Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

The present work was submitted to the LuFG Theory of Hybrid Systems

MASTER OF SCIENCE THESIS

PRE- AND INPROCESSING IN SMT SOLVING FOR REAL ALGEBRA

Laura-Sophie Kirchner

Communicated by
Prof. Dr. Erika Ábrahám

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Jürgen Giesl

Additional Advisor:
Valentin Promies

Aachen, 28.09.2023

Abstract

Satisfiability Modulo Theories (SMT) solvers can be used to decide the satisfiability of some First-Order (FO) logic formulas with respect to some FO-theory. In particular, SMT solvers can be used to solve formulas of the theory of quantifier-free non-linear real arithmetic (QFNRA).
In 2020 Brown et al. proposed a partial theory solver aiming to reduce the complexity of the given QFNRA constraints in DNF. This partial solver could be used as preprocessing for DNF formulas. The goal of this thesis is to modify the introduced algorithms to work as inprocessing using incrementality and backtracking.
After implementing the simplification algorithms in their original and modified form in SMT-RAT, we show that they highly improve the performance of the solver. Our modifications do not only provide a significantly higher number of solved instances, but also a runtime improvement compared to its non-incremental equivalent.

Contents

Chapter 1

Introduction

1.1 Motivation

Satisfiability Modulo Theories (SMT) solvers are tools to decide the Boolean satisfiability (SAT) of some First-Order (FO) logic formulas over some FO theories [DMB08]. SMT solvers can for example be used for symbolic software model checking and program verification [BdM14].

Interesting theories are for instance Linear Arithmetic and Non-Linear Arithmetic over the reals or integers [Seb07]. Sets of quantifier-free linear real arithmetic (QFLRA) formulas are decidable in polynomial time, while the complexity of sets of quantifier-free non-linear real arithmetic (QFNRA) formulas depends on the complexity of its polynomials and can reach exponential complexity [Pro14]. Usually, SMT solvers are composed of a SAT solver and theory solvers. Additionally, one can initially simplify the constraints before the input is processed. This is called preprocessing. If this step occurs between the SAT solver and the theory solver, it is called inprocessing. Currently, there is not much literature on pre- or inprocessing for QFNRA formulas, but since non-linear arithmetic is more complex to solve than linear arithmetic, a simplification of non-linear constraints could bring a performance advantage.

For that reason, Brown et al. [BVE20] describe a partial solver based on DNF formulas. They developed ways to derive a combination of simpler constraints from complex QFNRA constraints. Either the algorithms deduce unsatisfiability or they return some simpler constraints that are derived from more complex input constraints and can be used to ease the later solving process. Usually, SMT solvers work incrementally and with backtracking. This means that prior calculations are reused for new input constraints and the solver can return to a working state when constraints are removed.
Brown et al. do not use incrementality and backtracking in their stand-alone solver, which is the primary focus and contribution of this thesis.

1.2 Related Work

This thesis builds heavily on the papers by Brown et al. [Bro09], [BS10], [Bro12], [BVE20]. In these, primarily so-called Blackbox and Whitebox simplification algorithms were developed. Blackbox simplification focuses on the factor structure of the involved polynomials, treating each factor as a variable and thus considering the factors themselves as "blackboxes". Whitebox simplification on the other hand tries to derive information about the sign of each factor from its polynomial structure and prior knowledge about variable signs (e.g. $x + y < 0$ or $x + y \geq 0$). In [BVE20], Brown et al. have built a partial solver based on these simplification strategies. This article is the foundation for this thesis and is described in more detail in Chapter 3.

1.3 Research Questions

The aim of this thesis is divided into two tasks. The first step is to implement the simplification described in [BVE20] in the SMT-RAT solver [smtb]. The second task is to adapt this approach for incrementality and backtracking. For this, we use the incrementality of the theory solver, thus not replacing any constraints which were already given to the backend. We will analyze the impact of this adaption on the solver and investigate the question to which extent this inprocessing brings general advantages.

1.4 Outline

In Chapter 2 we will give a short overview over some preliminaries. This includes the introduction of lazy SMT solvers as well as quantifier-free non-linear real arithmetic. In Chapter 3 we will explain the simplification steps as described in [BVE20]. The adaption to incrementality and backtracking is given in Chapter 4. In Chapter 5 we evaluate the describes techniques, including successfulness in solving the test instances and the runtime. Finally, in Chapter 6 we will give a short summary, discussion as well as outlook on promising future work.

Chapter 2

Preliminaries

2.1 SMT Solvers

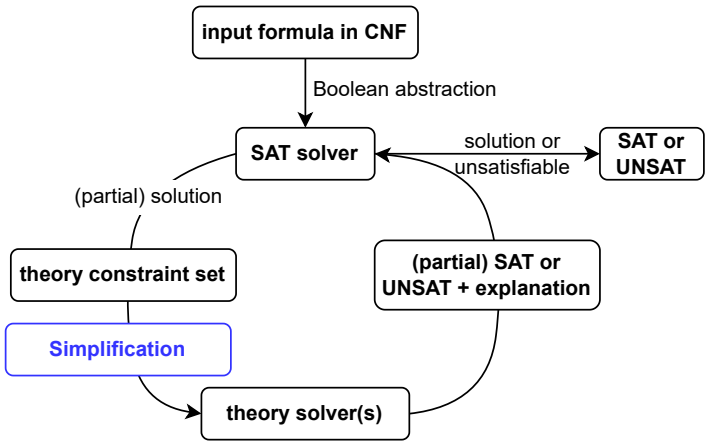


Figure 2.1: Mechanism of lazy SMT solving, as described in [ÁAB+16], with the addition of simplification (black), describing the inprocessing considered in this thesis.

Many solvers like SMT-RAT implement lazy SMT solving [Seb07]. These combine a SAT solver, handling the Boolean structure, with one or more theory solvers [Seb07].

Definition 2.1.1 (Conjunctive Normal Form [oM])**.** *A quantifier-free first-order logic formula over a theory is in conjunctive normal form (CNF) if it has the form*

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} C_{i,j},$$

where $n \in \mathbb{N}$, and for each $i \in \{1, \ldots, n\}$ we have $m_i \in \mathbb{N}$ and $C_{i,j}$ is either an atomic theory constraint or its negation.

The input formula in Conjunctive Normal Form (CNF) is mapped to its Boolean skeleton, which is then checked by the SAT solver for satisfiability. The Boolean skeleton

represents the Boolean Structure of the Formula, where each constraint is replaced by a fresh variable. Assume for simplicity that the input CNF contains no negation. If the Boolean skeleton is already unsatisfiable, this also applies to the original formula. Otherwise, the SAT solver finds a solution which needs to be checked for consistency in the respective theory. For this, the conjunction of the constraints that the SAT solver has assigned as true[1] is passed on to the theory solver. If the solution is not applicable to the theory, the theory solver returns a preferably small subset of the passed constraints that cannot be satisfied at the same time. This set is called the UNSAT core. If the SAT solver cannot find a new assignment considering the learned UNSAT cores, the formula is unsatisfiable (UNSAT). If the theory solver finds that the theory is consistent and the SAT solver does not need to add new constraints, the formula is satisfiable (SAT). This solver structure is depicted in Fig. 2.1. Note that we added the inprocessing (simplification) considered in Section 3.4.

The standard procedure for the SAT solver is the Davis-Putnam-Logemann-Loveland (DPLL) procedure [Seb07]. In this procedure, decisions about the assignment of literals are made and propagated. If a contradiction occurs, conflict resolution takes place, whereby a preferably small set of constraints is selected that explains the current conflict. The constraints of this set are learned as a new clause.

Example 2.1.1 (SMT Solving)**.** *Consider a QFNRA formula* $\varphi = a \leq 5 \wedge (a < 10 \vee b > 5) \wedge b \leq 0$.
The SAT solver considers the Boolean skeleton $\psi = x_1 \wedge (x_2 \vee x_3) \wedge x_4$. *A possible assignment by the DPLL algorithm is* $x_1 : true$, $x_2 : false$, $x_3 : true$, $x_4 : true$. *Thus, the theory solver receives the formula* $a \leq 5 \wedge b > 5 \wedge b \leq 0$, *which is unsatisfiable with the UNSAT core* $\{b > 5, b \leq 0\}$. *Thus* $\neg(x_3 \wedge x_4)$ *is added to* ψ *and the next iteration starts. A satisfiable assignment is* $x_1 : true, x_2 : true, x_3 : false, x_4 : true$ *with* $a = 5$ *and* $b = 0$.

A less lazy SMT solver has three important main properties [ÁAB+16]:
Incrementality: After checking the consistency of a set of constraints that result should be used to re-check the set after adding additional constraints.
Explanation: In case of unsatisfiability, the algorithm should return a set of contradictory constraints which should be as small as possible.
Backtracking: It should be possible to remove constraints in reverse chronological order without losing all progress.

These properties are helpful, because inconsistencies in the theory can be found earlier, without losing unnecessary time at the SAT solver. Reusing past calculations saves runtime.

The combination of multiple theory solvers has the advantage that i.e. for QFNRA formulas, the linear constraints can be checked with less complex, faster methods, before the non-linear constraints are handled with a second theory solver.

[1] If no negation appears in the input formula then false constraints do not need to be considered.

2.2 QFNRA

In this thesis we focus on quantifier-free nonlinear real arithmetic (QFNRA), which is defined inductively via formulas, constraints and polynomials [Pro14]:

Definition 2.2.1 (QFNRA)**.** *We fix the variables* $\{x_1,...,x_n\}$*. Then*

$$\begin{aligned} p &:= r \mid x \mid (p+p) \mid (p-p) \mid (p \cdot p) \\ c &:= p < 0 \mid p = 0 \\ \varphi &:= c \mid (\varphi \wedge \varphi) \mid \neg\varphi \end{aligned}$$

where $r \in \mathbb{Q}$ *is a constant and* $x \in \{x_1,...,x_n\}$ *is a variable. We call* p *polynomials and define* $\mathcal{C}$ *to be the set of constraints* c*.*

Definition 2.2.2 (Degree)**.** *For a polynomial* $p = a_1 x_1^{e_{1,1}} \cdot ... \cdot x_n^{e_{n,1}} + ... + a_k x_1^{e_{1,k}} \cdot ... \cdot x_n^{e_{n,k}}$ *the degree of* p *is defined as* $deg(p) := \max_{1 \leq j \leq k}(\sum_{i=1}^{n} e_{i,j})$ *[CLJÁ12].*

Constraints of the form $p \geq 0, p \leq 0, p > 0, p \neq 0$, as well as formulas of the form $(\varphi \vee \varphi)$ are syntactic sugar and thus derivable from this grammar.

Example 2.2.1 (Syntactic Sugar)**.**

$$(\varphi \vee \varphi) := \neg(\neg\varphi \wedge \neg\varphi)$$

$$p > 0 := 0 - p < 0$$

$$p \leq 0 := (p < 0) \vee (p = 0)$$

If all polynomials in φ have a maximum degree of 1, they are called linear and φ becomes a quantifier-free *linear* real arithmetic (QFLRA) formula, for which faster methods, e.g. Simplex [Nab09], for satisfiability checking exist.

Chapter 3

Simplification

In 2020, Brown et al. designed a partial solver for QFNRA formulas [BVE20]. The aim of this solver is to either deduce UNSAT or less complex constraints implying the given formula. These constraints can then for example be handed to a complete solver. What exactly defines the simplicity of constraints is not clear. There are many points of consideration, like number of variables, number of terms or the degree of the involved polynomials. We will discuss this topic in more detail in Section 3.4. In general, a why-mapping is created, which indicates for constraints from which combination of other constraints they can be derived.

Definition 3.0.1 (why-map)**.** *The* ***why*** *map is used to collect all derived implications.*

$$\boldsymbol{why} : \mathcal{C} \to \left\{ \bigwedge \mathcal{C} \right\}$$

$$\forall f \in \mathcal{C} : \forall g \in \boldsymbol{why}(f) : g \Rightarrow f$$

Constraints f which were handed to inprocessing by the SAT solver are called given. ***why*** *is initialized as*

$$\boldsymbol{why}(f) := \begin{cases} \{given\} & \text{if } f \text{ is given} \\ \{\} & \text{else} \end{cases}$$

We update implications in ***why*** *as*

$$\boldsymbol{why}(f) := \boldsymbol{why}(f) \cup \{g\}$$

or write

$$g \Rightarrow f \text{ is added to } \boldsymbol{why} \ .$$

The partial solver is composed of mainly three algorithms: Blackbox, Whitebox and SimpleSubstitution. Blackbox considers the factors of the constraints as "blackboxes" and thus focuses on their factor structure. Whitebox derives signs for the factors based on the polynomial structure and the known variable signs. Lastly, SimpleSubstitution is used to replace variables using simple equations. According to Brown et al. these three algorithms are executed until there are no new derivations. For the case that UNSAT was found, they also explain a strategy to find an UNSAT core. Otherwise, the constraints to give to the next solver have to be selected, for which the algorithm

Simplify was designed. In the following we will illustrate the Blackbox, Whitebox and SimpleSubstitution as described in [BVE20]. Lastly, the Simplify algorithm is used to narrow down the constraints to pass to the theory solver and the creation of UNSAT cores is clarified.

3.1 Blackbox

The first deduction approach is Blackbox simplification. It is important that polynomials of the form $\sum_j a_j x_j^{d_j}$ need to be factorized to be of the form $\Pi_i f_i^{d_i}$. Blackbox simplification focuses solely on the factor structure, which means that coefficients, exponents and variables in the factors are not relevant for these deductions. Thus, as the name suggests, it considers the factors to be blackboxes. First, the data is preprocessed for the actual Blackbox algorithm, which also leads to the first deductions. Afterwards, StrictBlackbox is used to simplify strict constraints and MinWtBasis does the same for non-strict constraints.

Example 3.1.1. *For a running example, we will consider the set* $F^* = (a+c^2)(a^2+b) < 0 \wedge (a+c^2)^2(a^2+b) < 0 \wedge a+b \leq 0 \wedge (a+c^2)^4(a+b) \geq 0 \wedge b+c = 0 \wedge a^3 > 0 \wedge b < 0 \wedge c > 0$. *Depending on the impact on the current derivations, subformulas may be omitted.*

3.1.1 Preprocessing

Definition 3.1.1 (Monomial Inequalities, Combinatorial Mapping)**.**
Let $A_i^* = \Pi_j f_j^{d_j} \sigma 0$, *where* f_j *are the distinct factors of* F^*, $d_j \in \mathbb{N}$ *and* $\sigma \in S_{relop}$. S_{relop} *is defined in Section 3.2.1.*
Let $\{x_1,...,x_k\}$ *be a set of distinct fresh variables, one for each factor in* $A_1^* \wedge ... \wedge A_m^*$.
We define $A_i := \Pi_{j=1}^k x_j^{d_j} \sigma 0$

Example 3.1.2 (Combinatorial Mapping)**.** *For the running example, we map the factors as follows:*

$$(a+c^2) \mapsto x_1, (a^2+b) \mapsto x_2, (a+b) \mapsto x_6, (b+c) \mapsto x_7, a \mapsto x_3, b \mapsto x_4, c \mapsto x_5$$

This leads to a new formula, relevant for the rest of the Blackbox algorithm:

$$F = \underbrace{x_1x_2 < 0}_{A_1} \wedge \underbrace{x_1^2x_2 < 0}_{A_2} \wedge \underbrace{x_6 <= 0}_{A_3} \wedge \underbrace{x_1^4x_6 >= 0}_{A_4} \wedge \underbrace{x_7 = 0}_{A_5} \wedge \underbrace{x_3^3 > 0}_{A_6} \wedge \underbrace{x_4 < 0}_{A_7} \wedge \underbrace{x_5 > 0}_{A_8}$$

Strict and non-strict constraints have separate derivation processes. Thus, we have to differentiate between strict and non-strict constraints and variables. A variable is strict, if it occurs in at least one strict constraint. If the variable has the exponent 0, the constraint would evaluate to 0 as well, making it non-strict.

Definition 3.1.2 (Variable Strictness)**.** *Let a variable be strict if it occurs in at least one constraint* A_i *with* $A_i = \Pi_j f_j^{d_j} \sigma 0$ *and* $\sigma \in \{<, >, \neq\}$. *We reorder the variables so that* $X_s = \{x_1,...,x_s\}$ *are the strict variables and* $X_n = \{x_{s+1},...,x_n\}$ *are the non-strict variables.*

For all strict variables, the strictness and its reason, i.e. the strict constraints it occurs in, are documented in the mapping. Thus, for all $x_j \in X_s$

$$\mathbf{why}(x_j \neq 0) = \{A_i \mid x_j \text{ appears in } A_i \text{ and } A_i \text{ is strict}\}$$

is added to **why** .

Example 3.1.3 (Variable Strictness)**.** *In our running example*

$$X_s = \{x_1, x_2, x_3, x_4, x_5\}, X_n = \{x_6, x_7\}$$

$$\begin{array}{lll} \boldsymbol{why}(x_1 \neq 0) = \{A_1, A_2\} & \boldsymbol{why}(x_2 \neq 0) = \{A_1, A_2\} & \\ \boldsymbol{why}(x_3 \neq 0) = \{A_6\} & \boldsymbol{why}(x_4 \neq 0) = \{A_7\} & \boldsymbol{why}(x_5 \neq 0) = \{A_8\} \end{array}$$

As the product of strict variables is always strict, we normalize non-strict constraints in such a way that they become strict, if all of their variables are strict.

Definition 3.1.3 (Non-strict Normalization)**.** *Let A_i be of the form $m\sigma 0$ with $\sigma \in \{LEOP, GEOP, EQOP\}$. Then they are transformed to their strict equivalent, if $\forall v \in Vars(A_i) \cap X_s$:*

$$A_i' := \begin{cases} m < 0 & \text{if } A_i = (m \leq 0) \\ m > 0 & \text{if } A_i = (m \geq 0) \\ FALSE & \text{if } A_i = (m = 0) \end{cases}$$

These normalizations are documented in **why** as

$$\mathbf{why}(A_i') = \mathbf{why}(A_i') \cup \left\{ m\sigma 0 \wedge \bigwedge_{x_j \in Vars(m)} x_j \neq 0 \right\}$$

$$\mathbf{why}(A_i) = \mathbf{why}(A_i) \cup \{A_i'\}$$

For all x_j of m the mapping is adapted as well:

$$\mathbf{why}(x_j \neq 0) = \mathbf{why}(x_j \neq 0) \cup \{A_i'\}$$

The strictness of a variable also impacts its normalization. Strict variables with an even exponent may be omitted, as they do not impact the sign of the polynomial. Non-strict variables on the other hand may still be zero and thus also impact the sign. Even exponents should therefore not be omitted for non-strict variables.

Definition 3.1.4 (Variable Normalization)**.** *Given a monomial inequality $A = \Pi_j^{d_j} \sigma 0$ we define*

$$\mathcal{N}(A) := \Pi_j^{d_j'} \sigma 0$$

with d_j' defined according to its strictness as

$$d_j' := \begin{cases} d_j \mod 2 & x_j \in X_s \\ 2 - (d_j \mod 2) & x_j \in X_n \end{cases}$$

For strict variables we add

$$\mathbf{why}(\mathcal{N}(A_i)) = \mathbf{why}(\mathcal{N}(A_i)) \cup \{A_i \wedge \bigwedge_{\text{strict } x_j \in Vars(A_i)} x_j \neq 0\}$$

$$\mathbf{why}(A_i) = \mathbf{why}(A_i) \cup \{\mathcal{N}(A_i) \wedge \bigwedge_{\text{strict } x_j \in Vars(A_i)} x_j \neq 0\}$$

For non-strict variables on the other hand it suffices to add

$$\mathbf{why}(\mathcal{N}(A_i)) = \mathbf{why}(\mathcal{N}(A_i)) \cup \{A_i\}$$

$$\mathbf{why}(A_i) = \mathbf{why}(A_i) \cup \{\mathcal{N}(A_i)\}$$

Example 3.1.4 (Variable Normalization)**.** *In our running example, variable normalization only impacts A_2, A_4 and A_6:*

$$\mathcal{N}(A_2) := x_2 < 0 \qquad \mathcal{N}(A_4) := x_6 >= 0 \qquad \mathcal{N}(A_6) := x_3 > 0$$

Thus the resulting formula is

$$\mathcal{N}(F) := \underbrace{x_1 x_2 < 0}_{A'_1} \wedge \underbrace{x_2 < 0}_{A'_2} \wedge \underbrace{x_6 <= 0}_{A'_3} \wedge \underbrace{x_6 >= 0}_{A'_4} \wedge \underbrace{x_7 = 0}_{A'_5} \wedge \underbrace{x_3 > 0}_{A'_6} \wedge \underbrace{x_4 < 0}_{A'_7} \wedge \underbrace{x_5 > 0}_{A'_8}$$

We add the following to ***why*** *:*

$$\begin{array}{lll} \boldsymbol{why}(A'_2) = \{A_2 \wedge x_1 \neq 0\} & \boldsymbol{why}(A'_4) = \{A_4 \wedge x_1 \neq 0\} & \boldsymbol{why}(A'_6) = \{A_6\} \\ \boldsymbol{why}(A_2) = \{A'_2 \wedge x_1 \neq 0\} & \boldsymbol{why}(A_4) = \{A'_4 \wedge x_1 \neq 0\} & \boldsymbol{why}(A_6) = \{A'_6\} \end{array}$$

The primary means of deduction in the Blackbox algorithm is Gaussian elimination. Gaussian elimination is only executed in the way that rows of a matrix are added modulo 2. Therefore, Brown et al. define a vector representation for the normalized constraints using a bijection denoted as Γ.

Definition 3.1.5 (Vector Representation)**.** *Let $A_i := \Pi_j x_j^{d_j} \sigma 0$. For two vectors u, v let $u \oplus v$ denote the concatenation. We define the Vector Representation of A_i as*

$$\Gamma(A_i) := \begin{cases} [d_1,...,d_r,0] \oplus [d_{r+1},...,d_n] & \text{if } \sigma \in \{>, \geq\} \\ [d_1,...,d_r,1] \oplus [d_{r+1},...,d_n] & \text{if } \sigma \in \{<, \leq\} \\ [0,...,0,0] \oplus [0,...,0] & \text{if } \sigma \in \{\neq\} \\ \Gamma(\Pi_j x_j^{2d_j} \leq 0) & \text{if } \sigma \in \{=\} \end{cases}$$

This definition is extended to a bijection from a formula to a matrix, denoted by $\Gamma^{-1}(A_i)$.

As a zero-vector does not have any impact on the deductions, it is not necessary to add formulas with $\sigma \in \{\neq\}$ to the matrix. Γ works for equalities, as even exponents always lead to a value ≥ 0. If a factor consisting only of values ≥ 0 is restricted to be ≤ 0, the only possible solution is 0.

Example 3.1.5 (Vector Representation)**.** *As A'_6, A'_7 and A'_8 are strict constraints only containing a single variable which does not occur in any other constraint, we will omit them for simplicity for the rest of Blackbox deductions.*

Applying Γ to $\mathcal{N}(F)$ leads to:

	x_1	x_2	σ	x_6	x_7
A'_1	1	1	1	0	0
A'_2	0	1	1	0	0
A'_3	0	0	1	1	0
A'_4	0	0	0	1	0
A'_5	0	0	1	0	2

This preprocessed matrix is then used for the actual Blackbox deductions. These are separated according to the strictness into StrictDeductions and MinWtBasis.

3.1.2 Strict Deductions

Algorithm 1 StrictBlackbox, given in [BVE20]

Input: B: set of normalized vectors with strict variables x_1,...,x_s and non-strict variables x_{s+1},...,x_n, **why** as defined in Definition 3.0.1

Output: B_s, B_n so that either $\Gamma^{-1}(B)$ is UNSAT and B_s = FALSE or $\Gamma^{-1}(B)$ is SAT and B_s, B_n are vector sets where B_s is strict and in row-echelon form and B_n is non-strict and $\Gamma^{-1}(B) \equiv \Gamma^{-1}(B_s) \wedge \Gamma^{-1}(B_n)$

1: split B into its strict B' and non-strict B_n row vectors
2: form matrix M from vectors of B'
3: produce M' by Gaussian elimination of M, matrix K: $K \cdot M = M'$
4: **for** i from 1 to $|B'|$ **do**
5: $\quad N :=$ set of column indices of non-zero entries of row i of K
6: $\quad E := \{M_{i,\_} \mid i \in N\}$ ▷ *The elements of E sum to w: the ith row of M'. As K was constructed by Gaussian elimination steps, E is minimal in the sense that no strict subset sums to w*
7: $\quad w :=$ row i of M'
8: $\quad$ **if** w is [0,...,0,1] $\oplus$ [0,...,0] **then**
9: $\quad\quad$ add $\mathbf{why}(FALSE) = \mathbf{why}(FALSE) \cup \{\bigwedge_{u \in E} \Gamma^{-1}(u)\}$
10: $\quad\quad$ **return** $B_s :=$ FALSE, B_n
11: $\quad$ **else if** w is not zero **then**
12: $\quad\quad$ add $\mathbf{why}(\Gamma^{-1}(w)) = \mathbf{why}(\Gamma^{-1}(w)) \cup \{\bigwedge_{u \in E} \Gamma^{-1}(u)\}$
13: $\quad\quad$ **for** $v \in E$ **do**
14: $\quad\quad\quad$ add $\mathbf{why}(\Gamma^{-1}(v)) = \mathbf{why}(\Gamma^{-1}(v)) \cup \{\bigwedge_{u \in E-\{v\}\cup\{w\}} \Gamma^{-1}(u)\}$
15: $\quad$ **else**
16: $\quad\quad$ **for** $v \in E$ **do**
17: $\quad\quad\quad$ add $\mathbf{why}(\Gamma^{-1}(v)) = \mathbf{why}(\Gamma^{-1}(v)) \cup \{\bigwedge_{u \in E-\{v\}} \Gamma^{-1}(u)\}$
18: $B_s := \bigwedge_{v \in D} v$, where D is set of non-zero rows in M'
19: **return** B_s, B_n

StrictBlackbox (Algorithm 1) is structured in a way that only the first three lines of code represent the actual computations. The rest of the algorithm describes the

possible derivations.
For strict deductions, only the rows of the matrix representing a strict monomial inequality are considered. Those are all the rows with zero v part: $v = [d_{r+1},...,d_n] = [0,...,0]$.

Example 3.1.6 (Strict Matrix)**.** *In our running example, we select the following strict rows:*

	x_1	x_2	σ	x_6	x_7
A'_1	1	1	1	0	0
A'_2	0	1	1	0	0

From the following examples for StrictBlackbox we will omit the first column and row for simplicity.

Lemma 3.1.1 ([BVE20])**.** *For the mapping $F \Rightarrow A \wedge A'$ with $\Gamma(A) = u \oplus v$ and $\Gamma(A') = u' \oplus v'$ addition mod 2 holds:*

$$F \Rightarrow \Gamma^{-1}(\mathcal{N}((u + u') \oplus (v + v')))$$

Lemma 3.1.1 is used as the basis for the StrictBlackbox algorithm as described in Algorithm 1.

Lemma 3.1.2 (StrictBlackbox UNSAT [BVE20])**.** *F is satisfiable iff* [0,...,0,1] $\oplus$ [0,...,0] *is not deducible from M.*

In StrictBlackbox, M is transformed to reduced row echelon form [Leo09], where σ is never used to pivot and rows are not swapped. The algorithm returns UNSAT if [0,...,0,1] $\oplus$ [0,...,0] is deduced.

Every row of M', the matrix in reduced row echelon form, is a deduction, for which the explanation is tracked in the **why**-mapping.

Lemma 3.1.3. *If u is a row of M' and w is a vector so that $wM = u$ and m_i are the rows of M, then according to [BVE20] it follows that*

$$\bigwedge_{i \ s.t. \ w_i=1} \Gamma^{-1}(m_i) \Rightarrow \Gamma^{-1}(u)$$

These deductions are added to **why**. Additionally, any row of M producing u is implied by u in combination with the other rows summing to u.

Example 3.1.7 (StrictBlackbox)**.** *Performing Gaussian elimination on*

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

leads to the new matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

*Thus from $x_1x_2 < 0 \wedge x_2 < 0$ we deduced $x_1 > 0 \wedge x_2 < 0$. We append **why** as follows:*

$$\begin{aligned} \textbf{\textit{why}}(x_1 > 0) &= \textbf{\textit{why}}(x_1 > 0) \cup \{x_1x_2 < 0 \wedge x_2 < 0\} \\ \textbf{\textit{why}}(x_1x_2 < 0) &= \textbf{\textit{why}}(x_1x_2 < 0) \cup \{x_1 > 0 \wedge x_2 < 0\} \\ \textbf{\textit{why}}(x_2 < 0) &= \textbf{\textit{why}}(x_2 < 0) \cup \{x_1x_2 < 0 \wedge x_1 > 0\} \end{aligned}$$

Why does Gaussian elimination of the exponents work? Let $A_1 = \Pi x_j^{d_j} < 0$ and $A_2 = \Pi x_j^{e_j} < 0$. Then $\Pi x_j^{d_j+e_j} > 0$.

Example 3.1.8 (Gaussian Elimination)**.** *Again we consider $x_1x_2 < 0$ and $x_2 < 0$. If we now multiply e.g. the first constraint ($x_1x_2 < 0$) with the second polynomial (x_2), we reach $x_1x_2^2$. As x_2 is negative according to constraint 2, we multiplied the first constraint by a negative number. Thus, the sign is turned around and the resulting constraint is $x_1x_2^2 > 0$. As we already used for variable normalization, even exponents can be omitted and thus we reach the result of $x_1 > 0$.*

For the rest of Blackbox (Algorithm 3), the rows making up the strict part of M will be replaced by the resulting matrix of StrictBlackbox. Brown et al. state that the implications of Algorithm 1 prove that $\Gamma^{-1}(B) \equiv \Gamma^{-1}(B_s) \wedge \Gamma^{-1}(B_n)$, where B is the vector form of F, and B_s and B_n are the strict/non-strict submatrices of B.

3.1.3 Non-strict Deductions

Similar to the strict deductions, the goal of non-strict deductions (Algorithm 2) is to either find implications between the constraints or derive less complex constraints. In the latter case, this would always be an equality. As non-strict inequalities are always satisfiable by setting a non-strict variable to 0, if F was unsatisfiable, it would have already been deduced during the strict deductions. Thus, it is not the goal to find unsatisfiability in MinWtBasis (Algorithm 2).

In the strict deductions, Gaussian elimination was used to reduce the total occurrences of strict variables in the constraints. MinWtBasis tries to achieve the same for non-strict variables with a slightly different approach. For this, we define a support $S(w)$ of w which is the set of indices of the non-strict part at which w is non-zero. The weight of the vector w is defined as $wt(w) = |S(w)|$, the number of non-zero entries of the non-strict part of w. MinWtBasis produces constraints which try to minimize the sum of weights, leading to fewer occurrences of non-strict variables.

The MinWtBasis algorithm (Algorithm 2) maintains two vector sets: B is the given vector set (M), while B_f contains derived or used constraints. As a first step, we select a row w with the highest weight (line 2). If no row with $wt(w) > 0$ exists, the weights can not be minimized any more and we return $B_f \cup B$ as the resulting set. Then we drop w from B and define $B_{\leq}, B_{<}$ as the rows b, for which $S(b)$ is a (strict) subset of $S(w)$ (lines 3-5). Now we want to find out if other constraints already include all the information w can provide or if the non-strict variables in w actually evaluate to 0.

Example 3.1.9. *Given*

	x_1	x_2	σ	x_6	x_7
A_1'	1	0	0	0	0
A_2'	0	1	1	0	0
A_3'	0	0	1	1	0
A_4'	0	0	0	1	0
A_5'	0	0	1	0	2

we select $w = A'_3$. *Thus* $B = \{A'_1, A'_2, A'_4, A'_5\}$, $B_{\leq} = \{A'_1, A'_2, A'_4\}$ *and* $B_{<} = \{A'_1, A'_2\}$.

Algorithm 2 MinWtBasis from [BVE20]

Input: B: set of vectors that are images of inequalities in normalized conjunction F
Output: B_f: minimum-weight set of vectors subject to $\bigwedge_{b \in B_f} \Gamma^{-1}(\mathcal{N}(b)) \equiv F$

1: $B_f := \{\}$
2: $w :=$ a maximum weight element of B with $wt(w) > 0$, if none exists: **return** $B_f \cup B$
3: $B := B/\{w\}$
4: $B_{\leq} := \{b \in B \mid S(b) \subseteq S(w)\}$
5: $B_{<} := \{b \in B \mid S(b) \subset S(w)\}$
6: check whether exists subset $T \subseteq B_{<}$ such that

$$\sum_{t \in T} t \equiv [0,...,0,1] \oplus [0,...,0] (mod\ 2)$$

if yes: goto step 2
7: ▷ *w removed from B and not added to B_f, as $\mathcal{N}(\sum_{t \in T} t)$ defines equation on a subset of variables appearing in the monomial w defines. This gives a new explanation for w. Assuming T is a minimal subset of $B_{<}$ summing to $[0,...,0,1] \oplus [0,...,0] \mod 2$. Add $\bigwedge_{t \in T} \Gamma^{-1}(t) \Rightarrow \Gamma^{-1}(w)$ to* ***why*** ◁
8: form matrix M over GF(2) whose rows are the elements of $B_{\leq}$ modulo 2
9: transform M into row-echelon form by Gaussian elimination
10: $w' :=$ result of reducing $w \mod 2$ by rows of M
11: **if** w' or some row of M equals $[0,...,0,1] \oplus [0,...,0]$ **then**
12: add $w^* := \mathcal{N}(2w + [0,...,0,1] \oplus [0,...,0])$ to B_f
13: ▷ *If $w^* = w$, we have not deduced anything new at this point. We have merely noticed that w is an equation. Otherwise we have deduced the equation $\Gamma^{-1}(w^*)$ and learned that Γ^{-1} is implied by it. In that case, let T^* be a minimal subset of $B_{\leq} \cup \{w\}$ with sum modulo 2 equal to $[0,...,0,1] \oplus [0,...,0]$, which can be constructed with linear algebra over GF(2). Add $\bigwedge_{t \in T^*} \Gamma^{-1}(t) \Rightarrow \Gamma^{-1}(w^*)$ and $\Gamma^{-1}(w^*) \Rightarrow \Gamma^{-1}(w)$ to* ***why*** ◁
14: remove from B any element with the same support as w^*
15: ▷ *Vector v is removed from B if its support is the same as w^*. For each such v we add $\Gamma^{-1}(w^*) \Rightarrow \Gamma^{-1}(v)$ to* ***why*** ◁
16: **else if** $w' \neq [0,...,0,1] \oplus [0,...,0]$ **then**
17: $B_f := B_f \cup \{w\}$
18: ▷ *If fall through both "if"'s, $w' = 0$. So some subset of $B_{\leq}$ sum to be equivalent to $w \mod 2$. Let T' be a minimum sized subset with this property, which is constructable via linear algebra over GF(2). Add $\bigwedge_{t \in T'} \Gamma^{-1}(t) \Rightarrow \Gamma^{-1}(w)$ to* ***why*** ◁
19: goto step 2

If Gaussian elimination mod 2 of $B_{<}$ can produce $[0,...,0,1] \oplus [0,...,0]$, we found a new derivation for w (line 6). Opposed to StrictBlackbox, $[0,...,0,1] \oplus [0,...,0]$ does not mean that UNSAT was found. For non-strict constraints the 1 in the operator column stands for $\leq$ and since the calculations are executed modulo 2, we simply

derived an equality. Therefore, if this test succeeded, we know that already constraints with smaller weights contain all the information of w. Therefore, by removing w no information is lost but the number of occurrences of non-strict constraints is reduced. If T is the minimum set of rows leading to [0,...,0,1] $\oplus$ [0,...,0] we add

$$\mathbf{why}(\Gamma^{-1}(w)) = \mathbf{why}(\Gamma^{-1}(w)) \cup \{\bigwedge_{t\in T} \Gamma^{-1}(t)\}$$

If this test was successful, we found a new derivation for w and therefore do not need w in the result set. We start again by selecting a new w in line 2. Otherwise, we continue by transforming $B_{\leq}$ to reduced row echelon form (M') and row reducing w mod 2 by M' (lines 8-10). Let w' be the result of the row reduction. There are three possible results:

1. w' or some row of M' is [0,...,0,1] $\oplus$ [0,...,0] (lines 11-15).
   We derived that the product of the variables in w is 0. Thus, we add $w^* = \mathcal{N}(2w + [0,...,0,1] \oplus [0,...,0])$ to B_f and remove all rows $v \in B$ with $S(v) = S(w^*)$ from B. This removal can be done as $\Pi_{j\in S(w^*)} x_j = 0$ and therefore all constraints containing all non-strict variables of w^* evaluate to 0 as well. We add the implication of the dropped rows as

   $$\mathbf{why}(\Gamma^{-1}(v)) = \mathbf{why}(\Gamma^{-1}(v)) \cup \{\Gamma^{-1}(w^*)\}$$

   Additionally, if $w \neq w^*$ we derived a new equality and save its derivation as:

   $$\mathbf{why}(\Gamma^{-1}(w)) = \mathbf{why}(\Gamma^{-1}(w)) \cup \{\Gamma^{-1}(w^*)\}$$
   $$\mathbf{why}(\Gamma^{-1}(w^*)) = \mathbf{why}(\Gamma^{-1}(w^*)) \cup \{\bigwedge_{t\in T} \Gamma^{-1}(t)\}$$

2. $w' = [0,...,0,0] \oplus [0,...,0]$ (line 18).
   The sum of some rows T sum to $w \mod 2$. Thus, we found a new derivation for w:

   $$\mathbf{why}(\Gamma^{-1}(w)) = \{\bigwedge_{t\in T} \Gamma^{-1}(t)\}$$

   w is not added to the result set again as T contains constraints implying w.

3. In all other cases, there are no new derivations and w is added to B_f (lines 16-17).

Example 3.1.10. *The test for $B_<$ fails for $w = A'_3$. w' is constructed as $w' = A'_3 + A'_4 = ([0,0,1] \oplus [1,0]) + ([0,0,0] \oplus [1,0]) = [0,0,1] \oplus [0,0] (\mod 2)$.*
This is consistent with the first case distinction. Thus $w^* = \mathcal{N}(2 * ([0,0,1] \oplus [1,0]) + ([0,0,1] \oplus [0,0])) = [0,0,1] \oplus [2,0]$. A'_4 is dropped from B, as it has the same support as w^*. We derived:

$$\boldsymbol{why}(\Gamma^{-1}(x_6 \geq 0)) = \boldsymbol{why}(\Gamma^{-1}(x_6 \geq 0)) \cup \{\Gamma^{-1}(w^*)\}$$
$$\boldsymbol{why}(\Gamma^{-1}(x_6 \leq 0)) = \boldsymbol{why}(\Gamma^{-1}(x_6 \leq 0)) \cup \{\Gamma^{-1}(x_6 = 0)\}$$
$$\boldsymbol{why}(\Gamma^{-1}(x_6 = 0)) = \boldsymbol{why}(\Gamma^{-1}(x_6 = 0)) \cup \{x_6 \leq 0 \wedge x_6 \geq 0\}$$

The cycle continues with $w = A'_5$.

Algorithm 3 Blackbox from [BVE20]

Input: F^*: conjunction of real polynomial constraints

Output: $G^*, \mathbf{why}^*$: if F^* identified as UNSAT G^* is FALSE, otherwise G^* simplified conjunction of polynomial constraints equivalent to F^*, $\mathbf{why}^*$ maps inequalities discovered to be implied by F^* to sets of deductions that imply them

1: $F :=$ combinatorial part of F^*
2: Initialize **why** with given $\Rightarrow A_i$ for each A_i in F
3: $F^1 :=$ normalization of F
4: $B^1 := \Gamma(F^1)$
5: $B^2_s, B^2_n :=$ StrictBlackbox(B^1)
6: **if** $B^2_s =$ FALSE **then**
7: ⌊ **return** FALSE, **why**
8: $B^3 :=$ MinWtBasis($B^2_s \cup B^2_n$)
9: $B^4_f := B^3$ with non-strict elements reduced by the elements of B^2_s
10: ▷ *In this context reduced means row-reduced, as in Gaussian elimination. If w is reduced to w^* and T is the minimal subset of B^2_s such that $w + \sum_{t\in T} t = w^*$, then the deductions added are: $\Gamma^{-1}(w) \wedge \bigwedge_{t\in T} \Gamma^{-1}(t) \Rightarrow \Gamma^{-1}(w^*)$ and $\Gamma^{-1}(w^*) \wedge \bigwedge_{t\in T} \Gamma^{-1}(t) \Rightarrow \Gamma^{-1}(w)$* ◁
11: $G := \Gamma^{-1}(B^2_s \cup B^4_f)$
12: add $x_i \neq 0$ to G, for each strict variable x_i not already appearing in a strict inequality in G
13: apply the reverse of the mapping that produced F from F^* in order to recover G^* and $\mathbf{why}^*$ from G and **why**
14: **return** $G^*, \mathbf{why}^*$

3.1.4 General Algorithm

Finally, Algorithm 3 combines these simplification algorithms and adds some final deductions. Lines 1-8 correspond to the algorithm steps described so far. After MinWtBasis is completed, the resulting non-strict rows are row-reduced by the strict rows (line 9). For row w to be reduced to w^*, we find a minimal subset T of the strict rows so that $w + \sum_{t\in T} t = w^*$. This is documented as

$$\mathbf{why}(\Gamma^{-1}(w^*)) = \mathbf{why}(\Gamma^{-1}(w^*)) \cup \{\Gamma^{-1}(w) \wedge \bigwedge_{t\in T} \Gamma^{-1}(t)\}$$

$$\mathbf{why}(\Gamma^{-1}(w)) = \mathbf{why}(\Gamma^{-1}(w)) \cup \{\Gamma^{-1}(w^*) \wedge \bigwedge_{t\in T} \Gamma^{-1}(t)\}$$

For strict variables not appearing in a strict constraint of the matrix (0 column), $x_i \neq 0$ is added to the result set (line 12). This might for example be the case for some constraint with $\sigma \in \{\neq\}$. In the end, the matrix is transformed back to a conjunction of constraints and the combinatorial mapping is reversed (line 13).

Example 3.1.11 (Blackbox)**.** *The final matrix is*

x_1	x_2	σ	x_6	x_7
1	0	0	0	0
0	1	1	0	0
0	0	1	2	0
0	0	1	0	2

Using Γ^{-1} *we get* $F' = x_1 > 0 \wedge x_2 < 0 \wedge x_6 = 0 \wedge x_7 = 0$*. This corresponds to* $F = (a+c^2) > 0 \wedge (a^2+b) < 0 \wedge (a+b) = 0 \wedge (b+c) = 0$*, leaving out* $a > 0 \wedge b < 0 \wedge c > 0$ *for simplicity.*

3.2 Whitebox

Whitebox deductions focus on the factor structure. We want to deduce information about the sign of the polynomial based on the factor structure and the sign of the variables. This is useful in two ways: First of all, we may deduce a sign for a single factor, which is then helpful for further deductions in Blackbox. Secondly, we may find interdependencies between different polynomials, possibly leading to a smaller final set of constraints for the theory solver. Note that we will not go into as much detail about Whitebox as we did for Blackbox, as this algorithm does not need much adaption for Incrementality and Backtracking. Additional algorithms can be found in Appendix A. First, we will give a definition of the used operators.

3.2.1 Relational Operators

Considered are conjunctions of monomial inequalities $F = A_1 \wedge ... \wedge A_n$. These are defined as $a_1^{e_1} \cdot ... \cdot a_k^{e_k} \sigma 0$, where a_i are variables, $e \in \mathbb{N}$ and σ is a relational operator.

Definition 3.2.1 (Relational Operators)**.** *The set of relational operators is given as* $S_{op} = \{NOOP, LTOP, EQOP, LEOP, GTOP, NEOP, GEOP, ALOP\}$*, where* $NOOP := \bot := FALSE$ *and* $ALOP := \top := TRUE$ *and all others are the common operators. E.g.* $LTOP :=<$ *and* $GEOP :=\geq$*. Additionally Brown et al. defined* $S_{op}^{+} = S_{op} - \{NOOP\}$ *and* $S_{relop} = S_{op} - \{NOOP, ALOP\}$*.*

For Whitebox deductions it is important to have an ordering of relational operators.

Definition 3.2.2 (Strength)**.** *For* $\sigma, \tau \in S_{op} : \sigma \preceq \tau$ *if* $\forall x \in \mathbb{R} : [x\sigma 0 \Rightarrow x\tau 0]$*.* σ *is stronger than* τ *if* $\sigma \prec \tau$*, i.e.* $\sigma \preceq \tau$ *and* $\sigma \neq \tau$*.*

Definition 3.2.3 (Maximal Weakness)**.** *Transferring Definition 3.2.2 to vectors of relational operators, for* $u, v \in S_{op}^n : u \preceq v$ *if* $\forall i \in \{1,...,n\} : u_i \preceq v_i$*.* v *is maximally weak with regards to some property* P*, if there is no weaker vector* w *satisfying this property* P*.*

Figure 3.1 shows a partial ordering of relational operators in regards to strength.

Definition 3.2.4. *For any* $x \in \mathbb{R}$ *Brown et al. define* $op(x)$ *as the strongest element of* S_{op} *satisfied by* x*:*

$$op(x) := \begin{cases} LTOP & x < 0 \\ GTOP & x > 0 \\ EQOP & else \end{cases}$$

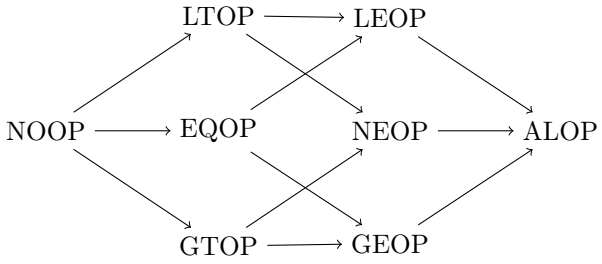


Figure 3.1: Ordering of relational operators as given by [BVE20]

For the Whitebox algorithm, it is necessary to be able to do arithmetic and logical operations. Therefore, it is also important to define $\cdot, +, \wedge, \vee$.

Definition 3.2.5. *For $\sigma, \tau \in S_{op}$ $\sigma \cdot \tau$ is defined as the unique strongest $\gamma \in S_{op}$ so that $\forall x,y \in \mathbb{R} : [x\sigma 0 \wedge y\tau 0 \Rightarrow x \cdot y\gamma 0]$. $+$ is defined analogously. $\sigma \wedge \tau$ is defined as a unique $\gamma \in S_{op}$ such that $\forall x \in \mathbb{R} : [x\sigma 0 \wedge x\tau 0 \Leftrightarrow x\gamma 0]$. $\sigma \vee \tau$ is again defined analogously.*

Table 3.1 represents tables of which σ, τ lead to which γ for $\cdot, +, \wedge, \vee$.

$\times$	$\bot$	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$<$	$\bot$	$>$	$=$	$\geq$	$<$	$\neq$	$\leq$	$\top$
$=$	$\bot$	$=$	$=$	$=$	$=$	$=$	$=$	$=$
$\leq$	$\bot$	$\geq$	$=$	$\geq$	$\leq$	$\top$	$\leq$	$\top$
$>$	$\bot$	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$	$\top$
$\neq$	$\bot$	$\neq$	$=$	$\top$	$\neq$	$\neq$	$\top$	$\top$
$\geq$	$\bot$	$\leq$	$=$	$\leq$	$\geq$	$\top$	$\geq$	$\top$
$\top$	$\bot$	$\top$	$=$	$\top$	$\top$	$\top$	$\top$	$\top$

$+$	$\bot$	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$<$	$\bot$	$<$	$<$	$<$	$\top$	$\top$	$\top$	$\top$
$=$	$\bot$	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$	$\top$
$\leq$	$\bot$	$<$	$\leq$	$\leq$	$\top$	$\top$	$\top$	$\top$
$>$	$\bot$	$\top$	$>$	$\top$	$>$	$\top$	$>$	$\top$
$\neq$	$\bot$	$\top$	$\neq$	$\top$	$\top$	$\top$	$\top$	$\top$
$\geq$	$\bot$	$\top$	$\geq$	$\top$	$>$	$\top$	$\geq$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

$\wedge$	$\bot$	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$<$	$\bot$	$<$	$\bot$	$<$	$\bot$	$<$	$\bot$	$<$
$=$	$\bot$	$\bot$	$=$	$=$	$\bot$	$\bot$	$=$	$=$
$\leq$	$\bot$	$<$	$=$	$\leq$	$\bot$	$<$	$=$	$\leq$
$>$	$\bot$	$\bot$	$\bot$	$\bot$	$>$	$>$	$>$	$>$
$\neq$	$\bot$	$<$	$\bot$	$<$	$>$	$\neq$	$>$	$\neq$
$\geq$	$\bot$	$\bot$	$=$	$=$	$>$	$>$	$\geq$	$\geq$
$\top$	$\bot$	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$	$\top$

$\vee$	$\bot$	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$	$\top$
$\bot$	$\bot$	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$	$\top$
$<$	$<$	$<$	$\leq$	$\leq$	$\neq$	$\neq$	$\top$	$\top$
$=$	$=$	$\leq$	$=$	$\leq$	$\geq$	$\top$	$\geq$	$\top$
$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\top$	$\top$	$\top$	$\top$
$>$	$>$	$\neq$	$\geq$	$\top$	$>$	$\neq$	$\geq$	$\top$
$\neq$	$\neq$	$\neq$	$\top$	$\top$	$\neq$	$\neq$	$\top$	$\top$
$\geq$	$\geq$	$\top$	$\geq$	$\top$	$\geq$	$\top$	$\geq$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Table 3.1: Arithmetic and logical operations on relational operators as described in [BVE20]

3.2.2 Algorithm

After Blackbox focused on derivations from the factor structure, the Whitebox algorithm (Algorithm 4) focuses on the polynomial structure of each factor. Thus the factors are not considered as variables like in Blackbox (Algorithm 3). Note that the only relevant property of the variables occurring in the factors is their sign, not their actual value.

Algorithm 4 Whitebox from [BVE20]

Input: F: formula of the form $F = A_1 \wedge ... \wedge A_r$ over the variables x_1,...,x_n, where each A_i is of the form $q\sigma 0$

Output: Conjunction G, which is equivalent to F and **why** as described before

1: create empty **why** map
2: for each A_i add $\mathbf{why}(A_i) = \{given\}$
3: $\alpha := [ALOP,...,ALOP]$
4: For each x_i set α_i to the strongest σ such that $x_i\sigma 0$ appears in F or $ALOP$ if none exists
5: $P :=$ set of all irreducible factors of the A_is and variables
6: **for** $p \in P$ **do**
7: $\quad \beta :=$ PolynomialSign(p,α)
8: $\quad \alpha' :=$ PolynomialSignProof(p,α, β)
9: $\quad I := \{i \in \{1,...,n\} \mid \alpha'_i \neq ALOP\}$
10: $\quad$ Add $\mathbf{why}(p\beta 0) = \mathbf{why}(p\beta 0) \cup \{\bigwedge_{i\in I} x_i\alpha'_i 0\}$
11: **for** p,$q \in P$, where $p \neq q$ **do**
12: $\quad \kappa :=$ strongest sign condition on q appearing in F
13: $\quad (\alpha', \gamma, t, \beta) :=$ DeduceSignExplain(p,q,κ,α)
14: $\quad I := \{i \in \{1,...,n\} \mid \alpha'_i \neq ALOP\}$
15: $\quad$ Add $\mathbf{why}(p\gamma 0) = \mathbf{why}(p\beta 0) \cup \{q\kappa 0 \wedge \bigwedge_{i\in I} x_i\alpha'_i 0\}$
16: $G :=$ conjunction of all inequalities appearing as keys in **why**
17: **return** G,**why**

Whitebox (Algorithm 4) is structured in two main parts. The first part (lines 6-10) focuses on the sign deductions possible for each single factor considering the given variable signs (α). PolySign (Appendix A, Algorithm 12) derives the sign of the factor (β) given the variable signs in α and the prior defined arithmetic (Table 3.1) in line 7. Note that if we do not know a variable sign, it is assigned to ALOP, meaning it could be any sign. In line 8, PolySignProof (Appendix A, Algorithm 14) then derives the weakest possible variable signs (α') necessary to derive β. The corresponding **why** map entries are

$$\mathbf{why}(p\beta 0) = \mathbf{why}(p\beta 0) \cup \{\bigwedge_{i\in I} x_i\alpha'_i 0\}.$$

Note that all α'_i are weaker or equal to α_i. To be able to properly use Simplify (Algorithm 7) later, we use the following addition:

$$\forall i \in I : \mathbf{why}(x_i\alpha'_i 0) = \mathbf{why}(x_i\alpha'_i 0) \cup \{x_i\alpha_i 0\}$$

Example 3.2.1 (Whitebox Part 1)**.** *Let P defined as in Algorithm 4. Then $P = \{a + c^2, a^2 + b, a + b, b + c, a, b, c\}$ and $\alpha = [GTOP, LTOP, GTOP]$. Let $p = a + c^2$. Then we can deduce $\beta = GTOP$ using $\alpha' = [GTOP, ALOP, ALOP]$.*

$$\boldsymbol{why}(a + c^2 > 0) = \boldsymbol{why}(a + c^2 > 0) \cup \{a > 0\}$$

The second part of Whitebox (lines 11-15) uses the approach that it may be possible to derive a sign for one factor considering the sign of another factor. Thus we try to determine the sign of p using $q\kappa 0$ and α. κ may have already been derived in Blackbox or in the first derivation part of Whitebox. If we cannot find a $\kappa \neq ALOP$, we do not consider that q. Brown et al. describe that for a proper selection of t, it is often possible to derive something about p using $p + tq$. Thus in DeduceSignExplain (Appendix A, Algorithm 17) we find intervals of t, in which $p+tq$ leads to a common sign β (line 13). Knowing this β, κ and t we can derive a sign γ for p, by extracting the corresponding value from the following tables:

$T_{ded}[\kappa,LTOP,\beta]$

$\kappa\backslash\beta$	$\bot$	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$<$	$\bot$	$<$	$<$	$<$	$\top$	$\top$	$\top$	$\top$
$=$	$\bot$	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$	$\top$
$\leq$	$\bot$	$<$	$\leq$	$\leq$	$\top$	$\top$	$\top$	$\top$
$>$	$\bot$	$\top$	$>$	$\top$	$>$	$\top$	$>$	$\top$
$\neq$	$\bot$	$\top$	$\neq$	$\top$	$\top$	$\top$	$\top$	$\top$
$\geq$	$\bot$	$\top$	$\geq$	$\top$	$>$	$\top$	$\geq$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

$T_{ded}[\kappa,GTOP,\beta]$

$\kappa\backslash\beta$	$\bot$	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$<$	$\bot$	$\top$	$>$	$\top$	$>$	$\top$	$>$	$\top$
$=$	$\bot$	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$	$\top$
$\leq$	$\bot$	$\top$	$\geq$	$\top$	$>$	$\top$	$\geq$	$\top$
$>$	$\bot$	$<$	$<$	$<$	$\top$	$\top$	$\top$	$\top$
$\neq$	$\bot$	$\top$	$\neq$	$\top$	$\top$	$\top$	$\top$	$\top$
$\geq$	$\bot$	$<$	$\leq$	$\leq$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Table 3.2: Strongest sign condition for p given κ,β and t as described in [BVE20]. Here, LTOP and GTOP are the sign of t.

Again we find the weakest α' necessary to deduce γ and add the deduction to **why** :

$$\mathbf{why}(p\gamma 0) = \mathbf{why}(p\gamma 0) \cup \{q\kappa 0 \wedge \bigwedge_{i \in I} x_i \alpha_i' 0\}$$

Note that for $t = 0$, the deduction did not depend on q, but solely on p. Thus $\gamma = \beta$ and $q\kappa 0$ is not necessary for the deduction.

Example 3.2.2 (Whitebox Part 2)**.** *Again* $P = \{a + c^2, a^2 + b, a + b, b + c, a, b, c\}$, $\alpha = [GTOP, LTOP, GTOP]$ *and* $p = a + c^2$*. Let* $q = a + b$ *and* $\kappa = EQOP$*. Then we can deduce* $\gamma = GTOP$ *from using* $q\kappa 0$ *and* $\alpha' = [ALOP, LTOP, ALOP]$*.*

$$\boldsymbol{why}(a + c^2 > 0) = \boldsymbol{why}(a + c^2 > 0) \cup \{a + b = 0 \wedge b < 0\}$$

Repeating Blackbox at a later point will have the advantage that single variables (factors) already derived a sign using Whitebox, leading to even simpler deductions for other constraints.

3.3 Simple Substitution

The final simplification method is SimpleSubstitution (Algorithm 6). For simple substitution, variables are eliminated if they are equated to a constant value or a multiple of another variable. If variables are multiples of each other, one variable must be selected to be substituted by the other variable.

Algorithm 5 SubstitutionMapping

Input: F, a formula of the form $A_1 \wedge ... \wedge A_k$ over the variables x_1,...,x_n, where $A_i = \sum_j x_j^{d_j} \sigma 0$, $\sigma \in S_{relop}$

Output: $(V, sub, reason)$ so that for all $x \in V : [reason[x] \Rightarrow x = sub[x]], V \subseteq \{x_1,...,x_n\}$

```
1:  K_1 := {A_i | i ∈ {1,...,k} ∧ σ_i = EQOP ∧ |Vars(A_i)| = 1 ∧ A_i linear}
2:  K_2 := {A_i | i ∈ {1,...,k} ∧ σ_i = EQOP ∧ |Vars(A_i)| = 2 ∧ A_i linear}
3:  V := ∅
4:  for k ∈ K_1 of the form ax + c = 0 do
5:      if x ∈ V then
6:          continue
7:      V := V ∪ {x}
8:      sub[x] := -c/a
9:      add k to reason[x]
10: for k ∈ K_2 of the form ax_i + bx_j = 0 do
11:     if x_i ∉ V ∧ x_j ∉ V then
12:
13:         V := V ∪ {x_i}
14:         sub[x_i] := -b/a x_j
15:         add k to reason[x_i]
16:     else if x_i ∈ V ∧ x_j ∉ V then
17:         V := V ∪ {x_j}
18:         sub[x_j] := -a/b sub[x_i]
19:         add k ∧ reason[x_i] to reason[x_j]
20:     else if x_i ∉ V ∧ x_j ∈ V then
21:         Analogue to case "x_i ∈ V ∧ x_j ∉ V"
22:     else
23:         if sub[x_i], sub[x_j] are both constant or both not constant then
24:             continue
25:         else if sub[x_i] is constant then
26:             sub[x_j] := -a/b sub[x_i]
27:             overwrite reason[x_j] by reason[x_i] ∧ k
28:         else if sub[x_j] is constant then
29:             Analogue to case "sub[x_i] is constant"
30: return (V,sub,reason)
```

Brown et al. define V, the set of variables to be substituted, $sub[\cdot]$ a mapping from the elements of V to the value/expression by which it is to be substituted, and $reason[\cdot]$, a mapping from the elements of V to a conjunction of the form $A_{i_1} \wedge ... \wedge A_{i_k}$ such that $reason[x] \Rightarrow x = sub[x]$. Accordingly, reason is an explanation for the substitution.

As Brown et al. do not define the creation of V, *sub* and *reason*, Algorithm 5 gives an own approach for the derivation.

First, all variables that can be mapped to constants are included in V and the corresponding constraints are stored in *reason* (lines 4-9). If they are already in V, *sub* and *reason* are not updated (lines 5-6). Possible contradictions will be found in Simplify (Algorithm 7). Then, variables that are equated with other variables are considered (lines 10-29). If none of the variables are already used for a substitution, a

variable is selected and V, *sub* and *reason* are adjusted accordingly (lines 11-15). If a variable is already included, it is substituted in the constraint and the other variable is mapped to the result (lines 16-21). If both variables are already substituted, a distinction must be made between cases (lines 22-29). If the *sub* mapping for one of the variables is constant, the constant value can be inserted into the equation and the result is used instead of the original mapping of the other variable (lines 25-29). We do not consider the case where both sub mappings are not constant, we then just keep the already existing mapping. SimpleSubstitution handles the case when both variables are constant but the equation is unsatisfiable after substitution. The result of SubstitutionMapping combined with the input formula is used as the input for SimpleSubstitution (Algorithm 6).

Algorithm 6 SimpleSubstitution from [BVE20]

Input: F: formula of the form $F = A_1 \wedge ... \wedge A_r$, V: set of variables to eliminate, $sub[\cdot]$: map of each $x \in V$ to the value/expression that will be substituted for x, $reason[\cdot]$: map from each $x \in V$ to a conjunction of the form $A_{i_1} \wedge ... \wedge A_{i_k}$ such that $reason[x] \Rightarrow x = sub[x]$.

Output: F': formula such that $F' \wedge \bigwedge_{x \in V} x = sub(x)$ is equivalent to F, **why** , where for each A_j in F that does not appear in F', there is an atom A'_j in F' such that **why**$[A_j]$ and **why**$[A'_j]$ are both non-empty.

1: $F' :=$ TRUE
2: **for** A_j in F **do**
3: $\quad V_j :=$ Vars$(A_j) \cap V$
4: $\quad A'_j :=$ result of substituting variables V_j in A_j using $sub[\cdot]$
5: $\quad$ **if** $A'_j =$ FALSE **then**
6: $\quad\quad$ add $\bigwedge_{x \in V_j} reason[x] \Rightarrow \neg A_j$ and $A_j \wedge \neg A_j \Rightarrow FALSE$ to **why**
7: $\quad$ **else if** A_j in in $reason[x]$ for some $x \in V_j$ **then**
8: $\quad\quad$ **continue**
9: $\quad$ **else if** $A_j = A'_j$ **then**
10: $\quad\quad F' := F' \wedge A'_j$
11: $\quad$ **else**
12: $\quad\quad F' := F' \wedge A'_j$
13: $\quad\quad$ Add **why**$(A'_j) =$ **why**$(A'_j) \cup \{A_j \wedge \bigwedge_{x \in V_j} reason[x] \Rightarrow A'_j\}$
14: $\quad\quad$ Add **why**$(A_j) =$ **why**$(A_j) \cup \{A'_j \wedge \bigwedge_{x \in V_j} reason[x] \Rightarrow A_j\}$
15: **return** F', **why**

SimpleSubstitution iterates over all constraints and substitutes the variables according to their *sub* mapping (line 4). If the resulting constraint is FALSE, the reasons are used for the **why** mapping (lines 5-6). Constraint which appear in *reason*, should not be substituted, as their original form is necessary for the deductions (lines 7-8) and they would just evaluate to TRUE. The substituted constraints are added to the formula and *reason* is used for the **why** mapping (lines 11-14). The final set of constraints consists of all reason constraints, unimpacted and substituted constraints.

Example 3.3.1 (SimpleSubstitution)**.** *In our running example we find:*

$$V = \{a, b\}$$
$$sub[b] = -c \qquad reason[b] = [b + c = 0]$$
$$sub[a] = c \qquad reason[a] = [a + b = 0, b + c = 0]$$

After substitution, we reach for instance:

$$\textbf{\textit{why}}(c + c^2 > 0) = \textbf{\textit{why}}(c + c^2 > 0) \cup \{a + c^2 > 0 \wedge a + b = 0 \wedge b + c = 0\}$$

3.4 Constraint Selection and UNSAT Core

3.4.1 Selection

The goal of the algorithms introduced by Brown et al. [BVE20] is to either find UNSAT or to simplify the constraints before giving them to the next solver. Simplify (Algorithm 7) is used to select preferably noncomplex constraints based on **why**. For this purpose, all constraints (original and derived) are sorted according to a criterion. Many options are possible for this sorting. For instance one could consider

Algorithm 7 Simplify from [BVE20]

Input: **why** as defined in Definition 3.0.1
Output: G: conjunction of inequality subset from which all other inequalities are derivable by deductions in **why**

```
L := list of all constraints in why
sort L from simplest to most complex
F_R := conjunction of all reason implications, except givens, in propositional form
F_G := conjunction of all constraints in why in propositional form
for A from L in reverse order do
    remove propositional form of A from F_G
    if A not implied by F_R ∧ F_G add propositional form of A back to F_G
```

the number of terms, number of variables or degree of the respective polynomial. In [BVE20], weights are assigned to variables depending on whether they were eliminated in SimpleSubstitution. This weight increases with each application of the algorithm. Variables that have not been eliminated have the highest weight. The sorting is now primarily determined by the variable weights of each constraint. The tie breaker is defined in Definition 3.4.1.

Definition 3.4.1 (Constraint Sorting)**.** *Let a constraint be of the form $f_1 \cdot ... \cdot f_k \sigma 0$ and m_σ is 1 if $\sigma \in \{\leq, \geq, \neq\}$ and 0 otherwise.*

$$m_\sigma + \sum_{i=1}^{k}(12 + \sum_{t \in terms(f_i)} (7 + 2|vars(t)|))$$

From the bottom up, the constraints are then removed one by one and only added again if they are not implied by the remaining set of constraints as determined by **why**.

Example 3.4.1 (Simplify)**.** *As an example we give the **why** mapping after one iteration of Blackbox, Whitebox and SimpleSubstitution after sorting, where the mapping is illustrated referencing the constraint id instead of the constraint itself. Note that $a^3 > 0$ was automatically converted to $a > 0$ and is thus handled as a given.*

id	constraint	why
7	$c > 0$	*given*
12	$c \neq 0$	$7, 4 \wedge 10 \wedge 16, 4 \wedge 11$
17	$c(c-1) \neq 0$	$4 \wedge 8 \wedge 16$
20	$c(c-1) < 0$	$4 \wedge 13 \wedge 16$
18	$c(c+1) \neq 0$	$4 \wedge 9 \wedge 16$
19	$c(c+1) > 0$	$4 \wedge 15 \wedge 16$
22	$c^2(c-1)(c+1)$	$0 \wedge 4 \wedge 16$
21	$c^3(c-1)(c+1)^2 < 0$	$1 \wedge 4 \wedge 16$
5	$a > 0$	*given*
10	$a \neq 0$	$5, 4 \wedge 12 \wedge 16$
9	$a + c^2 \neq 0$	$0, 1, 4 \wedge 16 \wedge 18$
15	$a + c^2 > 0$	$0 \wedge 13, 5, 6 \wedge 16, 4 \wedge 16 \wedge 19$
6	$b < 0$	*given*
11	$b \neq 0$	$6, 4 \wedge 12$
13	$a^2 + b < 0$	$1 \wedge 9, 0 \wedge 15, 4 \wedge 16 \wedge 20$
8	$a^2 + b \neq 0$	$0, 1, 4 \wedge 16 \wedge 17$
16	$a + b = 0$	$2 \wedge 14$
4	$b + c = 0$	*given*
2	$a + b \leq 0$	*given*, 16
14	$a + b \geq 0$	$3 \wedge 9, 16$
0	$(a + c^2)(a^2 + b) < 0$	*given*, $13 \wedge 15, 4 \wedge 16 \wedge 22$
1	$(a + c^2)^2(a^2 + b) < 0$	*given*, $9 \wedge 13, 4 \wedge 16 \wedge 21$
3	$(a + c^2)^4(a + b) \geq 0$	*given*, $6 \wedge 14$

Starting at the bottom, we drop 3 and check if it is still implied. As 3 is still implied by $6 \wedge 14$ we do not add it again. In the same way we remove 1, 0, 14 and 2, as at least one of their implications is still fully contained. Constraint 4 has to be kept, as given does not suffice to be implied in this algorithm. Therefore we also know that we have to keep 7, 5 and 6 as they are not implied by anything else. We have to keep 16, as 2 (and 14) is not in the set any more. We continue this way until we reach the following conclusion:

Dropped:	$\{0, 1, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15\}$
Result set:	$\{4, 5, 6, 7, 16, 17, 18, 19, 20, 21, 22\}$

With further iterations we could simplify the constraints even more.

3.4.2 UNSAT Core

If we found UNSAT, we want to check if the SAT solver may find a different (partial) solution. To make sure that the current constraints leading to UNSAT are not repeated, the SAT solver learns the UNSAT core. As the SAT solver only knows of *given* constraints, it is important to give an UNSAT core consisting of the original constraints. For this purpose, it is possible to backtrack through the **why** mapping from FALSE until the necessary original constraints have been found. Brown et al. do not specify how to backtrack the implications. We implemented **why** as a mapping of a constraint to a vector of constraint sets. These vectors are filled in the order they are created. Therefore we always use the first implication of the vector to reach the

given constraints as fast as possible.

We go through **why** by starting with FALSE and selecting the first element of **why**($FALSE$). Then each derived constraint is also replaced with its first implication until all constraints are implied by *given*.

Example 3.4.2 (UNSAT Core)**.** *Let the following table be the result of the simplification:*

id	constraint	why
0	$xz^2w^3 \leq 0$	$given, 2 \wedge 5$
1	$(x+y)^3(z-1) < 0$	$given$
2	$z+1 > 0$	$given$
3	$w-2 \neq 0$	$given$
4	$x^2z - xw + 1 < 0$	$given, 8$
5	$(z+1)^3x \leq 0$	$given, 6$
6	$x = 0$	$2 \wedge 5$
7	$x^2z - xw + 1 > 0$	$6, 8$
8	$FALSE$	$4 \wedge 7$

Now we start with FALSE and backtrack until all used id's are implied by given:

$$\begin{aligned} & FALSE \\ \Rightarrow\ & 4 \wedge 7 \\ \Rightarrow\ & 4 \wedge 6 \\ \Rightarrow\ & 4 \wedge 2 \wedge 5 \end{aligned}$$

As $2, 4$ *and* 5 *are given, the returned UNSAT core is* $2 \wedge 4 \wedge 5$*, which is* $\{z+1 > 0, x^2z - xw + 1 < 0, (z+1)^3x \leq 0\}$.

Chapter 4

Incrementality & Backtracking

After knowing how the simplification algorithm designed by Brown et al. in [BVE20] works, we will now adapt this algorithm to comply with incrementality and backtracking. This means, that for incrementality we want to be able to add a new constraint after a subset was already processed and reuse the prior calculations. The goal in backtracking is to reach a valid state for the remaining constraints after one is removed. For example, in Blackbox we need to add a new row to the processed matrix and also reach a valid matrix state, where all rows are implied in backtracking. For this, we will first give the general algorithm used as the basis for backtracking, before considering the adaptions necessary for Blackbox (Algorithm 3), Whitebox (Algorithm 4), SimpleSubstitution (Algorithm 6) and Simplify (Algorithm 7).

Definition 4.0.1. *Let C be the set of constraints added to the algorithm. This can be either as a given constraint or a newly derived constraint. We denote a constraint in this set as $c \in C$.*
Let F be the set of constraints processed before. Thus, after one call to Simplify $F := F \cup C$.

4.1 General Backtracking

When a *given* constraint c is backtracked, we have to make sure that all constraints depending on this constraint are also backtracked. If a constraint is implied by another constraint which does not depend on c, it does not have to be backtracked.
The goal of general backtracking is to clear the **why** map in a way that there is no residue of a backtracked constraint any more. For this, there are a few main steps:

1. *given* is removed from $\mathbf{why}(c)$: $\mathbf{why}(c) = \mathbf{why}(c)/\{given\}$.
2. If $\mathbf{why}(c) = \emptyset$ we remove all implications containing c. Otherwise, we stop backtracking if step 4 does not apply.
3. We drop all constraints for which **why** is mapped to $\emptyset$. These are not implied by any conjunction of constraints any more.
4. There may be loops of constraints implying each other. This may happen e.g. in line 15 of StrictBlackbox (Algorithm 1). Thus, if a constraint is only implied a loop with no path to a *given* constraint, it has to be backtracked.

5. All removed constraints are backtracked the same way.

Example 4.1.1 (Backtracking)**.** *Backtracking constraint 2 from Example 3.4.2, we also backtrack 6, as it solely depends on 2. The corresponding* ***why*** *mapping looks as follows:*

id	constraint	why
0	$xz^2w^3 \leq 0$	*given*
1	$(x+y)^3(z-1) < 0$	*given*
3	$w - 2 \neq 0$	*given*
4	$x^2z - xw + 1 < 0$	*given*, 8
5	$(z+1)^3x \leq 0$	*given*
7	$x^2z - xw + 1 > 0$	8
8	$FALSE$	$4 \wedge 7$

As $FALSE$ is still included, we may try to find the new UNSAT core:

$$\begin{aligned} &FALSE \\ &\Rightarrow 4 \wedge 7 \\ &\Rightarrow 4 \wedge 8 \end{aligned}$$

We reached a loop. FALSE implies itself, even though it is not a given. If we find this kind of loop (not necessarily starting from FALSE), we backtrack the looping constraint. Here we could start from constraint 8, leading to constraint 7 being backtracked as well.

id	constraint	why
0	$xz^2w^3 \leq 0$	*given*
1	$(x+y)^3(z-1) < 0$	*given*
3	$w - 2 \neq 0$	*given*
4	$x^2z - xw + 1 < 0$	*given*
5	$(z+1)^3x \leq 0$	*given*

The final table does not contain any loops.

Example 4.1.2. *Let $F = \{x \neq 0, xy < 0\}$ be the current set of constraints and $c = (x \neq 0)$. Then it is still valid that* ***why****$(x \neq 0) = \{xy < 0\}$. Thus, we have to keep c and just consider it as a derived constraint instead of given.*

This approach is the basis for our backtracking. In the following, we will explain the additional steps necessary to use the algorithms on incremental inputs and backtrack the simplification algorithm. Note that every following adaption only happens, if c is not implied anymore. Otherwise, we stop after removing *given*.

4.2 Blackbox

From all the algorithms, Blackbox is the hardest to adapt to Incrementality and Backtracking. For instance, the differing normalization for strict and non-strict variables cannot handle changes in strictness.

Example 4.2.1 (Strictness Change)**.** *Let* $F = \{x^2y \leq 0, x > 0\}$*. Then* $X_s = \{x\}$*,* $X_n = \{y\}$ *and we reach the following matrix:*

x	σ	y
0	1	1
1	0	0

As x *is strict,* x^2 *is interpreted as* x^0*. Thus, this reads as* $y \leq 0 \wedge x > 0$*.*
Now we backtrack $x > 0$*. Then* $X_s = \emptyset, X_n = \{x, y\}$*. Thus, we move* x *behind* σ *and remove the second row:*

σ	x	y
1	0	1

What we want to read is $x^2y \leq 0$*, as* $x \neq 0$ *is not implied any more. But what we read is* $y \leq 0$*. This is because* x *was strict before and there is no way of knowing if the* 0 *is actually a* 0 *or a former* 2*.*

To deal with this problem, we use a different normalization when writing into the matrix, than reading from it. $\mathcal{N}$ is the normalization as described in Section 3.1, while $\mathcal{N}'$ is the normalization formerly used for non-strict constraints. Now we use $\mathcal{N}$ for writing and $\mathcal{N}'$ for reading from a matrix.

Example 4.2.2 (Strictness Change)**.** *Let* $F = \{x^2y \leq 0, x > 0\}$*. Then* $X_s = \{x\}$*,* $X_n = \{y\}$ *and we reach the following matrix:*

x	σ	y
2	1	1
1	0	0

This reads as $y \leq 0 \wedge x > 0$*.*
Now we backtrack $x > 0$*. Then* $X_s = \emptyset, X_n = \{x, y\}$*. Thus, we move* x *behind* σ *and remove the second row:*

σ	x	y
1	2	1

This reads as $x^2y \leq 0$*.*

For the following adaptions we also define:

Definition 4.2.1. *Let* k *be the number of rows of the matrix. Then*

$O_1 : \{1,...,k\} \mapsto \mathcal{C}$	*is a mapping of a row of the matrix to the formula which was originally written to that line, normalized with* $\mathcal{N}'$*.*
$O_2 : \{1,...,k\} \mapsto \mathcal{C}$	*is* O_1*, but normalized with* $\mathcal{N}$*.*

For each newly created row i (e.g. MinWtBasis), we add

$$\mathbf{why}(O_1[i]) = \mathbf{why}(O_1[i]) \cup \{O_2[i]\}$$
$$\mathbf{why}(O_2[i]) = \mathbf{why}(O_2[i]) \cup \{O_1[i] \wedge \bigwedge_{v \in X_s, v \in Vars(O_1[i]), v \notin Vars(O_2[i])} v \neq 0\}$$

4.2.1 Incrementality

First, the new constraints are normalized as usual. Note that even though we adapted the normalization, we still also imply the constraints following from the initial normalization as before, to ensure that everything used at a later point was implied. The first significant difference is the adaption of the matrix.

1. c contains a variable which is not included in the matrix:
   We add a new zero-column in the matrix, corresponding to the constraints' strictness, before adding a new row for the new constraint.

2. c is strict and contains a variable $x \in X_n$:
   We move the corresponding column in front of σ and imply all constraints c' which are impacted. Let c'' be the impacted constraints after x was added to X_s and moved.
   $$\mathbf{why}(c'') = \mathbf{why}(c'') \cup \{c' \wedge x \neq 0\}$$
   Additionally, O_2 is updated and we again imply O_1 by the new O_2 and the strictness of the variables.

Example 4.2.3 (Incremental Blackbox)**.** *Let the current state of the matrix be*

idx	x_1	x_2	σ	x_3
0	1	2	0	1

with $O_1[0] = x_1x_2^2x_3 \geq 0, O_2[0] = x_1x_3 \geq 0$.
Now we add the constraint $w^4z > 0$*. This leads to* $O_1[1] = w^2z > 0, O_2[1] = z > 0$.
We update ***why*** *as follows:*

$$\begin{aligned}
&\boldsymbol{why}(w^4z > 0) = \boldsymbol{why}(w^4z > 0) \cup \{w^2z > 0\}\\
&\boldsymbol{why}(w^2z > 0) = \boldsymbol{why}(w^2z > 0) \cup \{w^4z > 0, z > 0 \wedge w \neq 0\}\\
&\boldsymbol{why}(z > 0) = \boldsymbol{why}(z > 0) \cup \{w^2z > 0\}\\
&\boldsymbol{why}(w \neq 0) = \boldsymbol{why}(w \neq 0) \cup \{w^4z > 0, w^2z > 0\}\\
&\boldsymbol{why}(z \neq 0) = \boldsymbol{why}(z \neq 0) \cup \{w^4z > 0, w^2z > 0, z > 0\}
\end{aligned}$$

The transformed matrix is

idx	x_1	x_2	x_4	x_3	σ
0	1	2	0	1	0
1	0	0	2	1	0

Thus $O_2[0]$ *changed to* $O_2[0] = x_1x_3 > 0$ *and*

$$\begin{aligned}
&\boldsymbol{why}(O_2[0]) = \boldsymbol{why}(O_2[0]) \cup \{O_1[0] \wedge x_1 \neq 0 \wedge x_2 \neq 0 \wedge x_3 \neq 0\}\\
&\boldsymbol{why}(O_1[0]) = \boldsymbol{why}(O_1[0]) \cup \{O_2[0]\}.
\end{aligned}$$

Next we consider changes to StrictBlackbox. As the matrix before adding new constraints was already in reduced row-echelon form, it is not necessary to perform Gaussian elimination on all rows. We also do not need to find more deductions for known constraints, as they are already solved at the backend. Instead, let I_1 be the set of

Algorithm 8 StrictBlackboxIncr, Adaption to Algorithm 1

Input: M: matrix over normalized vectors with strict variables x_1,...,x_s and non-strict variables x_{s+1},...,x_n, **why** as defined in Definition 3.0.1, I_1 *set of new strict rows of* M, K *original unit matrix*

Output: (M, is_sat) so that either $\Gamma^{-1}(M)$ is UNSAT and is_sat = FALSE or $\Gamma^{-1}(M)$ is SAT and $is\_sat = TRUE$

1: $I_2 := \{j \in \mathbb{N} \mid j \notin I_1 \wedge j \text{ strict row of } M\}$
2: produce M' by Gaussian elimination of *rows* I_1 of M and *row reduction of the result by rows* I_2
3: update K, so that $K \cdot M_O = M'$, where M_O is the original matrix without any Gaussian elimination.
4: $K' :=$ matrix so that $K' \cdot M = M'$
5: **for** $i \in I_1$ **do**
6: $\quad N_1 :=$ set of column indices of non-zero entries *of row* i *of* K
7: $\quad N_2 :=$ set of column indices of non-zero entries *of row* i *of* K'
8: $\quad E_1 := \{\Gamma(O_2[i]) \mid i \in N_1\}$
9: $\quad E_2 := \{M_{i,\_} \mid i \in N_2 \wedge i \notin I_1\} \cup \{\Gamma(O_2[i]) \mid i \in N_2 \wedge i \in I_1\}$
10: $\quad w :=$ row i of M'
11: $\quad$ **if** w is [0,...,0,1] $\oplus$ [0,...,0] **then**
12: $\quad\quad$ add $\bigwedge_{u \in E_1} \Gamma^{-1}(u) \Rightarrow$ FALSE to **why**
13: $\quad\quad$ add $\bigwedge_{u \in E_2} \Gamma^{-1}(u) \Rightarrow$ FALSE to **why**
14: $\quad\quad$ **return** M', FALSE
15: $\quad$ **else if** w is not zero **then**
16: $\quad\quad$ add $\bigwedge_{u \in E_1} \Gamma^{-1}(u) \Rightarrow \Gamma^{-1}(w)$ to **why**
17: $\quad\quad$ for each $v \in E_1$ add $\bigwedge_{u \in E-\{v\}\cup\{w\}} \Gamma^{-1}(u) \Rightarrow \Gamma^{-1}(v)$ to **why**
18: $\quad\quad$ add $\bigwedge_{u \in E_2} \Gamma^{-1}(u) \Rightarrow \Gamma^{-1}(w)$ to **why**
19: $\quad\quad$ for each $v \in E_2$ add $\bigwedge_{u \in E-\{v\}\cup\{w\}} \Gamma^{-1}(u) \Rightarrow \Gamma^{-1}(v)$ to **why**
20: $\quad$ **else**
21: $\quad\quad$ for each $v \in E_1$ add $\bigwedge_{u \in E-\{v\}} \Gamma^{-1}(u) \Rightarrow \Gamma^{-1}(v)$ to **why**
22: $\quad\quad$ for each $v \in E_2$ add $\bigwedge_{u \in E-\{v\}} \Gamma^{-1}(u) \Rightarrow \Gamma^{-1}(v)$ to **why**
23: **return** $M', TRUE$

row-indices of new strict rows and I_2 be the set of row indices of all other strict rows. Then we perform Gaussian elimination on the rows of I_1, before row-reducing them by the rows of I_2.

Now we define two kinds of implications. The first one implies interdependencies for original constraints (i.e. O_2). Thus, the rows of the existing transformation matrix for the rows used for creating w are added to the row w. Secondly, we imply w by other derived constraints. K' is a new transformation matrix, showing which of the current rows sum to w. For new rows, we always use O_2, as they may have been changed as well.

Example 4.2.4 (Incremental StrictBlackbox)**.** *Let rows* $0, 1, 2$ *be given and rows* $3, 4$ *be new:*

idx	x_1	x_2	x_3	x_4	σ
0	0	0	1	0	0
1	1	2	0	1	0
2	0	0	2	1	0
3	1	1	2	0	1
4	0	1	0	1	1

Row 2 was derived using row 0 and $O_2[2] = x_3x_4 > 0$*. Now first we perform Gaussian elimination on the new rows:*

idx	x_1	x_2	x_3	x_4	σ
0	0	0	1	0	0
1	1	2	0	1	0
2	0	0	2	1	0
3	1	2	2	1	0
4	0	1	0	1	1

Next, the resulting rows are row reduced by rows $0, 1, 2$

idx	x_1	x_2	x_3	x_4	σ
0	0	0	1	0	0
1	1	2	0	1	0
2	0	0	2	1	0
3	2	2	2	2	0
4	0	1	2	2	1

The deduced constraint is $x_2 < 0$*. As the new row 3 is a "zero"-row, we found a new deduction for the constraint. Now for the implications we have the sets*

$$
\begin{aligned}
E_{31} &= \{x_1x_2 < 0, x_2x_4 < 0, x_1x_4 > 0\} \\
E_{41} &= \{x_2x_4 < 0, x_3x_4 > 0, x_3 > 0\} \\
E_{32} &= \{x_1x_2 < 0, x_2x_4 < 0, x_1x_4 > 0\} \\
E_{42} &= \{x_2x_4 < 0, x_4 > 0\}
\end{aligned}
$$

Thus we derived

$$\begin{aligned}
&\textbf{\textit{why}}(x_1x_2 < 0) = \textbf{\textit{why}}(x_1x_2 < 0) \cup \{x_2x_4 < 0 \wedge x_1x_4 > 0\}\\
&\textbf{\textit{why}}(x_2x_4 < 0) = \textbf{\textit{why}}(x_2x_4 < 0) \cup \{x_1x_2 < 0 \wedge x_1x_4 > 0\}\\
&\textbf{\textit{why}}(x_1x_4 < 0) = \textbf{\textit{why}}(x_1x_4 < 0) \cup \{x_2x_4 < 0 \wedge x_1x_2 > 0\}
\end{aligned}$$

$$\begin{aligned}
&\textbf{\textit{why}}(x_2 < 0) = \textbf{\textit{why}}(x_2 < 0) \cup \{x_2x_4 < 0 \wedge x_3x_4 > 0 \wedge x_3 > 0, x_2x_4 < 0 \wedge x_4 > 0\}\\
&\textbf{\textit{why}}(x_2x_4 < 0) = \textbf{\textit{why}}(x_2x_4 < 0) \cup \{x_2 < 0 \wedge x_3x_4 > 0 \wedge x_3 > 0\}\\
&\textbf{\textit{why}}(x_3x_4 > 0) = \textbf{\textit{why}}(x_3x_4 > 0) \cup \{x_2 < 0 \wedge x_2x_4 < 0 \wedge x_3 > 0\}\\
&\textbf{\textit{why}}(x_3 > 0) = \textbf{\textit{why}}(x_3 > 0) \cup \{x_2 < 0 \wedge x_2x_4 < 0 \wedge x_3x_4 > 0\}\\
&\textbf{\textit{why}}(x_2x_4 < 0) = \textbf{\textit{why}}(x_2x_4 < 0) \cup \{x_2 < 0 \wedge x_4 > 0\}\\
&\textbf{\textit{why}}(x_4 > 0) = \textbf{\textit{why}}(x_4 > 0) \cup \{x_2 < 0 \wedge x_2x_4 < 0\}
\end{aligned}$$

Finally, for MinWtBasis, we simply change the selection of w, so that $S(w) \subseteq S(c)$ or $S(c) \subseteq S(w)$. Additionally, we do not actually delete rows in line 14, but save a list of rows to ignore in future steps together with the row leading to its removal.

Definition 4.2.2 (Removed Set)**.** *We define $L_R = \{(w_R, w) \in \mathbb{N}^2 \mid w$ leads to removal$\}$. In case a selected w is not returned to the result set in MinWtBasis (Algorithm 2), the tuple (w,w) is added to L_R.*

Example 4.2.5 (Incremental MinWtBasis)**.** *Let the matrix be*

idx	x_1	x_2	σ	x_3	x_4
0	0	0	1	0	1
1	2	0	1	1	2
2	0	2	0	1	0

with the new row being 2. Then only rows 1 and 2 can be selected for w, as $S(0) = \{x_4\} \not\subseteq \{x_3\} = S(2)$ and $S(2) \not\subset S(0)$. As $|S(1)|$ is maximal, we select w as row 1. We can derive [0,...,0,1] $\oplus$ [0,..,0] *(mod 2) using rows 1 and 2. Thus, we reach the matrix*

idx	x_1	x_2	σ	x_3	x_4
0	0	0	1	0	1
1	2	0	1	1	2
2	0	2	0	1	0
3	2	0	1	2	2

with $L_R = \{(1,1),(2,1)\}$ and

$$\begin{aligned}
&O_1[3] = x_1^2x_3x_4 = 0\\
&O_2[3] = x_3x_4 = 0
\end{aligned}$$

$$\begin{aligned}
&\textbf{\textit{why}}(O_1[3]) = \textbf{\textit{why}}(O_1[3]) \cup \{O_2[3]\}\\
&\textbf{\textit{why}}(O_2[3]) = \textbf{\textit{why}}(O_2[3]) \cup \{O_1[3] \wedge x_1 \neq 0\}
\end{aligned}$$

4.2.2 Backtracking

Backtracking in Blackbox focuses mainly on two aspects: Changes in strictness and how to deal with rows created using the backtracked constraint c. Let I be the set of row indices impacted by a change, identifiable by a 1 entry in the transformation matrix. First of all, if c is of the form $x \neq 0$, we backtrack the strictness of x. Thus x is moved behind σ and all rows with an entry $\neq 0$ are added to I. Furthermore, all rows, for which $\Gamma^{-1}(O_1(i)) = c$ or $\Gamma^{-1}(O_2(i)) = c$ have to backtracked and erased in the following way: All rows j, for which the entry in the transformation matrix $K[j,i] = 1$, have to be reset to their original form. Thus, if M_j is the j'th row of M, $M_j = \Gamma(O_1[j])$. The j'th row of K is reset as well and j is added to I. If only $\Gamma^{-1}(M_i) = c$, the i'th row itself is reset in the same way as j before and added to impacted. Now for all rows in I O_2 is adapted to the new level of strictness and implied using O_2:

$$\mathbf{why}(O_1[i]) = \mathbf{why}(O_1[i]) \cup \{O_2[i]\}$$
$$\mathbf{why}(O_2[i]) = \mathbf{why}(O_2[i]) \cup \{O_1[i] \wedge \bigwedge_{v \in X_s, v \in Vars(O_1[i]), v \notin Vars(O_2[i])} v \neq 0\}$$

All reset rows are considered as new rows for the next iteration of Blackbox. As a last step, we remove all tuples from L_R which contain a backtracked constraint at any position.

Example 4.2.6. *Let the following matrix be given as well as* $O_1[4] = x_0^2 x_1 x_2 \geq 0$ *and* $L_R = \{(1,1),(2,1)\}$.

idx	x_0	x_1	x_2	σ	x_3	x_4
0	0	1	2	1	0	0
1	0	2	0	1	1	2
2	0	0	2	0	1	0
3	0	2	0	1	2	2
4	2	2	1	1	0	0

We backtrack $\{x_2 \neq 0, x_3 x_4 = 0, x_1^2 x_3 x_4^2 \leq 0, x_1 < 0\}$. *Thus row 4 is reset using* O_1, *rows 0, 2 and 3 are erased and* x_2 *is moved:*

idx	x_0	x_1	σ	x_2	x_3	x_4
1	0	2	1	0	1	2
4	2	1	0	1	0	0

$$O_2[4] = x_1 x_2 \geq 0$$

$$\boldsymbol{why}(O_1[4]) = \boldsymbol{why}(O_1[4]) \cup \{O_2[4]\}$$
$$\boldsymbol{why}(O_2[4]) = \boldsymbol{why}(O_2[4]) \cup \{O_1[4] \wedge x_1 \neq 0\}$$

As row 1 was not impacted by any change, there is no need for an implication update. Lastly, as row 2 was backtracked, $L_R = \emptyset$.

Algorithm 9 BlackboxBacktracking, Adaption of Algorithm 3

Input: C: set of real polynomial constraints to backtrack, M matrix, K unit matrix, O_1, O_2, X_s, X_n over the variables $\{x_1,...,x_n\}$ as defined above
Output: M, **why***

$I := \emptyset$
for $c \in C$ **do**
  $c' := \mathcal{N}(c)$
  if c' of form $x_i \neq 0$ **then**
    $I := I \cup \{i \in \mathbb{N} \mid M_{i,j} > 0, j \in \{1,...,n\}\}$
    move x_i behind σ
    $X_s = X_s/\{x_i\}$, $X_n = X_n \cup \{x_i\}$
  else
    $I_B := \{i \in \mathbb{N} \mid O_1[i] = c' \vee O_2[i] = c'\}$
    $I_S := I_S \cup \{i \in \mathbb{N} \mid M_i = c'\}$
    $I_S := I_S \cup \{i \in \mathbb{N} \mid \exists j \in I_B : K[i,j] = 1\}$
    for $i \in I_S$ **do**
      $M_i = \Gamma(O_1[i])$
    $I := I \cup I_S$
for $i \in I$ **do**
  $O_2[i] = \Gamma^{-1}(M_i)$
  why$(O_1[i])$ = **why**$(O_1[i]) \cup \{O_2[i]\}$
  why$(O_2[i])$ = **why**$(O_2[i]) \cup \{O_1[i] \wedge \bigwedge_{v \in X_s, v \in Vars(O_1[i]), v \notin Vars(O_2[i])} v \neq 0\}$
return M, **why***

4.3 Whitebox

Opposed to Blackbox, Whitebox can be used with Incrementality and Backtracking with only few changes.

4.3.1 Incrementality

Whitebox is originally executed on all factors and variables. As the loops are still executed on the polynomials of the factors, no used algorithm like PolySign (Algorithm 12) or DeduceSignExplain (Algorithm 17) have to be changed. Thus, our goal is to restrict for which factors the two loops are executed. As a basis for this restriction, we exploit the fact that derivations are not necessary for known constraints, since these have either already been solved in the backend or are implied by constraints for which this is the case. We adapt three parts:

1. How we update the variable signs saved in α.
2. The definition of the set of factors and variables (P), for which the loops are executed.
3. For which combination of polynomials the second loop should be executed, which derives a sign for some $p \in P$ knowing the sign of some $q \in P$ using $p + tq$.

Let C be the set of newly considered constraints. First, we update α. Let $c \in C$ be a new constraint and a (possible) sign condition with $x\sigma 0$. Then $\alpha[x] = \sigma$ if $\sigma \prec \alpha[x]$.

Example 4.3.1 (Update α)**.** *Let $\alpha[x] = GEOP$ be the current state of α and $x > 0$ be a new constraint. Then we update $\alpha[x] = GTOP$.*

Next, instead of only defining P, we define two sets P_1 and P_2. P_1 is the main set, containing all new factors and all variables for which α was changed. P_2 on the other hand, is only used for the second loop and consists of all other factors and variables normally contained in P.

Example 4.3.2 (Factor Sets)**.** *Let $\{x^2 + y, z - y, x\}$ be the currents factors and $\{z + 1\}$ be the new factors. Additionally, $\alpha[x]$ changed. Then $P_1 = \{z + 1, x\}$ and $P_2 = \{x^2 + y, z - y, y, z\}$*

Now the first loop is executed for all $p \in P_1$. Meanwhile, the second loop is executed

1. for all $p \in P_1, q \in P_1$ with $p \neq q$ and
2. for all $p \in P_1, q \in P_2$.

With this approach, we derive as much as possible about new constraints (P_1) without having unnecessary derivations for known constraints.

4.3.2 Backtracking

Building on our General Backtracking (Section 4.1), only α needs to be updated for Whitebox. If $c = (x\sigma 0)$ is backtracked and $\alpha[x] = \sigma$, then $\alpha[x]$ is set to the strongest sign found in the remaining constraints. If no other sign constraint exists for x, $\alpha[x] = ALOP$ applies.

Example 4.3.3 (Backtrack α)**.** *Let $F = \{x^2 + y \leq 0, y \leq 0, y < 0, x > 0\}$ be the current set of constraints and $\alpha[y] = LTOP$ and $\alpha[x] = GTOP$. Now we backtrack $\{y < 0, x > 0\}$.*
As $y \leq 0$ still remains, we update $\alpha[y] = LEOP$. For x on the other hand, there is no remaining sign condition. Thus $\alpha[x] = ALOP$.

All factors impacted by a variable change are added to P_1 for the next iteration of incremental Whitebox.

4.4 Simple Substitution

4.4.1 Incrementality

Simple Substitution does not need much adaption to be incremental. First, we update *sub* and *reason* in Algorithm 5, using only the new constraints (given or derived). Then we distinguish two steps:

1. If *sub* and *reason* changed, we execute SimpleSubstitution for all prior constraints which are impacted by this change.

   Example 4.4.1. *Let $F = \{x - y = 0, y^2 + z \leq 0\}$ and we add $y - z = 0$. Then we change from $sub[x] = y, reason[x] = \{x - y = 0\}$ to $sub[x] = z, sub[y] = z, reason[x] = \{x - y = 0, y - z = 0\}, reason[y] = \{y - z = 0\}$.*
   Thus, we reach $F = \{x - y = 0, y - z = 0, z^2 + z \leq 0\}$

2. SimpleSubstitution is executed for all new constraints which were not used to adapt *sub* and *reason*.

   Example 4.4.2. *Let* $F = \{x - y = 0, y^2 + z \leq 0\}$ *and* $sub[x] = y, reason[x] = \{x - y = 0\}$ *and we add* $x + y^2 \geq 0$. *Then sub and reason are not adapted, as the new constraints is not a linear equation.*
   Thus, we reach $F = \{x - y = 0, y^2 + z \leq 0, y + y^2 \geq 0\}$.

It is important to list the constraints in *reason* in the right order, as this is required for backtracking.

4.4.2 Backtracking

Algorithm 10 BacktrackSubstitutionMapping

Input: c, a constraint of the form $ax_i + bx_j = 0$ with $b \neq 0 \wedge a \neq 0$, $V, sub, reason$ so that for all $x \in V : [reason[x] \Rightarrow x = sub[x]]$
Output: $(V, sub, reason)$ so that for all $x \in V : reason[x] \Rightarrow x = sub[x]$

```
1:  for v ∈ V : c ∈ reason[v] do
2:      R := [r_1,...,r_n] = reason[v]
3:      if r_1 = c then
4:          Remove v from V, sub, reason
5:          continue
6:      while r_{n+1} ≠ c do
7:          x'_i ∈ Vars(r_n) ∧ x'_i ∈ V
8:          x'_j ∈ Vars(r_n) ∧ x'_i ≠ x'_j
9:          sub[v] = sub[v]/sub[x'_i] x'_j
10:         Remove r_n from reason[v] and update r_n
11: return (V,sub,reason)
```

In backtracking, we want to derive the new *sub* and *reason* maps. For this, we traverse *reason* in reverse order and revert the substitution until we reach the first constraint before c. Note that we only consider constraints of the form $ax_i + bx_j = 0$ for backtracking, as all other constraints would not be used in *reason*. For $x_j = 1$, this also depicts constraints of the form $ax + c = 0$.

Lemma 4.4.1. *Let* $c = (a_i x_i + b_i y_i = 0)$ *and* $a_i, b_i \neq 0$.
Let $sub[x_k] = -\frac{b_j}{a_j} y_i$ *and* $reason[x_k] = [f_1,...,f_n,c]$. *Now let* $reason'[x_k] = [f_1,...,f_n]$.
Then $sub'[x_k] = \frac{sub[x_k]}{sub[x_i]} x_i$.

Proof. Let $ax + by = 0$ and $cy + dz = 0$ be constraints used for SubstitutionMapping.
Then $sub[y] = -\frac{d}{c} z$ and $sub[x] = -\frac{b}{a}(-\frac{d}{c} z) = -\frac{b}{a} sub[y]$ with $reason[y] = [cy + dz = 0]$ and $reason[x] = [ax + by = 0, cy + dz = 0]$.
We want to reach $sub[x] = -\frac{b}{a} y$ with $reason[x] = [ax + by = 0]$.
Now $\frac{sub[x]}{sub[y]} y = \frac{-\frac{b}{a}(-\frac{d}{c} z)}{-\frac{d}{c} z} y = -\frac{b}{a} y$. □

Example 4.4.3 (Backtrack SimpleSubstitution)**.** *Let* $sub[x] = -20z, reason[x] = \{6x + 12y = 0, -y + 10z = 0\}, sub[y] = 10z, reason[y] = \{-y + 10z = 0\}$. *Now we*

backtrack $-y + 10z = 0$.

$$sub[x] = \frac{sub[x]}{sub[y]}y = \frac{-20z}{10z}y = -2y$$
$$reason[x] = \{6x + 12y = 0\}$$

sub$[y]$ *and reason*$[y]$ *are deleted and* y *is dropped from* V.

Note that this does not apply to any $sub[x] = 0$, as division by 0 is prohibited. Thus, in the case of $sub[x] = 0$, we traverse $reason[x]$ in chronological order and substitute accordingly until we reach c.

4.5 Simplify

Lastly, Simplify is only adapted slightly. As the backend is also incremental, all constraints that were already transmitted to the backend have been solved at this point. Thus, it is more complex to backtrack contained constraints and to add simplifications of these constraints, than to just add simplifications for new constraints. Therefore, we execute the test of implication only for constraints which are not known to the backend.

Additionally, sorting is done in an adjoining way. This means that all newly added and newly derived constraints are sorted separately from the known constraints and then appended, so that the known constraints are on top. This is because all known constraints are either in the backend or implied by one of the backend constraints and will be removed during Simplify in any case. Having these constraints at the top has the advantage that new constraints can be implied by known constraints at any time.

Example 4.5.1. *Let* $L_{old} = \{x^2 + y \leq 0, x \neq 0, z \neq 0, y \leq 0, x + z < 0\}$, $L = \{z > 0, x < 0\}$ *and* $L_B = \{x^2 + y \leq 0, x + z < 0\}$.

id	constraint	why
0	$x^2 + y \leq 0$	*given*, $2 \wedge 4$
2	$x \neq 0$	$1, 6$
3	$z \neq 0$	$1, 5$
4	$y \leq 0$	$0 \wedge 1$
1	$x + z < 0$	*given*
5	$z > 0$	*given*
6	$x < 0$	$1 \wedge 5$

We select $z > 0$ *to add to the backend, as* $x < 0$ *is implied. This example shows that even though more complex constraints may be implied by less complex constraints, it is too late, if the theory solver already solved it so far. If the order of the given constraints changes slightly, that already impacts how complex the constraints given to the backend are:*

id	constraint	why
1	$x \neq 0$	$0, 6$
2	$z \neq 0$	$0, 5$
0	$x + z < 0$	*given*
4	$y \leq 0$	$0 \wedge 3$
3	$x^2 + y \leq 0$	$given, 2 \wedge 4$
5	$z > 0$	*given*
6	$x < 0$	$0 \wedge 5$

Now in the first call of Simplify, 0 *is selected, as it is only implied by given. In the second call* 4 *is selected, as* 3 *is implied by* $2 \wedge 4$*. In the third call we select* 5*, which results in the set* $\{0, 4, 5\} = \{x + z < 0, y \leq 0, z > 0\}$ *opposed to* $\{x + z < 0, x^2 + y \leq 0, z > 0\}$.

Algorithm 11 SimplifyIncr (Adaption of Algorithm 7)

Input: **why** so that from given inequalities all others are derivable, L_{old}: sorted list of old constraints

Output: G: conjunction of inequality subset from which all other inequalities are derivable by deductions in **why**

1: $L :=$ list of all *new* constraints in **why**
2: sort L from simplest to most complex
3: *Append L to* L_{old}
4: $L_B :=$ *list of all constraints known to backend*
5: $F_R :=$ conjunction of all reason implications except givens in propositional form
6: $F_G :=$ conjunction of all constraints in **why** in propositional form
7: **for** A from L_{old} in reverse order, $A \notin L_B$ **do**
8: remove propositional form of A from F_G
9: if A not implied by $F_R \wedge F_G$ add propositional form of A back to F_G

Chapter 5

Experiments

Finally, we want to evaluate the introduced methods. They were implemented as a module in SMT-RAT, a solver which was mainly developed for QFNRA and QFNIA formulas [CLJÁ12]. For arithmetic computations, the own library CArL was implemented [KÁ18]. From the variety of solver modules that SMT-RAT offers, we will use an implementation of the "Cylindrical Algebraic Covering" method [ÁDEK21] for the experiments, which we will refer to as Covering.
For compilation, g++ version 11.3.0 was used. The code was executed on the RWTH High Performance Computing Cluster containing 2 Intel Xeon Platinum 8160 Processors "SkyLake" (2.1 GHz, 24 cores each) and 1250 nodes with 48 cores and 192 GB main memory each (4 GB main memory per core). Accumulated the number of cores is 60000 [clu].
All experiments are evaluated on the benchmark set SMT-LIB QF_NRA [smta], containing 12134 test instances of QFNRA formulas.
We use a time limit of 10 minutes and a memory limit of 4 GB.
If this memory limit is reached, we categorize that run as memout. Note that memouts are sometimes wrongly categorized as segfaults.

In the following sections, we will take a closer look at the results produced by Simplify with and without incrementality and backtracking. To analyse the impact of Simplify on the whole solver process, we consider Covering without Simplify as a comparison.

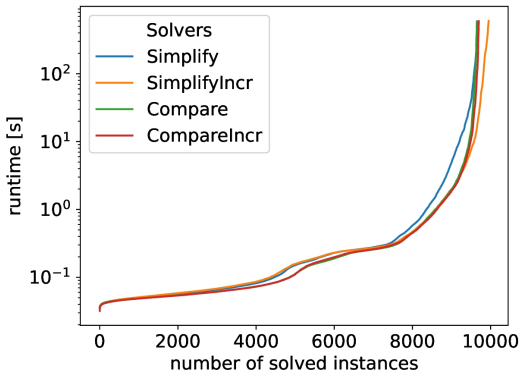
5.1 Result Comparison

We compare the results of Covering with (Simplify) and without (Compare) inprocessing. The solvers are each executed once incremental (Incr) and once without. Table 5.1 shows the results of SMT-RAT with Simplify (both incremental and non-incremental) and without Simplify. Note that at this point we set a limit of 3 iterations of the simplification algorithms.
Simplify solves more instances than the non-incremental comparison. There are more "sat" and "unsat" results and fewer "unknown". While the number of memouts (and segfaults) significantly decreased, the number of timeouts increased.
Incremental Simplify on the other hand solves more instances than any other considered approach. It has the highest number of "sat" and "unsat" and the lowest number of "unknown". Memouts and segfaults are slightly higher compared to Simplify, but

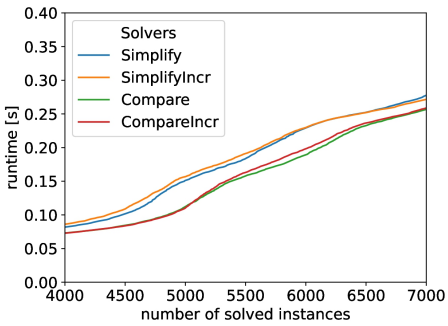
	CompareIncr	SimplifyIncr	Simplify	Compare	VB
sat	4919	4939	4900	4882	4957
unsat	4777	5011	4778	4766	5067
unknown	163	134	140	161	117
timeout	1367	1475	1887	1358	1630
memout	874	536	403	926	264
segfault	34	39	26	41	99
solved	9696	9950	9678	9648	10024

Table 5.1: Result comparison with and without Simplify, each incremental and non-incremental.

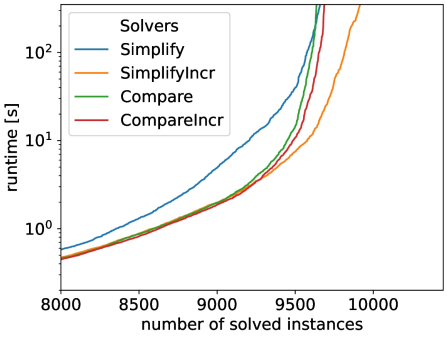
lower than CompareIncr. Opposed to this, the number of timeouts is lower than Simplify, but higher than CompareIncr.



(a) All



(b) Middle segment



(c) End segment

Figure 5.1: Performance profile of Simplify, SimplifyIncr, Compare and CompareIncr with logarithmic scale.

Fig. 5.1 shows the performance profile of each of these solvers. The plotted lines represent the number of solved instances vs. the cumulated runtime of the solved instances. First of all, we can see that the runtime behaviour is quite similar for all

solvers. Most instances can be solved fast, increasing only slowly, but a few hundred instances are more complex, taking almost all the given runtime. This change is relatively steep. Fig. 5.1 c) puts emphasis on the instances with a high runtime. Simplify and SimplifyIncr differ in the way that SimplifyIncr solves more instances, while Simplify partly has a less steep rise, showing that it has fewer instances solved quickly. b) zooms into the range of 4000 to 7000 solved instances. In this range, Simplify and Compare are almost identical to their incremental counterparts. Simplify and SimplifyIncr have a slightly higher runtime than the other solvers.

In the following, we will take a closer look at CompareIncr vs. SimplifyIncr and Simplify vs. SimplifyIncr.

5.2 SimplifyIncr vs. CompareIncr

We have established that SimplifyIncr performs better than CompareIncr considering the answers. Now we will first take a look at the runtimes before establishing how the answers changed from CompareIncr to SimplifyIncr.

5.2.1 Runtimes

Apart from the number of solved instances, the runtime is also an important assessment criterion. Fig. 5.2 shows the runtime of SimplifyIncr plotted against the runtime of CompareIncr with a logarithmic scale.

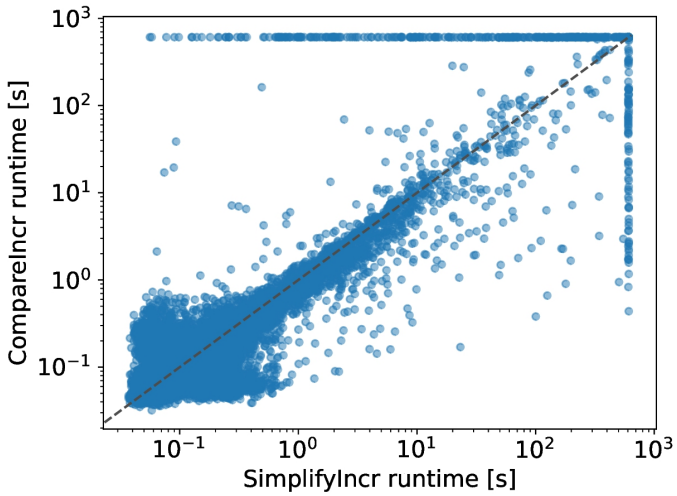


Figure 5.2: Runtime Comparison between SimplifyIncr and CompareIncr with logarithmic scale. Each dot represents one instance of the SMT-LIB benchmark set.

Note that memouts are automatically assigned a runtime of 605 s, considering the runtime limit of 600 s. Timeouts on the other hand are assigned a runtime of 610 s. Fig. 5.2 shows that mostly both solvers lead to similar runtimes. A large number of instances accumulate near the origin, which means that most instances can be solved quickly. On the other hand, there are only few solved instances with a high runtime,

without being categorized as memout or timeout. SimplifyIncr sometimes takes a little longer in the middle segment, which can be detected by the isolated dots on the right side of the middle line. CompareIncr regularly leads to timeouts/memouts when SimplifyIncr can solve the instances quickly. On the other hand, the timeouts/memouts of SimplifyIncr only start when CompareIncr needs a little more time than with the large number of simple instances. One reason why SimplifyIncr quickly solves instances that lead to timeouts or memouts in CompareIncr could be, that UNSAT has already been found in one of the algorithms described above (Chapter 3, Chapter 4).

Additionally to the general runtime, we would like to know how much the simplifications improved the runtime of the theory solver. Fig. 5.3 shows the runtime comparison of the backend for SimplifyIncr and Simplify for those instances which were solved and for which the backend was called at least once.

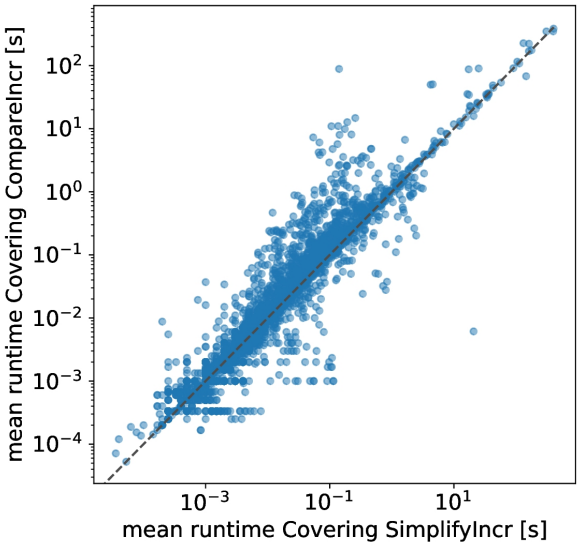


Figure 5.3: Comparison of backend runtime for SimplifyIncr and CompareIncr

Mostly all instances are oriented at the middle line, but clearly most instances are slightly faster in SimplifyIncr opposed to Compare. As this improvement is only slight and many instances are even slower in CompareIncr, the primary advantage of Simplify is not actually its simplification but the ability to identify UNSAT in a lot of cases.
As this runtime was measured in total, it is not clear, if Covering actually takes longer in CompareIncr or if it was just called less often.

Fig. 5.4 shows how often Covering was called for CompareIncr vs. SimplifyIncr. It is clear that SimplifyIncr calls Covering significantly less than CompareIncr. This underlines the point that the main advantage of SimplifyIncr is its solving capability. A more detailed analysis of the runtime of SimplifyIncr is given in Section 5.3.

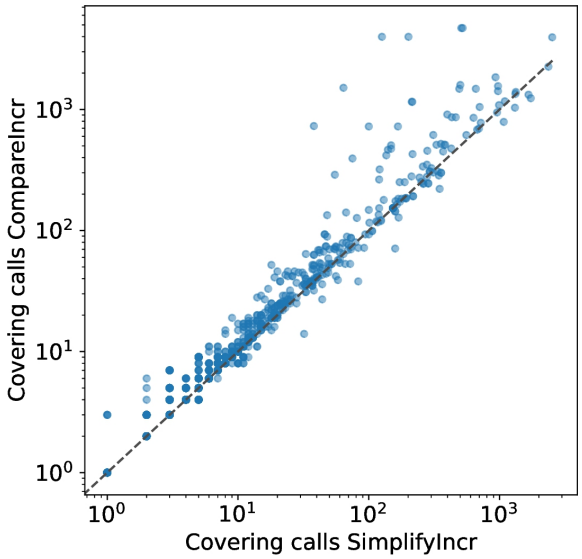


Figure 5.4: Number of backend calls compared for SimplifyIncr and CompareIncr with logarithmic scale

5.2.2 Answer Transitioning

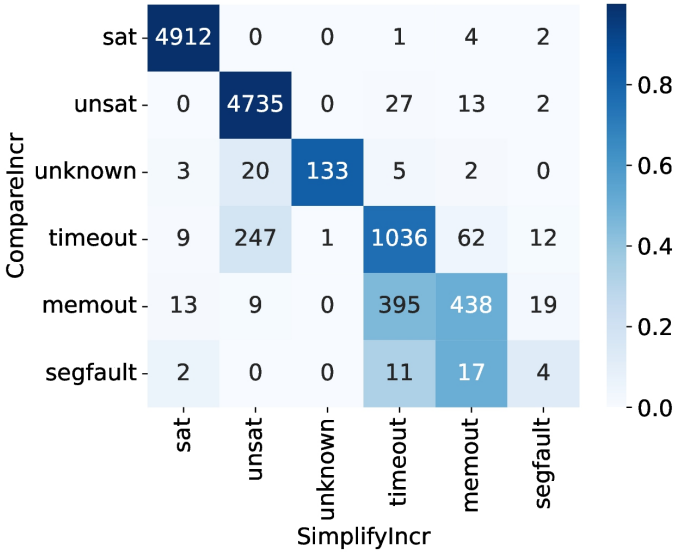


Figure 5.5: Transition of CompareIncr (y-axis) to SimplifyIncr (x-axis) normalized with colour scheme as percentage normalized to 100% per row

Additionally to the runtime, we also want to establish how the different answers changed. A theory could be that most of the memouts of CompareIncr became timeouts for SimplifyIncr. One question could be where the additionally solved instances

come from? There are 281 instances solved by SimplifyIncr but not by CompareIncr. As the number of memouts decreased, while the number of timeouts increased for SimplifyIncr, did memouts become timeouts?

Fig. 5.5 shows the number of instances transitioning from an answer for CompareIncr to an answer of SimplifyIncr. The colour scheme is normalized to show the percentage of transitions per row. The number of instances which had a valid answer for CompareIncr but transitioned to timeout, memout or segfault are exceedingly small. On the other hand, 20 instances, making up 12%, of unknowns and 247 instances, making up 18%, of timeouts became unsat. The number of newly found satisfiable instances is relatively small as well. Many memouts transitioned to timeouts, which is consistent with the changes in Table 5.1. That segfaults primarily became memouts and timeouts is expected, as these segfaults are wrongly categorized as segfaults and actually represent memouts.

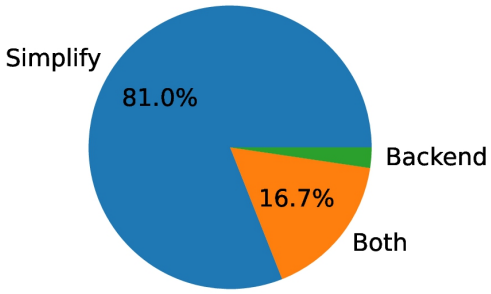


Figure 5.6: Number of instances solved by SimplifyIncr but not by CompareIncr with UNSAT result by Simplify or the theory solver

Fig. 5.6 only considers instances, which returned UNSAT at some point during inprocessing or the theory solver and shows how many of the instances newly solved by SimplifyIncr had UNSAT as a result from Simplify, the theory solver or both. The grand majority of 81% were only found to be UNSAT by Simplify. An additional 16.7% were found UNSAT by both. Thus 97.7% of newly solved instances are (partially) solved by Simplify. The remaining 2.3% were then solved only by the backend with the help of simplified constraints.

5.2.3 UNSAT Cores

How do the sizes of the UNSAT cores differ? Fig. 5.7 depicts the sizes of the UNSAT cores of SimplifyIncr and CompareIncr against each other, which was measured as the maximum size of the UNSAT cores of each instance. Considered are only cases in which both solvers created an unsat core.
The range of sizes varies between 1 and 45, where the maximum value of 45 is only

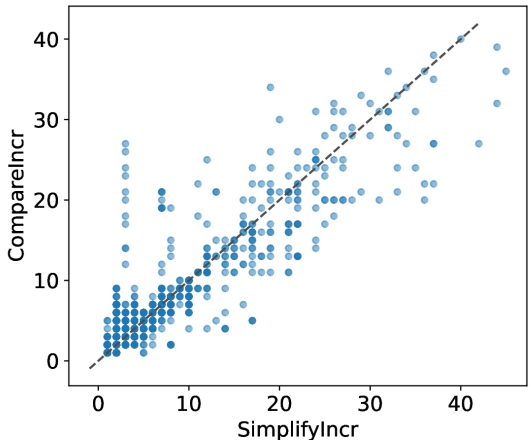


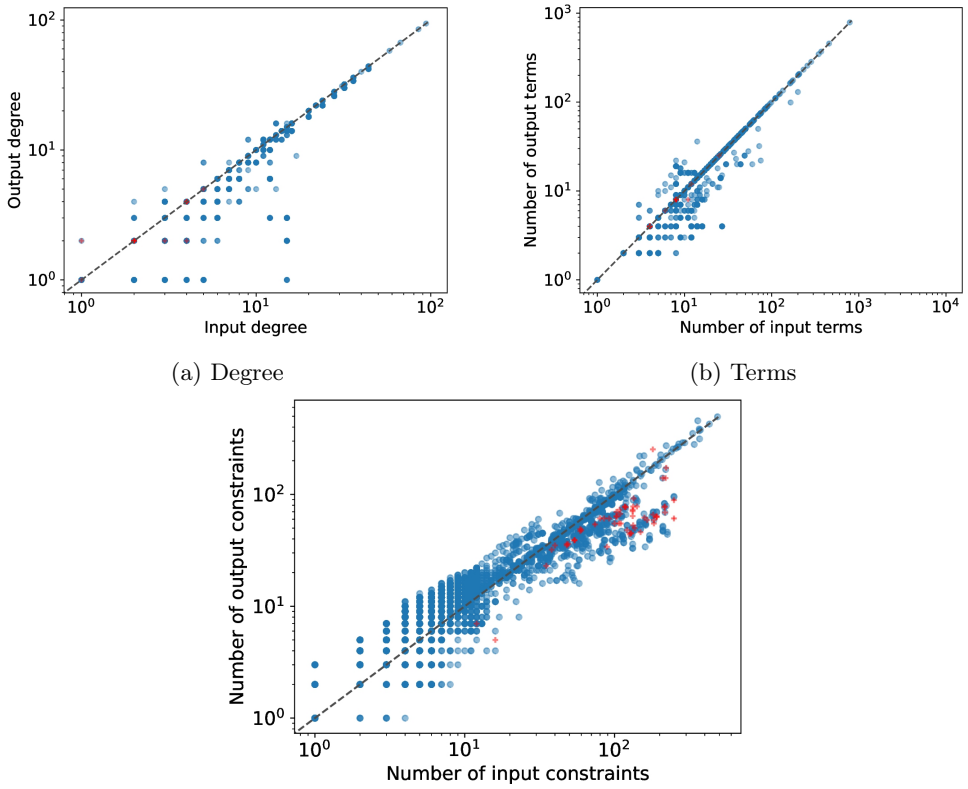
Figure 5.7: Comparison of size of UNSAT cores of SimplifyIncr and CompareIncr

reached by SimplifyIncr. We can see a clear tendency towards the middle with relatively close deviations on both sides. Thus the sizes of UNSAT cores are quite similar in most cases. Nevertheless, more instances have a bigger UNSAT core in SimplifyIncr. The deeper colour close to the origin also signifies that most UNSAT cores are relatively small. The deviations that stand out are visible in the vertical line on the left side of the graph. These indicate that in some cases the UNSAT core of SimplifyIncr is significantly smaller than the one of CompareIncr.

5.2.4 Benefit Correlation

Also of interest to investigate is whether there are certain characteristics for which SimplifyIncr is particularly helpful. To this end, we examine the maximum number of constraints that are given, the maximum degree of of these constraints and the maximum number of terms in log scale. Note that statistics are only collected for instances answered with ”unsat”, ”sat” or ”unknown” and since $log[0]$ is undefined, instances solved by Simplify itself are not visible.

Fig. 5.8 depicts the degree, number of terms and number of constraints compared for given constraints and constraints given to the theory solver in the end.
In a) we can see that the output degree is usually reduced quite well. Even relatively small changes in the degree can have a significant impact. There are only few instances with a higher output than input degree. With a high majority, newly solved instances lie on the middle line, meaning that the input and output degree are equivalent. They also tend to have a low input degree.
For instances solved by both solvers, we can also see a clear tendency towards the middle line in b). Thus, the number of terms does not change much for those instances. As we saw before in Fig. 5.6, most newly solved instance are solved by SimplifyIncr itself, meaning that the output terms are zero. For these, the number of input terms can be much higher. Generally, the number of terms is reduced.
In c) it is noticeable that most instances have a lower maximum number of constraints



(a) Degree

(b) Terms

(c) Constraints

Figure 5.8: Comparison of complexity characteristics of constraints given for inprocessing and created for the theory solver with logarithmic scale. Newly solved instances marked in red.

in the input than in the output. That strikes as surprising, as usually multiple constraints are used to imply one given constraint. It may be the case that mostly the highest input constraints were created in an iteration where SimplifyIncr found UNSAT, and the maximum output constraints are smaller in all other iterations. Another idea would be that the given constraints are interdependent enough to be implied by the same simpler constraints. Opposed to a) and b), the newly solved instances are more scattered and primarily occur for a higher number of input constraints.

When is SimplifyIncr especially helpful? Apparently, the newly solved instances mostly have a relatively low degree and a limited number of constraints. It seems like SimplifyIncr is especially helpful for a high number of terms.

5.2.5 Standard Preprocessing

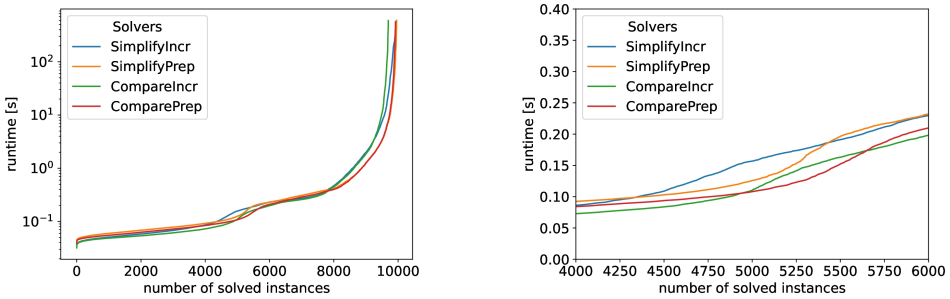
By default, SMT-RAT uses a preprocessing module called "FPPModule". This calls various preprocessing modules in succession, which are intended to simplify the constraints before the SAT solver as much as possible [fpp]. In this context, the last

question of this section is whether this module improves performance further or even makes Simplify redundant.
We consider SimplifyIncr and CompareIncr both in a preprocessed version and refer to the corresponding solvers as SimplifyPrep and ComparePrep.

	CompareIncr	SimplifyIncr	SimplifyPrep	ComparePrep
sat	4919	4939	5062	5057
unsat	4777	5011	4892	4862
unknown	163	134	86	85
timeout	1367	1475	1337	1045
memout	874	536	715	1033
segfault	34	39	42	52
solved	9696	9950	9954	9919

Table 5.2: Result comparison for SMT-RAT with Simplify and without Simplify (Compare) each with and without preprocessing

Table 5.2 again shows the answer given by SimplifyIncr, CompareIncr, SimplifyPrep and ComparePrep. Note that all considered solvers are incremental. ComparePrep is highly improved opposed to CompareIncr. Nevertheless, SimplifyIncr still outperforms ComparePrep. SimplifyPrep is a slight improvement for SimplifyIncr, visible in the number of solved instances. Also, the number of timeouts notably decreased after preprocessing, but the number of memouts increased.



(a) All

(b) Middle segment

Figure 5.9: Performance profile of SimplifyIncr, SimplifyPrep, CompareIncr and ComparePrep with logarithmic scale.

Fig. 5.9 again plots the cumulated runtime of the solved instances for SimplifyIncr, SimplifyPrep, CompareIncr and ComparePrep. All of these solvers have similar runtime behaviour, but the steep runtime increase happens later for solvers with preprocessing. It is again visualized that the number of solved instances are similar for SimplifyIncr, SimplifyPrep and ComparePrep and much higher than for CompareIncr. In b) we can see that solvers show a similar curve depending on whether preprocessing is used or not. In these two pairs, Simplify is the slower one in this segment, before being surpassed by Compare.

5.3 SimplifyIncr vs. Simplify

In this section, we investigate how much the adaptions of Chapter 4 improved the solver. In this regard, we take a look at the runtime comparison, both in general and with a focus on the different algorithms. Moreover, as in Section 5.2, we will also evaluate the transitions of the answers in Table 5.1 and the sizes of the UNSAT cores. Lastly we will take a look at the sizes of the **why** maps, before investigating if a limit on the iterations of the algorithms improves the results and if so which limit should be used. For all initial investigations, we consider a limit of 3 iterations.

5.3.1 Memory

Table 5.1 shows a shift from timeouts to memouts from Simplify to SimplifyIncr. This makes sense, as the incrementality is dependent on a lot of saved information. This includes the strictness of variables, the matrices, O_1 and O_2, the mapping between factors and variables for Blackbox, α for Whitebox and $sub, reason$ for SimpleSubstitution to name a few. For Simplify, we generate these in each iteration and algorithm application again.

5.3.2 Runtimes

Fig. 5.10 shows a runtime comparison between Simplify and SimplifyIncr. Note that the results are also impacted by the fact that Simplify uses a non-incremental version of Covering, opposed to the incremental Covering of SimplifyIncr.

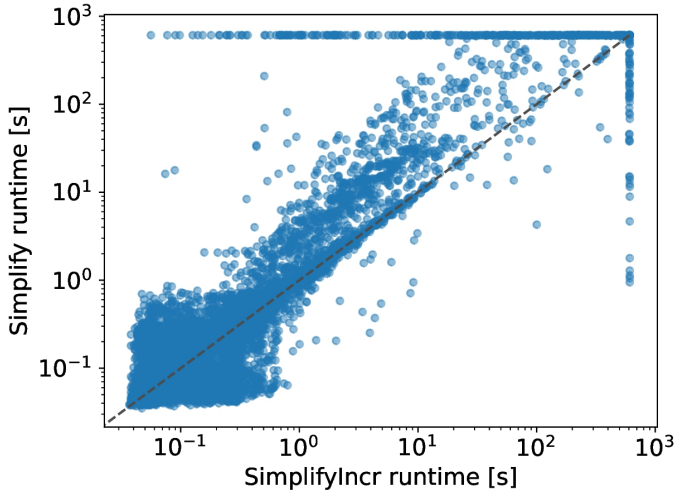


Figure 5.10: Runtime comparison between SimplifyIncr and Simplify with logarithmic scaling

Opposed to Fig. 5.2 we can clearly see a tendency towards a worse runtime for Simplify. Both solvers are sometimes better or worse for very easily solved instances. Yet, there are clear deviations along the middle line, indicating faster solving for SimplifyIncr. Additionally, similar to Fig. 5.2, while SimplifyIncr only has timeouts or

memouts for instances for which Simplify also needed a bit of time, Simplify produced timeouts/memouts for instances which were easily solved by SimplifyIncr.

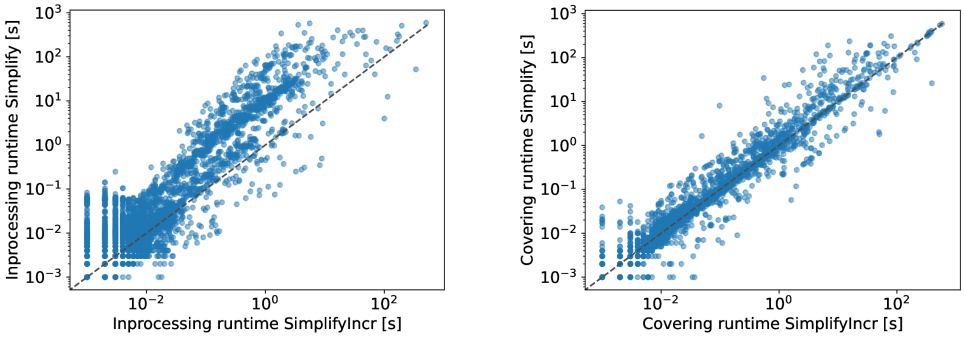
Now, even though we identified SimplifyIncr to be faster than Simplify, we do not know the impact of the runtime of the inprocessing itself or the theory solver.

		SAT Solver		Inprocessing		Theory Solver	
		[s]	%	[s]	%	[s]	%
A	Simplify	0.36	8.24	2.50	57.21	1.51	34.55
	SimplifyIncr	0.37	19.37	0.41	21.47	1.13	59.16
B	Simplify	0.28	5.51	2.99	58.86	1.81	35.63
	SimplifyIncr	0.31	13.72	0.52	23.01	1.43	63.27

Table 5.3: Mean runtime for the different solver parts of Simplify and SimplifyIncr. A stands for all instances solved by both solvers, while B is the subset of instances, for which the backend was actually called.

Table 5.3 shows the runtime influence of the SAT Solver, inprocessing and the theory solver, where A was created on all data, while B was created on those instances, for which the theory solver was actually used. We can see that the SAT solver only varies slightly between Simplify and SimplifyIncr. As SimplifyIncr is significantly faster than Simplify, the percentage of the SAT solver is of course higher for SimplifyIncr. Inprocessing takes much longer in Simplify, while the theory solver is only slightly faster in SimplifyIncr. This might be influenced by the incrementality of the theory solver in SimplifyIncr. From A to B only the runtime of the SAT Solver decreases, while the runtime for inprocessing and the theory solver slightly rises.

As Table 5.3 is based on the mean of the runtimes, it introduces a bias. Therefore, we also consider the runtime of inprocessing and the theory solver per instance (Fig. 5.11).



(a) Inprocessing (b) Theory Solver

Figure 5.11: Runtime comparison between Simplify and SimplifyIncr for inprocessing and the theory solver.

a) shows the runtime for inprocessing. It can be established that in general SimplifyIncr is much faster than Simplify. There are only few outliers where Simplify

outperforms SimplifyIncr. Opposed to this, the performance of Covering appears to be more similar for both solvers. b) shows that the instances mostly align on the middle line, while SimplifyIncr seems to have an advantage for faster instances. This may be due to the fact that SimplifyIncr also has an incremental backend.

Now it is of interest which parts of Simplify were slower compared to SimplifyIncr. For the following analysis, first there are a few important details about SMT-RAT. All constraints are saved in the form $\sum_i a_i x_i^{d_i} \sigma 0$. To be able to work with Blackbox, we need to factorize those constraints. This may take a lot of time in some cases, but once it was calculated for the first time, the factors are stored for future use. Covering also uses factorization, thus this runtime influence is also valid for Compare and CompareIncr, even though likely in a lesser form. Additionally, the constraints are returned to their standard form after Blackbox, which means that the factors are multiplied, which can also take a significant amount of time.

SMT-RAT modules provide three main functionalities called CheckCore, AddCore and RemoveCore. AddCore adds a new constraint and some initial steps may be performed. Here, the constraints are added to the matrix for Blackbox, we check for a new SubstitutionMapping and α may be updated. CheckCore performs all main algorithms, while in RemoveCore all backtracking steps are executed. As a first step, we take a look at the runtime influence of CheckCore, AddCore and RemoveCore for SimplifyIncr.

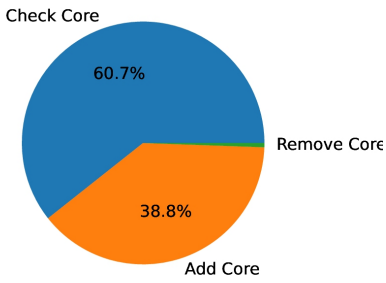


Figure 5.12: Percentage of CheckCore, AddCore and RemoveCore on the runtime.

Fig. 5.12 shows the percentages CheckCore, AddCore and RemoveCore contribute to the runtime. Even though in CheckCore the theory solver is executed as well, the corresponding runtime was subtracted for this plot. We can see that the portion of the overall runtime spent on backtracking is very low. CheckCore has the majority, but AddCore also has a significant influence on the runtime of the inprocessing. This is probably due to the fact that all initial factorizations take place in AddCore. In comparison to Simplify, it should be noted that apart from incrementality all steps of AddCore also take place in Simplify in some form. The only real change not occurring in Simplify is RemoveCore, which as we can see does not affect the runtime much.

To investigate which algorithms lead to the runtime improvement, we again consider the mean runtime contribution of the separate algorithms for Simplify and SimplifyIncr.

	Theory Solver		Blackbox		Whitebox		Rest	
	[s]	%	[s]	%	[s]	%	[s]	%
Simplify	1.69	37.56	0.77	17.11	1.98	44.00	0.06	1.33
SimplifyIncr	1.30	73.03	0.04	2.25	0.43	24.16	0.01	0.56
SimplifyIncr (all)	2.09	35.85	2.86	49.06	0.86	14.75	0.02	0.34

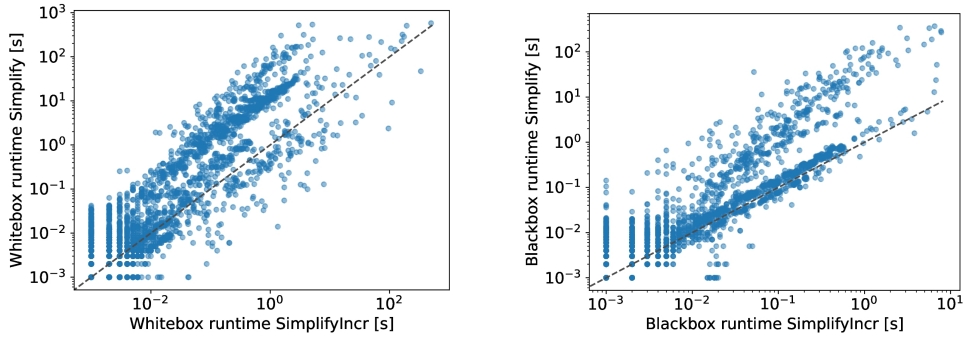
Table 5.4: Runtime contribution to inprocessing from the separate algorithms. The first two lines only consider instances solved by both Simplify and SimplfyIncr. The last line also includes instances not solved by Simplify.

Table 5.4 shows the runtime impacts of the separate algorithms, measured through mean. SimplifyIncr has a slightly faster backend, likely due to its incrementality. But as SimplifyIncr is faster than Simplify, the theory solver still makes up the high majority in regard to the percentage of runtime. Both Blackbox and Whitebox take more time in Simplify than in SimplifyIncr. In both solvers, Whitebox takes longer than Blackbox. Why is that?

The loops of Whitebox were not changed for SimplifyIncr. Only the sets for which these loops are executed were adapted. For Simplify, the first loop is executed for each existing factor and variable and the second loop for each combination of those. SimplifyIncr only executes the first loop on new factors or factors impacted by a variable change. The second loop is executed for each combination of this set and for each combination of this set with any factor or variable not included in this set. Let P be the set for Simplify with $|P| = n$ and P_1, P_2 be the sets for SimplifyIncr with $|P_1| = n_1, |P_2| = n_2$. Then the second loop of Simplify is executed $n * (n-1)$ times and the first loop n times. Thus, there are n^2 loop executions for Simplify. The first loop of SimplifyIncr is also executed n_1 times, while the second loop is executed $n_1 * (n_1 - 1 + n_2)$ times, leading to $n_1 * (n_1 + n_2)$ loop executions.

Example 5.3.1 (Loop executions)**.** *Let $n = n_1 + n_2$ and $n = 100$, $n_1 = 10, n_2 = 90$ and each loop have a runtime of $l = 0.0003s$. Then the runtime of Simplify's Whitebox is $r_1 = n^2 * l = 100^2 * 0.0003s = 30s$. The runtime of SimplifyIncr's Whitebox is $r_2 = n_1 * (n_1 + n_2) * l = 10 * 100 * 0.0003s = 0.3s$. Thus, r_1 is 100 times higher than r_2.*

Thus, we can determine that the incremental adaptions of Whitebox strongly improved its runtime.
The measure mean used in Table 5.4 introduces a bias, because very few instances with very high runtimes may influence the results significantly. To counter this bias, we also consider Fig. 5.13, which shows the runtimes per instance. Fig. 5.13 a) again highlights the runtime differences of Whitebox and shows that SimplifyIncr is faster in the majority of instances. There are only few cases in which Simplify is faster in Whitebox.
b) shows that Blackbox is very much faster for SimplifyIncr. Apart from the fact that almost all instances are on the left side of the middle line, the upper runtime for



(a) Whitebox (b) Blackbox

Figure 5.13: Runtime comparison for Blackbox and Whitebox for instances solved by Simplify and SimplifyIncr.

Simplify is more than 350 opposed to a value around 8 in SimplifyIncr. This is likely due to the repetition of Blackbox operations in each loop, as in Simplify there is no incrementality in the matrices at all.

Table 5.4 shows that for instances newly solved by SimplifyIncr, the percentage of time spent in Blackbox highly increases. The performance of Blackbox is likely influenced by the factorization and multiplication (map back).

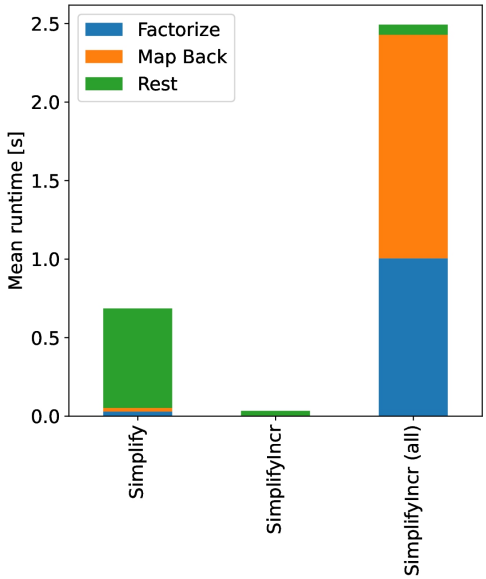


Figure 5.14: Mean runtime impact of factorization and final multiplication with for Simplify and SimplifyIncr, where (all) stands for all instances, including those not solved by Simplify.

Fig. 5.14 shows the runtime influence of the factorization of the constraints and the return to the standard structure through multiplication of factors. It is clear that these influence Simplify only marginally. For instances solved by both solvers, Blackbox obviously takes longer for Simplify than for SimplifyIncr. But for newly solved instances, the factorization and mapping to the original form takes up the grand majority of the runtime of Blackbox. Not only that, but due to this, the runtime of Blackbox is also about 3 times as high as for Simplify. This shows that constraints with a complex factor structure are harder to solve in Simplify and SimplifyIncr.

5.3.3 Answer Transitioning

In addition to the runtime, we are interested in the origin of the newly solved instances. For this reason, we created Fig. 5.15 the same way as Fig. 5.5.

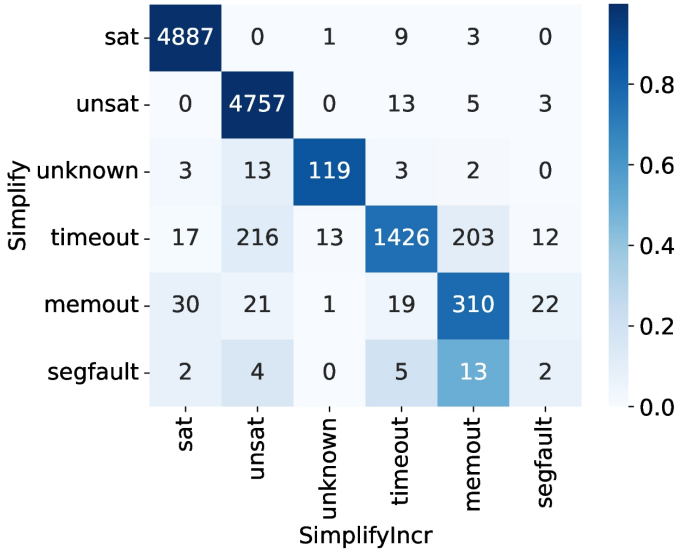


Figure 5.15: Transition of Simplify (y-axis) to SimplifyIncr (x-axis) normalized with colour scheme as percentage normalized to 100% per row

We can see that quite a few new UNSAT instances were found from the set of not successfully solved instances. Another interesting aspect is that for memouts and segfaults more new SAT than UNSAT results were found. Opposed to Fig. 5.5 the turnover between memouts and timeouts is less significant. Again, the transition from successfully solved instances to unsolved instances is neglectable.

We again want to consider how many new UNSAT instances were identified at which point of the solvers.
Fig. 5.16 shows the number and percentage of instances solved with the help of Simplify, the backend or both. We only consider instances for which both solvers returned UNSAT. For both, the majority of instances is solved using the theory solver, but the percentage and number of instances solved using the inprocessing is higher in SimplifyIncr, while the value for the Backend is lower. This shows that the inprocessing algorithm identifies more instances in SimplifyIncr.

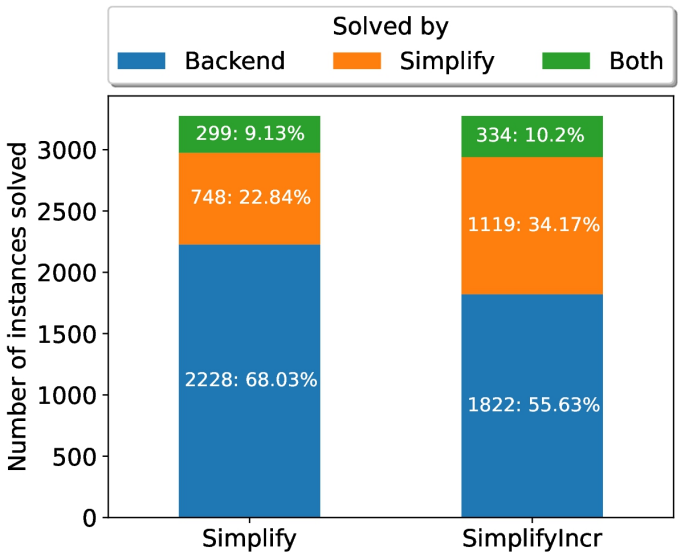


Figure 5.16: Box plot of number of instances solved by inprocessing, the theory solver or both.

5.3.4 UNSAT Cores

In this part of the analysis, we are interested in the differences of the UNSAT cores.

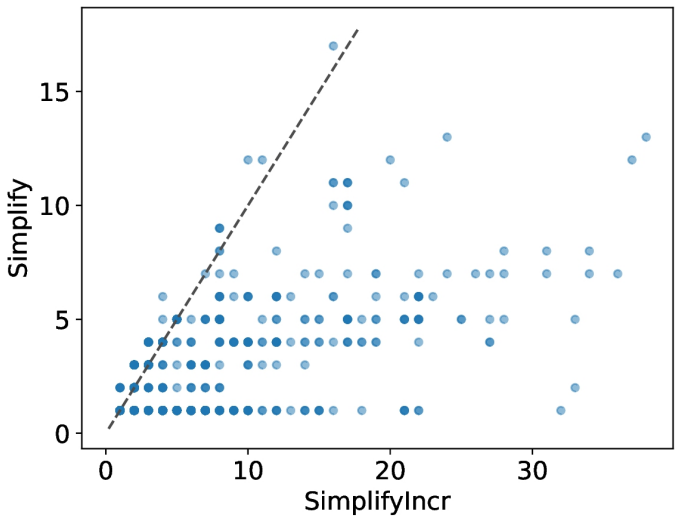


Figure 5.17: Comparison of size of UNSAT cores of SimplifyIncr and Simplify

Fig. 5.17 visualizes the sizes of UNSAT cores of Simplify against SimplifyIncr. Interestingly, not only is there a clear trend towards bigger UNSAT cores for SimplifyIncr compared to Simplify, but the highest size for SimplifyIncr is also 2.6 times higher

than for Simplify with 38 vs 17. Note that we only considered instances which were solved by both Simplify and SimplifyIncr. This might be explainable by the way the UNSAT cores are generated. In Simplify, the entries of **why** are filled in the order the constraints are created. Thus, the first implication is usually the fastest way to get to the given constraints. Since implications are deleted in SimplifyIncr, this is not necessarily the case for SimplifyIncr. We select the path to the givens dependent on the size of the newest implication. Thus, it might provide useful to develop a proper heuristic for this path selection for SimplifyIncr in the future.

5.3.5 Complexity Reduction

As the runtime and answers of SimplifyIncr are better than for Simplify, one could wonder how the complexity of the constraints given to the theory solver changes.

Fig. 5.18 shows information about the maximum degree, number of constraints and number of terms selected by Simplify and SimplifyIncr within one instance. Newly solved instances are marked in red. This means that for the plots of Simplify, the red marks are instances not solved by SimplifyIncr. The degree (visible in a) and b) of Fig. 5.18) are quite similar. Both have either a similar output and input degree, visible as the instances on the middle line, or reduce the complexity in the area below 20. The newly solved instances are also in a similar region, meaning that the degree does not seem to be the main factor in which SimplifyIncr is better. Simplify seems to have higher values for outliers in which the output degree is higher than the input degree.
In c) and d) a lot of instances are again on the middle line. The behaviour in the area before 10^2 is similar as well. On the other hand, the instances solved by SimplifyIncr itself are not visible which can reach much higher numbers of terms than Simplify. This is probably, because of the time spent on factorization. Simplify also has a few outliers with way more output terms than input terms. How can that happen?

Example 5.3.2 (Term increase)**.** *Let the following table be the initial strict matrix:*

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Then the resulting matrix would be

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

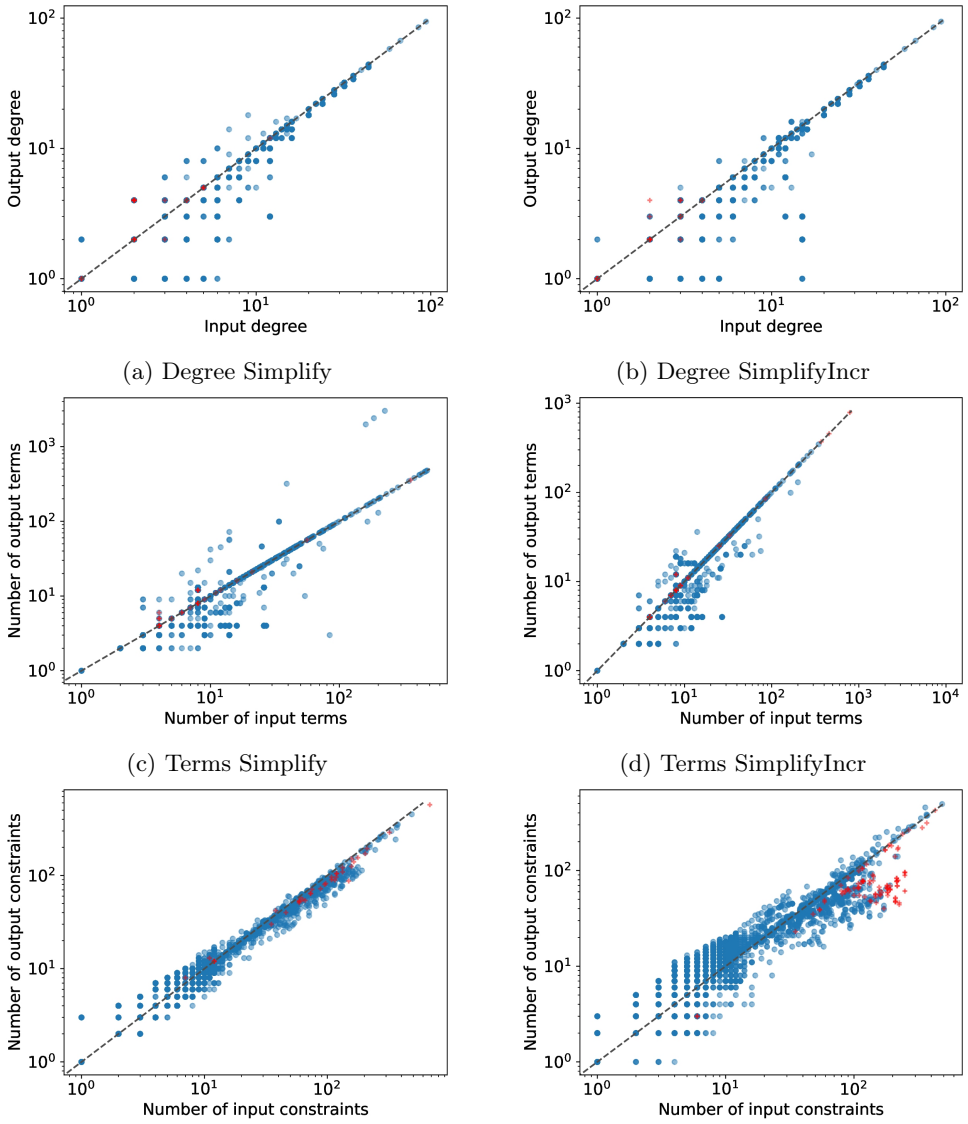
Thus the number of terms very likely increased.

These instances could belong to the instances solved by inprocessing in SimplifyIncr.

e) and f) show the interesting fact that the number of constraints actually seem to be reduced in most cases. This is surprising because the idea was to use several simpler constraints to imply a complex constraint. This shows that the constraints are interdependent enough to be implied by the same set of constraints. For Simplify, the newly solved instances are mainly at the middle line, while in SimplifyIncr the number of constraints for the newly solved instances is more spread. Nevertheless, SimplifyIncr also has more instances with a higher number of output constraints.

This may be due to the changed selection process. For both solvers, the newly solved instances tend to have a high number of constraints.

Our main takeaway is that the incrementality appears to be most helpful for constraints with a high number of terms.



(a) Degree Simplify

(b) Degree SimplifyIncr

(c) Terms Simplify

(d) Terms SimplifyIncr

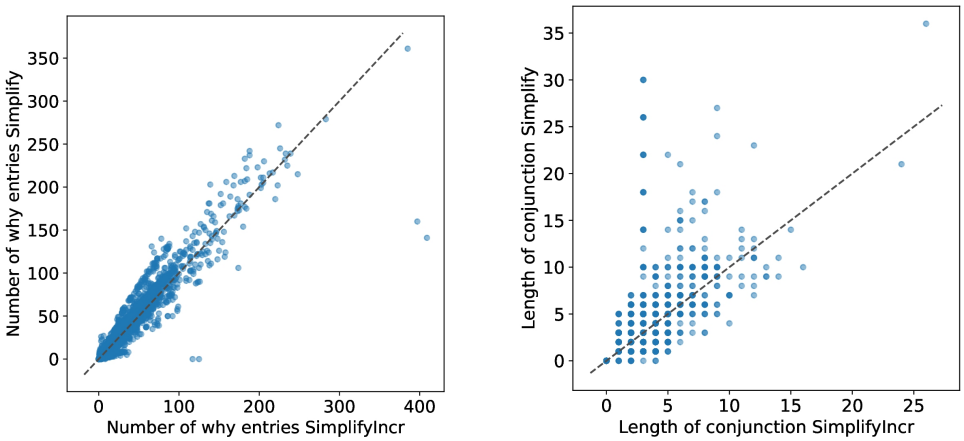
(e) Constraints Simplify

(f) Constraints SimplifyIncr

Figure 5.18: Comparison of complexity reduction for Simplify and SimplifyIncr with logarithmic scale.

5.3.6 WHY

How is the size of **why** affected by incrementality? Considering the fact that the incremental version has to imply O_1, both types of normalization have to be implied during preprocessing and StrictBlackbox (Algorithm 8) uses two types of implications, it makes sense to assume that the size of **why** increases.



(a) Entries (b) Implications

Figure 5.19: Comparison of maximum number of **why** entries and the longest conjunction in **why** .

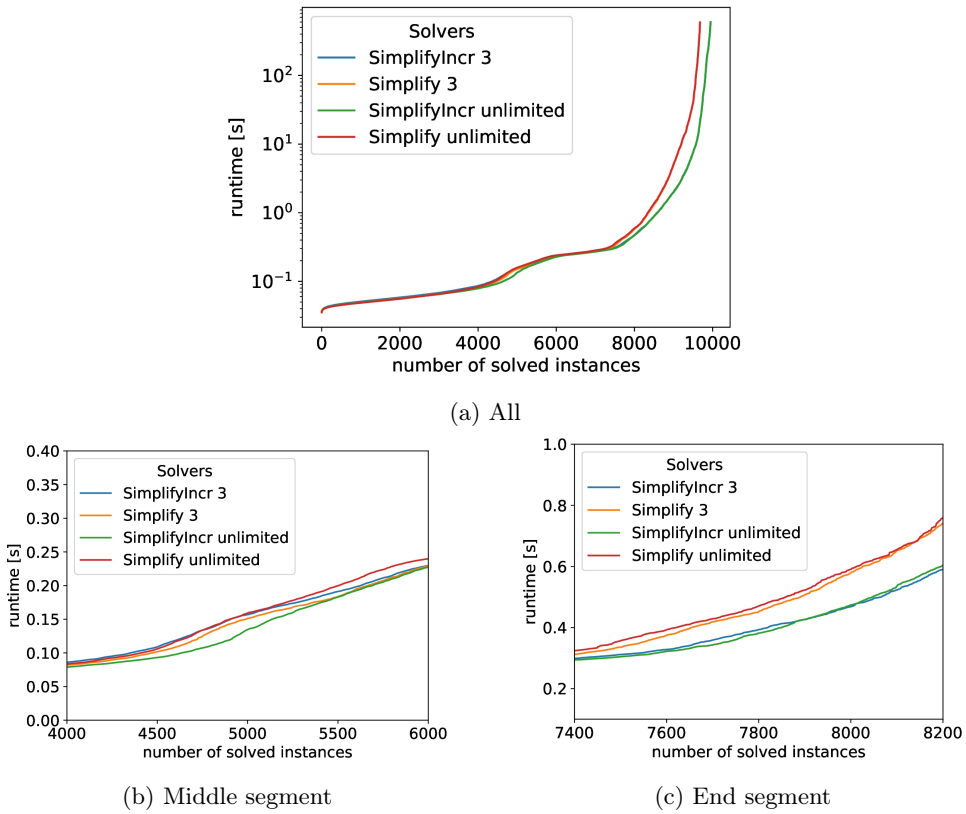
Fig. 5.19 visualizes the statistics of **why** for SimplifyIncr and Simplify. Here, a) visualized the highest number of **why** entries occurring for an instance, while b) shows the longest conjunction of constraints implying another constraint in **why** . Surprisingly, SimplifyIncr does not appear to have much more **why** entries. a) shows that the number of entries are very similar, but most deviations show more entries for Simplify. On the other hand, there are a very few outliers with much more entries for SimplifyIncr, beating the highest number of entries appearing in Simplify.
The highest number of conjunctions in **why** appears to stray more from the middle line. Fig. 5.19 b) shows that even though the instances appear to divert quite symmetrically, there are clearly longer conjunctions in Simplify than SimplifyIncr.

5.3.7 Loop Executions

So far, we used an upper limit of 3 executions for each described algorithm. Thus, Blackbox, Whitebox and SimpleSubstitution are executed until either the limit of 3 is up or there is nothing new to be found. Now we want to evaluate if it is beneficial to use this limit or iterate as long as we find new results.
Table 5.5 summarizes the answers provided by Simplify and SimplifyIncr, each with a limit of 3 and unlimited. Both solvers are not impacted much. There are slight interchanges between the unsolved instances. Setting no limit seems to be beneficial for SAT instances but not for UNSAT instances, as the number of SAT increases, while UNSAT decreases for SimplifyIncr. To see if there is a higher impact on the runtime behaviour, Fig. 5.20 shows the performance profile for these solvers.

	SimplifyIncr		Simplify	
	Unlimited	3	3	Unlimited
sat	4940	4939	4900	4899
unsat	5009	5011	4778	4778
unknown	134	134	140	139
timeout	1484	1475	1887	1896
memout	529	536	403	397
segfault	38	39	26	25
solved	9949	9950	9678	9677

Table 5.5: Result comparison for Simplify and SimplifyIncr, each with a limit of 3 iterations or unlimited.



(a) All

(b) Middle segment

(c) End segment

Figure 5.20: Performance profile of SimplifyIncr and Simplify, each with a loop limit of 3 and unlimited with logarithmic scale. To discern overlapping lines, b) and c) zoom into more interesting segments.

Again, the runtime behaviour of these solvers is very similar. To ensure the overlapping lines in the final section, c) zooms into the area between 7400 and 8200. We can see that Simplify is slower than SimplifyIncr in both versions and the final cumulated

runtime is almost 100% overlapping within each solver. b) shows the segment between 4000 and 6000 solved instances. It is clear that the unlimited SimplifyIncr appears to solve more instances with a lower runtime.

Chapter 6

Conclusion

Finally, we will give a conclusion to this thesis. This includes a short summary and discussion, before briefly addressing possible future work.

6.1 Summary

This thesis was based on [BVE20] and the idea to reduce the complexity of constraints before the theory solver. The deductions and their implications are saved in a so-called why-map. The algorithms introduced by Brown et al. were mainly based on Blackbox, Whitebox and SimpleSubstitution.

Blackbox considers the factors making up a constraint as blackboxes and focuses solely on derivations based on the factor structures. After normalizing the constraints and converting them to a matrix form, multiple derivations based on the strictness of the constraints and variables are possible using Gaussian elimination.

Whitebox on the other hand makes use of the polynomial structure of the factors itself. Using the knowledge about variable signs and also signs of other polynomials, new signs for factors and variables are derived.
SimpleSubstitution uses simple equations to replace variables by constants or other variables.

The main contribution of this thesis was, to adapt the algorithms by Brown et al. to enable incrementality and backtracking. For Whitebox this mainly included changing for which factors the algorithm is executed. We had to figure out how to derive a new substitution mapping for SimpleSubstitution when a constraint in *reason* is backtracked. Blackbox was the most complex to adapt. The normalization introduced by Brown et al. lead to information loss. Changes in strictness had to be implied by something. For backtracking, we needed to track the impact of the backtracked constraint and return the impacted constraints to their original form.

To evaluate the impact of these algorithms, we analysed a multitude of experiments. This included comparing the incremental version of Simplify (SimplifyIncr) with incremental Covering (CompareIncr) and Simplify.

6.2 Discussion

In Chapter 5 we found out that SimplifyIncr solves by far the most instances. Especially the number of UNSAT instances increases, due to instances identified as such in inprocessing. In $\sim 90\%$ of solved instances, inprocessing was involved in finding UNSAT partial solutions. Additional preprocessing before the SAT solver increases the performance again, but does not make inprocessing redundant.

The incrementality adaptions did not only increase the number of successfully solved instances, but also reduced the runtime. The largest runtime impact is on Whitebox. Whitebox impacts the runtime of inprocessing significantly less using the adapted algorithms. As SimplifyIncr solves additional instances which Simplify cannot solve in time the mean runtime of Blackbox increases.

The complexity for instances solved by both solvers was very similar. Nevertheless, it was again shown that SimplifyIncr has a big advantage for constraints with a high number of terms.

Limiting the number of loop executions does not impact the result much. There is one more solved instance and in the intermediate region there is a slightly higher lower for unlimited SimplifyIncr.

All in all, SimplifyIncr appears to be most helpful for constraints with many terms and a complicated factorization. At the same time, it highly depends on the runtime of the factorization which is likely also valid for Covering.

6.3 Future Work

Currently, we use two types of implications for Blackbox. For future work, one might analyse whether both constraints are necessary and otherwise explore which implication should best be used.

Moreover, it might be interesting to develop an efficient way to select the UNSAT Core. As a multitude of constraints may imply another constraint, there is also a high number of paths to original constraints. Thus, it may make a difference which one is selected.

As Covering also uses factors at some point, it might improve the runtime if instead of returning to the original structure, the constraints are given to Covering in their factor structure, which means that we do not need to multiply out the derived constraints. Also, the compatibility to other solvers may be tested. It might be especially interesting to see how SimplifyIncr performs for a Backend not using factorization at all. There are solvers, like the default of SMT-RAT, using the Model-Constructing Satisfiability (MCSAT) calculus [DMJ13], which is not strictly structured in SAT Solver and theory solver. In future work, one might adapt Simplify further to be of use for solvers using MCSAT.

Bibliography

[ÁAB+16] Erika Ábrahám, John Abbott, Bernd Becker, Anna M Bigatti, Martin Brain, Bruno Buchberger, Alessandro Cimatti, James H Davenport, Matthew England, Pascal Fontaine, et al. SC^2: Satisfiability checking meets symbolic computation: (project paper). In *Intelligent Computer Mathematics: 9th International Conference, CICM 2016, Bialystok, Poland, July 25-29, 2016, Proceedings*, pages 28–43. Springer, 2016.

[ÁDEK21] Erika Ábrahám, James H Davenport, Matthew England, and Gereon Kremer. Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings. *Journal of Logical and Algebraic Methods in Programming*, 119:100633, 2021.

[BdM14] Nikolaj Bjørner and Leonardo de Moura. Applications of SMT solvers to program verification. *Notes for the Summer School on Formal Techniques*, 2014.

[Bro09] Christopher W Brown. Fast simplifications for Tarski formulas. In *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*, pages 63–70, 2009.

[Bro12] Christopher W Brown. Fast simplifications for Tarski formulas based on monomial inequalities. *Journal of Symbolic Computation*, 47(7):859–882, 2012.

[BS10] Christopher W Brown and Adam Strzeboński. Black-Box/White-Box Simplification and Applications to Quantifier Elimination. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, pages 69–76, 2010.

[BVE20] Christopher W Brown and Fernando Vale-Enriquez. From Simplification to a Partial Theory Solver for Non-Linear Real Polynomial Constraints. *Journal of Symbolic Computation*, 100:72–101, 2020.

[CLJÁ12] Florian Corzilius, Ulrich Loup, Sebastian Junges, and Erika Ábrahám. SMT-RAT: An SMT-compliant nonlinear real arithmetic toolbox: (tool presentation). In *Theory and Applications of Satisfiability Testing–SAT 2012: 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings 15*, pages 442–448. Springer, 2012.

[clu] HPC cluster hardware specifications. `https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/article/fbd107191cf14c4b8307f44f545cf68a/`. Accessed: 2023-08-27.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008. Proceedings 14*, pages 337–340. Springer, 2008.

[DMJ13] Leonardo De Moura and Dejan Jovanović. A Model-Constructing Satisfiability Calculus. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 1–12. Springer, 2013.

[fpp] FPPModule documentation. `https://ths-rwth.github.io/smtrat/d1/d98/classsmtrat_1_1FPPModule.html`. Accessed: 2023-08-20.

[KÁ18] Gereon Kremer and Erika Ábrahám. Modular strategic SMT solving with SMT-RAT. *Acta Universitatis Sapientiae, Informatica*, 10(1):5–25, 2018.

[Leo09] Steven J. Leon. *Linear Algebra with Applications*. Pearson Prentice Hall, 8'th edition, 2009.

[Nab09] Hédi Nabli. An overview on the simplex algorithm. *Applied Mathematics and Computation*, 210(2):479–489, 2009.

[oM] Encyclopedia of Mathematics. Conjunctive normal form. `http://encyclopediaofmath.org/index.php?title=Conjunctive_normal_form&oldid=35078`. Accessed: 2023-09-06.

[Pro14] Prof. Dr. Erika Ábrahám. Satisfiability Checking: Interval Constraint Propagation. `https://ths.rwth-aachen.de/wp-content/uploads/sites/4/teaching/vorlesung_satchecking/ws14_15/09e_icp_handout.pdf`, 2014. Accessed: 2023-07-21.

[Seb07] Roberto Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3(3-4):141–224, 2007.

[smta] SMT-LIB-benchmarks QF_NRA. `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NRA`. Accessed: 2023-05-23.

[smtb] SMT-RAT. `https://ths-rwth.github.io/smtrat/`. Accessed: 2023-08-20.

Appendix A

Whitebox Algorithms

Algorithm 12 PolynomialSign from [BS10]

Input: Polynomial $p = a_1M_1 + ... + a_kM_k$, where the M_i are power products over $x_1,...,x_n$, $\alpha_1,...,\alpha_n \in S^+_{op}$
Output: $\beta \in S_{relop}$ such that $\bigwedge_{i=1}^{n} x_i\alpha_i 0 \Rightarrow p\beta 0$

```
1: β := EQOP
2: for i ∈ {1,...,k} do
3:     β := sgn(a_i)·MonomialSign(M_i, (α_1,...,α_n))+β
4: return β
```

Algorithm 13 MonomialSign from [BS10]

Input: Power product $M = x_1^{e_1} \cdot ... \cdot x_n^{e_n}$, $\alpha_1,...,\alpha_n \in S^+_{op}$
Output: β: strongest element of S_{relop} , such that $\bigwedge_{i=1}^{n} x_i\alpha_i 0 \Rightarrow M\beta 0$

```
1: β := GTOP
2: for i ∈ {1,...,n} do
3:     if e_i even then
4:         β := α_i · α_i · β
5:     else
6:         β := α_i · β
7: return β
```

Algorithm 14 PolynomialSignProof from [BVE20]

Input: Polynomial $p = a_1M_1 + ... + a_kM_k$, where the M_i are power products over $x_1,...,x_n$, $\alpha_1,...,\alpha_n \in S^+_{op}, \beta \in S_{relop}$

Output: FAIL or α'. If $wb(\alpha,p,\beta)$, return maximally weak α', so that α strengthens α' and $wb(\alpha',p,\beta)$ α' encodes the deduction $\bigwedge_{i=1}^{n} x_i\alpha'_i 0 \Rightarrow p\beta 0$. Else return FAIL

$\alpha' := [ALOP,...,ALOP]$
if $\beta \in \{LTOP,GTOP\}$ **then**
  $N := \emptyset$
  for $i \in \{1,...,k\}$ **do**
    $\gamma :=$ MonomialSignProof$(\alpha, M_i, op(a_i) \cdot \beta)$
    $\sigma :=$ MonomialSignProof$(\alpha, M_i, op(a_i) \cdot (\beta \vee EQOP))$
    if $\gamma =$ FAIL **then**
      if $\sigma =$ FAIL **then**
        return FAIL
    else
      $N := N \cup \{(i,\gamma)\}$
  if $N = \emptyset$ **then return** Fail
  $(i,\gamma) :=$ choose element of N, so that there is no $(i',\gamma') \in N$ for which the variables in $M_{i'}$ are a subset of the variables in M_i
  return $\alpha' := \gamma \wedge$PolynomialSignProof$(p,\alpha,\beta \vee EQOP)$
else if $\beta = NEOP$ **then**
  if $k = 1$ **then return** MonomialSignProof$(\alpha,M_i,NEOP)$
  $\alpha' :=$ PolynomialSignProof$(p,\alpha,GTOP)$
  if $\alpha' \neq$ FAIL **then return** α'
  return $\alpha' :=$ PolynomialSignProof$(p,\alpha,LTOP)$
else if $\beta = EQOP$ **then**
  $T = \bigcup_{i \text{ s.t. } M_i \text{ non-constant}} \{\{j \in \{1,...,n\} \mid deg_{x_j}(M_i) > 0 \wedge \alpha_i = EQOP\}\}$
  if $\emptyset \in T$ **then return** FAIL
  $H :=$ minimal hitting set for T
  $\alpha'_i := EQOP \ \forall i \in H$
  return α'
else if $\beta \in \{LEOP,GEOP\}$ **then**
  see Algorithm 15
  return α'

Algorithm 15 Case $\beta \in \{LEOP,GEOP\}$ from PolynomialSignProof14 from [BVE20]

$X := \{x_i \mid \alpha_i \neq EQOP\}$
$Y := \{x_i \mid \alpha_i = EQOP\}$
$Z := \emptyset$
$\gamma :=$ n-vector, where $\gamma_i = ALOP$ if $x_i \in Y$, $\gamma_i = \alpha_i \wedge NEOP$ if $x_i \in X$
$\sigma :=$ n-vector, where $\sigma_i = ALOP$ if $x_i \in Y$, $\sigma_i = EQOP$ if $x_i \in X$
$p' :=$ restriction(p,X)
$\pi := [NEOP,...,NEOP]$
if $p' \neq 0$ **then**
    $\pi := \bigwedge_{\text{terms } b_i N_i \text{ in } p'}$ MonomialSignProof(N_i,γ,$op(b_i)(\beta \wedge NEOP)$)
if $NOOP$ appears in π **then**
    return FAIL
for $x_i \in Y$ **do**
    $p'_{new} :=$ restriction(p,$X \cup \{x_i\}$)
    $t^+, t^- :=$ TRUE
    $\pi^+, \pi^- := \pi$, but with $\pi_i^+ := GTOP, \pi_i^- := LTOP$
    for terms $b_j N_j$ in $p'_{new} - p'$ **do**
        if MonomialSignProof$(N_j, \pi^+, op(b_j) \cdot (\beta \wedge NEOP))$= FAIL **then**
            $t^+ := FALSE$
        if MonomialSignProof$(N_j, \pi^-, op(b_j) \cdot (\beta \wedge NEOP))$= FAIL **then**
            $t^- := FALSE$
    if $t^+ \wedge t^-$ **then** $X := X \cup \{x_i\}, \pi_i := NEOP$
    if $t^+ \wedge \neg t^-$ **then** $X := X \cup \{x_i\}, \pi_i := GTOP$
    if $\neg t^+ \wedge t^-$ **then** $X := X \cup \{x_i\}, \pi_i := LTOP$
    if $\neg t^+ \wedge \neg t^-$ **then** $Z := Z \cup \{x_i\}, \pi_i := EQOP$
    $Y := Y - \{x_i\}, \sigma_i := EQOP, p' := p'_{new}$
$\alpha' := \pi \vee \sigma$

Algorithm 16 MonomialSignProof as given in [BVE20]

Input: M: power product $x_1^{e_1} \cdot ... \cdot x_n^{e_n}$, α, n-vector over S_{op}^{+} , $\beta \in S_{relop}$
Output: FAIL or α'. If $\bigwedge_{i=1}^{n} x_i \alpha_i 0 \Rightarrow M\beta 0$ return α', such that α strengthens the maximally weak α' and $\bigwedge_{i=1}^{n} x_i \alpha'_i 0 \Rightarrow M\beta 0$. Otherwise, FAIL is returned.

$\alpha' := [ALOP,...,ALOP]$
if $\beta \in \{LTOP, GTOP\}$ **then**
  $d := GTOP$
  for $i \in \{1,...,n\}$ with $e_i \neq 0$ **do**
    if $\alpha_i \in \{ALOP,LEOP,GEOP,EQOP\}$ or (e_i odd $\wedge \alpha_i = NEOP$) **then**
      return FAIL
    if e_i even **then** $\alpha'_i := NEOP$
    else
      $\alpha'_i := \alpha_i$
      $d := d \cdot \alpha'_i$
  if $d = \beta$ **then return** α'
  else return FAIL
else if $\beta \in \{LEOP,GEOP\}$ **then**
  $d := GEOP$
  $T := \{i \mid \alpha_i = EQOP, e_i \neq 0\}$
  if $\alpha_i \in \{NEOP,ALOP\}$ for some odd e_i **then**
    return MonomialSignProof(M,α,$EQOP$)
  for $i \in \{1,...,n\}$ with $e_i \neq 0$ **do**
    if e_i odd **then**
      if $\alpha_i \in \{LTOP,LEOP\}$ **then** $\alpha'_i := LEOP$
      else $\alpha'_i := GEOP$
      $d := d \cdot \alpha'_i$
  if $d = \beta$ **then return** α'
  else if $T \neq \emptyset$ **then**
    return MonomialSignProof(M,α,$EQOP$)
  else
    return FAIL
else if $\beta = NEOP$ **then**
  for $i \in \{1,...,n\}$ with $e_i \neq 0$ **do**
    if $\alpha_i \in \{ALOP,LEOP,GEOP,EQOP\}$ **then**
      return FAIL
    $\alpha'_i := NEOP$
  return α'
else if $\beta = EQOP$ **then**
  $T := \{i \mid \alpha_i = EQOP, e_i \neq 0\}$
  if $T = \emptyset$ **then**
    return FAIL
  $\alpha'_i := EQOP$ for some heuristically chosen $i \in T$
  return α'

Algorithm 17 DeduceSignExplain from [BVE20]

Input: Polynomials $p = a_1M_1 + ... + a_kM_k$ and $q = b_1M_1 + ... + b_kM_k$, where the M_i are power products over x_1,...,x_n, $\alpha = \alpha_1,...,\alpha_n \in S^+_{op}$, $\kappa \in S^+_{op}$
Output: α',γ,t,β so that $\alpha \preceq \alpha'$ and $q\kappa 0 \wedge \bigwedge_{i=1}^{n} x_i\alpha' 0 \Rightarrow p\gamma 0$

Q := FindIntervals(p,q,α)
Q' := []
for $(I,\beta) \in Q$ **do**
    t := midpoint(I)
    α' := PolynomialSignProof($p + tq$,α, β)
    $\gamma = T_{ded}[\kappa, op(t), \beta]$
    add $(\alpha', \gamma, t, \beta)$ to Q'
choose $(\alpha', \gamma, t, \beta)$ from Q' so that γ is minimal w.r.t $\preceq$, breaking ties by preferring larger α'.
return $\alpha', \gamma, t, \beta$

Algorithm 18 FindIntervals from [BVE20]

Input: Polynomials $p = a_1M_1 + ... + a_kM_k$ and $q = b_1M_1 + ... + b_kM_k$, where the M_i are power products over x_1,...,x_n, $\alpha = \alpha_1,...,\alpha_n \in S^+_{op}$
Output: Q: ordered list of disjoint non-empty-interval / relational operator pairs such that $(I,\beta) \in Q \Rightarrow \forall t \in I : [p + tq\beta 0]$

Q := [((−∞,0),$EQOP$),((0,∞),$EQOP$),]
for $i \in \{1,...,k\}$ **do**
    m := MonomialSign(M_i,α)
    if $b_i = 0$ **then**
        if $a_i > 0$ **then** $c := GTOP$
        else $c := LTOP$
        $L := \{((-\infty,\infty), c \cdot m)\}$
    else
        if $b_i > 0$ **then** $(s_l,s_r) := (LTOP,GTOP)$
        else$(s_l,s_r) := (GTOP,LTOP)$
        $L := [((-\infty, -\frac{a_i}{b_i}), s_l \cdot m), ([-\frac{a_i}{b_i}, -\frac{a_i}{b_i}], EQOP), ((-\frac{a_i}{b_i}, \infty), s_r \cdot m)]$
    Q' := []
    while Q,L both non-empty **do**
        (I_Q,β_Q) := first element of Q
        (I_L, β_L) := first element of L
        if $I_q \cap I_L \neq \emptyset \wedge \beta_Q + \beta_l \neq ALOP$ **then**
            append $(I_q \cap I_L, \beta_Q + \beta_l)$ to Q'
        if right endpoint of I_Q is less than right endpoint of I_L **then**
            remove first element of Q
        else
            remove first element of L
    $Q = Q'$
return Q