Linearization Techniques for Nonlinear Arithmetic Problems in SMT

Linearisierungstechniken für nichtlineare, arithmetische Probleme in SMT

vorgelegt von

Ömer Sali

Angefertigt am
Lehrstuhl i2 für Informatik
bei
Prof. Dr. Erika Ábrahám

Zweitgutachter:
Prof. Dr. Peter Rossmanith
Abstract

Polynomial constraint solving plays a prominent role in several areas of soft- and hardware verification, optimization and planning. Unfortunately, the nonlinear constraint solving problem over the integers is undecidable. The situation is not much better when considering the reals since, although the problem is decidable as it was shown for the first-order theory of real closed fields by Tarski, using the related algorithms in practice is unfeasible due to their complexity. More efficient, but incomplete decision procedures implementing sufficient conditions only are hence applied to decide simpler instances prior to a call of more elaborate decision procedures. In this thesis we present the theoretical foundations of two new incomplete modules for the Satisfiability Modulo Theories (SMT) toolbox SMT-RAT, namely the subtropical and the case-splitting methods. They are dedicated to proving satisfiability of nonlinear real and integer arithmetic formulas by encoding them into an SMT problem considering only linear arithmetic. These linearizations are in turn solved using linear arithmetic solvers implementing a Simplex or Branch-and-Bound approach, respectively. Extensive experiments on the SMT-LIB benchmarks demonstrate that these methods are not strong decision procedures by themselves but valuable heuristics to use within a portfolio of techniques.
Contents

Abstract iii

Introduction vii

Chapter 1. Preliminaries 1

1.1. Satisfiability Modulo Theories . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2. DPLL-based SMT solving . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.3. SMT-RAT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

Chapter 2. Subtropical Satisfiability 7

2.1. Limiting behaviour of multivariate polynomials . . . . . . . . . . . . . . . . 7
2.2. Restriction process as geometric projection . . . . . . . . . . . . . . . . . . 9
2.3. Exploiting the linear separability of frame vertices . . . . . . . . . . . . . . 10
   2.3.1. Strictly separable frame vertices . . . . . . . . . . . . . . . . . . . . 10
   2.3.2. Weakly separable frame vertices . . . . . . . . . . . . . . . . . . . . 11
   2.3.3. Linearly inseparable frame vertices . . . . . . . . . . . . . . . . . . 14
2.4. Application to the SMT problem . . . . . . . . . . . . . . . . . . . . . . . . 14
   2.4.1. Single constraint with an arbitrary relation . . . . . . . . . . . . . . 15
   2.4.2. Common solution of multiple constraints . . . . . . . . . . . . . . . 15
   2.4.3. Extension to mixed-integer problems . . . . . . . . . . . . . . . . . 16
2.5. Benchmarking results and conclusion . . . . . . . . . . . . . . . . . . . . . 17

Chapter 3. The Case-Splitting Method 23

3.1. Case-splits for monomial equalities . . . . . . . . . . . . . . . . . . . . . . 23
3.2. Purification of nonlinear constraints . . . . . . . . . . . . . . . . . . . . . . 24
   3.2.1. Discretization of real-valued variables . . . . . . . . . . . . . . . . . 24
   3.2.2. Extraction of nonlinear monomial equations . . . . . . . . . . . . . 25
3.3. Case-splitting for variables with bounded domains . . . . . . . . . . . . . . 26
   3.3.1. Handling small domains . . . . . . . . . . . . . . . . . . . . . . . . 26
   3.3.2. Handling large domains . . . . . . . . . . . . . . . . . . . . . . . . 26
3.4. Unsatisfiability and learning for unbounded domains . . . . . . . . . . . . . 28
   3.4.1. Unsatisfiability and learning . . . . . . . . . . . . . . . . . . . . . . 28
Introduction

The *satisfiability problem* poses the question of whether there exists an assignment to the variables of a given logical formula such that the later becomes true. Propositional logic is well-suited for a broad range of problems like the verification of logic programs or the bounded model checking of discrete systems. Accordingly, a lot of effort has been put into the development of fast solvers for the propositional satisfiability problem (SAT). Other inherently continuous problems in the areas of system analysis and verification require the expressiveness of theories. Therefore, propositional logic is extended with first-order theory constraints to so called *Satisfiability Modulo Theories* (SMT).

Especially polynomial constraints are ubiquitous and it is paramount to have efficient automatic tools that, given a polynomial constraint with integer or real indeterminates, either return a solution or notify that the constraint is unsatisfiable. Unfortunately, the polynomial constraint solving problem over the integers is undecidable. The situation is not much better when considering the reals since, although the problem is decidable as it was shown for the first-order theory of real closed fields by Tarski, using the related algorithms in practice is unfeasible due to their complexity. Therefore, all methods used in practice for both integer and real solution domains are incomplete. There are two approaches, namely focusing on proving satisfiability or focusing on proving unsatisfiability. In general, the decision on the approach is guided by the problem in hand. Current techniques focusing on satisfiability encode the problem into SAT known as bit-blasting. Following the success of the translation into SAT, it is reasonable to consider whether there is a better target language than propositional logic to keep as much as possible the arithmetic structure of the source language. Thus, in this thesis we consider methods for solving nonlinear constraints based on encoding the problem into an SMT problem over linear real or integer arithmetic. An interesting feature of this approach is that, in contrast to SAT translations, by having linear arithmetic built into the language, negative values and sums can be handled without additional codification effort.
Chapter 1.

Preliminaries

In this chapter we give an overview of the classical approaches in SMT solving both in general and in the context of the SMT-RAT framework. For this purpose, we first define the notion of SMT problems for arbitrary underlying theories and introduce the existential fragments of nonlinear real and integer arithmetics as two of their most important instances. We then sketch the basic scheme of DPLL-based SMT solving and show its impact on the modular design of the SMT-RAT framework. This will clarify the setting in which our own implementationary work is settled.

1.1. Satisfiability Modulo Theories

The Satisfiability Modulo Theories (SMT) problem is a generalization of the well-known satisfiability (SAT) problem. A SAT problem instance consists of a formula $\Phi$ in propositional logic, which is a combination of Boolean-valued variables $b_1, \ldots, b_m$ with connectives $\neg, \land, \lor$ and $\rightarrow$. It asks for an interpretation $I : \{b_1, \ldots, b_m\} \rightarrow \{\text{True}, \text{False}\}$ of the variables, such that $\Phi$ evaluates under $I$ to True, which is abbreviated by $I \models \Phi$.

The SAT problem replaces the Boolean-valued variables in favor of constraints $c_1, \ldots, c_m$ that are expressed in the context of a theory $T$, consisting of a domain $D$ (like $\mathbb{R}$) alongside with interpretations for all function symbols $f_1, \ldots, f_k$ (like $+$) and predicate symbols $\sim_1, \ldots, \sim_l$ (like $<$). The variables $x_1, \ldots, x_d$ in $\Phi$ are no longer Boolean-valued, but range over the domain $D$. Solving the SMT instance $\Phi$ in the theory $T$ means deciding whether an interpretation $I : \{x_1, \ldots, x_d\} \rightarrow D$ exists, such that $I \models \Phi$ with respect to $T$.

The SAT problem is known to be NP-hard, though decidable, since we may simply enumerate all possible interpretations for a given instance. The decidability of the SMT problem, on the other hand, depends heavily on the underlying theory $T$ and additional restriction to specific fragments of the first-order logic. The focus of this thesis lies the quantifier-free fragment of the nonlinear real and integer arithmetic:

**Definition 1.1** The syntax of a formula in the quantifier-free fragment of the nonlinear
real and integer arithmetic is defined by the following grammar:

\[
\begin{align*}
\text{formula} & ::= \text{constraint} \mid (\neg \text{formula}) \mid (\text{formula} \land \text{formula}) \mid (\text{formula} \lor \text{formula}) \\
\text{constraint} & ::= \text{term} \sim \text{term} \quad \text{for} \quad \sim \in \{\leq, =, \neq, \geq, >\} \\
\text{term} & ::= v \mid c \mid \text{term} + \text{term} \mid \text{term} \cdot \text{term} \quad \text{for} \quad v \in \{x_1, \ldots, x_d\}, \ c \in \mathbb{R}
\end{align*}
\]

Depending on the domain \(D = \mathbb{R}\) or \(D = \mathbb{Z}\), we distinguish the nonlinear real (QF_NRA) from the integer (QF_NIA) arithmetic and simply write QF_NA to denote any of these nonlinear arithmetic problems.

The QF_NA formulas \(\Phi\) are hence arbitrarily shaped Boolean combinations of polynomial inequalities. For a concise description of these formulas, we will subsequently rely on the outcome of the following lightweight normalization steps:

(i) For an exponent vector \(p = (p_1, \ldots, p_d) \in \mathbb{R}^d\) and a vector of real- or integer-valued variables \(x = (x_1, \ldots, x_d)\), we denote by \(x \cdot p := \sum_{i=1}^{d} x_i p_i\) the usual dot product and by \(x^p := \prod_{i=1}^{d} x_i^{p_i}\) the monomial exponent. Every multivariate polynomial \(f(x) \in \mathbb{R}[x_1, \ldots, x_d]\) can now be written in a \textit{sparse distributive notation} as

\[
f(x) = \sum_{p \in \text{fr}(f)} f_p x^p \quad \text{with} \quad \text{fr}(f) := \{p \in \mathbb{Z}^d \mid f_p \neq 0\},
\]

where the \textit{frame} \(\text{fr}(f)\) denotes its supporting set. Every constraint \(c_i\) in \(\Phi\) can thus be written as \(f_i(x) \sim_i 0\) for a relation symbol \(\sim_i \in \{<, \leq, =, \neq, \geq, >\}\).

(ii) The given QF_NA formula \(\Phi\) can be transformed in linear time into an equisatisfiable formula \(\Phi_{\text{CNF}}\) in conjunctive normal form by using Tseitin’s encoding to get

\[
\Phi_{\text{CNF}} = \bigwedge_{i=1}^{k} \bigvee_{j=1}^{l_i} \ell_{ij} \quad \text{with} \quad \ell_{ij} \in \{c_{ij}, \neg c_{ij}\}
\]

for QF_NA constraints \(c_{ij}\). Next, the negations in negative literals \(\ell_{ij} = \neg c_{ij}\) with a constraint \(c_{ij} = f_{ij}(x) \sim_{ij} 0\) can be eliminated by pushing them to the theory constraints: \(<, \leq\) and \(=\) get replaced with \(\geq, >\) and \(\neq\) and vice versa.

In the following, we will assume that these transformations were already done and that \(\Phi\) is a CNF formula consisting of unnegated constraints in sparse distributive representation

\[
\Phi = \bigwedge_{i=1}^{k} \bigvee_{j=1}^{l_i} c_{ij} \quad \text{with} \quad c_{ij} = f_{ij}(x) \sim_{ij} 0.
\]

For \(i = 1, 2\) let \(\Phi_i := \bigwedge_{j=1}^{k_i} \omega_{i,j}\) be two of these CNF formulas with clauses \(\omega_{i,j}\). For a
1.2. DPLL-based SMT solving

Several tools for deciding the satisfiability of SMT formulas over a quantifier-free first-order theory \( T \) rely on the DPLL(\( T \)) framework: They combine a Boolean satisfiability solver based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure to resolve the Boolean structure of a given formula, and a dedicated theory solver capable of verifying the consistency of theory constraints conjunctions. In what follows, we have a closer look on the DPLL(QF NA) approach (see \[KS08\] Chapter 2).

From a normalized input formula \( \Phi \) as described in the last subsection, the Boolean abstraction \( \Phi_{\text{Bool}} \) is constructed by introducing a fresh Boolean variable \( e_{ij} \) for every constraint \( c_{ij} \) and keeping the Boolean skeleton intact, which gives

\[
\Phi_{\text{Bool}} = \bigwedge_{i=1}^{k} \bigvee_{j=1}^{l_i} e_{ij}.
\]

A DPLL-based SAT solver operating in less lazy mode now systematically tries to find partial interpretations \( I \) for the variables \( e_{ij} \) that do not contradict the Boolean skeleton \( \Phi_{\text{Bool}} \) (see Figure [1.1]). After each variable assignment, the constraints conjunction

\[
\Phi_{\text{Theory}} = \bigwedge_{I|=e_{ij}} c_{ij}
\]

is handed over to the theory solver and checked for consistency. Recall that the Boolean abstraction \( \Phi_{\text{Bool}} \) does not contain any negations and is therefore a monotone formula. Hence, the original formula \( \Phi \) must be satisfiable if and only if \( \Phi_{\text{Theory}} \) is consistent. If the theory solver fails to find a solution of the given constraints conjunction \( \Phi_{\text{Theory}} \), it provides a preferably minimal unsatisfiable core \( \Phi_{\text{Inf}} \subseteq \Phi_{\text{Theory}} \) as explanation to the SAT solver, which is used to narrow down the search for feasible assignments. Depending on the answer of the theory solver on this constraint conjunction, the SAT solver can adjust its partial solution until a complete assignment is found. The formula is declared to be unsatisfiable, if the SAT solver is not able to find any further interpretations for \( \Phi_{\text{Bool}} \).
This DPLL(QF_NA) approach in less lazy mode requires a theory solver for the QF_NA theory that supports the following minimal functionality known as \textit{SMT-compliance}:

\textbf{Incrementality} It has to manage an internal state to make use of previous consistency checks as the input formulas $\Phi_\text{Theory}$ do not vary too much between two successive invocations of the theory solver. It should therefore allow the belated assertion of new and removal of already asserted constraints to the constraints conjunction.

\textbf{Correctness and Termination} The consistency check over all asserted constraints must terminate in finite time and return a \textit{SAT/UNSAT} answer. In case of satisfiability a model must be constructed and returned. Otherwise, it must provide a preferably minimal unsatisfiable core $\Phi_\text{Inf}$ as explanation for the unsatisfiability of $\Phi_\text{Theory}$.

For many theory solvers that only implement a sufficient satisfiability condition like our own linearization approaches, the second point is illusory. We therefore allow a third answer \textit{Unknown} that can be returned, if the consistency of $\Phi_\text{Theory}$ is undecidable.

\section{1.3. SMT-RAT}

The SMT toolbox \texttt{SMT-RAT} \cite{Cor+12} is an open-source project written in C++ for SMT solving over several background theories. The toolbox is structured into its basic architectural components called \textit{modules} that provide SMT compliant implementations of decision procedures. Every module maintains a list $C_{\text{rec}}$ of received formulas whose conjunction needs to be checked for consistency the next time when the \texttt{check} method is called. This list can be modified in an incremental fashion with the use of the \texttt{assert} and \texttt{remove} methods to add new and remove already added formulas. Different modules can be stacked together with the help of \textit{managers} to a complete solving strategy. Every
module decides himself which formulas to delegate to succeeding modules that we call his backends. Figure 1.2 shows a strategy in which the prototypical linearization module LinModule is a placeholder for any of our two modules STropModule and CSplitModule. It is preceded by an instance of a SATModule that implements a DPLL-based SAT solver to resolve the Boolean skeleton of the input formula $\Phi$. Our linearization modules therefore do not receive arbitrary formulas, but only conjunctions of polynomial constraints as described in Section 1.2. They perform a linearization of the nonlinear input and pass the result to an internal linear arithmetic solver LRAModule. Since the later also expects its own input to be a constraints conjunction, the linearization must be piped beforehand into a second instance of a SATModule. The linearization modules are both sound, but incomplete, for which reason they call their Backends on their complete input $C_{rec}$ in case they are unable to decide the consistency themselves. In our experiments, the Backend strategy is a combination of the following decision procedures already implemented as SMT-RAT modules:

**LRAModule** This is a misnomer, since it not only implements the Simplex method to tackle linear real arithmetic problems, but also performs Branch-and-Bound on all integer-valued variables to effectively handle any linear mixed-integer problems.

**ICPModule** Interval Constraint Propagation uses the given constraints to iteratively contract the search space until an interval for every variable is reached that tightly over-approximates the solution set satisfying some preset precision requirement.

**VSModule** The Virtual Substitution method exploits the existence of closed form solutions for univariate polynomials up to degree four to successively eliminate variables.

**CADModule** The Cylindrical Algebraic Decomposition algorithm decomposes the search space into a finite number of connected sets called cells, on which each polynomial of the input constraints has constant sign. The satisfiability can then be decided by testing their consistency at single sample points in each cell.
Chapter 2.

Subtropical Satisfiability

In this chapter we provide the theoretical foundations for our SMT-RAT implementation of the subtropical module (STropModule) that is fully presented in Appendix A. It is inspired by the incomplete but terminating subtropical real root finding method as described in [Stu15] that identifies roots of very large multivariate polynomials. The algorithm takes an abstract view of a polynomial as the set of its exponent vectors tagged with sign information on the corresponding coefficients. It then examines the limiting behaviour of the polynomial in the direction of a normal vector to find real zeros. The search for such a normal vector is translated into a linear problem in the space of the polynomial’s exponent vectors and in turn solved using a linear real arithmetic solver. In the context of nonlinear real arithmetic problems in SMT, this algebraic root finding method was first generalized in [Fon+17] to find solutions of a conjunction of strict polynomial inequalities. In the following sections, we will describe further enhancements that could improve the completeness of the original method with slightly more overhead.

2.1. Limiting behaviour of multivariate polynomials

In the following we will examine sufficient conditions for the existence of solutions to the problem description given below. These results based on the subtropical real root finding method will be used afterwards to decide the satisfiability of a conjunction of nonlinear real and integer arithmetic constraints in the context of Sat Modulo Theories solving.

Problem 2.1 Let \( f(x) \in \mathbb{R}[x_1, \ldots, x_d] \) be a multivariate polynomial. Find a real-valued variable assignment \( a \in \mathbb{R}^d \) such that the inequality \( f(a) > 0 \) is fulfilled.

The idea of an incomplete algorithm for Problem 2.1 is best described in the one-dimensional case \( d = 1 \): For a univariate polynomial \( f(x) \in \mathbb{R}[x] \) consider the limiting process \( \lim_{a \to L} f(a) \), as \( a \) approaches one of the one-sided limits \( L \in \{-\infty, \infty\} \). Repeatedly test the sign of \( f(a) \) until finally \( f(a) > 0 \) is fulfilled, if this happens at all. The examination of four different limit values \( L \) can be reformulated as the choice of a sign \( s \in \{-1, 1\} \) and an
Chapter 2. Subtropical Satisfiability

Chapter 2. Subtropical Satisfiability

Figure 2.1.: Visualization of the polynomial $f(x, y) := -4x^4y^4 - x^3 - 3x^2y^2 + 2xy^3 + y$.

(a) Momentum curve $m(t \mid s, n) := (t^{-2}, t^3)$ on its surface that eventually becomes positive.

(b) Projection of frame vertices onto the hyperplane in direction of $n = (-2, 3)$.

Example 2.3 Consider the bivariate polynomial

$$f(x, y) := -4x^4y^4 - x^3 - 3x^2y^2 + 2xy^3 + y \in \mathbb{R}[x, y].$$
Figure 2.1a shows an example for a good choice of a normal vector \( \mathbf{n} = (-2, 3) \) and a sign variant \( \mathbf{s} = (1, 1) \) such that the resulting Laurent polynomial

\[
L(t \mid f, s, n) = 2t^7 - 4t^4 + t^3 - 3t^2 - t - 1 \in \mathbb{R}[t, t^{-1}]
\]

eventually becomes positive, for instance at \( t = 2 \). A corresponding satisfying assignment for the inequality \( f(x, y) > 0 \) in Problem 2.1 is then given by \( \mathbf{a} := \mathbf{m}(t \mid s, n) = (2^{-2}, 2^3) \).

The search for a variable assignment \( \mathbf{a} \in \mathbb{R}^d \) for the strict inequality \( f(x) > 0 \) hence boils down to a search for a normal vector \( \mathbf{n} \in \mathbb{Z}^d \) and a sign variant \( \mathbf{s} \in \{\pm 1\}^d \) such that the resulting polynomial \( L(t \mid f, s, n) \) has a positive leading coefficient. A corresponding variable assignment for the inequality can then be reconstructed as the image of the momentum curve \( \mathbf{m}(t \mid s, n) \) for a large enough \( t \in \mathbb{R}_+ \), where \( L(t \mid f, s, n) > 0 \) is fulfilled.

### 2.2. Restriction process as geometric projection

The coefficients of \( L(t \mid f, s, n) \) are composed of those of the multivariate polynomial \( f(x) \). To make this calculation process explicit, take the above definition of the Laurent polynomial and reorder its terms according to the same integral exponents to get

\[
L(t \mid f, s, n) = \sum_{p \in fr(f)} s^p f_p t^{n_p} = \sum_{k \in \mathbb{Z}} L_k t^k \quad \text{with} \quad L_k := \sum_{p \in fr(f) \atop n_p = k} s^p f_p.
\]

Graphically speaking, the exponent vector \( \mathbf{n} \) of the momentum curve defines a hyperplane in the \( \mathbb{Z} \)-lattice of all possible exponent vectors \( \mathbb{Z}^d \) onto which the frame vertices of \( f(x) \) are projected. Those frame vertices \( \mathbf{p} \in fr(f) \) with equal projection length \( \mathbf{n} \cdot \mathbf{p} = k \) to the origin contribute to the same coefficient \( L_k \) weighted with the sign \( s^p \). Let us therefore partition the frame \( fr(f) \) into a variant positive and a variant negative frame by

\[
fr^+(f, s) := \{ \mathbf{p} \in fr(f) \mid s^p f_p > 0 \} \quad \text{and} \quad fr^-(f, s) := \{ \mathbf{p} \in fr(f) \mid s^p f_p < 0 \}.
\]

**Example 2.4** To better understand the choice of the exponent vector \( \mathbf{n} = (-2, 3) \) in Example 2.3 consider the visualization of the frame \( fr(f) \) in Figure 2.1b and its projection to the hyperplane defined by \( \mathbf{n} \). The frame vertex \( \mathbf{p} = (1, 3) \) has the largest projection length \( \mathbf{n} \cdot \mathbf{p} = 7 \) and hence constitutes the leading coefficient \( L_7 = s^p f_p = 2 \). Since this is positive, we can deduce the positive divergence of \( L(t \mid f, s, n) \) for a large enough \( t \in \mathbb{R}_+ \).

Let us enumerate all existing projection lengths descendingly by \( \mathbf{n} \cdot fr(f) = \{k_1, \ldots, k_l\} \) with \( k_1 > \ldots > k_l \). As this example suggests, we have a special interest in the sign of
the coefficient $L_{k_1}$ with the largest existing projection length $k_1 := \max_{p \in \text{fr}(f)} n \cdot p$. But note that this is not necessarily the leading coefficient of $L(t | f, s, n)$, since the signed coefficients $s_p f_p$ it is composed of may cancel out each other yielding an overall value of zero. In the worst case, the whole Laurent polynomial can vanish through the projection making it impossible to draw any conclusions about the limiting behaviour of $f(x)$ for this particular choice of the normal vector $n$ and the sign variant $s$. Nonetheless the approaches we will review in the following sections rely on a positivity check of this coefficient with largest projection length only. Linearization based methods that deduce the sign of the true leading coefficient of $L(t | f, s, n)$ performed poorly in all of our experiments.

### 2.3. Exploiting the linear separability of frame vertices

The positivity condition on the coefficient $L_{k_1}$ can be formulated as a linear real arithmetic formula $\Phi_{\text{Constr}}$ without the need of calculating the entire Laurent polynomial $L(t | f, s, n)$. In the following subsections we will derive two variants of this formula $\Phi_{\text{Constr}}$ dedicated to different linearization approaches to test the positivity of $L_{k_1} = \sum_{n \cdot p = k_1} s_p f_p$:

- The linearization $\Phi^{\text{Str}}_{\text{Constr}}$ of Subsection 2.3.1 is based on the original method in [Fon+17]. It verifies whether the coefficient $L_{k_1}$ is composed of variant positive frame vertices $p \in \text{fr}^+(f, s)$ only, since this trivially implies its own positivity.

- In the concluding remarks of [Fon+17] it is therefore left as a research question to find a linearization method that also allows the summation over variant negative frame vertices as long as the overall value of $L_{k_1}$ is positive. In Subsection 2.3.2 we propose a novel linearization $\Phi^{W}_{\text{Constr}}$ that solves this issue with a more sophisticated analysis of the projected frame vertices. It increases the completeness of the original method while its consistency check is still feasible in a reasonable amount of time.

#### 2.3.1. Strictly separable frame vertices

For a frame vertex $p \in \text{fr}(f)$ the sign of $s^p$ is fully determined by only those $s_i$ with an odd exponent $p_i$. Treating negative signs as True and encoding the sign variant accordingly as a Boolean vector, it can be calculated as the parity of the individual signs by the formula

$$\Phi_{\text{sgn}}(s | p) := \bigoplus_{i=1, \ldots, d, \ p_i \ \text{odd}} s_i.$$
Since the coefficients \( f_p \) are already known constants at the time of linearization, we can encode the membership \( p \in \text{fr}^+(f, s) \) as one static branch of the following case distinction

\[
\Phi_{\text{PosFrm}}(s \mid f, p) := \begin{cases} 
-\Phi_{\text{sgn}}(s \mid p), & \text{if } f_p > 0, \\
\Phi_{\text{sgn}}(s \mid p), & \text{if } f_p < 0.
\end{cases}
\]

The coefficient \( L_{k_1} = \sum_{n \cdot p = k_1} s^p f_p \) with the largest distance \( k_1 := \max_{p \in \text{fr}(f)} n \cdot p \) solely consists of positive frame vertices if and only if there exists a threshold \( b \in \mathbb{R} \) such that

(i) for all negative frame vertices \( p \in \text{fr}^-(f, s) \) it holds that \( n \cdot p \leq b \), and

(ii) there exists at least one positive frame vertex \( q \in \text{fr}^+(f, s) \) with \( n \cdot q > b \).

By strictly separating at least one positive from all negative frame vertices, these conditions test whether all frame vertices \( p \in \text{fr}(f) \) with a distance \( n \cdot p = k > b \) are positive. This implies that especially all the frame vertices with the largest existing distance \( k_1 > b \) out of which \( L_{k_1} \) is composed of must be positive as well. The following formula directly encodes these conditions and can be handed over to a LRA solver for a consistency check:

\[
\Phi_{\text{Const}}(s, n, b \mid f > 0) := \begin{cases} 
\bigwedge_{p \in \text{fr}(f)} \Phi_{\text{PosFrm}}(s \mid f, p) \lor n \cdot p \leq b, & \text{and} \\
\bigvee_{q \in \text{fr}(f)} \Phi_{\text{PosFrm}}(s \mid f, q) \land n \cdot q > b.
\end{cases}
\]

Note that we face a linear real arithmetic problem, since we do not insist \( n = (n_1, \ldots, n_d) \) to be a vector of integral variables. A given solution for a linear formula stays valid even if all variable values are scaled by a common factor. If this linearization is satisfiable, we simply scale the resulting normal vector assignment to get back an integral solution.

**Example 2.5** For our running Example 2.4, we have marked a possible choice for a threshold \( b \) in Subfigure 2.1b. It separates the projected positive frame vertex \( p = (1, 3) \) from all negative frame vertex projections and hence proves that the coefficient \( L_{k_1} \) with \( k_1 > b \) must be composed of positive frame vertices only.

### 2.3.2. Weakly separable frame vertices

As already mentioned, the coefficient \( L_{k_1} = \sum_{n \cdot p = k_1} s^p f_p \) does not need to consist of variant positive frame vertices only to have a positive overall value:

**Example 2.6** Consider the bivariate polynomial

\[
f(x, y) := -x^3 y + 3x^2 y^2 - xy^3 - 1 \in \mathbb{R}[x, y]
\]
Chapter 2. Subtropical Satisfiability

(a) Weakly separable frame vertices of the polynomial $f(x, y) := -x^3y + 3x^2y^2 - xy^3 - 1$.

(b) Linearly inseparable frame of the polynomial $f(x, y) := -x^4 + 2x^2y^2 - y^4 + xy^2 + x^2y - 1$.

Figure 2.2.: Projections onto the plane given by $n = (1, 1)$ with sign variant $s = (1, 1)$ whose frame is visualized in Subfigure 2.2a for a fixed sign variant $s = (1, 1)$. There is no choice for a normal vector $n$ such that $L_{k_1}$ consists of variant positive frame vertices only. However, if we choose the normal vector $n = (1, 1)$, then we get

$$L_{k_1} = f_{(3,1)} + f_{(2,2)} + f_{(1,3)} = -1 + 3 - 1 = 1 > 0$$

which is still positive although we have negative frame vertex contributions.

The idea for a more sophisticated linearization lies in a better analysis of the frame vertices projected onto the threshold border $b \in \mathbb{R}$ in the last subsection. We similarly claim that

(i') for all negative frame vertices $p \in fr^- (f, s)$ it holds that $n \cdot p \leq b$, and

(ii') there exists at least one positive frame vertex $q \in fr^+ (f, s)$ with $n \cdot q \geq b$.

Notice the decisive difference in (ii') compared to the original condition (ii) where we now allow the projected frame vertex to lie on the threshold border through the use of a weak relation. The situation on the threshold border needs additional attention:

(a) If there exists a positive frame vertex $q \in fr^+ (f, s)$ with $n \cdot q > b$, then the strict separability conditions (i) and (ii) of Subsection 2.3.1 are satisfied and $L_{k_1}$ is positive.

(b) Otherwise we also have $n \cdot q \leq b$ for all variant positive frame vertices $q \in fr^+ (f, s)$ and by condition (ii') there is at least one such vertex with $n \cdot q = b$. It follows that $k_1 = b$ and we therefore claim the coefficient $L_b = \sum_{n.p=b} f_p$ to be positive.

Both of these cases can be handled simultaneously by the following reformulated condition that furthermore allows a very elegant encoding into a linear real arithmetic formula:
2.3. Exploiting the linear separability of frame vertices

(iii') The total rating of \( f(x) \) at the threshold border \( b \) is defined by

\[
    r(f \mid s, n, b) := \sum_{p \in fr(f)} r_p
\]

with

\[
    r_p := \begin{cases} 
        +\infty, & \text{if } n \cdot p > b, \\
        s^p f_p, & \text{if } n \cdot p = b, \\
        0, & \text{if } n \cdot p < b.
    \end{cases}
\]

Suppose that the weak separability conditions (i') and (ii') above are given. Then the two cases (a) and (b) are equivalent to the inequality \( r(f \mid s, n, b) > 0 \).

Treating the \( r_p \) as additional indeterminates, an encoding of condition (iii') is given by

\[
    \Phi_{Rtg}(s, n, b, r \mid f) := \sum_{p \in fr(f)} r_p > 0 \land \left[ \bigwedge_{p \in fr(f)} n \cdot p < b \rightarrow r_p = 0 \right] \land \\
    \left[ \bigwedge_{p \in fr(f)} n \cdot p = b \rightarrow (\Phi_{Sgn}(s \mid p) \land r_p = -f_p) \lor (-\Phi_{Sgn}(s \mid p) \land r_p = f_p) \right],
\]

Note that the remaining case \( n \cdot p > b \rightarrow r_p = +\infty \) is superfluous and deliberately not encoded to reduce the size of the linearization. If the premise \( n \cdot p > b \) is satisfied, then the variable \( r_p \) is not fixed by the above formula. The LRA solver is able to assign an arbitrarily large value to \( r_p \) in order to fulfill the total rating constraint \( \sum_{p \in fr(f)} r_p > 0 \), which would also be the desired effect of the conclusion \( r_p = +\infty \). The full linearization taking also the weak separability conditions (i') and (ii') into account is now given by

\[
    \Phi_{WkConstr}(s, n, b, r \mid f > 0) := \Phi_{Rtg}(s, n, b, r \mid f) \land \\
    \left[ \bigwedge_{p \in fr(f)} \Phi_{PosFrm}(s \mid f, p) \lor n \cdot p \leq b \land \bigvee_{q \in fr(f)} \Phi_{PosFrm}(s \mid f, q) \land n \cdot q \geq b \right].
\]

A brief inspection of this formula shows that condition (ii') is already included in condition (iii'). If there is no positive frame vertex \( q \in fr^+(f, s) \) with \( n \cdot q \geq b \), then the total rating of \( f(x) \) cannot be positive. Eliminating this redundancy from the linearization however lead to an extraordinary increase of the runtime for its consistency check in all our experiments. In case of a conflict the total rating formula \( \Phi_{Rtg}(s, n, b, r \mid f) \) provides very few information for its resolution. The redundant encoding of condition (ii') excludes obviously unsatisfiable choices for the indeterminates \( s, n \) and \( b \) before the total rating formula gets evaluated.

**Example 2.7** Reconsider the Example 2.6 and its visualization in Subfigure 2.2a. There is no threshold \( b \) that strictly separates at least one positive frame vertex from all negative frame vertices. But if we choose the threshold \( b = 4 \), then we get the vertex ratings...
Chapter 2. Subtropical Satisfiability

\[ r_{(1,3)} = -1, \ r_{(2,2)} = 3, \ r_{(3,1)} = -1, \text{ and } r_{(0,0)} = 0. \] The total rating of \( f(x) \) is therefore given by \( r(f \mid s, n, b) = L_{k_1} = 1 \) which fulfills the total rating condition \((iii')\).

2.3.3. Linearly inseparable frame vertices

As already mentioned, the coefficient \( L_{k_1} \) corresponding to the largest projection length \( k_1 \) does not necessarily represent the leading coefficient of \( L(t \mid f, s, n) \), since it may vanish.

Example 2.8 Consider the visualization of the bivariate polynomial

\[ f(x, y) := -x^4 + 2x^2y^2 - y^4 + xy^2 + x^2y - 1 \in \mathbb{R}[x, y] \]

in Subfigure 2.2b. If we choose \( n = (1, 1) \) and \( s = (1, 1) \), then we get

- \( L_4 = f_{(4,0)} + f_{(2,2)} + f_{(0,4)} = -1 + 2 - 1 = 0 \),
- \( L_3 = f_{(1,2)} + f_{(2,1)} = 1 + 1 = 1 \),
- \( L_0 = f_{(0,0)} = -1 \),

and the full Laurent polynomial is given by \( L(t \mid f, s, n) = t^3 - 1 \). The true leading coefficient \( L_3 \) is positive, but the coefficient \( L_4 \) with the largest projective distance vanishes. Choosing another sign variant \( s \) is not an option, since the variant negative frame \( fr^{-}(f, s) \) consists of vertices with even parity and hence is invariant to sign changes.

This problem can get even worse if not only \( L_{k_1} \) is zero, but a whole sequence of coefficients \( L_{k_i} \) for \( i = 1, 2, \ldots \) up to the point where the whole Laurent polynomial vanishes. To overcome this issue, we were able to find a linearization that starting with \( i = 1, 2, \ldots \)

(i) tests whether \( L_{k_i} > 0 \) with a threshold border \( b_i \) using the weak separability method,

(ii) in case \( L_{k_i} = 0 \) uses the threshold \( b_i \) to discard all frame vertices \( p \in fr(f) \) with \( n \cdot p \geq b \) for the next iteration of the weak separability method.

The consistency check of the resulting linearization was infeasible in a reasonable amount of time even on hand-crafted toy examples with only three variables. This generalization was therefore discarded from our final STropModule code base.

2.4. Application to the SMT problem

The so far presented approaches identify real-valued assignments, where only a single multivariate polynomial becomes positive. We will use the derived linearizations in the strict and weak encoding variants \( \Phi_{\text{Constr}}^{\text{Str}}(s, n, b \mid f > 0) \) and \( \Phi_{\text{Constr}}^{\text{Wk}}(s, n, b, r \mid f > 0) \),
2.4. Application to the SMT problem

respective, as interchangeable building blocks to solve the DPLL based SMT Problem. Let us denote any of these two formulas by $\Phi_{\text{Constr}}(s, n, b[, r] \mid f > 0)$ for convenience.

2.4.1. Single constraint with an arbitrary relation

A single constraint $f(x) \sim 0$ with an arbitrary relation symbol $\sim \in \{<, \leq, =, \neq, \geq, >\}$ can be reduced to the already known case by applying the following rewriting rules:

$$
\Phi_{\text{Constr}}(s, n, b[, r] \mid f \sim 0) := \begin{cases} 
\Phi_{\text{Constr}}(s, n, b[, r] \mid -f > 0), & \text{if } \sim \in \{<, \leq\} \\
\Phi_{\text{Constr}}(s, n, b[, r] \mid -f > 0) \lor \Phi_{\text{Constr}}(s, n, b[, r] \mid f > 0), & \text{if } \sim \in \{\neq\} \\
\Phi_{\text{Constr}}(s, n, b[, r] \mid f > 0), & \text{if } \sim \in \{\geq, >\}
\end{cases}
$$

It is worth mentioning that the relation symbols $\{<, \leq\}$ and $\{\geq, >\}$ define classes with the same limiting behaviour and hence are mapped to the same linearization. Furthermore, the lack of a rewriting rule for the case of an equality relation $=\neq$ is no mistake: As seen in Section 2.3 our linearization method is based on the linear separability of positive and negative frame vertices. But rewriting $f(x) = 0$ as the conjunction $-f(x) \geq 0 \land f(x) \geq 0$ and encoding both clauses independently from each other will lead to a linearization

$$
\Phi_{\text{Constr}}(s, n, b[, r] \mid -f > 0) \land \Phi_{\text{Constr}}(s, n, b[, r] \mid f > 0)
$$

that is always inconsistent: By the coincidence $fr^+(f, s) = fr^-(\neg f, s)$ there is no normal vector $n$ such that the linear separability conditions in Subsections 2.3.1 and 2.3.2 are fulfilled. But from the unsatisfiability of this linearization we are unable to draw any conclusions about the consistency of the original constraint $f(x) = 0$ without the aid of back-end solvers, since the limiting behaviour analysis is insufficient to exclude solutions within a bounded support. Hence, the presence of equality constraints leads to an immediate abort of our subtropical method with an unknown result. In every other case, provided the consistency check verifies the satisfiability of the linearization $\Phi_{\text{Constr}}(s, n, b[, r] \mid f \sim 0)$, a solution for the original constraint $f(x) \sim 0$ can be reconstructed as the image of the momentum curve $m(t \mid s, n)$ for a large enough $t \in \mathbb{R}_+$, where $L(t \mid f, s, n) \sim 0$ is fulfilled.

2.4.2. Common solution of multiple constraints

Let a constraint conjunction $C = \bigwedge_{i=1}^n f_i(x) \sim_i 0$ with polynomials $f_i(x) \in \mathbb{R}[x_1, \ldots, x_d]$ and relations $\sim_i \in \{<, \leq, =, \neq, \geq, >\}$ be given. If the independently linearized formulas $\Phi_{\text{Constr}}(s, n, b[, r] \mid f_i \sim_i 0)$ share their sign variant $s \in \{\pm 1\}^d$ and their normal vector
\( n \in \mathbb{R}^d \), then in case of satisfiability a common solution for all constraints will be given by \( m(t \mid s, n) \) for a large enough \( t \in \mathbb{R}_+ \), where \( L(t \mid f_i, s, n) \sim_i 0 \) for all \( i = 1, \ldots, m \) is fulfilled. We hence linearize \( C \) by

\[
\Phi_{\text{SMT}}(s, n, b_1, \ldots, b_m, r_1, \ldots, r_m) \mid \bigwedge_{i=1}^m f_i \sim_i 0 := \bigwedge_{i=1}^m \Phi_{\text{Constr}}(s, n, b_i, r_i \mid f_i \sim_i 0).
\]

In addition to the already described short-circuiting for equality constraints in Subsection \ref{sec:short-circuit}, the combination of multiple constraints can help avoiding a consistency check in many more cases. For this purpose we normalize the left hand side of every constraint \( f_i(x) \sim_i 0 \) by enforcing a unit leading coefficient and turning the relation symbol accordingly. Constraints with the same left hand side \( f(x) \) but different relations are

- **unsatisfiable** if the relations contradict each other in the cases \( \{<,=\}, \{<,>\}, \{<,\ge\}, \{\le,>\} \) and \( \{=,>\} \). From contradicting relations \( \{\sim_1, \sim_2\} \) an unsatisfiable core \( f(x) \sim_1 0 \land f(x) \sim_2 0 \) can be generated before terminating with an **Unsat** result.

- **undecidable** if the relations resemble an equality in the cases \( \{=\} \) and \( \{\le,\ge\} \). We therefore skip the application of the subtropical method and directly call the backend solvers of the defined strategy tree on the given constraint conjunction \( C \).

Interestingly, when these two sources for a fast skip of the subtropical method are excluded, only a single relation for the same left hand side \( f(x) \) can be active. Take as an example the relations \( \{\ne, \ge, >\} \), where we only need to use the strictest relation \( > \) for a linearization. These simple optimizations accelerated the consistency check on the benchmarks presented in Section \ref{sec:benchmarks} by more than ten times. We discovered that a large number of the constraint conjunctions given to our **STropModule** can be decided or skipped by these simple pre-tests without an invocation of the LRA solver on the linearizations.

### 2.4.3. Extension to mixed-integer problems

The so far presented subtropical method is defined as a lightweight decision procedure for real arithmetic problems only. We propose the following extension to the original algorithm that works pretty well on mixed-integer problems with a small number of integer-valued coordinates. Remember that in case of satisfiability of the constructed linearization \( \Phi_{\text{SMT}}(s, n, \{b_i[r_i]\}^m_{i=1} \mid \{f_i \sim_i 0\}^m_{i=1}) \) a corresponding variable assignment for the original constraint conjunction \( C = \bigwedge_{i=1}^m f_i(x) \sim_i 0 \) is given by the image of the momentum curve

\[
a := m(t \mid s, n) = (s_1 t^{m_1}, \ldots, s_d t^{m_d}) \quad \text{for} \quad t \gg 1.
\]
2.5. Benchmarking results and conclusion

A coordinate $x_i$ will receive an integral solution $a_i$ if and only if we can ensure the integrity of the expression $s_i t^m$. As a sufficient condition, we enforce the normal vector component $n_i$ and the sample point $t$ to take non-negative integral values. We therefore assert

$$\Phi_{\text{Int}}(n \mid x) := \bigwedge_{i=1}^{d} n_i \geq 0$$

and only use test points $t \in \mathbb{N}$ for the later variable assignment reconstruction. Restricting a single normal vector value $n_i$ to the positive axis effectively halves the solution search space for the linearization. We hence note that with a growing number of integer-valued coordinates $x_i$ our STropModule is less likely to give a satisfiable answer.

2.5. Benchmarking results and conclusion

We tested our STropModule implementation on the QF_NRA division of the SMT-LIB benchmarking library \cite{BFT16} stemming from the industrial and academic world. Every benchmark consists of an input formula and the expected answer of its consistency check, if the status is already defined. They are grouped into families of similar problems like termination proofs or elementary function approximations and have a variable complexity regarding the formula sizes and the number of variables. We performed the experiments in Table 2.1 on a 2.7 GHz Intel Core i7-4800MQ CPU with a timeout of 60 seconds and 4 GB memory per benchmark. If any of these resource limits is exceeded, the corresponding run gets terminated with a Resout answer by our benchmarking scheduler. Otherwise, we report the number of benchmarks (Num) and the average runtime (Avg) in milliseconds for each possible answer Sat, Unsat or Unknown and for each of the following strategies:

**STrop only**: SATModule $\rightarrow$ STropModule

**Backends only**: SATModule $\rightarrow$ ICPModule $\rightarrow$ VSModule $\rightarrow$ CADModule

**STrop + Backends**: SATModule $\rightarrow$ STropModule $\rightarrow$ ICPModule $\rightarrow$ VSModule $\rightarrow$ CADModule

The strict and weak variants denote the two linearization methods from Section 2.3. The strategy that combines the strict variant of our STropModule with the standard backends consistently outperforms the pure backends on almost every benchmark family. Even for those benchmarks, where the number of answers of one class are kept stable, we observe a considerable decrease of the average runtime. As an example for this behaviour, compare the Sat answers for the kissing family or the Unsat answers for the Sturm MBO/MGC family. The later shows another interesting property of DPLL based theory.
solvers in general: Although the subtropical method only focuses on proving satisfiability, it is able to accelerate also unsatisfiable answer deductions. It happens that many of the partial CNF formulas passed to the theory solver are satisfiable, even if the complete formula that gets checked is unsatisfiable. On the remaining instances, our STropModule fails quickly with an *Unknown* result and thus generates a very small overhead. To see this, consider the pure subtropical method on the Heizmann Ultimate Invariant Synthesis and Ultimate Automizer families. It is able to generate *Unknown* answers, where the full strategies timeout, proving its efficiency. For the final question which linearization variant to use, consider the meti-tarski and zankl families: As a theory solver on its own, the additional completeness of the weak variant comes with a 25 fold and a 84 fold increase of the average runtime, respectively. If it is used upfront to more sophisticated theory solvers for the nonlinear real arithmetic, its linearization complexity degrades the performance of the full strategy compared to the strict variant. In an environment like SMT-RAT, where already more complete decision procedures exist, we therefore highly recommend the strict variant as a lightweight heuristic to decide simpler input instances.
2.5. Benchmarking results and conclusion
<table>
<thead>
<tr>
<th>Benchmark Family</th>
<th>Answer</th>
<th>STrop strict only</th>
<th>STrop weak only</th>
<th>STrop strict + Backends</th>
<th>STrop weak + Backends</th>
<th>Backends only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sturm MBO/MGC (414)</td>
<td>Sat (107)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Unsat (292)</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>9</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td>Unknown (15)</td>
<td>412</td>
<td>7475</td>
<td>412</td>
<td>7722</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Resout</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Heizmann Ultimate Invariant Synthesis (69)</td>
<td>Sat (0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Unsat (0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Unknown (69)</td>
<td>51</td>
<td>4255</td>
<td>51</td>
<td>4466</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Resout</td>
<td>18</td>
<td>60156</td>
<td>18</td>
<td>60032</td>
<td>69</td>
</tr>
<tr>
<td>hong (20)</td>
<td>Sat (0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Unsat (20)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Unknown (0)</td>
<td>20</td>
<td>206</td>
<td>20</td>
<td>239</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Resout</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>hycomp (2752)</td>
<td>Sat (191)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Unsat (2191)</td>
<td>1898</td>
<td>1419</td>
<td>1898</td>
<td>1507</td>
<td>1804</td>
</tr>
<tr>
<td></td>
<td>Unknown (370)</td>
<td>19</td>
<td>5220</td>
<td>19</td>
<td>5505</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Resout</td>
<td>835</td>
<td>57952</td>
<td>835</td>
<td>58011</td>
<td>924</td>
</tr>
<tr>
<td>kissing (45)</td>
<td>Sat (42)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Unsat (3)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Unknown (0)</td>
<td>45</td>
<td>41</td>
<td>45</td>
<td>41</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Resout</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>35</td>
</tr>
</tbody>
</table>
### Table 2.1: Benchmarking results of the STropModule on the QF_NRA division of the SMT-LIB.

<table>
<thead>
<tr>
<th></th>
<th>Sat (121)</th>
<th>Unsat (133)</th>
<th>Unknown (567)</th>
<th>Resout</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LassoRanker</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>821</td>
</tr>
<tr>
<td><strong>(821)</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>60020</td>
</tr>
<tr>
<td><strong>meti-tarski</strong></td>
<td>1277</td>
<td>1346</td>
<td>4179</td>
<td>821</td>
</tr>
<tr>
<td><strong>(7006)</strong></td>
<td>11</td>
<td>281</td>
<td>184</td>
<td>60012</td>
</tr>
<tr>
<td><strong>Ultimate</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>821</td>
</tr>
<tr>
<td><strong>Automizer</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>60005</td>
</tr>
<tr>
<td><strong>(61)</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>60005</td>
</tr>
<tr>
<td><strong>zankl</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>(166)</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>(11354)</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.1: Benchmarking results of the STropModule on the QF_NRA division of the SMT-LIB.
Chapter 3.

The Case-Splitting Method

In this chapter we illustrate the implemented functionality behind our case-splitting module (CSplitModule) that is presented in Appendix B. In their first publication Borralleras et al. proposed a reduction method for nonlinear to linear integer arithmetic formulas generalizing the idea previously known from bit-blasting to a higher-order target logic. It is based on the linearization of nonlinear monomials by a repeated application of a case analysis on the possible values that some of the variables in the monomial can take. To ensure completeness, this method requires the domains of variables used for case-splits to be finite. In reality, this basic idea quickly loses its termination power for bounded, but large variable domains. In the follow-up paper this issue is addressed by replacing the unary encoding of large domains through an improved encoding in a positional numeral system. Additionally, the authors present a novel method to handle entirely unbounded domains via an incremental approach to introduce and expand artificial bounds in a clever way. This allows us to give unsatisfiability answers for certain input formulas even in the later case and therefore improves the completeness of the original method.

3.1. Case-splits for monomial equalities

The main idea of the case-splitting method is the linearization of nonlinear monomials by a repeated application of a case analysis on the possible values that some of the variables in the monomial can take. Consider for example the nonlinear integer arithmetic formula

\[ x = abc \land 5 \leq x \leq 10 \land 2 \leq a, b, c \leq 3. \]

Since the variable \( a \) is integral, it can only take the values \( a = 2 \) or \( a = 3 \), and we get an equisatisfiable formula by replacing the constraint \( x = abc \) by a simple case distinction as

\[
\begin{align*}
  y &= bc \land a = 2 \rightarrow x = 2y \land 5 \leq x \leq 10 \\
  &\land a = 3 \rightarrow x = 3y \land 2 \leq a, b, c \leq 3.
\end{align*}
\]
Note the additional substitution of the nonlinear expression \( bc \) by a fresh intermediate variable \( y \). This produces a new nonlinear equality \( y = bc \) with a smaller total degree, and apart from that linear constraints only. Another case-split on the variable \( b \) gives

\[
\begin{align*}
 b = 2 & \rightarrow y = 2c \quad \land \quad a = 2 \rightarrow x = 2y \quad \land \quad 5 \leq x \leq 10 \\
\land \quad b = 3 & \rightarrow y = 3c \quad \land \quad a = 3 \rightarrow x = 3y \quad \land \quad 2 \leq a, b, c \leq 3,
\end{align*}
\]

which can be checked by a linear integer arithmetic solver such as the Branch-and-Bound method for satisfiability. The resulting model \( \{ a = 2, b = 2, c = 2, y = 4, x = 8 \} \) also satisfies the original formula, where the additional assignment \( y = 4 \) for the intermediate variable can be dropped. This trivial example points to the main challenges of the method:

- In the foregoing example, there were only two possible values for the variables \( a \) and \( b \) used for case distinctions. In the following sections, we will develop improved linearization techniques for case variables with large but bounded domains.

- If the domains of variables used for case distinctions are even unbounded, we need to introduce new bounds and hence we lose completeness. A proper analysis of the unsatisfiable core of the background linear arithmetic solver however will allow us to prove unsatisfiability even in this case. Furthermore, it will yield an efficient incremental method to choose the variable bounds that need to be enlarged in order to continue the search for a satisfying assignment for the original set of constraints.

In Section 3.2 we first describe some lightweight preprocessing steps to extract monomial equalities from a given conjunction of constraints. The remaining sections are then devoted to the linearization of those nonlinear monomial equalities.

### 3.2. Purification of nonlinear constraints

Let \( C := \bigwedge_{i=0}^m f_i(x) \sim_i 0 \) be a conjunction of constraints with multivariate polynomials \( f_i(x) \in \mathbb{R}[x_1, \ldots, x_d] \) and relations \( \sim_i \in \{<, \leq, =, \neq, \geq, >\} \). Consider the mixed-integer problem of finding a satisfying assignment \( a \in \mathbb{R}^k \times \mathbb{Z}^{d-k} \).

#### 3.2.1. Discretization of real-valued variables

Since the main case-splitting method is designed to solve integer arithmetic problems only, we first need to get rid of the real-valued variables \( x_1, \ldots, x_k \). In [Bor+12] several discretization techniques are considered:

**Constant Denominator** Fix a common denominator \( D \in \mathbb{Z}, D \neq 0 \). For \( i = 1, \ldots, k \)
choose fresh integer-valued variables \( n_i \) and perform the substitution \( x_i := \frac{n_i}{D} \). Whenever this method is applied, we definitely lose completeness by excluding point solutions that cannot be written as a quotient with denominator \( D \).

**Full Quotient** For \( i = 1, \ldots, k \) choose fresh integer-valued variables \( n_i \) and \( d_i \). Perform the substitution \( x_i := \frac{n_i}{d_i} \) and eliminate the denominators in \( C \) by multiplying the constraints with a power of \( d_i \) to get back an integer arithmetic formula.

Although the full quotient method is more expressive, the explosion of the monomial degrees turned out to be computationally infeasible in all our experiments. The number of intermediate variables needed to linearize the formula results in a very poor performance of the linear arithmetic solver. This confirms the concerns in [Bor+12] regarding this approach. For our **CSplitModule** the constant denominator approach was therefore finally chosen. From now on, let \( C \) denote an already discretized integer arithmetic problem.

### 3.2.2. Extraction of nonlinear monomial equations

For each constraint \( f(x) \sim 0 \) in \( C \) we perform the following preprocessing steps: Replace every nonlinear monomial \( x^p \) in \( f(x) \) with \( d_p := \|p\|_1 > 1 \) by a fresh integer-valued variable \( y_p \) to get the linear part of the original constraint by

\[
L(f(x) \sim 0) := \sum_{p : \text{fr}(f), d_p \leq 1} f_p x^p + \sum_{p : \text{fr}(f), d_p > 1} f_p y_p \sim 0.
\]

We want the remaining monomial equations of the form \( y_p = x^p \) to have a total degree of two. For this purpose, choose an index \( b_p \in \{1, \ldots, d\} \) with \( p_{b_p} > 0 \) and perform the split into a binary equation \( y_p = x_{b_p} \cdot z_p \) and \( z_p = x^p - e_{b_p} \) for a fresh intermediate variable \( z_p \). Repeat this binarization process with \( z_p = x^p - e_{b_p} \) until all resulting monomial equations have a total degree of two. The depicted iteration constructs a so-called reduction sequence \( b_p = (b_{p,1}, \ldots, b_{p,d_p}) \) of indices \( b_{p,j} \in \{1, \ldots, d\} \) that decomposes the exponent into \( p = \sum_{j=1}^{d_p} e_{b_{p,j}} \). The nonlinear part of the constraint \( f(x) \sim 0 \) is then given by

\[
N(f(x) \sim 0) := \bigwedge_{p : \text{fr}(f), d_p > 1} \left[ \bigwedge_{j=1}^{d_p-2} y_{p,j} = x_{b_{p,j}} \cdot y_{p,j+1} \right] \land y_{p,d_p-1} = x_{b_{p,d_p-1}} \cdot x_{b_{p,d_p}}
\]

for a sequences of intermediate variables \( y_p = (y_{p,1}, \ldots, y_{p,d_p-1}) \). The complexity of the final linearization is highly dependent on the right choice of these reduction sequences and we therefore devote Subsection 3.4.2 to this problem. The purification of the constraints
conjunction $C$ is now given by the decomposition

$$L := \bigwedge_{i=1}^{m} L(f_i(x) \sim_i 0) \quad \text{and} \quad N := \bigwedge_{i=1}^{m} N(f_i(x) \sim_i 0)$$

Note that $C' := L \land N$ is still a conjunction of constraints and equisatisfiable to $C$. From now on, let $C = C'$ denote an already purified input formula in which the only nonlinearities arise as binary monomial equations of the form $x = v \cdot w$ in $N$.

### 3.3. Case-splitting for variables with bounded domains

Suppose that for every variable $v$ in $C$ we have a maximal domain $D_v := [L_v, U_v] \subseteq \mathbb{Z}$ that restricts its solution search space. In our CSplitModule, these domains are extracted directly from the input $C$ exploiting linear bounding constraints of the form $v \sim 0$ in $L$ with $\sim \in \{<, \leq, =, \geq, >\}$. For future development an enhanced bounds extraction routine should be implemented that is able to take also nonlinear constraints like $v^2 \leq 10$ into account. We fix a constant number $T \in \mathbb{N}$, $T \geq 2$, and subdivide all variable domains into the following classes: We denote $D_v$ as small if $|D_v| \leq T$, large if $T < |D_v| < +\infty$, and unbounded otherwise. In this section, we present linearization techniques for the binary monomial equations $x = v \cdot w$ in $N$ with bounded domains $D_v$.

#### 3.3.1. Handling small domains

Suppose for the moment that the variable $v$ used for case-splits has a small domain $D_v$. The simple linearization rule seen in the introductory Section 3.1 can be restated as

$$\Phi_{\text{Small Monomial}}(x, v, w) := U_v \land \alpha = L_v (v = \alpha \rightarrow x = \alpha \cdot w)$$

For a single monomial equation $x = v \cdot w$ it produces $|D_v| \in \mathcal{O}(T)$ binary linear clauses and is therefore inappropriate for the linearization of large domains for $T \gg 1$. If the variable $v$ is restricted to the domain $D_v$, then the monomial equation is equisatisfiable to its linearization. Remember, that this premise is fulfilled within the formula $C$, since the variable domain $D_v$ was extracted from the bounding constraints contained in it.

#### 3.3.2. Handling large domains

The problem with the presented linearization rule for small domains lies in the unary encoding of the variable domain $D_v = [L_v, U_v]$, where every binary clause represents a
single value that the case variable \( v \) can take. To overcome this issue, we fix another integer \( B \in \mathbb{N}, 2 \leq B \leq T \), and subsequently encode the variable domain \( D_v \) in a positional numeral system to the base \( B \). To this end, we take fresh integer-valued variables \( q \) and \( r \) and perform a symbolic division of \( v \) modulo \( B \) by introducing the linear formula

\[
\Phi_{\text{Digit}}(v, q, r) := v = B \cdot q + r \land \left\lfloor \frac{v}{B} \right\rfloor \leq q \leq \left\lceil \frac{v}{B} \right\rceil \land 0 \leq r \leq B - 1.
\]

The added variable domains \( D_q = \left\lfloor \frac{v}{B} \right\rfloor, \left\lceil \frac{v}{B} \right\rceil \) and \( D_r = [0, B - 1] \) constitute a smallest possible over-approximation of \( D_v \) in the usual sense of interval analysis: Every value \( \hat{v} \in D_v \) can be written as a linear combination \( \hat{v} = B \cdot \hat{q} + \hat{r} \) for suitable choices of \( \hat{q} \in D_q \) and \( \hat{r} \in D_r \), and the two domains \( D_q \) and \( D_r \) are minimal with respect to this condition. Hence, substituting the expression \( B \cdot q + r \) for \( v \) in the original monomial equation \( x = v \cdot w \) does not exclude any solutions and we obtain

\[
x = v \cdot w = (B \cdot q + r) \cdot w = B \cdot q \cdot w + r \cdot w.
\]

In order to get rid of the remaining nonlinear monomials on the right hand side, we

- replace \( q \cdot w \) by a fresh integer-valued variable \( y \) and get a new monomial equation \( y = q \cdot w \) that still needs to be linearized. But the qualitative difference between the later and the original constraint \( x = v \cdot w \) is the domain size reduction \( |D_q| \leq \left\lfloor \frac{|D_v|}{B} \right\rfloor \).

- perform an unary case-split on the monomial \( r \cdot w \) using \( r \) as the case variable like in Subsection 3.3.1. Note that the variable domain \( D_r \) is small since \( |D_r| = B - 1 \leq T \).

In summary, we replace the monomial equation \( x = v \cdot w \) with the linearization

\[
\Phi_{\text{Expansion}}(x, y, v, q, r, w) := \bigwedge_{\alpha=0}^{B-1} (r = \alpha \rightarrow x = B \cdot y + \alpha \cdot w) \land \Phi_{\text{Digit}}(v, q, r)
\]

and repeat this linearization process on the remaining constraint \( y = q \cdot w \). In each iteration, the domain size of \( D_q \) is reduced by a factor of \( B \). After at most \( k := \left\lfloor \log_B |D_v| \right\rfloor \) iterations, the variable domain \( D_q \) is small and we encode the constraint \( y = q \cdot w \) as seen in Subsection 3.3.1. To formalize this described iteration process, choose sequences of integer-valued variables \( x = (x_0, \ldots, x_k), q = (q_0, \ldots, q_k) \) and \( r = (r_1, \ldots, r_k) \). With the identification \( x_0 := x \) and \( q_0 := v \), the full linearization of \( x = v \cdot w \) is given by

\[
\Phi_{\text{Large Monomial}}(x, q, r, w) := \bigwedge_{i=1}^{k} \Phi_{\text{Expansion}}(x_{i-1}, x_i, q_{i-1}, q_i, r_i, w) \land \Phi_{\text{Small Monomial}}(x_k, q_k, w)
\]

For a single monomial equation \( x = v \cdot w \) with bounded domain \( D_v \) this linearization
Chapter 3. The Case-Splitting Method

rule produces $O(B \log_B |D_v| + T)$ at most binary linear clauses. Suppose that the pure nonlinear formula $C$ is transformed into $C'$ by one application of this linearization rule. Since the variable domain $D_v$ was derived from $C$, both formulas are equisatisfiable.

3.4. Unsatisfiability and learning for unbounded domains

As one source of incompleteness of the case-splitting method we have already identified the discretization of real-valued variables. If a case variable used for linearization lacks a finite upper or lower bound, then we have to introduce artificial bounds and again we lose completeness at first glance. In this section, we will therefore address the problem of the right choice of case variables and present a method based on the analysis of unsatisfiable cores to guide this bounding process in a clever way. This will allow us to prove unsatisfiability in many cases and attenuate the incompleteness issue of the second kind.

3.4.1. Unsatisfiability and learning

Let $C = L \land N$ be a pure conjunction of constraints. Such a constraints conjunction is always a special case of a CNF formula with a single literal per clause. The CNF property is invariant under any application of the presented linearization rules in Section 3.3. If the domains of all case variables in $C$ are bounded, such that its nonlinear part $N$ can be completely linearized to produce the formula $L_N$, we will therefore get an equisatisfiable CNF formula $D := L \land L_N$ in linear integer arithmetic. Recall the set theoretic notations for arbitrary CNF formulas from Subsection 1.1. The next Theorem relates the unsatisfiable cores of the linearization $D$ to those of the original formula $C$.

**Theorem 3.1** Let the input $C$ and hence its linearization $D$ be unsatisfiable. If $U_D$ is an unsatisfiable core for $D$, then $U_C := (U_D \cap L) \land N$ is an unsatisfiable core for $C$.

**Proof.** By the definition of an unsatisfiable core, we have $U_D \subseteq D$, which gives

$$U_D = U_D \cap D = U_D \cap (L \land L_N) = (U_D \cap L) \land (U_D \cap L_N) \subseteq (U_D \cap L) \land L_N.$$ 

The rightmost CNF formula is equisatisfiable to $(U_D \cap L) \land N = U_C$. Altogether, this proves that $U_C$ must contain a subformula that is equisatisfiable to the unsatisfiable core $U_D$. Since further $U_C \subseteq C$ holds, it follows that $U_C$ is an unsatisfiable core of $C$. □

For many input formulas $C$, the so far claimed boundedness for all variable domains that are used for case distinctions during the linearization process is not fulfilled. In this case, we need to introduce a conjunction of additional bounding constraints $B$ and consider...
the input CNF formula $C' := B \land C$ instead. This makes our method incomplete, since only \textsc{sat} answers for $C'$ imply the satisfiability of the original input $C$. A first strategy to choose the newly added bounds in $B$ as large as possible is foredoomed, as it easily produces a too hard problem for the internal linear arithmetic solver even if the logarithmic encoding of variable domains from Subsection 3.3.2 is used. An alternative idea is to start with bounds that make the domains small and enlarge them incrementally if necessary. Instead of enlarging all added bounds, we can further analyze the unsatisfiable core of the linearization to identify the bounds that need to be adapted. The core of this approach is the following refinement of Theorem 3.1 in the presence of bounding constraints.

**Corollary 3.2** Let $B$ be a conjunction of bounding constraints such that $C' := B \land C$ can be linearized into the CNF formula $D'$. If $U_{D'}$ is an unsatisfiable core for $D'$ with $U_{D'} \cap B = \emptyset$, then $U_C := (U_{D'} \cap L) \land N$ is an unsatisfiable core for the original input $C$.

**Proof.** The purification of $C$ is given by the decomposition $C = L \land N$ into its linear and nonlinear parts $L$ and $N$, respectively. Since the bounding constraints in $B$ are linear by definition, the corresponding purification of $C'$ is given by $C' = L' \land N'$ with

$$L' := B \land L \quad \text{and} \quad N' := N.$$

If we apply Theorem 3.1 on $C'$ instead of $C$ and insert the given premise $U_{D'} \cap B = \emptyset$, we obtain the unsatisfiable core for $C'$ given by

$$U_{C'} := (U_{D'} \cap L') \land N' = (U_{D'} \cap (B \land L)) \land N = (U_{D'} \cap L) \land N.$$

From the rightmost representation of $U_{C'}$ we can easily see the relation $U_{C'} \subseteq L \land N = C$. Hence, $U_C := U_{C'}$ is also an unsatisfiable core of $C$. \hfill $\square$

The benefit of Corollary 3.2 for our \textsc{CSplitModule} is twofold:

(i) If the unsatisfiable core $U_{D'}$ of the linearization $D'$ does not contain any of the auxiliary bounds in $B$, we can deduce the unsatisfiability of the original formula $C = L \land N$ and generate a corresponding unsatisfiable core $U_C := (U_{D'} \cap L) \land N$.

(ii) If, on the other hand, the unsatisfiable core $U_{D'}$ has a non-empty intersection with the auxiliary bounds in $B$, then the constraints in $U_{D'} \cap B$ are the candidate bounds that need to be enlarged before the next invocation of the linear arithmetic solver.

Unfortunately, the incremental enlargement of bounds in (ii) does not terminate for many input formulas after a finite number of iterations by reaching case (i). Our experiments showed that the initial choice of reduction sequences has the greatest impact on the
termination of the algorithm in case of unsatisfiable input formulas. We will therefore look into this problem in more detail in the next Subsection 3.4.2. Here, we give a brief summary of additional implementation details of our CSplitModule that lead to a slight performance increase for both, satisfiability and unsatisfiability answers:

- As a first obvious strategy to ensure termination, we limit the maximal number of bounds refinement iterations, before the consistency check gets finally aborted with an Unknown result. In a single iteration, a subset of candidate bounds in $U_D \cap B$ gets bloated and the linear arithmetic solver is called on the adapted linearization. This step is very time critical and needs to be implemented efficiently. Let $v$ be a variable whose domain is changed from $D_v^{\text{old}}$ to $D_v^{\text{new}}$. An inspection of the $\Phi_{\text{Monomial}}$ formulas from Section 3.3 in which $v$ is involved as the case variable shows that for
  - small domains we only need to add the case distinction clauses for the values in $D_v^{\text{new}} \setminus D_v^{\text{old}}$ and modify the bounding constraints accordingly.
  - for large domains the majority of $\Phi_{\text{Expansion}}$ subformulas stay completely untouched and only the bounds in the $\Phi_{\text{Digit}}$ subformulas need to be adapted.

We implemented a recursive algorithm that simultaneously calculates the expansions in a positional numeral system to the base $B$ for $D_v^{\text{old}}$ and $D_v^{\text{new}}$ and modifies precisely the changed clauses to transform the resulting linearization into a consistent state.

- For some linearizations, the internal linear arithmetic solver terminates quickly even for variable domain sizes in the order of millions, for others, variable domain sizes of five or less are already time critical. We therefore start for all variables $v$ with an initial interval $D_v = [0, 1]$ and bloat the candidate domains in two phases:
  
  (i) In the first phase, the domains are enlarged linearly in both directions with a step size of one until a certain threshold is reached.

  (ii) If all candidates have exceeded the threshold size, we start an exponential bloating of the domains that are used for case-splits and activate the maximal domain for variables that are not involved in the case analysis.

Furthermore, we limit the maximal number of candidates that are bloated in a single iteration, since we observed input formulas with 400 variables and more. From all potential bloating candidates in $U_D \cap B$, we prefer those with the smallest domains.

All of the mentioned parameters that control the behaviour of our CSplitModule are not hard-coded into it but can be set centrally in its corresponding settings file. We will report the best configuration that we found with the help of a sparse grid search in Section 3.5.
3.4. Unsatisfiability and learning for unbounded domains

3.4.2. Optimal choice of reduction sequences

So far we have only considered the final linearization step for binary monomial equations of the form \( x = v \cdot w \) for bounded and unbounded domains \( D_v \). These binary monomial equations were produced in Subsection 3.2.2 from monomial equations \( y_{\mathbf{p}} = x_{\mathbf{p}} \) of arbitrary total degree \( d_{\mathbf{p}} = \| \mathbf{p} \|_1 > 2 \) by the choice of reduction sequences \( b_{\mathbf{p}} = (b_{\mathbf{p},1}, \ldots, b_{\mathbf{p},d_{\mathbf{p}}}) \). These sequences define the order in which the variables are removed from the nonlinear monomial \( x_{\mathbf{p}} \). When multiple monomial equations \( y_{\mathbf{p}_1} = x_{\mathbf{p}_1}, \ldots, y_{\mathbf{p}_k} = x_{\mathbf{p}_k} \) are involved, the choice of reduction sequences cannot be considered in isolation anymore. The interdependency of the reduction sequences \( b_{\mathbf{p}_1}, \ldots, b_{\mathbf{p}_k} \) has a vast impact on the number of intermediate variables and thus the number of clauses in the final linearization.

Example 3.3 Consider the system of monomial equations \( y_{\mathbf{p}_i} = x_{\mathbf{p}_i} \) for \( i = 1, \ldots, 3 \) with

\[
\mathbf{p}_1 = (1, 2, 2, 0), \quad \mathbf{p}_2 = (2, 0, 2, 1), \quad \mathbf{p}_3 = (1, 0, 2, 2).
\]

Figure 3.1 shows the reduction trees for two different sets of reduction sequences that lead to a different number of intermediate nonlinear monomial equations:

(a) \( b_{\mathbf{p}_1} = (1, 2, 2, 3, 3), \ b_{\mathbf{p}_2} = (1, 1, 3, 4, 4), \ b_{\mathbf{p}_3} = (1, 3, 3, 4, 4) \) with 12 equations.

(b) \( b_{\mathbf{p}_1} = (1, 2, 2, 3, 3), \ b_{\mathbf{p}_2} = (1, 1, 4, 3, 3), \ b_{\mathbf{p}_3} = (4, 1, 4, 3, 3) \) with 8 equations.

The most desirable set of reduction sequences is the one that minimizes the number of intermediate nonlinear monomials in the reduction tree. It is easy to see that this minimization problem is NP-complete and thus too expensive as a subproblem of our linearization algorithm. In [Bor+12] this problem discussion closes with a reference to a "greedy approximation algorithm" without any information on the implementation details. After numerous experiments, we chose the following method as a tradeoff between the desired minimal cardinality of intermediate monomials and the avoidance of case variables with unbounded domains, since the later are the reason for a lack of termination. Let the set of exponents \( E_0 := \{ \mathbf{p}_1, \ldots, \mathbf{p}_k \} \) be sorted ascendingly in degree lexicographic order. Perform the following steps for \( i = 1, \ldots, k \):

(i) In the monomial \( x_{\mathbf{p}_i} \) we have exactly one free choice of a variable that will not be used for case-splits during the linearization. We therefore select the last index \( b_{d_{\mathbf{p}_i}} \) of the reduction sequence corresponding to the variable \( x_{b_{d_{\mathbf{p}_i}}} \) with the largest domain.

(ii) Among all exponents \( \mathbf{p} \in E_{i-1} \), choose the one with maximal degree \( d_{\mathbf{p}} \) such that it

- is componentwise smaller than \( p_i \) and hence \( x_{\mathbf{p}_i} \) is reducible to \( x_{\mathbf{p}} \).
- contains the variable \( x_{b_{d_{\mathbf{p}_i}}} \). By induction, it follows that \( b_{d_{\mathbf{p}}} = b_{d_{\mathbf{p}_i}} \) and during the reduction process of \( x_{\mathbf{p}} \) the variable \( x_{b_{d_{\mathbf{p}_i}}} \) was successfully avoided.
(iii) Construct the reduction sequence \( b_{p_i} \) that reduces \( p_i \) to \( p \) and then follows the reduction sequence \( b_p \). Add all used intermediate exponents to \( E_{i-1} \) to get \( E_i \).

3.5. Benchmarking results and conclusion

We tested our CSplitModule on the QF_NIA division of the SMT-LIB benchmarking library \([BFT16]\) and report our results in Table 3.1. The experiments were performed on a 2.7 GHz Intel Core i7-4800MQ CPU with a timeout of 60 seconds and 4 GB memory per benchmark. In \([Bor+12]\), only the families AProVE, calypto and leipzig are considered with a timeout limit of 1200 seconds to conclude the supremacy of the case-splitting method. But our results clearly indicate that these three families were cherry-picked as they are the only with a reasonable performance. The strategies that we tested are:

**CSplit only:** SATModule → CSplitModule

**Backends only:** SATModule → LRAModule → VSModule → CADModule

**CSplit + Backends:** SATModule → CSplitModule → LRAModule → VSModule → CADModule

To find the best parameter combination, we performed a grid search on a validation subset of 500 benchmarks and finally picked the following setting:

- We choose \( T = 32 \) as the threshold between small and large domain sizes and also \( B = 32 \) as the base for the logarithmic encoding of large domains.
- The maximum number of bounds refinements is limited to 50 iterations. In every iteration, we choose at most three candidates whose bounds get enlarged. Candidates with a domain size of 300 or more get discarded and not considered for bloating.
3.5. Benchmarking results and conclusion

- All variable domains are initially restricted to an interval of size one near to the zero point as many input formulas seem to have solutions near to the origin. They are bloated linearly with a step size of one until the threshold of three is reached. Afterwards, the exponential bloating routine starts.

It is worth mentioning that we were able to find parameter sets that gave better results when we restricted ourselves to one of the three above mentioned benchmark families. In calypoto, the linear arithmetic solver terminates quickly independent from the domain sizes. Hence, it is advisable to select a much higher number of iterations and entirely remove the limit size of 300 for candidate domains. This behaviour is perpendicular to the leipzig family, where a fast rejection with an Unknown answer gives better results in later consistency checks of the outer DPLL loop.
<table>
<thead>
<tr>
<th>QF_NIA Benchmarks</th>
<th>Benchmark Family</th>
<th>Answer</th>
<th>Num</th>
<th>Avg</th>
<th>Num</th>
<th>Avg</th>
<th>Num</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>AProVE</td>
<td>Sat (1663)</td>
<td>894</td>
<td>1451</td>
<td></td>
<td>890</td>
<td>1392</td>
<td></td>
<td>627</td>
</tr>
<tr>
<td></td>
<td>Unsat (320)</td>
<td>2</td>
<td>14</td>
<td></td>
<td>4</td>
<td>5960</td>
<td></td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>Unknown (426)</td>
<td>5</td>
<td>12898</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Resout</td>
<td>1508</td>
<td>59995</td>
<td></td>
<td>1515</td>
<td>59993</td>
<td></td>
<td>1503</td>
</tr>
<tr>
<td>calypto</td>
<td>Sat (80)</td>
<td>44</td>
<td>5036</td>
<td></td>
<td>45</td>
<td>4797</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Unsat (97)</td>
<td>6</td>
<td>28</td>
<td></td>
<td>9</td>
<td>1147</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>Unknown (0)</td>
<td>21</td>
<td>21859</td>
<td></td>
<td>17</td>
<td>11798</td>
<td></td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>Resout</td>
<td>106</td>
<td>59986</td>
<td></td>
<td>106</td>
<td>60004</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>CInteger</td>
<td>Sat (858)</td>
<td>10</td>
<td>27391</td>
<td></td>
<td>10</td>
<td>24151</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Unsat (150)</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Unknown (810)</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Resout</td>
<td>1808</td>
<td>59991</td>
<td></td>
<td>1808</td>
<td>59993</td>
<td></td>
<td>1813</td>
</tr>
<tr>
<td>ITS</td>
<td>Sat (9473)</td>
<td>32</td>
<td>17754</td>
<td></td>
<td>33</td>
<td>19957</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Unsat (2360)</td>
<td>0</td>
<td>0</td>
<td></td>
<td>5</td>
<td>12160</td>
<td></td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>Unknown (5213)</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Resout</td>
<td>17014</td>
<td>59997</td>
<td></td>
<td>17008</td>
<td>59989</td>
<td></td>
<td>16992</td>
</tr>
<tr>
<td>LassoRanker</td>
<td>Sat (4)</td>
<td>4</td>
<td>4568</td>
<td></td>
<td>3</td>
<td>50</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Unsat (100)</td>
<td>0</td>
<td>0</td>
<td></td>
<td>15</td>
<td>244</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Unknown (2)</td>
<td>59</td>
<td>3646</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Resout</td>
<td>43</td>
<td>59992</td>
<td></td>
<td>48</td>
<td>60002</td>
<td></td>
<td>88</td>
</tr>
</tbody>
</table>
### Table 3.1: Benchmarking results of the CSplitModule on the QF_NIA division of the SMT-LIB.

<table>
<thead>
<tr>
<th></th>
<th>Sat (162)</th>
<th>73</th>
<th>5902</th>
<th>74</th>
<th>5889</th>
<th>15</th>
<th>7693</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unsat (5)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Unknown (0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Resout</td>
<td>94</td>
<td>59990</td>
<td>93</td>
<td>59986</td>
<td>152</td>
<td>59991</td>
</tr>
<tr>
<td>mcm (186)</td>
<td>Sat (25)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>43761</td>
</tr>
<tr>
<td></td>
<td>Unsat (0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Unknown (161)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Resout</td>
<td>186</td>
<td>59983</td>
<td>186</td>
<td>59992</td>
<td>185</td>
<td>59989</td>
</tr>
<tr>
<td>SAT14 (1926)</td>
<td>Sat (1853)</td>
<td>11</td>
<td>6068</td>
<td>10</td>
<td>3180</td>
<td>11</td>
<td>3929</td>
</tr>
<tr>
<td></td>
<td>Unsat (63)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>16337</td>
</tr>
<tr>
<td></td>
<td>Unknown (10)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Resout</td>
<td>1915</td>
<td>59995</td>
<td>1916</td>
<td>59997</td>
<td>1904</td>
<td>60001</td>
</tr>
<tr>
<td>Ultimate Automizer/</td>
<td>Sat (6)</td>
<td>5</td>
<td>589</td>
<td>6</td>
<td>944</td>
<td>6</td>
<td>209</td>
</tr>
<tr>
<td>LassoRanker (39)</td>
<td>Unsat (33)</td>
<td>7</td>
<td>8214</td>
<td>25</td>
<td>10891</td>
<td>27</td>
<td>7762</td>
</tr>
<tr>
<td></td>
<td>Unknown (0)</td>
<td>1</td>
<td>2828</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Resout</td>
<td>26</td>
<td>59991</td>
<td>8</td>
<td>59937</td>
<td>6</td>
<td>59918</td>
</tr>
<tr>
<td>Total (23874)</td>
<td>Sat (14124)</td>
<td>1073</td>
<td>2684</td>
<td>1071</td>
<td>2641</td>
<td>778</td>
<td>1071</td>
</tr>
<tr>
<td></td>
<td>Unsat (3128)</td>
<td>15</td>
<td>3846</td>
<td>58</td>
<td>6395</td>
<td>172</td>
<td>3232</td>
</tr>
<tr>
<td></td>
<td>Unknown (6622)</td>
<td>86</td>
<td>8622</td>
<td>17</td>
<td>11798</td>
<td>124</td>
<td>167</td>
</tr>
<tr>
<td></td>
<td>Resout</td>
<td>22700</td>
<td>59996</td>
<td>22728</td>
<td>59991</td>
<td>22800</td>
<td>59991</td>
</tr>
</tbody>
</table>
Appendix A.

STropModule source code

Listing A.1: STropSettings.h

```cpp
/**
 * @file STropSettings.h
 * @author Ömer Sali <oemer.sali@rwth-aachen.de>
 * @version 2018-04-04
 * Created on 2017-09-13.
 */
#pragma once

#include “../../solver/ModuleSettings.h”
#include “../../solver/Manager.h”
#include “../SATModule/SATModule.h”
#include “../LRAModule/LRA.h”

namespace smtrat
{
enum class SeparatorType { STRICT = 0, WEAK = 1};

struct STropSettings1
{
    /// Name of the Module
    static constexpr auto moduleName = "STropModule<STropSettings1>”;
    /// Type of linear separating hyperplane to search for
    static constexpr SeparatorType separatorType = SeparatorType::STRICT;
    /// Linear real arithmetic solver to call for the linearized formula
    struct LRASolver : public Manager
    {
        LRASolver() : Manager()
        {
            setStrategy(
                addBackend<LRA< LRASolver >>()
            );
        }
    };
};
}
```

Listing A.2: STropModule.h

```cpp
/**
 * @file STropModule.h
 * @author Ömer Sali <oemer.sali@rwth-aachen.de>
 * @version 2018-04-04
 * Created on 2017-09-13.
 */
```
*/
#pragma once

#include "../../solver/Module.h"
#include "STropStatistics.h"
#include "STropSettings.h"

namespace smtrat {

template<typename Settings>
class STropModule : public Module {

private:
#ifndef SMTRAT_DEVOPTION_Statistics
STropStatistics mStatistics;
#endif

/**
* Represents the normal vector component and the sign variable
* assigned to a variable in an original constraint.
*/
struct Moment {
    // Normal vector component of the separating hyperplane
    const carl::Variable mNormalVector;
    // Boolean variable representing the sign variant
    const carl::Variable mSignVariant;
    // Flag that indicates whether this moment is used for linearization
    bool mUsed;

    Moment() : mNormalVector(carl::freshRealVariable()),
               mSignVariant(carl::freshBooleanVariable()),
               mUsed(false) {
    }

    // Maps a variable to the components of the moment function
    std::unordered_map<carl::Variable, Moment> mMoments;

    /**
* Represents a term of an original constraint and assigns him a rating variable if a weak separator is searched.
*/
    struct Vertex { 
        // Coefficient of the assigned term
        const Rational mCoefficient;
        // Monomial of the assigned term
        const carl::Monomial::Arg mMonomial;
        // Rating variable of the term for a weak separator
        const carl::Variable mRating;

        Vertex(const TermT& term) :
            mCoefficient(term.coeff()),
            mMonomial(term.monomial()),
            mRating(
                Settings::separatorType == SeparatorType::WEAK ?
                carl::freshRealVariable() : carl::Variable::NO_VARIABLE)
        {}
    };

    // Subdivides the relations into classes with the same linearization result
    enum class Direction {NONE = 0, BOTH = 0, NEGATIVE = 1, POSITIVE = 2};
};
/**
 * Represents the class of all original constraints with the same
 * left hand side after a normalization. Here, the set of all
 * received
 * relations of constraints with the same left hand side is stored.
 * At any
 * one time only one relation can be active and used for
 * linearization.
 */

struct Separator
{
    /// Bias variable of the separating hyperplane
    const carl::Variable mBias;
    /// Vertices for all terms of the normalized left hand side
    const std::vector<Vertex> mVertices;
    /// Relations of constraints with the same left hand side
    std::set<carl::Relation> mRelations;
    /// Direction currently used for linearization
    Direction mActiveDirection;

    Separator(const Poly & normalization)
    : mBias(carl::freshRealVariable())
    , mVertices(normalization.begin(), normalization.end())
    , mRelations()
    , mActiveDirection(Direction::NONE)
    {};

    // Maps a normalized left hand side of a constraint to its separator
    std::unordered_map<Poly, Separator> mSeparators;
    // Stores the Separators that were updated since the last check call
    std::unordered_set<Separator *> mChangedSeparators;
    // Counts the number of relation pairs that prohibit an application
    // of this method
    size_t mRelationalConflicts;
    // Stores the sets of separators that were found to be undecidable
    // by the LRA solver
    typedef std::vector<std::pair<const Separator *, const Direction>> Conflict;
    std::vector<Conflict> mLinearizationConflicts;
    // Stores whether the last consistency check was done by the
    // backends
    bool mCheckedWithBackends;
    // Handle to the linear real arithmetic solver
    typename Settings::LRASolver mLRASolver;

public:
    typedef Settings SettingsType;

    std::string moduleName() const
    {
        return SettingsType::moduleName;
    }

    STrOpModule(const ModuleInput* _formula, RuntimeSettings* _settings,
                Conditionals& _conditionals, Manager* _manager = nullptr);

    /**
     * The module has to take the given sub-formula of the received
     * formula into account.
     * @param _subformula The sub-formula to take additionally into
     * account.
     * @return False, if it can be easily decided that this sub-formula
     * causes a conflict with
     * the already considered sub-formulas;
     * True, otherwise.
     */
    bool addCore(ModuleInput::const_iterator _subformula);
/**
 * Removes the subformula of the received formula at the given
 * position to the considered ones of this module.
 * Note that this includes every stored calculation which depended on
 * this subformula, but should keep the other
 * stored calculation, if possible, untouched.
 * @param _subformula The position of the subformula to remove.
 */
void removeCore(ModuleInput::const_iterator _subformula);

/**
 * Updates the current assignment into the model.
 * Note, that this is a unique but possibly symbolic assignment maybe
 * containing newly introduced variables.
 */
void updateModel() const;

/**
 * Checks the received formula for consistency.
 * @return SAT, if the received formula is satisfiable;
 * UNSAT, if the received formula is not satisfiable;
 * UNKNOWN, otherwise.
 */
Answer checkCore();

private:
/**
 * Creates the linearization for the given separator with the active
 * relation.
 * @param separator The separator object that stores the construction
 * information.
 * @return Formula that is satisfiable iff such a separating
 * hyperplane exists.
 */
inline FormulaT createLinearization(const Separator& separator);

/**
 * Creates the formula for the hyperplane that linearly separates at
 * least one
 * variant positive frame vertex from all variant negative frame
 * vertices. If a
 * weak separator is searched, the corresponding rating is included.
 * @param separator The separator object that stores the construction
 * information.
 * @param negated True, if the formula for the negated polynomial
 * shall be constructed.
 * False, if the formula for the original polynomial shall be
 * constructed.
 * @return Formula that is satisfiable iff such a separating
 * hyperplane exists.
 */
FormulaT createSeparator(const Separator& separator, bool negated);

/**
 * Asserts/Removes the given formula to/from the LRA solver.
 * @param assert True, if formula shall be asserted;
 * False, if formula shall be removed.
 */
inline void propagateFormula(const FormulaT& formula, bool assert);
}
namespace smtrat
{
    template<class Settings>
    STropModule<Settings>::STropModule(const ModuleInput* _formula,
    RuntimeSettings*, Conditionals& _conditionals, Manager* _manager)
    : Module(_formula, _conditionals, _manager)
    , mMoments()
    , mSeparators()
    , mRelationalConflicts(0)
    , mLinearizationConflicts()
    , mCheckedWithBackends(false)
    #ifdef SMTRAT_DEVOPTION_Statistics
    , mStatistics(Settings::moduleName)
    #endif
    {}
}

bool STropModule<Settings>::addCore(ModuleInput::const_iterator _subformula)
{
    addReceivedSubformulaToPassedFormula(_subformula);
    const FormulaT & formula{_subformula->formula()};
    if (formula.getType() == carl::FormulaType::FALSE)
        mInfeasibleSubsets.push_back({formula});
    else if (formula.getType() == carl::FormulaType::CONSTRAINT)
    {
        // Normalize the left hand side of the constraint and turn the
        // relation accordingly
        const ConstraintT constraint{formula.constraint()};
        const Poly normalization{constraint.lhs().normalize()};
        carl::Relation relation{constraint.relation()};
        if (carl::isNegative(constraint.lhs().lcoeff()))
            relation = carl::turnAroundRelation(relation);

        // Store the normalized constraint and mark the separator object as
        // changed
        Separator& separator{mSeparators.emplace(normalization, normalization
            ).first->second};
        separator.mRelations.insert(relation);
        mChangedSeparators.insert(&separator);

        // Check if the asserted constraint prohibits the application of
        // this method
        if (relation == carl::Relation::EQ
            || (relation == carl::Relation::LEQ
                && separator.mRelations.count(carl::Relation::GEQ))
            || (relation == carl::Relation::GEQ
                && separator.mRelations.count(carl::Relation::LEQ)))
            ++mRelationalConflicts;

        // Check if the asserted relation trivially conflicts with other
        // asserted relations
        switch (relation)
        {
            case carl::Relation::EQ:
                if (separator.mRelations.count(carl::Relation::NEQ))
                    mInfeasibleSubsets.push_back({
                        FormulaT(normalization, carl::Relation::EQ),
                        formula});
                break;
            case carl::Relation::LEQ:
                if (separator.mRelations.count(carl::Relation::GEQ))
                    mInfeasibleSubsets.push_back({
                        FormulaT(normalization, carl::Relation::LEQ),
                        formula});
                break;
            case carl::Relation::GEQ:
                if (separator.mRelations.count(carl::Relation::LEQ))
                    mInfeasibleSubsets.push_back({
                        FormulaT(normalization, carl::Relation::GEQ),
                        formula});
                break;
            default:
                break;
        }
    }
}
Appendix A. STropModule source code

```cpp
2
FormulaT(normalization, carl::Relation::NEQ)
3
});
4
if (separator.mRelations.count(carl::Relation::LESS))
5  mInfeasibleSubsets.push_back({
6      FormulaT(normalization, carl::Relation::EQ),
7      FormulaT(normalization, carl::Relation::LESS)
8  });
9
10 if (separator.mRelations.count(carl::Relation::GREATER))
11  mInfeasibleSubsets.push_back({
12      FormulaT(normalization, carl::Relation::EQ),
13      FormulaT(normalization, carl::Relation::GREATER)
14  });
15
16 break;
17 case carl::Relation::NEQ:
18 if (separator.mRelations.count(carl::Relation::EQ))
19  mInfeasibleSubsets.push_back({
20      FormulaT(normalization, carl::Relation::NEQ),
21      FormulaT(normalization, carl::Relation::EQ)
22  });
23
24 break;
25 case carl::Relation::LESS:
26 if (separator.mRelations.count(carl::Relation::EQ))
27  mInfeasibleSubsets.push_back({
28      FormulaT(normalization, carl::Relation::LESS),
29      FormulaT(normalization, carl::Relation::EQ)
30  });
31
32 if (separator.mRelations.count(carl::Relation::GEQ))
33  mInfeasibleSubsets.push_back({
34      FormulaT(normalization, carl::Relation::LESS),
35      FormulaT(normalization, carl::Relation::GEQ)
36  });
37
38 case carl::Relation::LEQ:
39 if (separator.mRelations.count(carl::Relation::GREATER))
40  mInfeasibleSubsets.push_back({
41      FormulaT(normalization, relation),
42      FormulaT(normalization, carl::Relation::GREATER)
43  });
44
45 break;
46 case carl::Relation::GREATER:
47 if (separator.mRelations.count(carl::Relation::EQ))
48  mInfeasibleSubsets.push_back({
49      FormulaT(normalization, carl::Relation::GREATER),
50      FormulaT(normalization, carl::Relation::EQ)
51  });
52
53 if (separator.mRelations.count(carl::Relation::LEQ))
54  mInfeasibleSubsets.push_back({
55      FormulaT(normalization, carl::Relation::GREATER),
56      FormulaT(normalization, carl::Relation::LEQ)
57  });
58
59 case carl::Relation::GEQ:
60 if (separator.mRelations.count(carl::Relation::LESS))
61  mInfeasibleSubsets.push_back({
62      FormulaT(normalization, relation),
63      FormulaT(normalization, carl::Relation::LESS)
64  });
65
66 break;
67 default:
68  assert(false);
69 }
70 return mInfeasibleSubsets.empty();
71}
72
73 template<class Settings>
74 void STropModule<Settings>::removeCore(ModuleInput::const_iterator _subformula)
75 {
const FormulaT& formula(_subformula->formula());

if (formula.getType() == carl::FormulaType::CONSTRAINT)
{
    /// Normalize the left hand side of the constraint and turn the
    /// relation accordingly
    const ConstraintT& constraint(formula.constraint());
    const Poly normalization(constraint.lhs().normalize());
    carl::Relation relation(constraint.relation());
    if (carl::isNegative(constraint.lhs().lcoeff()))
        relation = carl::turnAroundRelation(relation);

    // Retrieve the normalized constraint and mark the separator object
    // as changed
    Separator& separator(mSeparators.at(normalization));
    separator.mRelations.erase(relation);
    mChangedSeparators.insert(&separator);

    // Check if the removed constraint prohibited the application of
    /// this method
    if (relation == carl::Relation::EQ
        || (relation == carl::Relation::LEQ
            && separator.mRelations.count(carl::Relation::GEQ))
        || (relation == carl::Relation::GEQ
            && separator.mRelations.count(carl::Relation::LEQ)))
        --mRelationalConflicts;
}
}

/// Stores all informations retrieved from the LRA solver to
/// construct the model
struct Weight
{
    const carl::Variable& mVariable;
    Rational mExponent;
    const bool mSign;
    Weight(const carl::Variable& variable, const Rational& exponent, const bool sign)
    :
        mVariable(variable),
        mExponent(exponent),
        mSign(sign)
    {}
};

std::vector<Weight> weights;

/// Retrieve the sign and exponent for every active variable
const Model& LRAModel(mLRASolver.model());
Rational gcd(0);
for (const auto& momentsEntry : mMoments)
{
    const carl::Variable& variable(momentsEntry.first);
    const Moment& moment(momentsEntry.second);
    if (moment.mUsed)
    {
        auto signIter(LRAModel.find(moment.mSignVariant));
        weights.push_back(
            Weight(variable, moment.mExponent, moment.mSign));
    }
}
Appendix A. STropModule source code

weights.emplace_back(
  variable,
  LRAModel.at(moment.mNormalVector).asRational(),
  signIter != LRAModel.end() && signIter->second.asBool()
);
carl::gcd_assign(gcd, weights.back().mExponent);
}

/// Calculate smallest possible integer valued exponents
if (gcd != ZERO_RATIONAL && gcd != ONE_RATIONAL)
for (Weight& weight : weights)
  weight.mExponent /= gcd;

/// Find model by increasingly testing the sample base
Rational base(ZERO_RATIONAL);
do {
  ++base;
clearModel();
  for (const Weight& weight : weights)
    {
      Rational value(carl::isNegative(weight.mExponent) ? carl::reciprocal(base) : base);
carl::pow_assign(value, carl::toInt<carl::uint>(carl::abs(weight.mExponent)));
      if (weight.mSign)
        value *= MINUS_ONE_RATIONAL;
      mModel.emplace(weight.mVariable, value);
    }
  while (!rReceivedFormula().satisfiedBy(mModel));
  mModelComputed = true;
}

template<class Settings>
Answer STropModule<Settings>::checkCore() {
  /// Report unsatisfiability if the already found conflicts are still unresolved
  if (!mInfeasibleSubsets.empty())
    return Answer::UNSAT;
  
  /// Predicate that decides if the given conflict is a subset of the asserted constraints
  const auto hasConflict = [&](const Conflict& conflict) {
    return std::all_of(
      conflict.begin(),
      conflict.end(),
      [&](const auto& conflictEntry) {
        return ((conflictEntry.second == Direction::NEGATIVE
          || conflictEntry.second == Direction::BOTH)
          && (conflictEntry.first->mRelations.count(carl::Relation::LESS)
            || conflictEntry.first->mRelations.count(carl::Relation::LEQ)))
        || ((conflictEntry.second == Direction::POSITIVE
          && (conflictEntry.first->mRelations.count(carl::Relation::GREATER)
            || conflictEntry.first->mRelations.count(carl::Relation::GEQ))
          && (conflictEntry.second == Direction::BOTH
            && (conflictEntry.first->mRelations.count(carl::Relation::NEQ)));
      });
  };
/// Apply the method only if the asserted formula is not trivially
undecidable
if (!(mRelationalConflicts
&& rReceivedFormula().isConstraintConjunction()
&& std::none_of(mLinearizationConflicts.begin(),
mLinearizationConflicts.end(), hasConflict))
{
  // Update the linearization of all changed separators
  for (Separator *separatorPtr : mChangedSeparators)
  {
    Separator& separator{*separatorPtr};

    // Determine the direction that shall be active
    Direction direction;
    if (separator.mRelations.empty())
      direction = Direction::NONE;
    else if ((separator.mActiveDirection == Direction::NEGATIVE
      && ((separator.mRelations.count(carl::Relation::LESS)
      || separator.mRelations.count(carl::Relation::LEQ)))
      || (separator.mActiveDirection == Direction::POSITIVE
      && ((separator.mRelations.count(carl::Relation::GREATER)
      || separator.mRelations.count(carl::Relation::GEQ)))))
      direction = separator.mActiveDirection;
    else
      switch (*separator.mRelations.rbegin())
      {
      case carl::Relation::NEQ:
        direction = Direction::BOTH;
        break;
      case carl::Relation::LESS:
      case carl::Relation::LEQ:
        direction = Direction::NEGATIVE;
        break;
      case carl::Relation::GREATER:
      case carl::Relation::GEQ:
        direction = Direction::POSITIVE;
        break;
      default:
        assert(false);
      }

    // Update the linearization if the direction has changed
    if (separator.mActiveDirection != direction)
    {
      if (separator.mActiveDirection != Direction::NONE)
        propagateFormula(createLinearization(separator), false);
      separator.mActiveDirection = direction;
      if (separator.mActiveDirection != Direction::NONE)
        propagateFormula(createLinearization(separator), true);
    }
    mChangedSeparators.clear();

  // Restrict the normal vector component of integral variables to
  positive values
  for (auto& momentsEntry : mMoments)
  {
    const carl::Variable& variable{momentsEntry.first};
    Moment& moment{momentsEntry.second};
    if (variable.type() == carl::VariableType::VT_INT
      && moment.mUsed != receivedVariable(variable))
    {
      moment.mUsed = !moment.mUsed;
      propagateFormula(FormulaT(Poly(moment.mNormalVector), carl::
      Relation::GEQ), moment.mUsed);
    }
  }
}
/// Check the constructed linearization with the LRA solver
switch (mLRASolver.check(true))
{
    case Answer::SAT:
        mCheckedWithBackends = false;
        return Answer::SAT;
    case Answer::UNSAT:
        /// Learn the conflicting set of separators to avoid its recheck in
        the future
        const std::vector<FormulaSetT> LRAConflicts = mLRASolver.
infeasibleSubsets();
        for (const FormulaSetT& LRAConflict : LRAConflicts)
        {
            carl::Variables variables;
            for (const FormulaT& formula : LRAConflict)
                formula.allVars(variables);
            Conflict conflict;
            for (const auto& separatorsEntry : mSeparators)
            {
                const Separator& separator = separatorsEntry.second;
                if (separator.mActiveDirection != Direction::NONE
                    && variables.count(separator.mBias))
                    conflict.emplace_back(&separator, separator.mActiveDirection);
            }
            mLinearizationConflicts.emplace_back(std::move(conflict));
        }
    }

    /// Check the asserted formula with the backends
    mCheckedWithBackends = true;
    Answer answer = runBackends();
    if (answer == Answer::UNSAT)
        getInfeasibleSubsets();
    return answer;
}

template<class Settings>
inline FormulaT STropModule<Settings>::createLinearization(const Separator& separator)
{
    switch (separator.mActiveDirection)
    {
    case Direction::POSITIVE:
        return createSeparator(separator, false);
    case Direction::NEGATIVE:
        return createSeparator(separator, true);
    case Direction::BOTH:
        return FormulaT(carl::FormulaType::XOR,
                         createSeparator(separator, false),
                         createSeparator(separator, true));
    default:
        assert(false);
    }
}

template<class Settings>
FormulaT STropModule<Settings>::createSeparator(const Separator& separator, bool negated)
{
    Poly totalRating;
    FormulasT disjunctions, conjunctions;
    for (const Vertex& vertex : separator.mVertices)
    {
        /// Create the hyperplane and sign change formula
        Poly hyperplane = separator.mBias;
        }
FormulaT signChangeFormula;
if (vertex.mMonomial)
{
  FormulasT signChangeSubformulas;
  for (const auto& exponent : vertex.mMonomial->exponents())
  {
    const auto& moment=mMoments[exponent.first];
    hyperplane += Rational(exponent.second)*moment.mNormalVector;
    if (exponent.second%2)
      signChangeSubformulas.emplace_back(moment.mSignVariant);
  }
  signChangeFormula = FormulaT(carl::FormulaType::XOR, move(
        signChangeSubformulas));
}

/// Create the rating case distinction formula
if (Settings::separatorType == SeparatorType::WEAK)
{
  totalRating += vertex.mRating;
  conjunctions.emplace_back(
    carl::FormulaType::IMPLIES,
    FormulaT(hyperplane, carl::Relation::LESS),
    FormulaT(Poly(vertex.mRating), carl::Relation::EQ)
  );
  const Rational coefficient{negated ? -vertex.mCoefficient : vertex.
        mCoefficient};
  conjunctions.emplace_back(
    carl::FormulaType::IMPLIES,
    FormulaT(hyperplane, carl::Relation::EQ),
    FormulaT(carl::FormulaType::AND,
      FormulaT(carl::FormulaType::IMPLIES,
        signChangeFormula,
        FormulaT(vertex.mRating+coefficient, carl::Relation::EQ)
      ),
      FormulaT(carl::FormulaType::IMPLIES,
        signChangeFormula.negated(),
        FormulaT(vertex.mRating-coefficient, carl::Relation::EQ)
      )
    );
  }

/// Create the strict/weak linear separating hyperplane
bool positive{carl::isPositive(vertex.mCoefficient) != negated};
disjunctions.emplace_back(
  carl::FormulaType::IMPLIES,
  positive ? signChangeFormula.negated() : signChangeFormula,
  FormulaT(hyperplane, Settings::separatorType == SeparatorType::
        STRICT ? carl::Relation::LEQ : carl::Relation::LESS)
).negated()
);
conjunctions.emplace_back(
  carl::FormulaType::IMPLIES,
  positive ? move(signChangeFormula) : move(signChangeFormula.negated
() ),
  FormulaT(move(hyperplane), carl::Relation::LEQ)
);
if (Settings::separatorType == SeparatorType::WEAK)
conjunctions.emplace_back(totalRating, carl::Relation::GREATER);
return FormulaT(carl::FormulaType::AND,
  FormulaT(carl::FormulaType::OR, move(disjunctions)),}
Appendix A. STropModule source code

```cpp
FormulaT(carl::FormulaType::AND, move(conjunctions));

// Template function
template <class Settings>
inline void STropModule<Settings>::propagateFormula(const FormulaT& formula, bool assert)
{
    if (assert)
        mLRASolver.add(formula);
    else if (formula.getType() == carl::FormulaType::AND)
        { auto iter = mLRASolver.formulaBegin();
          for (const auto& subformula : formula.subformulas())
              iter = mLRASolver.remove(std::find(iter, mLRASolver.formulaEnd(), subformula));
        }
    else
        mLRASolver.remove(std::find(mLRASolver.formulaBegin(), mLRASolver.formulaEnd(), formula));
}

#include "Instantiation.h"
```
Appendix B.

CSplitModule source code

Listing B.1: Bimap.h

```cpp
/**
 * @file CSplitModule.h
 * @author Ömer Sali <oemer.sali@rwth-aachen.de>
 * @version 2018-04-04
 * Created on 2017-11-01.
 */
#pragma
#include <forward_list>
#include <set>
namespace smtrat
{
  /**
   * Container that stores expensive to construct objects and allows the
   * fast lookup with respect to two independent keys within the objects.
   */
  template < class Class, typename KeyType, KeyType Class::*FirstKey,
             KeyType Class::*SecondKey>
  class Bimap
  {
    public:
      typedef std::forward_list<Class> Data;
    typedef typename Data::iterator Iterator;
    typedef typename Data::const_iterator ConstIterator;
  
    private:
      // Comparator that performs a heterogeneous lookup on the first key
    struct FirstCompare
    {
      using is_transparent = void;

      bool operator()(const Iterator& lhs, const Iterator& rhs) const
      {
        return (*lhs).*FirstKey<(*rhs).*FirstKey;
      }

      bool operator()(const Iterator& lhs, const KeyType& rhs) const
      {
        return (*lhs).*FirstKey<rhs;
      }

      bool operator()(const KeyType& lhs, const Iterator& rhs) const
      {
        return lhs<(*rhs).*FirstKey;
      }
    };
```
/// Comparator that performs a heterogeneous lookup on the second key
struct SecondCompare
{
    using is_transparent = void;

    bool operator()(const Iterator& lhs, const Iterator& rhs) const
    {
        return (*lhs).SecondKey <(*rhs).SecondKey;
    }

    bool operator()(const Iterator& lhs, const KeyType& rhs) const
    {
        return (*lhs).SecondKey < rhs;
    }

    bool operator()(const KeyType& lhs, const Iterator& rhs) const
    {
        return lhs <(*rhs).SecondKey;
    }
};

Data mData;
std::set<Iterator, FirstCompare> mFirstMap;
std::set<Iterator, SecondCompare> mSecondMap;

public:
    Iterator begin() noexcept
    {
        return mData.begin();
    }

    ConstIterator begin() const noexcept
    {
        return mData.begin();
    }

    Iterator end() noexcept
    {
        return mData.end();
    }

    ConstIterator end() const noexcept
    {
        return mData.end();
    }

    Class& firstAt(const KeyType& firstKey)
    {
        return *(mFirstMap.find(firstKey));
    }

    Class& secondAt(const KeyType& secondKey)
    {
        return *(mSecondMap.find(secondKey));
    }

    Iterator firstFind(const KeyType& firstKey)
    {
        auto firstIter = mFirstMap.find(firstKey);
        if (firstIter == mFirstMap.end())
            return mData.end();
        else
            return *firstIter;
    }

    Iterator secondFind(const KeyType& secondKey)
    {
        auto secondIter = mSecondMap.find(secondKey);
    }
if (secondIter == mSecondMap.end())
    return mData.end();
else
    return *secondIter;
}

template< typename... Args>
Iterator emplace(Args&&... args)
{
    mData.emplace_front(std::move(args)...);
    mFirstMap.emplace(mData.begin());
    mSecondMap.emplace(mData.begin());
    return mData.begin();
}
};

Listing B.2: CSplitSettings.h

/*
 * @file CSplitSettings.h
 * @author Ömer Sali <oemer.sali@rwth-aachen.de>
 * @version 2018-04-04
 * @Created on 2017-11-01.
 */

#pragma

#include "../../solver/ModuleSettings.h"
#include "../../solver/Manager.h"
#include "../SATModule/SATModule.h"
#include "../LRAModule/LRAModule.h"

namespace smtrat
{
    struct CSplitSettings1
    {
        /// Name of the Module
        static constexpr auto moduleName = "CSplitModule<CSplitSettings1>";
        /// Limit size for the domain of variables that need to be expanded
        static constexpr size_t maxDomainSize = 32;
        /// Base number 2 <= expansionBase <= maxDomainSize for the expansion
        static constexpr size_t expansionBase = 32;
        /// Common denominator for the discretization of rational variables
        static constexpr size_t discrDenom = 16;
        /// Maximum number of iterations before returning unknown (0 = infinite)
        static constexpr size_t maxIter = 50;
        /// Radius of initial variable domains
        static constexpr size_t initialRadius = 1;
        /// Threshold radius to
        /// - start exponential bloating of variables used for case splits
        /// - activate full domains of variables not used for case splits
        static constexpr size_t thresholdRadius = 3;
        /// Maximal radius of domain that still gets bloated (0 = infinite)
        static constexpr size_t maximalRadius = 300;
        /// Maximal number of bounds to bloat in one iteration (0 = infinite)
        static constexpr size_t maxBloatedDomains = 3;
        /// Linear integer arithmetic module to call for the linearized
        /// formula
        struct LIASolver : public Manager
        {
            LIASolver() : Manager()
            {
                setStrategy({
                    addBackend< SATModule< SATSettings1>>{}}
            };
```cpp
addBackend<LRAModule<LRASettings1>>()
}
};

Listing B.3: CSplitModule.h

/**
 * @file CSplitModule.h
 * @author Ömer Sali <oemer.sali@rwth-aachen.de>
 * @version 2018-04-04
 * Created on 2017-11-01.
 */

#pragma once

#include "../../datastructures/VariableBounds.h"
#include "../../solver/Module.h"
#include "Bimap.h"
#include "CSplitStatistics.h"
#include "CSplitSettings.h"

namespace smtrat {

class CSplitModule : public Module {

private:

#ifdef SMTRAT_DEVOPTION_Statistics

CSplitStatistics mStatistics;
#endif

struct Purification {

/// Variable sequence used for the virtual positional notation
std::vector<carl::Variable> mSubstitutions;

/// Number of active constraints in which the monomial is included
size_t mUsage;

bool mActive;

Purification()
: mSubstitutions()
, mReduction(carl::Variable::NO_VARIABLE)
, mUsage(0)
, mActive(false)
{
    mSubstitutions.emplace_back(carl::freshIntegerVariable());
}

};

std::map<carl::Monomial::Arg, Purification> mPurifications;

```
/// - UNBOUNDED, if domain size = infinity
enum class DomainSize {SMALL = 0, LARGE = 1, UNBOUNDED = 2};
/**
 * Represents the quotients and remainders of a variable in
 * a positional notation to the basis Settings::expansionBase.
 */

struct Expansion
{
    /// Original variable to which this expansion is dedicated to and
    /// its discrete substitute
    const carl::Variable mRationalization, mDiscretization;

    Rational mNucleus;

    /// Size of the maximal domain
    DomainSize mMaximalDomainSize;

    /// Maximal domain deduced from received constraints and the
    /// currently active domain
    RationalInterval mMaximalDomain, mActiveDomain;

    /// Sequences of quotients and remainders used for the virtual
    /// positional notation
    std::vector<carl::Variable> mQuotients, mRemainders;

    /// Active purifications of monomials that contain the
    /// rationalization variable
    std::vector<Purification *> mPurifications;

    /// Flag that indicates whether the variable bounds changed since
    /// last check call
    bool mChangedBounds;

    Expansion(const carl::Variable & rationalization)
        : mRationalization (rationalization)
        , mDiscretization (rationalization.type() == carl::VariableType::VT_INT ? rationalization : carl::freshIntegerVariable())
        , mNucleus (ZERO_RATIONAL)
        , mMaximalDomainSize (DomainSize::UNBOUNDED)
        , mMaximalDomain (RationalInterval::unboundedInterval())
        , mActiveDomain (RationalInterval::emptyInterval())
        , mChangedBounds (false)
    {
        mQuotients.emplace_back (mDiscretization);
    }

    Bimap<Expansion, const carl::Variable, &Expansion::mRationalization, &Expansion::mDiscretization> mExpansions;

    /**
     * Represents the class of all original constraints with the same
     * left hand side after a normalization. Here, the set of all
     * relations of constraints with the same left hand side is stored.
     */

    struct Linearization
    {
        /// Normalization of the original constraint to which this
        /// linearization is dedicated to
        const Poly mNormalization, mLinearization;

        /// Flag that indicates a sign change between the leading
        /// coefficient of normalization and linearization
        const bool mParity;

        /// Purifications of the original nonlinear monomials
        const std::vector<Purification *> mPurifications;

        /// Flag that indicates whether the original constraint contains
        /// real variables
        const bool mHasRealVariables;

        /// Relations of constraints with the same left hand side
        std::unordered_set<carl::Relation> mRelations;
Appendix B. CSplitModule source code

```cpp
Linearization(const Poly& normalization, const Poly& linearization,
               std::vector<Purification*>&& purifications, bool hasRealVariables)
    : mNormalization(normalization),
      mLinearization(linearization.normalize()),
      mParity(carl::isNegative(linearization.lcoeff())),
      mPurifications(std::move(purifications)),
      mHasRealVariables(std::move(hasRealVariables))
{};

Bimap<Linearization, const Poly, &Linearization::mNormalization, &
      Linearization::mLinearization> mLinearizations;

/// Helper class that extracts the variable domains
vb::VariableBounds<FormulaT> mVariableBounds;
/// Stores the last model for the linearization that was found by the
/// LIA solver
Model mLIAModel;
/// Stores whether the last consistency check was done by the
/// backends
bool mCheckedWithBackends;
/// Handle to the linear integer arithmetic module
typename Settings::LIASolver mLIASolver;

public:

typedef Settings SettingsType;

std::string moduleName() const
{
    return SettingsType::moduleName;
}

CSplitModule(const ModuleInput* _formula, RuntimeSettings* _settings,
             Conditionals& _conditionals, Manager* _manager = nullptr);

/**
 * The module has to take the given sub-formula of the received
 * formula into account.
 * @param _subformula The sub-formula to take additionally into
 * account.
 * @return False, if it can be easily decided that this sub-formula
 * causes a conflict with
 * the already considered sub-formulas;
 * True, otherwise.
 */
bool addCore(ModuleInput::const_iterator _subformula);

/**
 * Removes the subformula of the received formula at the given
 * position to the considered ones of this module.
 * @param _subformula The position of the subformula to remove.
 */
void removeCore(ModuleInput::const_iterator _subformula);

/**
 * Updates the current assignment into the model.
 * Note, that this is a unique but possibly symbolic assignment maybe
 * containing newly introduced variables.
 */
void updateModel() const;

/**
 * Checks the received formula for consistency.
 * @return SAT, if the received formula is satisfiable;
 */
```

* UNSAT, if the received formula is not satisfiable;
* UNKNOWN, otherwise.
*
Answer checkCore();

private:
/**
 * Resets all expansions to the center points of the variable domains
 * and
 * constructs a new tree of reductions for the currently active
 * monomials.
 * @return True, if there exists a maximal domain with no integral
 * points;
 * False, otherwise.
 */
bool resetExpansions();

/**
 * Bloats the active domains of a subset of variables that are part
 * of the LIA solvers
 * infeasible subset, and indicates if no active domain could be
 * bloated, because the the
 * maximal domain of all variables were reached.
 * @param LIAConflict Infeasible subset of the LIA solver
 * @return True, if no active domain was bloated;
 * False, otherwise.
 */
bool bloatDomains(const FormulaSetT& LIAConflict);

/**
 * Analyzes the infeasible subset of the LIA solver and constructs an
 * infeasible
 * subset of the received constraints. The unsatisfiability cannot be
 * deduced if
 * the corresponding original constraints contain real valued
 * variables.
 * @param LIAConflict Infeasible subset of the LIA solver
 * @return UNSAT, if an infeasible subset of the received
 * constraints could be constructed;
 * UNKNOWN, otherwise.
 */
Answer analyzeConflict(const FormulaSetT& LIAConflict);

/**
 * Changes the active domain of a variable and adapts its positional
 * notation
 * to the basis Settings::expansionBase.
 * @param expansion Expansion data structure thats keeps all needed
 * informations.
 * @param domain The new domain that shall be active afterwards. Note
 * that the new
 * domain has to contain the currently active interval.
 */
void changeActiveDomain(Expansion& expansion, RationalInterval&&
  domain);

/**
 * Asserts/Removes the given formula to/from the LIA solver.
 * @param formula The formula to assert/remove to the LIA solver.
 * @param assert True, if formula shall be asserted;
 * False, if formula shall be removed.
 */
inline void propagateFormula(const FormulaT& formula, bool assert);
}
/**
 * @file CSplitModule.cpp
 * @author Ömer Sali <oemer.sali@rwth-aachen.de>
 * @version 2018-04-04
 * @Created on 2017-11-01.
 */

#include "CSplitModule.h"

namespace smtrat {

template<class Settings>
CSplitModule<Settings>::CSplitModule(const ModuleInput* _formula,
    RuntimeSettings*, Conditionals& _conditionals, Manager* _manager)
    : Module(_formula, _conditionals, _manager)
    , mPurifications()
    , mExpansions()
    , mLinearizations()
    , mVariableBounds()
    , mLIAModel()
    , mCheckedWithBackends(false)
    #ifdef SMTRAT_DEVOPTION_Statistics
    , mStatistics(Settings::moduleName)
    #endif
    {};

template<class Settings>
bool CSplitModule<Settings>::addCore(ModuleInput::const_iterator _subformula)
{
    addReceivedSubformulaToPassedFormula(_subformula);
    const FormulaT& formula{_subformula->formula()};
    if (formula.getType() == carl::FormulaType::FALSE)
        mInfeasibleSubsets.push_back({formula});
    else if (formula.isBound())
    {
        /// Update the variable domain with the asserted bound
        mVariableBounds.addBound(formula, formula);
        auto carl::Variable& variable{*formula.variables().begin()};
        if (expansionIter == mExpansions.end())
            expansionIter = mExpansions.emplace(variable);
        expansionIter->mChangedBounds = true;
        if (mVariableBounds.isConflicting())
            mInfeasibleSubsets.emplace_back(mVariableBounds.getConflict());
    }
    else if (formula.getType() == carl::FormulaType::CONSTRAINT)
    {
        /// Normalize the left hand side of the constraint and turn the
        relation accordingly
        const ConstraintT& constraint{formula.constraint()};
        const Poly normalization{constraint.lhs().normalize()};
        carl::Relation relation{constraint.relation()};
        if (carl::isNegative(constraint.lhs().lcoeff()))
            relation = carl::turnAroundRelation(relation);
        /// Purify and discretize the normalized left hand side to construct
        the linearization
        auto linearizationIter{mLinearizations.firstFind(normalization)};
        if (linearizationIter == mLinearizations.end())
        {
            Poly discretization;
            std::vector<Purification *> purifications;
            bool hasRealVariables{false};
            for (TermT term : normalization)
            {
if (!term.isConstant())
{
    size_t realVariables(0);
    for (const auto& exponent : term.monomial()->exponents())
        if (exponent.first.type() == carl::VariableType::VT_REAL)
            realVariables += exponent.second;
    if (realVariables)
        { 
            term.coeff() /= carl::pow(Rational(Settings::discrDenom),
                                      realVariables);
            hasRealVariables = true;
        }
    if (!term.isLinear())
    {
        Purification& purification(mPurifications[term.monomial()]);
        purifications.emplace_back(&purification);
    }
    else if (realVariables)
    {
        const carl::Variable variable(term.getSingleVariable());
        auto expansionIter(mExpansions.firstFind(variable));
        if (expansionIter == mExpansions.end())
            expansionIter = mExpansions.emplace(variable);
        term = term.coeff()*expansionIter->mQuotients[0];
    }
    discretization += term;
}
linearizationIter = mLinearizations.emplace(normalization,
                                           discretization, std::move(purifications), hasRealVariables);
Linearization& linearization{*linearizationIter};
propagateFormula(FormulaT(linearization.mLinearization, linearization.mParity ? carl::turnAroundRelation(relation) : relation), true);
    if (linearization.mRelations.empty())
        for (Purification *purification : linearization.mPurifications)
            ++purification->mUsage;
        linearization.mRelations.emplace(relation);
    /// Check if the asserted relation trivially conflicts with other asserted relations
    switch (relation)
    {
    case carl::Relation::EQ:
        if (linearization.mRelations.count(carl::Relation::NEQ))
            mInfeasibleSubsets.push_back({
                FormulaT(normalization, carl::Relation::EQ),
                FormulaT(normalization, carl::Relation::NEQ)});
    case carl::Relation::GREATER:
        if (linearization.mRelations.count(carl::Relation::LEQ))
            mInfeasibleSubsets.push_back({
                FormulaT(normalization, carl::Relation::EQ),
                FormulaT(normalization, carl::Relation::LEQ)});
    break;
    case carl::Relation::NEQ:
        if (linearization.mRelations.count(carl::Relation::EQ))
            mInfeasibleSubsets.push_back({
                FormulaT(normalization, carl::Relation::EQ),
                FormulaT(normalization, carl::Relation::NEQ)});
Appendix B. CSplitModule source code

```cpp
127   };
128   break;
129   case carl::Relation::LESS:
130     if (linearization.mRelations.count(carl::Relation::EQ))
131       mInfeasibleSubsets.push_back(
132         FormulaT(normalization, carl::Relation::LESS),
133         FormulaT(normalization, carl::Relation::EQ));
134     if (linearization.mRelations.count(carl::Relation::GEQ))
135       mInfeasibleSubsets.push_back(
136         FormulaT(normalization, carl::Relation::LESS),
137         FormulaT(normalization, carl::Relation::GEQ));
138   case carl::Relation::LEQ:
139     if (linearization.mRelations.count(carl::Relation::GREATER))
140       mInfeasibleSubsets.push_back(
141         FormulaT(normalization, relation),
142         FormulaT(normalization, carl::Relation::GREATER));
143     break;
144   case carl::Relation::GREATER:
145     if (linearization.mRelations.count(carl::Relation::EQ))
146       mInfeasibleSubsets.push_back(
147         FormulaT(normalization, carl::Relation::GREATER),
148         FormulaT(normalization, carl::Relation::EQ));
149     if (linearization.mRelations.count(carl::Relation::LEQ))
150       mInfeasibleSubsets.push_back(
151         FormulaT(normalization, carl::Relation::GREATER),
152         FormulaT(normalization, carl::Relation::LEQ));
153   case carl::Relation::GEQ:
154     if (linearization.mRelations.count(carl::Relation::LESS))
155       mInfeasibleSubsets.push_back(
156         FormulaT(normalization, relation),
157         FormulaT(normalization, carl::Relation::LESS));
158     break;
159   default:
160     assert(false);
161 }
162 return mInfeasibleSubsets.empty();
163 }
164
165 template<class Settings>
166 void CSplitModule<Settings>::removeCore(ModuleInput::const_iterator _subformula)
167 {
168   const FormulaT& formula{_subformula->formula()};
169   if (formula.isBound())
170     { // Update the variable domain with the removed bound
171       mVariableBounds.removeBound(formula, formula);
172       mExpansions.firstAt(*formula.variables().begin()).mChangedBounds =
173         true;
174     }
175   else if (formula.getType() == carl::FormulaType::CONSTRAINT)
176     { // Normalize the left hand side of the constraint and turn the
177       constraint{constraint(formula.constraint())};
178       const Poly normalization{constraint.lhs().normalize()};
179       carl::Relation relation{constraint.relation()};
180       if (carl::isNegative(constraint.lhs().lcoeff()))
181         relation = carl::turnAroundRelation(relation);
182   }
```
/// Retrieve the normalized constraint and mark the separator object as changed
Linearization& linearization = mLinearizations.firstAt(normalization);
propagateFormula(FormulaT(linearization.mLinearization, linearization.mParity ? carl::turnAroundRelation(relation) : false);
linearization.mRelations.erase(relation);
if (linearization.mRelations.empty())
    for (Purification* purification : linearization.mPurifications)
        ++purification->mUsage;
}

template<class Settings>
void CSplitModule<Settings>::updateModel() const
{
    if (!mModelComputed)
    {
        clearModel();
        if (mCheckedWithBackends)
        {
            getBackendsModel();
            excludeNotReceivedVariablesFromModel();
        }
        else
        {
            for (const Expansion& expansion : mExpansions)
                if (receivedVariable(expansion.mRationalization))
                    {
                        Rational value = mLIAModel.at(expansion.mDiscretization).asRational();
                        if (expansion.mRationalization.type() == carl::VariableType::VT_REAL)
                            value /= Settings::discrDenom;
                        mModel.emplace(expansion.mRationalization, value);
                    }
        }
        mModelComputed = true;
    }
}

template<class Settings>
Answer CSplitModule<Settings>::checkCore()
{
    /// Report unsatisfiability if the already found conflicts are still unresolved
    if (!mInfeasibleSubsets.empty())
        return Answer::UNSAT;

    /// Apply the method only if the asserted formula is not trivially undecidable
    if (rReceivedFormula().isConstraintConjunction())
        {
            Answer answer = Answer::UNKNOWN;
            mLIASolver.push();
            if (resetExpansions())
                {
                    Answer LIAAnswer = Answer::UNSAT;
                    for (size_t i = 1; LIAAnswer == Answer::UNSAT && (!Settings::maxIter || i <= Settings::maxIter); ++i)
                        {
                            LIAAnswer = mLIASolver.check(true);
                            if (LIAAnswer == Answer::SAT)
                                {
                                    mLIAModel = mLIASolver.model();
                                    answer = Answer::SAT;
                                }}}}
```cpp
else if (LIAAnswer == Answer::UNSAT)
{
    FormulaSetT LIAConflict{mLIASolver.infeasibleSubsets()[0]};
    if (bloatDomains(LIAConflict))
    {
        LIAAnswer = Answer::UNKNOWN;
        answer = analyzeConflict(LIAConflict);
    }
    mLIASolver.pop();
    if (answer != Answer::UNKNOWN)
    {
        mCheckedWithBackends = false;
        return answer;
    }
    // Check the asserted formula with the backends
    mCheckedWithBackends = true;
    Answer answer(runBackends());
    if (answer == Answer::UNSAT)
        getInfeasibleSubsets();
    return answer;
}
template<class Settings>
bool CSplitModule<Settings>::resetExpansions()
{
    /// Update the variable domains and watch out for discretization conflicts
    for (Expansion& expansion : mExpansions)
    {
        RationalInterval& maximalDomain{expansion.mMaximalDomain};
        if (expansion.mChangedBounds)
        {
            maximalDomain = mVariableBounds.getInterval(expansion.mRationalization);
            if (expansion.mRationalization.type() == carl::VariableType::VT_REAL)
                maximalDomain *= Rational(Settings::discrDenom);
            if (expansion.mMaximalDomain.isUnbounded())
                expansion.mMaximalDomainSize = DomainSize::UNBOUNDED;
            else if (expansion.mMaximalDomainSize > Settings::maxDomainSize)
                expansion.mMaximalDomainSize = DomainSize::LARGE;
            else
                expansion.mMaximalDomainSize = DomainSize::SMALL;
            expansion.mChangedBounds = false;
        }
        if (maximalDomain.isEmpty())
            return false;
        expansion.mActiveDomain = RationalInterval::emptyInterval();
        expansion.mPurifications.clear();
    }
    /// Activate all used purifications bottom-up
    for (auto purificationIter = mPurifications.begin(); purificationIter != mPurifications.end(); ++purificationIter)
    {
        Purification& purification{purificationIter->second};
        if (purification.mUsage)
        {
            carl::Monomial::Arg monomial{purificationIter->first};
```
Find set of variables with maximal domain size

```cpp
carl::Variables maxVariables;
DomainSize maxDomainSize(DomainSize::SMALL);
for (const auto& exponent : monomial->exponents()) {
    const carl::Variable& variable(exponent.first);
    auto expansionIter(mExpansions.firstFind(variable));
    if (expansionIter == mExpansions.end())
        expansionIter = mExpansions.emplace(variable);
    Expansion& expansion(*expansionIter);

    if (maxDomainSize <= expansion.mMaximalDomainSize)
        if (maxDomainSize < expansion.mMaximalDomainSize)
            maxVariables.clear();
        maxDomainSize = expansion.mMaximalDomainSize;
        maxVariables.emplace(variable);
}
```

Find a locally optimal reduction for the monomial

```cpp
const auto isReducible = [&](const auto& purificationsEntry) {
    return purificationsEntry.second.mActive
        && monomial->divisible(purificationsEntry.first)
        && std::any_of(
            maxVariables.begin(),
            maxVariables.end(),
            [&](const carl::Variable& variable) {
                return purificationsEntry.first->has(variable);
            });
};
```

Activate the sequence of reductions top-down

```cpp
if (reductionIter == mPurifications.end())
    monomial->divide(*maxVariables.begin(), guidance);
else
    monomial->divide(reductionIter->first, guidance);
```

```cpp
auto hintIter(purificationIter);
for (const auto& exponentPair : guidance->exponents()) {
    const carl::Variable& variable(exponentPair.first);
    Expansion& expansion(mExpansions.firstAt(variable));
    for (carl::exponent exponent = 1; exponent <= exponentPair.second;
        ++exponent) {
        hintIter->second.mActive = true;
        expansion.mPurifications.emplace_back(&hintIter->second);
        monomial->divide(variable, monomial);
        if (monomial->isAtMostLinear())
            hintIter->second.mReduction = mExpansions.firstAt(monomial->
                getSingleVariable()).mQuotients[0];
        else
            auto temp{mPurifications.emplace_hint(hintIter, std::piecewise_construct,
                std::make_tuple(monomial), std::make_tuple());
            hintIter->second.mReduction = temp->second.mSubstitutions[0];
        hintIter = temp;
    }
```
Appendix B. CSplitModule source code

```cpp
else
    purification.mActive = false;

/// Activate expansions that are used for case splits and deactivate
them otherwise
for (Expansion& expansion : mExpansions)
{
    /// Calculate the center point where the initial domain is located
    expansion.mNucleus = ZERO_RATIONAL;
    if (expansion.mMaximalDomain.lowerBoundType() != carl::BoundType::INFTY
        && expansion.mNucleus < expansion.mMaximalDomain.lower())
        expansion.mNucleus = expansion.mMaximalDomain.lower();
    else if (expansion.mMaximalDomain.upperBoundType() != carl::BoundType::INFTY
        && expansion.mNucleus > expansion.mMaximalDomain.upper())
        expansion.mNucleus = expansion.mMaximalDomain.upper();

    /// Calculate and activate the corresponding domain
    RationalInterval domain(0, 1);
    domain.mul_assign(Rational(Settings::initialRadius));
    domain.add_assign(expansion.mNucleus);
    domain.intersect_assign(expansion.mMaximalDomain);
    changeActiveDomain(expansion, std::move(domain));
}
return true;
}

template<class Settings>
bool CSplitModule<Settings>::bloatDomains(const FormulaSetT& LIAConflict)
{
    /// Data structure for potential bloating candidates
    struct Candidate
    {
        Expansion& mExpansion;
        const Rational mDirection;
        const Rational mRadius;
    }
    Candidate Candidate(Expansion& expansion, Rational&& direction, Rational&& radius)
        : mExpansion(expansion),
          mDirection(std::move(direction)),
          mRadius(std::move(radius))
    {
    }

    bool operator<(const Candidate& rhs) const
    {
        if (mDirection*rhs.mDirection == ONE_RATIONAL)
            return mRadius < rhs.mRadius;
        else if (mDirection == ONE_RATIONAL)
            return mRadius < Rational(Settings::thresholdRadius);
        else
            return rhs.mRadius >= Rational(Settings::thresholdRadius);
    }

    std::set<Candidate> candidates;

    /// Scan the infeasible subset of the LIA solver for potential
candidates
    for (const FormulaT& formula : LIAConflict)
    if (formula.isBound())
    {
        const ConstraintT& constraint{formula.constraint()};
    }
```
const carl::Variable& variable {*constraint.variables().begin()};
auto expansionIter{mExpansions.secondFind(variable)};
if (expansionIter != mExpansions.end())
{
  Expansion& expansion{*expansionIter};
  Rational direction;
  if (constraint.isLowerBound())
    && (expansion.mMaximalDomain.lowerBoundType() == carl::BoundType::INFTY
        || expansion.mMaximalDomain.lower() < expansion.mActiveDomain.
        lower()))
    direction = MINUS_ONE_RATIONAL;
  else if (constraint.isUpperBound())
    && (expansion.mMaximalDomain.upperBoundType() == carl::BoundType::INFTY
        || expansion.mMaximalDomain.upper() > expansion.mActiveDomain.
        upper()))
    direction = ONE_RATIONAL;
  if (direction != ZERO_RATIONAL)
  {
    Rational radius{(direction*(expansion.mActiveDomain-expansion.
      mNucleus)).upper()};
    if (! Settings::maximalRadius
        || radius <= Settings::maximalRadius)
      candidates.emplace(expansion, std::move(direction), std::move(
        radius));
    if (Settings::maxBloatedDomains
        && candidates.size() > Settings::maxBloatedDomains)
      candidates.erase(std::prev(candidates.end()));
  }
}

/// Change the active domain of the candidates with highest priority
for (const Candidate& candidate : candidates)
{
  RationalInterval domain;
  if (candidate.mRadius <= Settings::thresholdRadius)
    domain = RationalInterval(0, ONE_RATIONAL);
  else if (candidate.mExpansion.mPurifications.empty())
    domain = RationalInterval(0, carl::BoundType::WEAK, 0, carl::
      BoundType::INFTY);
  else
    domain = RationalInterval(0, candidate.mRadius);
  domain.mul_assign(candidate.mDirection);
  domain.add_assign(candidate.mExpansion.mActiveDomain);
  domain.intersect_assign(candidate.mExpansion.mMaximalDomain);
  changeActiveDomain(candidate.mExpansion, std::move(domain));
}

/// Report if any variable domain was bloated
return candidates.empty();

template<class Settings>
Answer CSplitModule<Settings>::analyzeConflict(const FormulaSetT&
LIAConflict)
{
  /// Construct an infeasible subset from the LIA conflict
  FormulaSetT conflict;
  for (const FormulaT& formula : LIAConflict)
  {
    if (formula.isBound())
    {  

auto expansionIter = mExpansions.secondFind(*formula.variables().begin());
if (expansionIter != mExpansions.end())
{
    const Expansion& expansion = *expansionIter;
    if (expansion.mRationalization.type() == carl::VariableType::VT_REAL
        || expansion.mMaximalDomain != expansion.mActiveDomain)
        return Answer::UNKNOWN;
    else
    {
        auto expansionIter { mExpansions.secondFind(*formula.variables().begin());
        if (expansionIter != mExpansions.end())
        {
            const Expansion & expansion {* expansionIter;}
        
            if (expansion.mRationalization.type() == carl::VariableType::VT_REAL
                || expansion.mMaximalDomain != expansion.mActiveDomain)
                return Answer::UNKNOWN;
        }
else if (formula.getType() == carl::FormulaType::CONSTRAINT)
        {
        const ConstraintT & constraint = formula.constraint();
        auto linearizationIter { mLinearizations.secondFind(constraint.lhs().normalize());
        if (linearizationIter != mLinearizations.end())
        {
            const Linearization & linearization {* linearizationIter;}
            if (linearization.mHasRealVariables)
                return Answer::UNKNOWN;
        else
            {
        carl::Relation relation { constraint.relation()};
        if (carl::isNegative(constraint.lhs().lcoeff()) != linearization.mParity)
            relation = carl::turnAroundRelation(relation);
        conflict.emplace(linearization.mNormalization, relation);
            }
        }
else if (!domain.isEmpty())
            mInfeasibleSubsets.emplace_back(std::move(conflict));
        return Answer::UNSAT;
    }
}

template <class Settings>
void CSplitModule<Settings>::changeActiveDomain(Expansion& expansion, RationalInterval && domain)
{
    RationalInterval activeDomain { move(expansion.mActiveDomain) };
    expansion.mActiveDomain = domain;

    /// Update the variable bounds
    if (!activeDomain.isEmpty())
    {
        if (activeDomain.lowerBoundType() != carl::BoundType::INFTY
            && (domain.lowerBoundType() == carl::BoundType::INFTY
                || domain.lower() != activeDomain.lower())
        {
            propagateFormula(FormulaT(expansion.mQuotients[0] - Poly(activeDomain.lower()), carl::Relation::GEQ), false);
        }
        if (activeDomain.upperBoundType() != carl::BoundType::INFTY
            && (domain.upperBoundType() == carl::BoundType::INFTY
                || domain.upper() != activeDomain.upper())
        {
            propagateFormula(FormulaT(expansion.mQuotients[0] - Poly(activeDomain.upper()), carl::Relation::LEQ), false);
        }
    }
}
\[
\begin{align*}
       \text{if} \ (\text{domain.lowerBoundType()} != \text{carl::BoundType::INFTY} \\
       \quad \&\& \ (\text{activeDomain.lowerBoundType()} == \text{carl::BoundType::INFTY} \\
       \quad \quad | | \ \text{activeDomain.lower()} != \text{domain.lower()} \\
       \quad \quad | | \ \text{activeDomain.isEmpty()})
       \quad \text{propagateFormula}(&\text{FormulaT}\text{(expansion.mQuotients[0]-Poly(domain.lower ()}, \\
                \text{carl::Relation::GEQ), true);} \\
       \text{if} \ (\text{domain.upperBoundType()} != \text{carl::BoundType::INFTY} \\
       \quad \&\& \ (\text{activeDomain.upperBoundType()} == \text{carl::BoundType::INFTY} \\
       \quad \quad | | \ \text{activeDomain.upper()} != \text{domain.upper()} \\
       \quad \quad | | \ \text{activeDomain.isEmpty()})
       \quad \text{propagateFormula}(&\text{FormulaT}\text{(expansion.mQuotients[0]-Poly(domain.upper ()}, \\
                \text{carl::Relation::LEQ), true);}
       \\
       \\
       \text{// Check if the digits of the expansion need to be encoded} \\
       \text{if} \ (\text{expansion.mPurifications.empty()})
       \quad \{ \\
       \quad \quad \text{activeDomain} = \text{RationalInterval::emptyInterval();} \\
       \quad \quad \text{domain} = \text{RationalInterval::emptyInterval();} \\
       \quad \} \\
       \\
       \text{// Update the case splits of the corresponding digits} \\
       \text{for} \ (\text{size_t} \ i = 0; \ \text{activeDomain != domain; ++i)}
       \quad \{ \\
       \quad \quad \text{if} \ (\text{domain.diameter()} \leq \text{Settings::maxDomainSize})
       \quad \quad \{ \\
       \quad \quad \quad \text{// Update the currently active linear encoding} \\
       \quad \quad \quad \text{Rational} \ \text{lower} \ {\text{activeDomain.isEmpty() ? domain.lower() :}} \\
       \quad \quad \quad \text{activeDomain.lower();} \\
       \quad \quad \quad \text{Rational} \ \text{upper} \ {\text{activeDomain.isEmpty() ? domain.lower() :}} \\
       \quad \quad \quad \text{domain.upper()}}+\text{ONE_RATIONAL}; \\
       \quad \quad \quad \text{for} \ (\text{const Purification} * \text{purification : expansion.mPurifications})
       \quad \quad \quad \{ \\
       \quad \quad \quad \quad \text{Rational} \ \text{alpha} = \text{domain.lower();} \ \text{alpha} < \text{lower; ++alpha)} \\
       \quad \quad \quad \text{propagateFormula(} \\
       \quad \quad \quad \quad \text{carl::FormulaType::IMPLIES,} \\
       \quad \quad \quad \quad \text{FormulaT}(\text{Poly(expansion.mQuotients[i]-Poly(alpha)}, \text{carl::}} \\
       \quad \quad \quad \quad \text{Relation::EQ),} \\
       \quad \quad \quad \quad \text{FormulaT}(\text{Poly(purification->mSubstitutions[i]-Poly(alpha)*Poly(} \\
       \quad \quad \quad \quad \text{purification->mReduction), \text{carl::Relation::EQ) }} \\
       \quad \quad \quad \quad \text{true),}} \\
       \quad \quad \quad \quad \text{true;}} \\
       \quad \quad \quad \text{for} \ (\text{Rational} \ \text{alpha} = \text{upper; alpha} \leq \text{domain.upper(); ++alpha}) \\
       \quad \quad \quad \text{propagateFormula(} \\
       \quad \quad \quad \quad \text{carl::FormulaType::IMPLIES,} \\
       \quad \quad \quad \quad \text{FormulaT}(\text{Poly(expansion.mQuotients[i]-Poly(alpha)}, \text{carl::}} \\
       \quad \quad \quad \quad \text{Relation::EQ),} \\
       \quad \quad \quad \quad \text{FormulaT}(\text{Poly(purification->mSubstitutions[i]-Poly(alpha)*Poly(} \\
       \quad \quad \quad \quad \text{purification->mReduction), \text{carl::Relation::EQ) }} \\
       \quad \quad \quad \quad \text{true,}} \\
       \quad \quad \quad \quad \text{true;}} \\
       \quad \}\} \\
       \text{else if} \ (\text{activeDomain.diameter()} \leq \text{Settings::maxDomainSize})
       \quad \{ \\
       \quad \quad \text{// Switch from the linear to a logarithmic encoding} \\
       \quad \quad \text{if} \ (\text{expansion.mQuotients.size()} \leq \text{i+1})
       \quad \quad \{ \\
       \quad \quad \quad \text{expansion.mQuotients.emplace_back(carl::freshIntegerVariable());} \\
       \quad \quad \quad \text{expansion.mRemainders.emplace_back(carl::freshIntegerVariable());} \\
       \quad \quad \} \\
       \quad \text{for} \ (\text{Purification} * \text{purification : expansion.mPurifications})
       \quad \}
\end{align*}
\]
Appendix B. CSplitModule source code

```cpp
{  
    if (purification->mSubstitutions.size() <= i+1)  
        purification->mSubstitutions.emplace_back(carl::freshIntegerVariable());  
    for (Rational alpha = activeDomain.lower(); alpha <= activeDomain.upper(); ++alpha)  
        propagateFormula(  
            FormulaT(  
                carl::FormulaType::IMPLIES,  
                FormulaT(Poly(expansion.mQuotients[i]) - Poly(alpha), carl::Relation::EQ),  
                FormulaT(Poly(purification->mSubstitutions[i]) - Poly(alpha)*Poly(purification->mReduction), carl::Relation::EQ)  
            ),  
            false  
        );  
    for (Rational alpha = ZERO_RATIONAL; alpha < Settings::expansionBase; ++alpha)  
        propagateFormula(  
            FormulaT(  
                carl::FormulaType::IMPLIES,  
                FormulaT(Poly(expansion.mRemainders[i]) - Poly(alpha), carl::Relation::EQ),  
                FormulaT(Poly(purification->mSubstitutions[i]) - Poly(Settings::expansionBase)*Poly(purification->mSubstitutions[i+1]) - Poly(alpha)*Poly(purification->mReduction), carl::Relation::EQ)  
            ),  
            true  
        );  
    propagateFormula(FormulaT(Poly(purification->mSubstitutions[i]) - Poly(expansion.mQuotients[i]), carl::Relation::EQ), false);  
    propagateFormula(FormulaT(Poly(purification->mSubstitutions[i]) - Poly(expansion.mRemainders[i]), carl::Relation::GEQ), false);  
    propagateFormula(FormulaT(Poly(purification->mSubstitutions[i]) - Poly(expansion.mRemainders[i]) - Poly(Settings::expansionBase-1), carl::Relation::LEQ), false);  
}  

/// Calculate the domain of the next digit  
if (!activeDomain.isEmpty())  
    if (activeDomain.diameter() <= Settings::maxDomainSize)  
        activeDomain = RationalInterval::emptyInterval();  
    else  
        activeDomain = carl::floor(activeDomain/Rational(Settings::expansionBase));  
if (!domain.isEmpty())  
    if (domain.diameter() <= Settings::maxDomainSize)  
        domain = RationalInterval::emptyInterval();  
    else  
        domain = carl::floor(domain/Rational(Settings::expansionBase));  
/// Update the variable bounds of the next digit  
if (!activeDomain.isEmpty())  
    if (domain.isEmpty() || domain.lower() != activeDomain.lower())  
        propagateFormula(FormulaT(expansion.mQuotients[i+1] - Poly(activeDomain.lower()), carl::Relation::GEQ), false);  
    if (domain.isEmpty() || domain.upper() != activeDomain.upper())  
        propagateFormula(FormulaT(expansion.mQuotients[i+1] - Poly(activeDomain.upper()), carl::Relation::LEQ), false);  
    if (!domain.isEmpty())  
    {  
        if (activeDomain.isEmpty() || activeDomain.lower() != domain.lower() )  
            propagateFormula(FormulaT(expansion.mQuotients[i+1] - Poly(domain.  
```
lower(), carl::Relation::GEQ), true);

if (activeDomain.isEmpty() || activeDomain.upper() != domain.upper())
    propagateFormula(FormulaT(expansion.mQuotients[i+1]-Poly(domain.
        upper())), carl::Relation::LEQ), true);
}
}
}

template <class Settings>
inline void CSplitModule<Settings>::propagateFormula(const FormulaT&
    formula, bool assert)
{
    if (assert)
        mLIASolver.add(formula);
    else
        mLIASolver.remove(std::find(mLIASolver.formulaBegin(), mLIASolver.
            formulaEnd(), formula));
}

#include "Instantiation.h"
Bibliography


Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Name, Vorname/Last Name, First Name

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/
Masterarbeit* mit dem Titel
I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

___________________________    ___________________________
Ort, Datum/City, Date      Unterschrift/Signature

Belehrung: Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

§ 161 StGB: Fahrlässige Falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

___________________________    ___________________________
Ort, Datum/City, Date      Unterschrift/Signature