canonicalisation_{par}$ have as effective as the ones generated by *sequential-canonicalise* in finding equivalent constraints.

Bibliography

[ABC+15] Andrea Aquino, Francesco A. Bianchi, Meixian Chen, Giovanni Denaro, and Mauro Pezzè. Reusing constraint proofs in program analysis. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '15, pages 305–315. ACM, 2015.

[ADP17] Andrea Aquino, Giovanni Denaro, and Mauro Pezzè. Heuristically matching solution spaces of arithmetic formulas to efficiently reuse solutions. In *Proceedings of the International Conference on Software Engineering*, ICSE '17, pages 427–437. IEEE Computer Society, 2017.

[BCD+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[BDH03] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The google cluster architecture. *Micro, IEEE*, 23(2):22–28, 2003.

[BDHJ14] Anton Belov, Daniel Diepold, Marijn JH Heule, and Matti Järvisalo. Proceedings of sat competition 2014. 2014.

[BDP13] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '13, pages 411–421. ACM, 2013.

[BDP16] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. JBSE: A symbolic executor for java programs with complex heap inputs. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '16, pages 1018–1022. ACM, 2016.

[BE13] Suhabe Bugrara and Dawson Engler. Redundant state detection for dynamic symbolic execution. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference*, pages 199–212. USENIX Association, 2013.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[BHJ17a] Tomáš Balyo, Marijn JH Heule, and Matti Järvisalo. Sat competition 2016: Recent developments. In *Association for the Advancement of Artificial Intelligence*, pages 5061–5063, 2017.

[BHJ+17b] Tomáš Balyo, Marijn JH Heule, Matti Järvisalo, et al. Proceedings of sat competition 2017. 2017.

[BHvM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[BS08] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the International Conference on Automated Software Engineering*, pages 443–446. IEEE Computer Society, 2008.

[CARB12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI '08, pages 209–224. USENIX Association, 2008.

[CDW16] David R Cok, David Déharbe, and Tjark Weber. The 2014 smt competition. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:207–242, 2016.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. CCS '06, pages 322–335. ACM, 2006.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 93–107. Springer, 2013.

[CKC12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The s2e platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)*, 30(1):2, 2012.

[Cla76] Lori A. Clarke. A program testing system. In *Proceedings of the 1976 Annual Conference*, ACM '76, pages 488–491. ACM, 1976.

[Com79] Douglas Comer. Ubiquitous b-tree. *ACM Computer Surveys*, 11(2):121–137, 1979.

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, February 2013.

[CTS08] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *Proceedings of the International Conference on Software Engineering*, ICSE '08, pages 281–290. ACM, 2008.

[DA14] Peter Dinges and Gul Agha. Solving complex path conditions through heuristic search on induced polytopes. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 425–436. ACM, 2014.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS/ETAPS '08, pages 337–340. Springer, 2008.

[Dut14] Bruno Dutertre. Yices 2.2. In *Proceedings of the International Conference on Computer Aided Verification*, CAV '2014, pages 737–744. Springer, 2014.

[EO11] Ikpeme Erete and Alessandro Orso. Optimizing constraint solving to better support symbolic execution. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '11, pages 310–315. IEEE Computer Society, 2011.

[GAC12] Sicun Gao, Jeremy Avigad, and Edmund M Clarke. Delta-decidability over the reals. In *Logic in Computer Science (LICS), 2012 27th Annual IEEE Symposium on*, pages 305–314. IEEE, 2012.

[GLM12] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *ACM Queue*, 10(1):20–27, 2012.

[JGY15] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '15, pages 177–187. ACM, 2015.

[Kin76] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KST12] Johannes Kobler, Uwe Schöning, and Jacobo Torán. *The graph isomorphism problem: its structural complexity*. Springer Science & Business Media, 2012.

[LLQ+16] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li. Symbolic execution of complex program driven by machine learning based constraint solving. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 554–559. ACM, 2016.

[Mei14] Chen Meixian. Reusing constraint proofs for scalable program analysis. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 449–452. ACM, 2014.

[MHL+13] Antonio Morgado, Federico Heras, Mark Liffiton, Jordi Planes, and Joao Marques-Silva. Iterative and core-guided maxsat solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013.

[MPS17] Leonardo Mariani, Mauro Pezzè, and Mauro Santoro. Gk-tail+ an efficient approach to learn software models. *IEEE Transactions on Software Engineering*, 43(8):715–738, 2017.

[PC13] Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In *Proceedings of the International Conference on Computer Aided Verification*, CAV '13, pages 53–68. Springer, 2013.

[PY07] Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.

[PYRK11] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '11, pages 504–515. ACM, 2011.

[Red21] Redis. Redis NoSQL database. `http://redis.io`, Accessed: 2017-12-21.

[RVBW06] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[SBdP11] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S Păsăreanu. Coral: solving complex constraints for symbolic pathfinder. In *NASA Formal Methods*, pages 359–374. Springer, 2011.

[SK10] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[TdH08] Nikolai Tillmann and Jonathan de Halleux. Pex: White box test generation for .NET. In *Proceedings of the International Conference on Tests and Proofs*, TAP '08, pages 134–153. Springer, 2008.

[TSBB17] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. Search-driven string constraint solving for vulnerability detection. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 198–208. IEEE, 2017.

[VGD12] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '12, pages 1–11. ACM, 2012.

[YDS09] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on World wide web*, pages 401–410. ACM, 2009.

[YPK12] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. Memoized symbolic execution. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '12, pages 144–154, 2012.