RWTH AACHEN
UNIVERSITY
INFORMATIK

DIPLOMA THESIS

# Virtual Substitution in SMT Solving

Florian Corzilius

*First supervisor:*
Prof. Dr. Erika Ábrahám

*Second supervisor:*
Prof. Dr. Jürgen Giesl

## Danksagung

Ich bedanke mich bei allen, die mich auf meinem Weg bis zum heutigen Tage begleitet haben. Jeder von Euch, ob Familienmitglied oder Freund, hat und/oder hatte seinen Einfluss auf das, was mich heute ausmacht.
Ich möchte mich auch insbesondere bei Erika Ábrahám und Ulrich Loup bedanken, die mir dieses wunderbare Thema anvertrauten, mich in diese Arbeit einführten und mir stets meine Fragen beantworteten.
Mein Dank gilt natürlich auch denjenigen, die mir bei der Korrektur halfen sowie Verbesserungsvorschläge bei der Gestaltung gaben.

## Erklärung

Hiermit versichere ich, dass ich die vorgelegte Diplomarbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

(Florian Corzilius)
Aachen, den 14.12.2010

SMT-solving combines two highly explored research topics, SAT-solving and theory solving. SAT-solving handles the logical part of the given problem and the theory solver resolves the arithmetic part. To optimize their interaction, the theory solver must provide *incrementality*, *backtracking* and *minimal infeasible subset* generation.

Current SMT-solvers have their main focus on linear arithmetic and just a few are capable of deciding over the more expressive but still decidable logics like the first-order theory of the reals with addition and multiplication – *real algebra*. However, their approaches are based on approximations or are not complete.

As one of the few approaches deciding real algebra, the *virtual substitution* method fits to embed it in an SMT-solver providing incrementality, backtracking and minimal infeasible subset generation. This thesis addresses a new implementation of the virtual substitution method, providing these abilities.

# Contents

# 1. Introduction

The *satisfiability problem* poses the question whether a given logical formula is satisfiable, i.e., whether we can assign values to the variables contained in the formula such that the formula becomes $\mathrm{True}$. The development of efficient algorithms and tools (*solvers*) for satisfiability checking forms an active research area in computer science. A lot of effort has been put into the development of fast solvers for the propositional satisfiability problem, called SAT. To increase expressiveness, extensions of the propositional logic with respect to first-order *theories* can be considered. The corresponding satisfiability problems are called *SAT-modulo-theories* problems, short *SMT*. SMT-solvers exist, e.g., for equality logic, uninterpreted functions, predicate logic, and linear real arithmetic.

In contrast to the above-mentioned theories, less activity can be observed for SMT-solvers supporting the first-order theory of the real ordered field, which we call *real algebra*. Our research goal is to develop an SMT-solver for real algebra, which is capable of solving Boolean combinations of polynomial constraints over the reals efficiently.

Even though decidability of real algebra has been known for a long time [Tar48], the first decision procedures were not yet practicable. Since 1974 we have known that the time complexity of deciding formulas of real algebra is in worst case doubly exponential in the number of variables (dimension) contained in the formula [DH88, Wei88, BD07].

Today, several methods are available which satisfy these complexity bounds, for example the cylindrical algebraic decomposition (CAD) [CJ98] , the Gröbner basis, and the virtual substitution method [Wei98]. An overview of these methods is given in [DSW97]. There are also tools available which implement these methods. The stand-alone application `QEPCAD` is a `C++` implementation of the CAD method [Bro03]. Another example is the `Redlog` package [DS97] of the computer algebra system `Reduce` based on `Lisp`, which offers an optimized combination of the virtual substitution, the CAD method, and real root counting.

Currently existing solvers are not suited to solve large formulas containing arbitrary combinations of real constraints. We want to combine the advantages of highly tuned SAT-solvers and the most efficient techniques currently available for solving conjunctions of real constraints, by implementing an SMT-solver for real algebra capable of efficiently solving arbitrary Boolean combinations of real constraints.

Theory solvers should satisfy specific requirements in order to embed them into an SMT-solver efficiently:

- *Incrementality:* The theory solver should be able to accept theory constraints one after the other. After it receives one or more new theory constraints it should check the conjunction of the new constraints with the earlier constraints for satisfiability. For efficiency it is important that the solver does not do unnecessary work and makes

use of the results of earlier checks.

- *(Minimal) infeasible subsets:* If the theory solver detects a conflict, it should give a reason for the unsatisfiability. The usual way is to determine an unsatisfiable subset of the constraints. Even better is if we find a minimal unsatisfiable subset in the sense that if we remove a constraint the remaining ones become satisfiable.
- *Backtracking:* If a conflict occurs, either in the Boolean or in the theory domain, the solver should be able to remove some constraints and continue the check at an earlier state.

To our knowledge, these functionalities are currently not supported by the available solvers for real algebra. In this thesis we extend the virtual substitution method to support incrementality, backtracking, and the generation of (minimal) infeasible subsets.

We have chosen the virtual substitution method because it is a restricted but very efficient decision procedure for a subset of real algebra. The restriction concerns the degree of polynomials. The method uses solution equations to determine the zeros of (multivariate) polynomials in a given variable. As such solution equations exist for polynomials of degree at most $4$, the method is a priori restricted in the degree of polynomials. In this thesis we restrict ourselves to polynomials of degree $2$. An extension to formulas containing higher degrees will be future work. The idea is to extend the degree to its theoritical maximum of $4$ and to pass polynomials of higher degrees to another implementation using the CAD method.

**Related work** We are only aware of the SMT-solvers `Z3` [dMB08], `HySAT` [FHT+07] and `ABsolver` [BPT07] which are able to handle nonlinear real arithmetic constraints. The algorithm implemented in `HySAT` and currently in its successor tool `iSAT` uses interval constraint propagation to check real constraints. This technique is only pseudo-complete, i.e. it sometimes cannot solve the problem with a clear satisfiability result. Nevertheless it is in practice more efficient compared to solvers based upon exact methods [FHT+07]. The structures of `ABsolver` and `Z3` are more similar to our planned SMT-solver. However to our knowledge, `Z3` does not support full real-algebraic constraints. The authors of `ABsolver` do not address the issues of incrementality and backtracking. Though `ABsolver` computes minimal infeasible subsets, we did not find any information about how they are generated.

**Thesis structure** The remaining part of the thesis is structured as follows: Chapter 2 introduces definitions and algorithms we use in this thesis. We formalize the SMT-solving mechanism in Chapter 3 and the virtual substitution method in Chapter 4. In Chapter 5 we implement the virtual substitution method such that it supports incrementality. In Chapter 6 we extend the resulting implementation by an interface to achieve a (minimal) infeasible subset. The remaining requirement we want the theory solver to fulfill, backtracking, is introduced in Chapter 7. We show the first experimental results in Chapter 8 and give an outlook on our future goals in Chapter 9.

# 2. Basic definitions

First we introduce the syntax and the semantics of real-algebraic formulas.

## 2.1. Syntax of real-algebraic formulas

In the following we give the syntax of real-algebraic formulas, being the quantified boolean combination of real-algebraic constraints. Those constraints compare a polynomial with a constant.

The virtual substitution method introduced later in this chapter also creates and handles extended polynomials containing roots and fractions. Such extended polynomials do not appear in real-algebraic formulas, but they form intermediate results during the virtual substitution. Therefore we first give a general syntax allowing roots and fractions. Real-algebraic formulas build a subset, which do not contain those constructs.

Assume in the following a set Var of real-valued variables.

**Definition 2.1.1 (Extended polynomials, polynomials)**
*An* extended (multivariate) polynomial $e$ *is constructed by the following abstract grammar* [1]*:*

$$e ::= c \quad | \quad x \quad | \quad (e+e) \quad | \quad (e \cdot e) \quad | \quad (e/e) \quad | \quad (\sqrt[i]{e})$$

*with* $x \in$ *Var,* $c \in \mathbb{Q}$*, and* $i \in \mathbb{N}$ *with* $i \geq 2$*. The extended polynomial* $(e_1 + e_2)$ *is the* sum *and* $(e_1 \cdot e_2)$ *is the* product of $e_1$ and $e_2$*; we call* $(\sqrt[i]{e})$ *the* $i$th root of $e$ *and* $(e_1/e_2)$ *is the* division of $e_1$ by $e_2$.

*With* $e(x_1, \ldots, x_n)$ *we denote that the extended polynomial* $e$ *contains only variables from the set* $\{x_1, \ldots, x_n\}$*. For* $k \geq 0$ *we define* $x^k$*,* the $k$th power of $x$*, by* $x^0 = 1$ *and* $x^k = x \cdot x^{k-1}$ *for* $k > 0$*. An extended polynomial of the form* $\sum_{i=0}^{k} e_i \cdot x^i$ *with* $x \in$ *Var and* $e_0, \ldots, e_k$ *extended polynomials not containing* $x$*, is called an* extended polynomial in $x$*. We call* $e_i$ *the* $i$th coefficient of $x$*.*

*A* (multivariate) polynomial $p$ *is an extended polynomial not containing any roots or divisions. The set* $\mathbb{R}[x_1, \ldots, x_n]$ *consists of all multivariate polynomials with variables from* $\{x_1, \ldots, x_k\}$. Polynomials in $x$ *and their coefficients are defined analogously to extended polynomials.*

Note that each extended polynomial containing a variable $x$ can be transformed into an equivalent[2] extended polynomial in $x$, that means, into a formula of the form $\sum_{i=0}^{k} e_i \cdot x^i$.

[1] For simplicity we do not distinguish between syntax and semantics of constants.

[2] With equivalent we mean that for each assignment both extended polynomials evaluate to the same value.

We use the notation $\sqrt{}$ instead of $\sqrt[2]{}$. We introduce the operator $-$ as syntactic sugar, with $(e_1 - e_2)$ defined as $(e_1 + (-1 \cdot e_2))$.

**Example 2.1.1**
*Let $p$ be the polynomial $(x + y) \cdot (x + 2 \cdot z) \in \mathbb{R}[x,y,z]$. The representation is not unique. The polynomial $p$ can be seen as a polynomial in $x$, $y$ or $z$. E.g., the polynomial $\hat{p} = x^2 + (y + 2 \cdot z) \cdot x + 2 \cdot y \cdot z$ in $x$ is equivalent to $p$. The* coefficients *of $x$ in $\hat{p}$ are $1$, $y + 2 \cdot z$, and $2 \cdot y \cdot z \in \mathbb{R}[y,z]$.*

**Definition 2.1.2 (Extended constraints, constraints)**
*An* extended (theory) constraint *$e \sim 0$ compares the extended polynomial $e$ to $0$, where $\sim \in \{=, \neq, <, >, \leq, \geq\}$. If $e$ is a polynomial, we call $e \sim 0$ a* (theory) constraint.

Note that $e_1 \sim e_2$ can be transformed to $e_1 + (-1) \cdot e_2 \sim 0$.

**Definition 2.1.3 (Syntax of extended-real-algebraic and real-algebraic formulas)**
Extended real-algebraic formulas *are built according to the following abstract grammar:*

$$\varphi \quad ::= \quad (e \sim 0) \qquad | \qquad (\neg\varphi) \qquad | \qquad (\varphi \wedge \varphi) \qquad | \qquad (\exists x\ \varphi)$$

*with $x \in$ Var and $e \sim 0$ an extended constraint.*

*We introduce the logical operators $\vee$, $\rightarrow$, ... with the expected meaning as syntactic sugar, and define $(\forall x\varphi)$ by $(\neg(\exists x(\neg\varphi)))$. We call "$\exists$" and "$\forall$"* existential *resp.* universal quantifiers. *We call $\varphi$ in $\exists x\varphi$ the* scope of *$x$ and occurrences of $x$ in $\varphi$* bound. *We call non-bound occurrences of a variable* free. *We exclude nested quantification of the same variable. Furthermore, we assume that in formulas variables do not have both bound and free occurrences.*

*The bound variables of the formula $\varphi$ form the set Bound$(\varphi)$, and the other variables occuring in the constraints are the* free variables *forming the set Free$(\varphi)$.*

*A constraint or its negation is called a* (positive resp. negative) literal. *A disjunction of literals is called a* clause. *A formula is in* conjunctive normal form (CNF) *if it is a conjunction of clauses. Formulas of the form*

$$(Q_1 x_1(\ldots(Q_n x_n \varphi)\ldots))$$

*with $\varphi$ being quantifier free, are said to be in* prenex normal form.

For simplicity we do not allow Boolean variables in real-algebraic formulas, but the logic could be easily extended to support their inclusion.

The virtual substitution extends the following standard substitution.

**Definition 2.1.4 (Substitution)**
*The* substitution *of a variable $x \in$ Var by an extended polynomial $e$ in an extended*

*real-algebraic formula $\varphi$, written $\varphi[e/x]$, is defined as follows:*

$$
\begin{array}{lcll}
x[e/x] & = & e & \\
y[e/x] & = & y & , \ y \neq x \\
c[e/x] & = & c & \\
(e_1 + e_2)[e/x] & = & (e_1[e/x] \ + \ e_2[e/x]) & \\
(e_1 \cdot e_2)[e/x] & = & (e_1[e/x] \ \cdot \ e_2[e/x]) & \\
(\sqrt[i]{e_1})[e/x] & = & (\sqrt[i]{e_1[e/x]}) & \\
(e_1/e_2)[e/x] & = & (e_1[e/x] \ / \ e_2[e/x]) & \\
(e_1 \sim 0)[e/x] & = & (e_1[e/x] \sim 0) & \\
(\neg\varphi_1)[e/x] & = & (\neg\varphi_1[e/x]) & \\
(\varphi_1 \ \wedge \ \varphi_2)[e/x] & = & (\varphi_1[e/x] \ \wedge \ \varphi_2[e/x]) & \\
(\exists y \ \varphi_1)[e/x] & = & (\exists y \ \varphi_1[e/x]) & , \ y \neq x \\
(\exists x \ \varphi_1)[e/x] & = & (\exists x \ \varphi_1) &
\end{array}
$$

*with $x, y \in$ Var, $c \in \mathbb{Q}$, $i \in \mathbb{N}$ with $i \geq 2$, $e, e_1, e_2$ extended polynomials, $e_1 \sim 0$ is an extended constraint, and $\varphi_1, \varphi_2$ extended real-algebraic formulas.*

In the following we sometimes omit parentheses, when it does not cause confusion, and use the standard arithmetic and logical binding order instead.

## 2.2. Semantics of extended real-algebraic constraints and formulas

The semantics of extended real-algebraic constraints and formulas is given by an evaluation function in the context of an assignment.

**Definition 2.2.1 (Assignment)**
*A function $\alpha : \ R \ \to \ \mathbb{R}$ with $R \subseteq$ Var is called a* (full) assignment *if $R =$ Var and a* partial assignment *otherwise. An assignment $\alpha' : \ R' \ \to \ \mathbb{R}$ with $R \subset R'$ is called an* extension of $\alpha$.

*We define $\alpha[x \mapsto v] : \ (R \cup \{x\}) \ \to \ \mathbb{R}$ with $v \in \mathbb{R}$ and $x \in$ Var by*

$$\alpha[x \mapsto v](y) \ = \ \begin{cases} \alpha(y) & \text{, if } x \neq y \\ v & \text{, otherwise.} \end{cases}$$

**Definition 2.2.2 (Semantics of extended real-algebraic constraints and formulas)**
*We define the semantics of real-algebraic formulas and extended polynomials in the context of an assignment $\alpha : \ Var \ \to \ \mathbb{R}$ as follows:*

$$
\begin{array}{lcll}
\alpha[\![c]\!] & = & c & \\
\alpha[\![x]\!] & = & \alpha(x) & \\
\alpha[\![e + \hat{e}]\!] & = & \alpha[\![e]\!] + \alpha[\![\hat{e}]\!] & \\
\alpha[\![e \cdot \hat{e}]\!] & = & \alpha[\![e]\!] \cdot \alpha[\![\hat{e}]\!] & \\
\alpha[\![\sqrt[i]{e}]\!] & = & \sqrt[i]{\alpha[\![e]\!]} & \text{, if } i \text{ is odd or } \alpha[\![e]\!] \geq 0 \\
\alpha[\![e/\hat{e}]\!] & = & \alpha[\![e]\!]/\alpha[\![\hat{e}]\!] & \text{, if } \alpha[\![\hat{e}]\!] \neq 0
\end{array}
$$

$$
\begin{array}{lcll}
\alpha[\![e \sim 0]\!] & = & \mathrm{True} & \text{, if } \alpha[\![e]\!] \sim 0 \\
\alpha[\![\neg\varphi]\!] & = & \mathrm{True} & \text{, if } \alpha[\![\varphi]\!] = \mathrm{False} \\
\alpha[\![\varphi \wedge \hat{\varphi}]\!] & = & \mathrm{True} & \text{, if } \alpha[\![\varphi]\!] = \mathrm{True} \text{ and } \alpha[\![\hat{\varphi}]\!] = \mathrm{True} \\
\alpha[\![\exists x\varphi]\!] & = & \mathrm{True} & \text{, if exists } v \in \mathbb{R} \text{ such that } \alpha[x \mapsto v][\![\varphi]\!] = \mathrm{True}
\end{array}
$$

*where $c \in \mathbb{Q}$, $x \in$ Var, $i \in \mathbb{N}$ with $i \geq 2$, $e, \hat{e}$ extended polynomials, $e \sim 0$ is an extended constraint and $\varphi, \hat{\varphi}$ extended real-algebraic formulas.*

*For an extended real-algebraic formula $\varphi$ we also write $\alpha \vDash \varphi$ instead of $\alpha[\![\varphi]\!] = \mathrm{True}$; we call $\alpha$ a* satisfying assignment *of $\varphi$ and $\varphi$* satisfiable*. If no satisfying assignment of $\varphi$ exists, it is called* unsatisfiable*.*

The arithmetic operations $+$, $\cdot$, $\sqrt[i]{}$, and $/$ in the above definition are the expected plus, times, $i$th root, and division, which we do not axiomatize here. The same holds for the relations $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$, such that a constraint comparing two real numbers is mapped to $\mathrm{True}$ or $\mathrm{False}$ as expected.

**Example 2.2.1**
*Consider the real-algebraic formulas*

$$
\begin{array}{lcl}
\varphi_1 & = & \exists x \exists y((x - 1 > 0 \ \vee \ x^2 - y \geq 0) \ \wedge \ (x^2 - 1 = 0 \ \vee \ y^2 + x < 0)) \\
\varphi_2 & = & \exists x \exists y(x^2 - y < 0 \ \wedge \ y < 0)
\end{array}
$$

*Let $\alpha$ be an assignment with $\alpha(x) = -1$ and $\alpha(y) = 1$, then every extension of $\alpha$ is a satisfying assignment of $\varphi_1$. The formula $\varphi_2$ is unsatisfiable.*

Now we are able to formalize the satisfiability problem for real-algebraic formulas.

**Definition 2.2.3 (Satisfiability problem for real-algebraic formulas)**
*The* satisfiability problem *for a real-algebraic formula $\varphi$ is the problem of deciding whether there exists a satisfiable assignment for $\varphi$.*

## 2.3. Graph theory

For the construction of an adequate data structure for the algorithm, we make use of the following notions.

**Definition 2.3.1 (Digraph)**
*A* digraph *or* directed graph *is a tuple of two sets $(V,E)$, where $V$ is a set of* nodes *and $E \subseteq V \times V$ is a set of directed* edges*.*

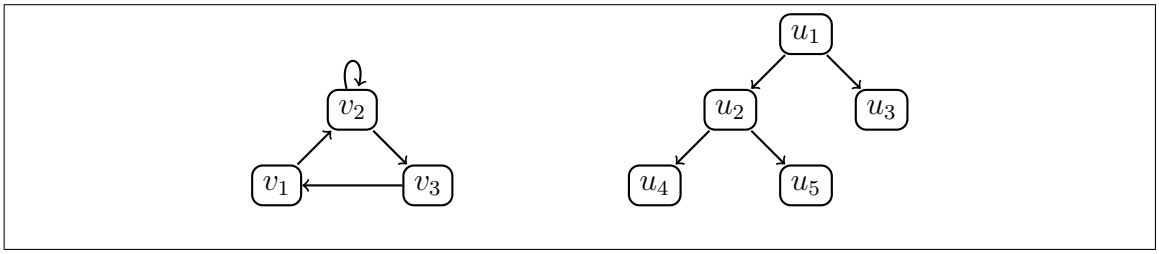


**Figure 2.1.:** The digraph $G := (\{v_1,v_2,v_3\},\{(v_1,v_2),(v_2,v_2),(v_2,v_3),(v_3,v_1)\})$ on the left; The directed tree $T := (\{u_1,u_2,u_3,u_4,u_5\}, \{(u_1,u_2),(u_1,u_3),(u_2,u_4),(u_2,u_5)\})$ on the right.

An example for a directed graph is shown in Figure 2.1.

**Definition 2.3.2 (Path)**
*A* path $p$ *in a digraph* $G = (V, E)$ *is a sequence*

$$p := v_0 v_1 \dots v_{n-1} v_n$$

*with the* length of the path $n \geq 0$ *such that*

1. $v_i \in V \quad (0 \leq i \leq n)$
2. $(v_i,v_{i+1}) \in E \quad (0 \leq i < n)$.

In the the right graph of Figure 2.1 the sequence $u_1u_2u_4$ is a path.

**Definition 2.3.3 (Directed tree)**
*A* directed tree $T = (V,E)$ *is a digraph such that the following conditions hold:*

1. *There exists exactly one* root $r \in V$ *with* $(v, r) \notin E$ *for all* $v \in V$.
2. *For all nodes* $v \in V\backslash\{r\}$ *there exists exactly one edge* $(u, v) \in E$ *and for this edge* $u \neq v$ *holds.*

*If* $(u,v) \in E$ *we call* $u$ *the* father *of* $v$ *and* $v$ *a* child *of* $u$. *Nodes without children are called* leaves. *We say that* $v$ *is* reachable *from* $u$, *if there is a path from* $u$ *to* $v$. *Let* $v \in V$ *and* $V_v \subseteq V$ *be the set containing all nodes reachable from* $v$ *in* $T$. *Then* $(V_v, E \cap (V_v \times V_v))$ *is called a* subtree of $T$ with root $v$.

# 3. SMT-solving

The propositional satisfiability problem, where the variables range over the values $\mathrm{True}$ and $\mathrm{False}$, is NP-complete, but SAT-solvers are quite efficient in practice due to a vast progress in SAT-solving during the recent decade. One of the main achievements in the field of SAT-solving is the DPLL-algorithm [MMZ+01]. Furthermore, its DPLL($T$) extension is capable of performing consistency checks for propositional logic extended with theories $T$. Thus, DPLL-based decision procedures can be applied to logics richer than propositional logic, by abstracting all non-propositional atomic formulas by propositional variables. This approach is called *SAT-modulo-theories* (*SMT*) solving.

## 3.1. Preprocessing of the input formula

The DPLL-based SMT-solving gets a real-algebraic formula as input.

1. We convert it into an equivalent formula in negation normal form (NNF), which contains negations only in literals.

2. We resolve the negations of the negative literals by changing the relation symbol according to Figure 3.1.

3. We achieve an *equisatisfiable*[1] formula in CNF using Tseitin's encoding [Tse83]. The complexity of the transformation is linear in time and space in the number of Boolean operators. However, it comes at the price of a new Boolean variable for each Boolean operator.

4. We create a Boolean skeleton of the input formula, replacing all theory constraints in the input formula by fresh Boolean variables.

The resulting formula is passed to the SAT-solver.

## 3.2. SAT-solving

The SAT-solver searches for a satisfying assignment for the Boolean skeleton. A naive approach could check all assignments whether they satisfy the formula. State-of-the-art SAT-solvers use more sophisticated approaches. In the following we describe a method based on propagation and conflict-driven non-chronological backtracking. In this work we

[1] Two formulas are equisatisfiable iff either both are satisfiable or both are unsatisfiable.

**Table 3.1.** Inverting a relation symbol.

| | | |
|---|---|---|
| $=$ | $\mapsto$ | $\neq$ |
| $\neq$ | $\mapsto$ | $=$ |
| $<$ | $\mapsto$ | $\geq$ |
| $>$ | $\mapsto$ | $\leq$ |
| $\leq$ | $\mapsto$ | $>$ |
| $\geq$ | $\mapsto$ | $<$ |

do not explain the heuristics used for the variable assignments; we use VSIDS (Variable State Independent, Decaying Sum), see [MMZ+01].

The input formula of the SAT-solver is in conjunctive normal form, that means it is a conjunction of clauses. Each of these clauses must be satisfied in order to satisfy the formula. To satisfy a clause, at least one literal in it must be $\mathrm{True}$, since a clause is a disjunction of literals. We classify the clauses with respect to a partial assignment as follows:

- If all literals of a clause are $\mathrm{False}$, it is called *conflicting*.
- If a clause has only one literal in it, which has not yet been assigned and all other literals are false, we call it *unit*.
- If at least one literal is $\mathrm{True}$, the clause is called *satisfied*.
- Otherwise, i.e., the clause is neither conflicting nor satisfied and it contains at least two unassigned literals, it is *unresolved*.

In the remaining of this section, we explain DPLL-based SAT-solving considering the pseudo code of Algorithm 1. The algorithm executes a main loop using the following submethods:

**bool CreateClauses(CNF $\varphi$)** Create a database of clauses for the Boolean CNF formula $\varphi$. Clauses with a single literal are not added, but those literals are assigned to $\mathrm{True}$. If an unsatisfied clause is found, return $\mathrm{False}$; Otherwise return $\mathrm{True}$.

**bool Decide()** If existing, choose an unassigned variable, assign it to a truth value, and return $\mathrm{True}$. This step is called a *decision*. The $i$th decision and the assignments made by the subsequent propagation (see below) build the $i$th *decision level*. If no more unassigned variable exists, return $\mathrm{False}$. The choice of the decision variable and its value is managed by a heuristics. Some example heuristics can be found in [Boo].

**Clause BCP()** This submethod implements *Boolean constraint propagation*: As long as there is a unit clause, assign its unassigned literal to $\mathrm{True}$. If an unsatisfied clause is found, return it. We call this a *conflict*. Otherwise if there are no more unit clauses, return NULL.

**(Clause, int) AnalyzeConflict(Clause $c$)** This submethod implements *conflict-driven non-chronological backtracking* [MMZ+01]: It generates a *conflict clause*, being an unsatisfied clause implying the unsatisfaction of $c$. This clause is *asserting*, that means it has exactly one literal assigned at the current decision level. After backtracking (see below) this clause becomes unit. The method returns the conflict clause and the second highest decision level $dl$ of its literals.

**void Backtrack(Int $dl$)** Erase all decision levels higher than $dl$. Set the current decision level to $dl$.

**void AddClause(Clause $c$)** Add the unit clause $c$ to the database of clauses and assign $\mathrm{True}$ to its unassigned literal.

**int DecisionLevel()** Return the current decision level.

**Algorithm 1** DPLL-SAT (1)

```
bool DPLL(CNF φ)
begin
    if CreateClauses(φ)=False then                          (1)
        return False;                                       (2)
    end if                                                  (3)
    while True do                                           (4)
        if BCP()≠NULL then                                  (5)
            if DecisionLevel()=0 then                       (6)
                return False;                               (7)
            else                                            (8)
                (conf, btLevel) := AnalyzeConflict();       (9)
                Backtrack(btLevel);                         (10)
                AddClause(conf);                            (11)
            end if                                          (12)
        else                                                (13)
            if Decide()=False then                          (14)
                return True;                                (15)
            end if                                          (16)
        end if                                              (17)
    end while                                               (18)
end
```

After creating the database for the formula, we enter in a loop performing a Boolean constraint propagation and, depending on the result of the propagation, either making a decision or conflict resolution and backtracking. The algorithm terminates either if a satisfying assignment is found or we found a conflict at decision level $0$, which implies that the formula is unsatisfiable.

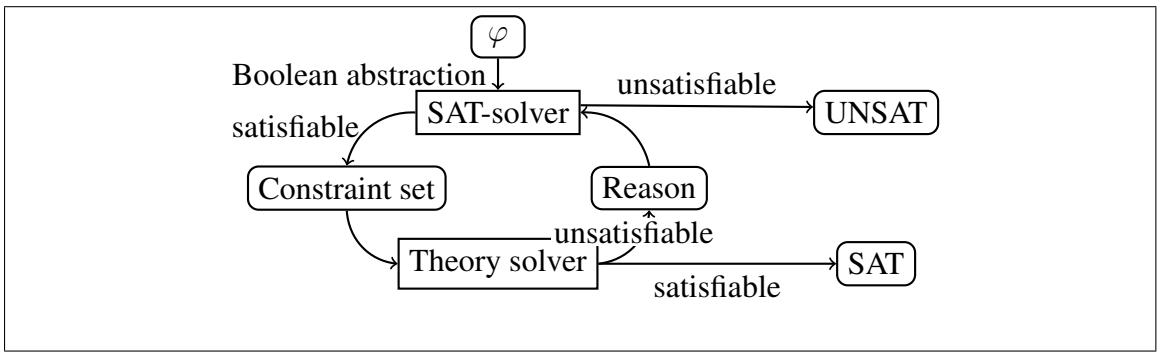


**Figure 3.1.:** The basic scheme of DPLL-based full lazy SMT-solving

## 3.3. Interaction between SAT-solving and theory solving

The SMT-solver uses the theory solver to check, whether the set of theory constraints, whose Boolean abstraction is assigned to $\mathrm{True}$, is consistent. Note that the preprocessing in Section 3.1 assures, that it is sufficient to check only those constraints for consistency, whose abstraction variables are assigned to $\mathrm{True}$. This relies on the fact that after preprocessing no constraint is (directly or indirectly) negated in the formula. Thus, if an assignment assigning $\mathrm{False}$ to a constraint's abstraction variable satisfies the Boolean skeleton, then also the assignment assigning $\mathrm{True}$ to that variable satisfies it.

A *full lazy* approach calls the theory solver each time, the SAT-solver has found a complete satisfying assignment. The SAT-solver we use permits a *less lazy* approach, which calls the theory solver each time the SAT-solver finishes a decision level.

Thus, after each completed decision level the theory solver receives the set of constraints whose abstraction variable got assigned $\mathrm{True}$ at that level. It checks, whether this set together with previously received constraints from earlier decision levels is consistent. Hence, for efficiency it is important that the theory solver is able to make use of previous consistency checks. This ability is called *incrementality*.

If the theory solver determines consistency, the assignment is either complete, which implies that the input formula of the SMT-solver is satisfiable or it is not complete and the SAT-solver performs another decision level. If the theory solver determines inconsistency, it should return an *infeasible subset* of the constraints checked, which is then used by the SAT-solver to refine the Boolean abstraction. The Boolean abstraction of the infeasible subset forms an unsatisfied clause, which is used for conflict resolution. Optimally, the infeasible subset is *minimal*, which means that taking away any constraint makes it consistent.

Thereafter, the SAT-solver retracts certain decision levels, which also provokes, that some constraints should be erased from the set of constraints the theory solver maintains. It is useful for the theory solver to have such a backtrack mechanism, which avoids deleting information, relevant for the future consistency checks of the remaining constraints.

Note that we strictly separate the satisfiability checks in the Boolean and in the theory domains, that means, we do not consider theory propagation embedded in the DPLL search

like, e.g., Yices does.

# 4. Virtual substitution

The virtual substitution method is a restricted but very efficient decision procedure for a subset of real algebra. In this thesis we adapt it to support incrementality, backtracking, and minimal infeasible subset generation (cf. Chapter 3). In this section we introduce virtual substitution in its original form.

## 4.1. The decision procedure

We are interested in checking satisfiability of pure-existentially quantified formulas in prenex normal form (PNF). The decision procedure based on virtual substitution produces a quantifier-free equivalent of a given input formula, by successively eliminating all bound variables starting with the innermost one. Below we explain how the innermost existentially bound variable is eliminated using virtual substitution.

Let $\exists y_1 \ldots \exists y_n \exists x \varphi$ be the input formula, where $\varphi$ is a quantifier-free Boolean combination of polynomial constraints as defined in Chapter 2. In this thesis we handle constraints, whose degree in $x$ is at most two ($y_1, \ldots, y_n$ may occur with higher degree). Thus we assume that all constraints in $\varphi$ are of the form $p \sim 0$, $\sim \in \{=, <, >, \leq, \geq, \neq\}$, where $p$ is a polynomial that is at most quadratic in $x$. The coefficients of $x$ are again polynomials, but do not contain $x$.

Considering the problem's domain, each constraint containing $x$ splits it into values which satisfy the constraint and values which do not. More precisely, the satisfying values can be merged to a finite number of intervals whose endpoints are elements of $\{\infty, -\infty\} \cup \mathbb{L}_x$, where $\mathbb{L}_x$ are the zeros of $p$ in $x$. Given a polynomial $p = ax^2 + bx + c$ and a constraint $p \sim 0$, $\sim \in \{=, <, >, \leq, \geq, \neq\}$, the finite endpoints of its satisfying intervals are the zeros of $p$:

| | | |
|---|---|---|
| Linear in $x$ : | $x_0 = -\frac{c}{b}$ | , if $a = 0 \ \wedge \ b \neq 0$ |
| Quadratic in $x$, first solution: | $x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a}$ | , if $a \neq 0 \ \wedge \ b^2 - 4ac \geq 0$ |
| Quadratic in $x$, second solution: | $x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$ | , if $a \neq 0 \ \wedge \ b^2 - 4ac > 0$ |

The conditions on the right are called *side conditions*. Note that in case $a = b = c = 0$ (constant in $x$) the solution interval for $x$ is $(-\infty, \infty)$, which does not have finite endpoints.

Let us consider the previously introduced polynomial $p$ and its zeros $x_0$, $x_1$ and $x_2$. The

following table shows all possible non-empty solution intervals for $p \sim 0$:

| constraints | | possible solution intervals ($0 \leq i, j \leq 2, i \neq j$) | | | | |
|---|---|---|---|---|---|---|
| $p = 0$ | | | $[x_i, x_i]$ | | | $(-\infty, \infty)$ |
| $p < 0$ | $p > 0$ | $(-\infty, x_i)$ | | $(x_i, x_j)$ | $(x_i, \infty)$ | $(-\infty, \infty)$ |
| $p \neq 0$ | | $(-\infty, x_i)$ | | | $(x_i, \infty)$ | $(-\infty, \infty)$ |
| $p \leq 0$ | $p \geq 0$ | $(-\infty, x_i]$ | $[x_i, x_i]$ | $[x_i, x_j]$ | $[x_i, \infty)$ | $(-\infty, \infty)$ |

**Example 4.1.1**
*Let* $c: \ y \cdot x^2 + z \cdot x \geq 0$ *with the finite endpoints* $x_0 = x_1 = 0$ *and* $x_2 = -\frac{z}{y}$. *The constraint* $c$ *has the following solution intervals:*

- *Constant case:* $(-\infty, \infty)$, *if* $y = z = 0$
- *Linear case:* $(-\infty, 0]$ *or* $[0, \infty)$, *if* $y = 0$ *and* $z \neq 0$
- *Quadratic case, first solution:* $(-\infty, 0]$ *or* $[0, \infty)$, *if* $y \neq 0$
- *Quadratic case, second solution:* $(-\infty, \ -\frac{z}{y}]$ *or* $[-\frac{z}{y}, \infty)$, *if* $y \neq 0$

Assume now that we have a set of constraints $\{c_1, \ldots, c_n\}$ each containing $x$. Each constraint $c_i$, $1 \leq i \leq n$, has a set of solution intervals $\{I_{i,1}, \ldots, I_{i,k_i}\}$ for $x$. If the constraints have a common solution for $x$, then for all $i \in \{1, \ldots, n\}$ there exists a $j_i \in \{1, \ldots, k_i\}$ with

$$I \ = \ ( \bigcap_{i \in \{1, \ldots, \ n\}} I_{i,j_i}) \ \neq \ \emptyset.$$

The intersection $I$ is an interval, whose endpoints are both endpoints of some of the intervals we intersect. If $I$ is left-closed, its left endpoint is in $I$; otherwise there exists an infinitesimal value we can add to the left endpoint, such that the result is an element of $I$. In both cases we found an element of $I$ being a solution for $x$. Candidates for the left endpoint of $I$ are the left endpoints of the possible solution intervals of the constraints. Considering the above table, those candidates are $-\infty$ and all finite endpoints $x_0$, $x_1$, $x_2$ for all constraints. When searching for a satisfying solution for $x$, it is sufficient to consider those candidates if they belong to a left-closed interval, or those candidates plus an infinitesimal if the corresponding interval is left-open.

In other words, we check if (1) one of the left endpoints of the left-closed solution intervals, or (2) one of the left endpoints plus an infinitesimal $\epsilon$ of the left-opened intervals or (3) a very small value, which we denote by $-\infty$, fulfills all constraints. We call those points belonging to (1), (2), or (3) *test candidates*. In [Wei88] you can find the corresponding theorem proving the soundness and correctness of the method.

Basically, the virtual substitution recursively eliminates all bound variables $x$ in $\varphi$ by (i) determining all test candidates for $x$ in all constraints in $\varphi$ containing $x$, and (ii) checking if one of these test candidates satisfies $\varphi$.

To check whether a test candidate $t$ for $x$ satisfies a constraint $\overline{p} \sim 0$ in $\varphi$, we substitute all occurrences of $x$ by $t$ in $\overline{p}$, yielding $\overline{p}[t/x] \ \sim \ 0$, and check the resulting constraint

under the test candidate's side conditions for consistency. Note that neither $\overline{p}[t/x] \sim 0$ nor the solution conditions refer to $x$, but they may contain other bound variables. Thus the consistency check may involve further quantifier eliminations.

Standard substitution could lead to terms not contained in real algebra, since the test candidates include $-\infty$, square roots, and infinitesimals $\epsilon$. Virtual substitution however, avoids these expressions in the resulting terms: it defines substitution rules yielding formulas of real algebra that are equivalent to the result of the standard substitution. However, these substitution rules may increase the degree of the polynomials.

## 4.2. Example

The virtual substitution method defines 18 substitution rules: There are six relation symbols and three possible types of test candidates corresponding to (1), (2), and (3) as described above. The according substitution rules can be found in Appendix A. In this section we show two examples of the rules to demonstrate the idea:

1. We first show the case for a test candidate being the left endpoint of a left-closed interval. So let $t$ be a test candidate for $x$ of type (1) and assume the constraint $\hat{p} = 0$ occurrs in $\varphi$. If we use standard substitution to replace $x$ by $t$ in $\hat{p} = 0$, the result can be transformed to the general form $\frac{r+s\cdot\sqrt{w}}{q} = 0$ (see Proof A.2.1), where $q$, $r$, $s$ and $w$ are polynomial terms of the real algebra.

   We distinguish between the cases of $s$ being 0 or not, i.e., if there is a square root in the term after substitution or not. In the case $s = 0$ the equation $\frac{r+s\cdot\sqrt{w}}{q} = 0$ simplifies to $\frac{r}{q} = 0$ and further to $r = 0$. In the case $s \neq 0$, the constraint $\frac{r+s\cdot\sqrt{w}}{q} = 0$ is satisfied iff $r + s \cdot \sqrt{w} = 0$, or equivalently, iff either both $r$ and $s$ equal 0, or they have different signs and $|r| = |s\sqrt{w}|$. Therefore the virtual substitution replaces the constraint $\hat{p} = 0$ by

$$(s = 0 \wedge r = 0) \vee (s \neq 0 \wedge r \cdot s \leq 0 \ \wedge \ r^2 - s^2 \cdot w = 0).$$

2. The second case we describe is the substitution for a test candidate of type (2) in an inequalitity $\hat{p} < 0$. The test candidate represents in this case the left endpoint $t$ of a left-open interval plus an infinitesimal value. The substitution of the test candidate for $x$ in $\hat{p} < 0$ is equivalent to the following:

$$\underbrace{\hat{p}[t/x] < 0}_{\text{Case 1}} \vee \underbrace{\hat{p}[t/x] = 0 \wedge \hat{p}'[t/x] < 0}_{\text{Case 2}} \vee \underbrace{\hat{p}[t/x] = 0 \wedge \hat{p}'[t/x] = 0 \wedge \hat{p}''[t/x] < 0}_{\text{Case 3}}$$

   where $\hat{p}'$ and $\hat{p}''$ are the first resp. second derivative of $\hat{p}$.

   Either $\hat{p}$ maps the endpoint to a negative value satisfying Case 1, where it is surrounded by negative values due to the density of $\mathbb{R}$ or it is zero and one of its derivatives in $x$ is negative satisfying Case 2 or 3, which implies that values of $\hat{p}$

in the right neighborhood of $t$ are negative. A visualization of the three cases for univariate polynomials is shown in Figure 2.

In the above formula, the substitutions $\hat{p}[t/x] = 0$ and $\hat{p}'[t/x] = 0$ are computed according to the first case above. The other substitutions $\hat{p}[t/x] < 0$, $\hat{p}'[t/x] < 0$, and $\hat{p}''[t/x] < 0$ are computed using the substitution rule for test candidates of type (1) and a strict inequality, not listed here.

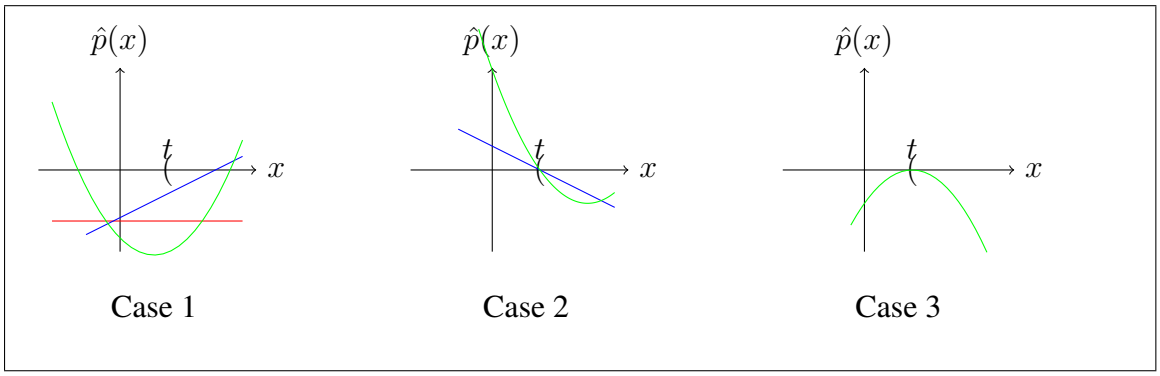


**Figure 4.1.:** Case distinction for the virtual substitution of a test candidate of type (2) into an inequaltity $\hat{p} < 0$.

Assume $T$ is the set of all possible test candidates for $x$. Given a test candidate $t \in T$ with side conditions $C_t$, the virtual substitution method applies the substitution rules to all constraints in the input formula $\varphi$ and conjugates the result with $C_t$. Considering all possible test candidates results in the formula

$$\exists y_1 \ldots \exists y_n \bigvee_{t \in T} (\varphi[t/x] \wedge C_t).$$

Note that test candidates of type (3) does not have side conditions. The virtual substitution method continues with the elimination of the next variable.

# 5. Incremental virtual substitution in SMT-solving

As discussed in Chapter 3, a theory solver should support incrementality in order to be suited for an efficient embedding into a less lazy SMT-solver. Note that theory solvers in the SMT-context only have conjunctions of theory constraints as input, instead of arbitrary combinations. If the theory solver already checked a set of theory constraints for consistency, it should be able to accept more constraints and to check whether the conjunction of the already added constraints and the new constraints is still consistent.

The original virtual substitution method does not provide these functionalities yet. Nevertheless, it can be embedded into an SMT-solver. Full lazy SMT-solving does not require incrementality, but is not very profitable compared to a less lazy approach with an incremental theory solver. We could also embed a non-incremental theory solver into a less lazy SMT-solver. However, in this case the theory solver has to re-do a lot of work. In this section we propose an incremental version of the virtual substitution method.

Assume that the original virtual substitution method checks the satisfiability of a formula and eliminates a variable $x$. The elimination yields a list of test candidates with corresponding side conditions. After the substitution step the result is a new formula being the disjunction of the substitution results for each test candidate of each constraint containing $x$ (for further details see Chapter 4).

If we want to support the belated addition of further constraints, possibly containing $x$, we must be able to belatedly substitute $x$ in the new constraints using the previously generated test candidates. Furthermore, we have to find the test candidates of the new constraints for $x$ and belatedly consider them for substitution. For this purpose we must be able to reconstruct the constraints we had before the substitution by the already generated test candidates in order to be able to substitute them by the test candidates the new constraints provide. We also need to store all determined test candidates with their corresponding side conditions to be able to apply them to the new constraints belatedly.

A naive approach would be to mimic the original virtual substitution method: we could store all the abovementioned information, apply all relevant previous substitutions to new constraints, and extend the formula with new disjunctive components using test candidates from the new constraint. However, this approach would lead to very large formulas, growing exponentially in the number of variables (see [Wei97]).

Furthermore we are interested in the satisfiability of the formula. The elimination of a variable, as we saw in Chapter 4, creates a disjunction of disjunctions of conjunctions. This formula is satisfied if one of these conjunctions is satisfied. This leads to the idea, that we do not need to consider the whole formula all of time, but could proceed with one of its conjunctions as long as it keeps being consistent and switch to another conjunction

otherwise.

The data model underlying the consistency check must fulfill certain requirements:

- It must keep the data to be stored to a minimum.
- It must support incrementality.
- It must support an informed search and not just a breadth-first search.

## 5.1. Data model

Remember that the virtual substitution starts with a formula and eliminates its variables successively using test candidate generations and substitutions. Both of these operations lead to branching on possible solutions:

- The generation of test candidates yields substitutions, which are mappings of variables to test candidates, and their corresponding side conditions.
- The substitution itself branches on possible substitution cases according to the virtual substitution method. An overview of all those cases can be extracted from Appendix A.

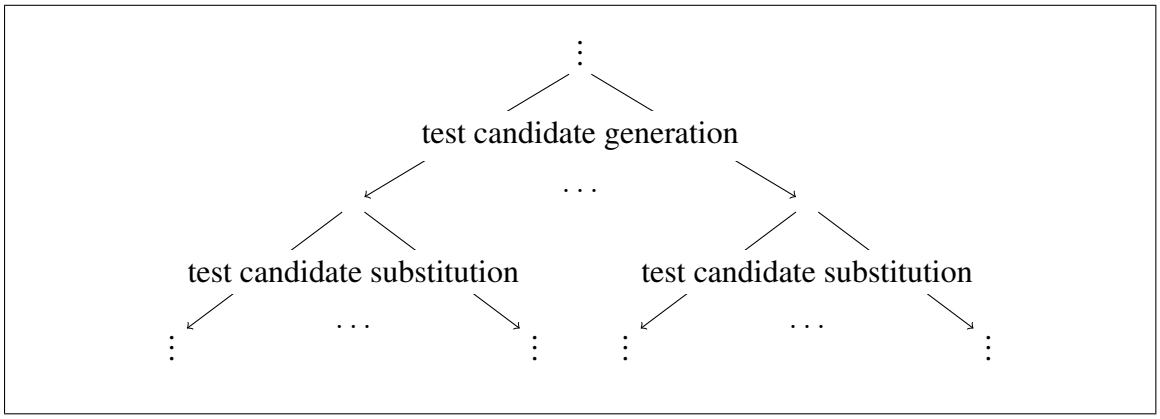


**Figure 5.1.:** The branching of the different cases of generating test candidates and substitutions considered as a tree.

Figure 5.1 gives an abstract illustration of this branching. As we want to be able to belatedly apply those operations to later arrived constraints, we must remember not only the current result but also the history of operations executed. Therefore the current solver state is stored in a tree as defined in Definition 2.3.3. Its root should consist of the set of constraints, which the theory solver has to check. An intermediate node results from its antecessor by applying either the generation of test candidates or the substitution of a variable by a test candidate in a constraint. Vice versa the children of an intermediate node relate either to the different test candidates the node's constraints provide or to the different cases a substitution provokes due to the applicable rule of Appendix A. Hence,

the main information a node contains is the conjunction of constraints it considers, which we represent in the following by a set. By reason of the incrementality we also need to remember the set of substitutions we have already achieved. These two sets form almost all information we need to remember in each node.

**Definition 5.1.1 (Decision tuple)**
*A node of the tree in the implementation of the virtual substitution is called a* decision tuple*. It has the following structure:*

$$\begin{pmatrix} Conditions \\ Substitutions \end{pmatrix}_{index}$$

*where:*

- $Conditions$ *is a set of indexed constraints*

  $$(p \sim 0)_{\mathit{flag}}$$

  *where flag is defined as follows:*

  $$\mathit{flag} = \begin{cases} \text{True} & \text{, if the constraint was used} \\ & \text{to generate test candidates} \\ \text{False} & \text{, otherwise.} \end{cases}$$

- $Substitutions$ *is a set of substitutions of variables by test candidates:*

  $$[test\ candidate\ /\ variable].$$

  *The variables of all substitutions in this set must be pairwise different.*

- *The* $index$ *has the following definition:*

  $$index = \begin{cases} var & \text{, the variable for which the test candidates} \\ & \text{in this decision tuple are generated} \\ \bot & \text{, if no variable is determined to generate} \\ & \text{test candidates for.} \end{cases}$$

In order to save space, we write in a text $(Conditions,\ Substitutions)_{index}$ instead of

$$\begin{pmatrix} Conditions \\ Substitutions \end{pmatrix}_{index}.$$

In the following we explain the evaluation of a node, which constructs its successors in the tree. Thereby it becomes clear why the information a decision tuple contains is necessary.

## 5.2. Evaluation of a decision tuple

Starting with one decision tuple, which consists just of the conditions corresponding to the constraints the theory solver wants to check, we want to achieve a tree as shown in Figure 5.1. Hence, we have to extend the tree by new decision tuples, more precisely we add children to the already existing decision tuples in the tree. The evaluation we introduce in this section considers a decision tuple of the tree and generates children of it by either applying test candidate generation or a substitution by a test candidate. Thus it previously has to decide, which operation it applies. This depends on the constitution of the considered decision tuple and on the strategy we pursue.

In this section we explain two different strategies of decision tuple evaluation. The first approach tries to minimize the operations we do in one evaluation step. It either generates all test candidates for a condition, which still has not served as test candidate provider, or substitutes one variable in one condition by a test candidate we have already generated. The second approach differs in the substitution, since it substitutes all occurrences of a variable by a test candidate we just generated. The first approach seems to be more *target oriented*, such that we get more freedom of decision and a heuristics influences more specific. However, it unfortunately repeats substitutions within the tree unnecessarily. We give a further explanation after introducing the approach. Nevertheless, it needs more investigation in terms of benchmarks and a formal proof, which states that it is less efficient in worst or even average case than the second so called *entire substitution approach*.

As a simplification we said that test candidate generations as well as substitutions in a decision tuple lead to the generation of new children (see Figure 5.1). Actually, there is no need to consider a decision tuple, when we had achieved its children by a substitution. So the evaluation of a decision tuple by a substitution will not lead to new children, but split it into new decision tuple according to the different cases of the substitution rule.

Note that we always discard conditions, which are variable-free and consistent, e.g. $(0 = 0)_{\mathit{flag}}$, $(4 \geq 0)_{\mathit{flag}}, \ldots$. As a decision tuple considers a conjunction of conditions, the consistency of its conditions depends on the consistency of the conditions, which are not yet satisfied, i.e. the conditions which are not variable-free and consistent.

Keep in mind that we want to check a set of constraints for consistency. We generate a tree of decision tuples by the evaluation approaches we introduce next. Its leaves are those decision tuples, to which we can neither apply a test candidate generation nor a substitution by a test candidate. Considering that we discard conditions, which are variable-free and consistent and assuming that no condition arises with a degree higher than $2$, the leaves must be decision tuples, which just contain variable-free inconsistent conditions, e.g. $(1 = 0)_{\mathit{flag}}$, $(2 \leq 0)_{\mathit{flag}}, \ldots$. [1] They are variable-free, since otherwise we can apply a substitution or a test candidate generation. They are inconsistent, since otherwise they have already been discarded. Thus, there are two kinds of leaves: one, which contains an variable-free inconsistent condition, and one, which does not contain any condition. The latter one means that we have proven the consistency of the constraints the theory

[1] Note that applying a substitution can increase the degree of the remaining variables. Hence, a condition with a degree higher than 2 can be created within the tree of decision tuples, even if the constraints in the root have all a degree smaller equal to 2 in all variables.

solver checks: The path from the root to a leaf provides for each variable exactly one test candidate and the leaf represents one case of the cases created by substituting all occurrences of all variables by these test candidates. Having a leaf without conditions means that it has no inconsistent conditions. Thus the corresponding case just consists of consistent constraints and is therefore fulfilled. The mentioned test candidates on the path of the root to such a *satisfying leaf* represents a satisfying assignment of the variables in the constraints the theory solver checks for consistency. However, this assignment maps to terms containing $-\infty$ and $t + \epsilon$. The former term represents a sufficiently small value and the latter term represents $t$ plus an infinitesimal value. Checking for consistency in terms of decision tuples then means searching for a satisfying leaf and the two approaches we introduce next take this into account.

Possibly, we create conditions during the generation of the tree of decision tuples, which have a degree higher than $2$ in at least one variable. If we eliminate this variable next, we must also generate test candidates of this condition. Unfortunately, we cannot generate them in this case (see Chapter 4). In such a case we continue in both evaluation approaches in one of the following two ways:

1. We set the flag of these conditions to $\mathrm{True}$ and do not consider their test candidates. However, it distorts the result of the consistency check: If it determines consistency, the constraints we checked are consistent; Otherwise they are either consistent or inconsistent. It depends on the fact, that we do not consider all test candidates in this case.

2. We stop the consistency check, since we cannot guarantee the correctness of the result, if the theory solver determines inconsistency.

This is why we call the restriction to the degree of the input constraints of the theory solver *quadratic and beyond*. It is hard to give an exact restriction. Some ideas of how we can restrict the input constraints are shown in [Wei97]. Nevertheless, it is essential that we can also deal with these conditions, which will be part of our future work.

### 5.2.1. Target oriented approach

We consider the decision tuple $(C, S)_v$. First we check if $C$ contains any variable-free inconsistent condition. If so, we delete the considered decision tuple, since we cannot get rid of a variable-free inconsistent condition neither by test candidate generations nor by substitutions. Hence, this decision tuple cannot be part of a path from the root to a satisfying leaf. If $C$ has no variable-free inconsistent condition, we choose any condition $(c)_{\mathit{flag}} \in C$, with $\mathit{flag} = \mathrm{False}$. We change the flag to $\mathrm{True}$, which indicates that we have already considered this condition for further processing. We have the following cases for the evaluation of$(C, S)_v$:

1. Given: The constraint $c$ contains a variable, which is mapped by one of the substitutions in $S$ and $v = \bot$.

In this case we apply a substitution to $c$ as follows: We choose the oldest substitution $s \in S$, which maps a variable of $c$. The substitution rules of Appendix A give for $c$ and $s$ a disjunction of conjunctions of real algebraic constraints as a result.

This is the only case where we do not generate children, but split the decision tuple. More precisely, we generate brothers and delete the decision tuple afterwards. For each conjunction we generate a brother $D_i = (C_i, S)_\perp$ of $D$. The conditions $C_i$ are formed by the constraints occuring in the conjunction or in $C$ excluding $c$. They all get the flag $\mathrm{False}$. (Note that in this case all conditions in $C$ had the flag $\mathrm{False}$.)

2. Given: The constraint $c$ contains a variable, which is mapped by one of the substitutions in $S$ and $v \neq \perp$.

   In this case we choose $c$ for test candidate generation. However, first we have to apply a substitution to $c$. We delay the test candidate generation, that is done later in the generated subtree.

   We choose the oldest substitution $s \in S$, which maps a variable of $c$. The substitution rules of Appendix A give for $c$ and $s$ a disjunction of conjunctions of real algebraic constraints as a result. For each conjunction we generate a child $D_i = (C_i, S)_v$ of $D$. The conditions $C_i$ are formed by the constraints occuring in the conjunction or in $C$ excluding $c$. The conditions of $C_i$ that are inherited from $C$ have the flag $\mathrm{True}$ and the others the flag $\mathrm{False}$. We set the flags of the inherited conditions to $\mathrm{True}$ in order to avoid that they serve as a test candidate providers in both $D$ and $D_i$.

3. Given: No variable in $c$ is mapped by a substitution in $S$ and $v = \perp$.

   In this case there is no belated substitution we have to apply to $c$ and also no variable fixed for elimination. We choose a variable $v'$ occuring in $c$ to generate test candidates for. Thereafter the index of $D$ gets set to $v'$ and we generate the test candidates of $c$ for $v'$.

   A constraint provides a set of finite test candidates and $-\infty$. In Chapter 4 we define the set of finite test candidates combined with their side conditions. For each test candidate, whose side conditions do not contain a variable-free inconsistent constraint, we generate a child $D_i = (C_i, S_i)_\perp$ of $D$. The conditions $C_i$ are formed by the constraints occuring in the side conditions or $C$ and all have the flag $\mathrm{False}$. The substitutions $S_i$ are those occuring in $S$ extended by the subsitution, which maps the chosen variable $v'$ to the test candidate we currently consider. The index of the new decision tuple is $\perp$, since we must determine a new variable for elimination.

4. Given: No variable in $c$ is mapped by a substitution of $S$, $v \neq \perp$, and $v$ is a variable which $c$ contains.

   The case is similar to the previous one, but $c$ is not the first constraint we use to generate test candidates for $v$. The first time we generated test candidates for $v$ we have already added a child for the test candidate $-\infty$. So we do not need to consider $-\infty$ a second time. We just generate the children for the finite test candidates, whose side conditions do not contain a variable-free inconsistent constraint.

5. Given: No variable in $c$ is mapped by a substitution of $S$, $v \neq \bot$, and $v$ is a variable, which does not occur in $c$.

   In this case $c$ is chosen for test candidate generation for the variable $v$, which it does not contain. That means $c$ is constant in $v$, yielding the only test candidate $-\infty$. However, since $c$ is not the first constraint we generate test candidates for ($v \neq \bot$), $\infty$ has already been considered (see case 3).

   We have already set the flag of the condition we consider to $\mathrm{True}$, which assures, that we do not choose it once again to generate test candidates for $v$. We do not make any further computations. We can interpret this step as generating zero (finite) test candidates.

In Algorithm 2 you can see the evaluation of a decision tuple according to this approach in a more compact way. Note, that along the path from the root to a leaf each variable is mapped at most once by a substitution, since this evaluation favors substitutions over test candidate generations.

**Algorithm 2** The algorithm to evaluate a decision tuple using the target oriented approach.

**void evaluate**$_{TO}$(Decision_tuple $D = (C, S)_v \in V$)
**begin**
  **if** $C$ contains variable-free inconsistent condition **then** (1)
    $V := V \setminus \{D\}$; (2)
    **if** $D$ is not the root **then** (3)
      $E := E \setminus \{(D_f, D)\}$; // ($D_f$ father of $D$) (4)
    **end if** (5)
  **else** (6)
    choose a condition $(c)_{\mathit{flag}} \in C$ with $\mathit{flag} = \mathrm{False}$; (7)
    $\mathit{flag} := \mathrm{True}$; (8)
    **if** no $s \in S$ maps a variable in $c$ **then** (9)
      generateTestCandidates$_{TO}(D, (c)_{\mathit{flag}})$; // Alg. 4 (10)
    **else** (11)
      $s \in S$ is oldest substitution mapping a variable in $c$; (12)
      substitute$_{TO}(D, (c)_{\mathit{flag}}, s)$; // Alg. 3 (13)
    **end if** (14)
  **end if** (15)
**end**

Global variables: Tree_of_decision_tuples $T = (V, E)$;

This approach unfortunately could repeat substitutions unnecessarily. Let us consider the decision tuple $D := (C \cup \{(c)_{\mathit{flag}}\}, S \cup \{s\})_v$. According to the just given evaluation method it is possible that we generate test candidates for all conditions in $C$ although $s$ maps a variable, which occurs in $C$. Though this substitution must be applied in each child we just generated or even worse in their antecessors. The later we apply this substitution the more we have to apply it.

**Algorithm 3** The algorithm to apply one subsitution to one condition using the target oriented approach.

**void substitute**$_{TO}$(Decision_tuple $D = (C, S)_v \in V$, Condition $(c)_{\mathit{flag}} \in C$, Substitution $s \in S$)
**begin**

$C_1 \vee \ldots \vee C_k \overset{\text{App. } A}{:=}$ result of $s$ applied to $c$; (1)
**if** $v = \bot$ **then** (2)
  **for all** $1 \leq i \leq k$ **do** (3)
    $D_i := ((C \setminus \{(c)_{\mathit{flag}}\}) \cup \{(\hat{c})_{\text{False}} | \hat{c} \in C_i\}, S)_\bot$; (4)
    $T := (V := V \cup \{D_i\}, E := E \cup \{(D_f, D_i)\})$; // ($D_f$ father of $D$) (5)
  **end for** (6)
  $T := (V := V \setminus \{D\}, E := E \setminus \{(D_f, D)\})$; // ($D_f$ father of $D$) (7)
**else** (8)
  **for all** $1 \leq i \leq k$ **do** (9)
    $D_i := (\{(\hat{c})_{\text{True}} | (\hat{c})_{\hat{\mathit{flag}}} \in C \setminus \{(c)_{\mathit{flag}}\}\} \cup \{(\hat{c})_{\text{False}} | \hat{c} \in C_i\}, S)_v$; (10)
    $T := (V := V \cup \{D_i\}, E := E \cup \{(D, D_i)\})$; (11)
  **end for** (12)
**end if** (13)
**end**

Variable-free consistent conditions get discarded.
Global variables: Tree_of_decision_tuples $T = (V, E)$;

## 5.2.2. Entire substitution approach

We call the second approach the *entire substitution approach*, since it applies a new generated substitution to all conditions in the considered decision tuple before further elimination steps. Thus it has to combine the results of each single substitution, using the following construction.

**Definition 5.2.1 (Combination of a set of sets of sets)**
*Let $M = \{M_1, \ldots, M_n\}$ be a set of sets, such that $M_i = \{M_{i,1}, \ldots, M_{i,k_i}\}$ is again a set of sets for all $i \in \{1, \ldots, n\}$. The* combination of $M$ *is the set of sets:*

$$\{ \bigcup_{i \in \{1,\ldots,n\}} M_{i,j_i} | \, (j_1, \ldots, j_n) \in \{1, \ldots, k_1\} \times \ldots \times \{1, \ldots, k_n\} \}$$

This corresponds to the reformulation of a formula of the form

$$\bigwedge_i \bigvee_j \bigwedge_{k_j} c_{i,j,k_j}$$

to a formula

$$\bigvee_{j_1,\ldots,j_n} \bigwedge_i \bigwedge_{k_{j_i}} c_{i,j_i,k_{j_i}}$$

The evaluation of a decision tuple $D = (C, S)_v$ has two cases:

**Algorithm 4** The algorithm to generate all test candidates for a variable of a condition using the target oriented approach.

**void generateTestCandidates**$_{TO}$(Decision_tuple $D = (C, S)_v \in V$, Condition $(c)_{\mathit{flag}} \in C$)
**begin**
**if** $v = \bot$ **then** (1)
$v := v'$; // ($v'$ variable in $c$) (2)
$D_{inf} := (\{(\hat{c})_{\mathrm{False}} | (\hat{c})_{\hat{\mathit{flag}}} \in C\}, S \cup \{[-\infty/v]\})_\bot$; (3)
$T := (V := V \cup \{D_{inf}\}, E := E \cup \{(D, D_{inf})\})$; (4)
**end if** (5)
$\{(t_1,C_1), \ldots, (t_k,C_k)\} \overset{Chap.\ 4}{:=}$ test candidates with side conditions of $c$ for $v$; (6)
**for all** $1 \leq i \leq k$ **do** (7)
**if** $C_i$ has no variable-free inconsistent conditions **then** (8)
$D_i := (\{(\hat{c})_{\mathrm{False}} | (\hat{c})_{\hat{\mathit{flag}}} \in C\} \cup \{(\hat{c})_{\mathrm{False}} | \hat{c} \in C_i\}, S \cup \{[t_i/v]\})_\bot$; (9)
$T := (V := V \cup \{D_i\}, E := E \cup \{(D, D_i)\})$; (10)
**end if** (11)
**end for** (12)
**end**

Variable-free consistent conditions get discarded.
Global variables: Tree_of_decision_tuples $T = (V, E)$;

1. Given: $v = \bot$.

   In this case we apply all substitutions in $S$ to $C$. As we do this after each elimination, just the most recent substitution of $S$, stamming from the last elimination, can be applied.

   This substitution applied to a single constraint in $C$ leads to a disjunction of conjunctions, which do not contain the variable to substitute anymore. The results of applying this substitution to all constraints in $C$ are combined to one disjunction of conjunctions according to Definition 5.2.1. For each of these conjunctions we add a brother $D_i = (C_i, S)_{v_i}$ to $D$. The conditions $C_i$ are formed by the constraints occuring in the conjunction and have the flag $\mathrm{False}$. The index $v_i$ gets set to a variable occuring in $C_i$. Finally, we delete $D$. The pseudo code for this method is given by Algorithm 6.

2. Given: $v \neq \bot$

   In this case we generate test candidates. Note that, as $v \neq \bot$ was set in the previous case, currently there are no pending substitutions.

   Firstly, we check if for the test candidate $-\infty$ a child of $D$ has already been generated. If not, we generate a child $D_{inf} = (C_{inf}, S_{inf})_\bot$, such that its conditions $C_{inf}$ consist of the constraints occuring in $C$ and have the flag $\mathrm{False}$ and its substitutions $S_{inf}$ are those occuring in $S$ extended by the substitution $[-\infty/v]$. Otherwise, we choose a

condition $(c)_{\mathit{flag}} \in C$ with $\mathit{flag} = \mathrm{False}$ and set $\mathit{flag}$to $\mathrm{True}$. According to Chapter 4 we determine the set of finite test candidates for $c$ and $v$ together with their side conditions. For each finite test candidate $t$, whose side conditions do not contain a variable-free inconsistent constraint, we generate a child $D_i = (C_i, S_i)_\perp$, such that its conditions $C_i$ are the constraints occuring in the side conditions or in $C$. Their flags are all $\mathrm{False}$. The substitutions $S_i$ are those occuring in $S$ extended by the substitution $[t/v]$. The pseudo code for this method is given by Algorithm 7.

**Algorithm 5** The algorithm to evaluate a decision tuple using the entire substitution approach.

```
void evaluate_ES(Decision_tuple D = (C, S)_v ∈ V)
begin
    if C contains a variable-free inconsistent condition then                    (1)
        V := V \ {D};                                                            (2)
        if D is not the root of T then                                           (3)
            E := E \ {(D_f, D)};        // (D_f father of D)                     (4)
        end if                                                                   (5)
    else if v =⊥ then                                                            (6)
        substitute_ES(D);                                  // Alg. 6             (7)
    else                                                                         (8)
        generateTestCandidates_ES(D);                      // Alg. 7             (9)
    end if                                                                       (10)
end
```

Variable-free consistent conditions get discarded.
Global variables: Tree_of_decision_tuples $T = (V, E)$;

In Algorithm 5 you can see the evaluation of a decision tuple according to this approach in a more compact way. The data stored in a decision tuple $D$ can be reduced by the not recent substitutions, since applying a substitution entirely makes it superfluous to remember the substitution in the successors of $D$.

## 5.3. Checking consistency of a set of constraints with decision tuples

Given a set of constraints $\{c_1, \ldots, c_k\}$, the theory solver should check whether it is consistent or not. In terms of decision tuples we initialize a tree by generating the root node

$$D_{root} := \begin{pmatrix} \{(c_1)_{\mathrm{False}}, \ldots, (c_k)_{\mathrm{False}}\} \\ \emptyset \end{pmatrix}_\perp .$$

We have already brought up that we want to expand the tree until one of its nodes has an empty condition set. Starting with $D_{root}$ we could follow the strategy to generate all its

**Algorithm 6** The algorithm to apply the most recent substitution of a decision tuple entirely.

**void substitute**$_{ES}$(Decision_tuple $D = (C, S)_v \in V$)
**begin**
$s :=$ most recent substitution in $S$; (1)
**for all** $(c_i)_{\mathit{flag}_i} \in C = \{(c_1)_{\mathit{flag}_1}, \ldots, (c_n)_{\mathit{flag}_n}\}$ **do** (2)
$Disj_i = C_{i,1} \vee \ldots \vee C_{i,k_i} \overset{App.\ A}{:=}$ result of $s$ applied to $c_i$; (3)
$\tilde{Disj}_i := \{\{(c)_{\mathrm{False}} | \ c \in C_{i,j}\} | 1 \leq j \leq k_i\}$; (4)
**end for** (5)
$\{C_1, \ldots, C_k\} \overset{Def.\ 5.2.1}{:=}$ combination of $\{\tilde{Disj}_1, \ldots, \tilde{Disj}_n\}$; (6)
**for all** $1 \leq i \leq k$ **do** (7)
$v_i :=$ any variable occuring in $C_i$; (8)
$D_i := (C_i, \ S)_{v_i}$; (9)
$T \ := \ (V := V \cup \{D_i\}, E := E \cup \{(D_f, D_i)\})$; // ($D_f$ father of $D$) (10)
**end for** (11)
$T := (V := V \setminus \{D\}, E := E \setminus \{(D_f, D)\}$; // ($D_f$ father of $D$) (12)
**end**

Variable-free consistent conditions get discarded.
Global variables: Tree_of_decision_tuples $T = (V, E)$;

children and then to proceed with each of its children in the same way. This would be a breadth-first search and similar to virtual substitution in its raw form, where in one step for one variable all test candidates are generated and all occurrences of the variable get substituted by each test candidate leading to a large disjunction. A structure, which embeds decision tuples, has a crucial advantage: there is no need to apply breadth-first search. The information a decision tuple stores allows us to have the arbitrary choice, which decision tuple we evaluate next.

Let us say we have a mechanism, which gives us the next decision tuple $D$ to evaluate. The consistency check progresses the decision tuple by distinguishing between three forms:

1. Given: The set of conditions in $D$ is empty.

   Procedure: It implies, that the set of constraints, which the theory solver received so far, is consistent. Hence the algorithm returns $\mathrm{True}$.

2. Given: The set of conditions in $D$ is not empty, but all conditions have the flag $\mathrm{True}$.

   Procedure: It implies, that all conditions are already entirely substituted according to the set of substitutions and that all test candidates in this decision tuple were already generated. Thus, we cannot generate further children for $D$ and there is no need to consider it anymore. Mark the decision tuple in order to avoid that the mechanism to choose the next decision tuple will choose this decision tuple again. Note that the two evaluation approaches both assure, that we first apply all substitutions to a condition, before we use it for further test candidate generations.

**Algorithm 7** The algorithm to generate all test candidates for a variable of a condition using the entire substitution approach.

**void generateTestCandidates**$_{ES}$(Decision_tuple $D = (C, S)_v \in V$)
**begin**
choose a condition $(c)_{\mathit{flag}} \in C$ with $\mathit{flag} = \mathrm{False}$; (1)
$\mathit{flag} := \mathrm{True}$; (2)
**if** test canidate $-\infty$ not yet considered in $D$ **then** (3)
$D_{inf} := (\{(\hat{c})_{\mathrm{False}} | (\hat{c})_{\hat{\mathit{flag}}} \in C\}, S \cup \{[-\infty/v]\})_{\perp}$; (4)
$T := (V := V \cup \{D_{inf}\}, E := E \cup \{(D, D_{inf})\})$; (5)
**end if** (6)
$\{(t_1,C_1), \ldots, (t_k,C_k)\} \overset{Chap.\ 4}{:=}$ test candidates with side conditions of $c$ for $v$; (7)
**for all** $1 \leq i \leq k$ **do** (8)
**if** $C_i$ contains no variable-free inconsistent conditions **then** (9)
$D_i := (\{(\hat{c})_{\mathrm{False}} | (\hat{c})_{\hat{\mathit{flag}}} \in C\} \cup \{(\hat{c})_{\mathrm{False}} | \hat{c} \in C_i\}, S \cup \{[t_i/v]\})_{\perp}$; (10)
$T := (V := V \cup \{D_i\}, E := E \cup \{(D, D_i)\})$; (11)
**end for** (12)
**end**

Variable-free consistent conditions get discarded.
Global variables: Tree_of_decision_tuples $T = (V, E)$;

3. Given: $D$ contains a condition $c$, which has the flag $\mathrm{False}$.

   Procedure: Depending on the evaluation approach we use, call either the method given by Algorithm 2 or Algorithm 5.

The algorithm repeats choosing a decision tuple and processing it according to the previous description until either it finds one without conditions or all decision tuples are marked. The termination of this approach depends on the finite number of the test candidates we generate and the fact that the substitution eliminates a variable of the initially finite number of variables in the given set of constraints. Furthermore we do not introduce new variables. A compact description of the just explained procedure is shown in Algorithm 9. Algorithm 8 is just temporarily necessary to explain the so far gained ideas and will be replaced later in order to make this approach incremental.

## 5.4. Choice of the next decision tuple to evaluate

The previously introduced algorithm used an unexplained mechanism to choose the next unmarked decision tuple to evaluate. Such a mechanism is implemented by a heuristics. An example-heuristics would be to take the most recently generated decision tuple next ($\sim$depth-first search) or vice versa the oldest decision tuple ($\sim$breadth-first search). We can achieve promising measures by analyzing the conditions and substitutions. In the following we list some properties, which could influence the heuristics:

**Algorithm 8** The algorithm to determine consistency of a set of constraints using decision tuples, where, depending on the evaluation approach, $X = TO$ resp. $X = ES$.

**bool is_consistent1**(Set_of_constraints $C$)
**begin**
$D_{root} := (\{(c)_{\mathrm{False}} | \ c \in C\}, \emptyset)_{\perp}$; (1)
$T := (\{D_{root}\}, \emptyset)$; (2)
**if** is_consistent$_X$() **then** (3)
**return** $\mathrm{True}$; (4)
**else** (5)
**return** $\mathrm{False}$; (6)
**end if** (7)
**end**

Variable-free consistent conditions get discarded.
Global variables: Tree_of_decision_tuples $T = (V, E)$;

- The number of conditions having the flag $\mathrm{False}$.
- The number of substitutions (already eliminated variables).
- The degrees of the conditions in the variable given by the index of the decision tuple. Here, we could also just consider the condition we are going to evaluate next.
- The relation symbol of the condition we are going to evaluate next.
- The number of variables occuring in all conditions or the conditions we are going to evaluate next.
- The number of occurrences of the variable to eliminate/substitute in the conditions.
- The types of the substitutions, resp. of the most recent one.

How good a heuristics is depends on how fast we find a decision tuple without conditions. Another important aspect is how much computional effort the heuristics causes. It obviously depends on the number of decision tuples in the tree, which could increase exponentially as you can see in [Wei97]. Our idea is to rank the decision tuples, which are still not marked. Each decision tuple maintains a value and a unique ID. Once distributed, the ID of a decision tuple will not change, but its value can change during the evaluation. Every time a decision tuple is generated it is inserted into the ranking according to its value combined with its ID. This assures that two decision tuples would never share a rank. When a decision tuple gets changed, we first erase it from the ranking and insert it afterwards according to the new value. The inserting operation of an element with a certain value-ID combination can be achieved in logarithmical time, since the ranking is sorted. The same holds for erasing. Let us say, that the best decision tuple according to a heuristics is the first element of the ranking. Finding the best element then needs just

**Algorithm 9** The algorithm to determine consistency of a tree of decision tuples, where, depending on the evaluation approach, $X = TO$ resp. $X = ES$.

```
bool is_consistent_X()
begin
    while exists an unmarked D = (C, S)_v ∈ V do                          (1)
        if D has a condition (c)_flag with flag = False then              (2)
            evaluate_X(D);                          // Alg. 2, Alg. 5     (3)
        else if C ≠ ∅ then                                                (4)
            mark D (to avoid choosing it again);                          (5)
        else                                                              (6)
            return True;                                                  (7)
        end if                                                            (8)
    end while                                                             (9)
    return False;                                                         (10)
end
```

Global variables: Tree_of_decision_tuples $T = (V, E)$;

constant time. Altogether, the heuristics produces an additional effort each time a decision tuple is generated, which is logarithmic regarding to the number of (unmarked) decision tuples, which is in worst case exponential in the number of bound variables in the given set of constraints (see [Wei97]). Thus, the heuristics extends the exponential worst case complexity by a linear factor if we consider the effort to create the value of a decision tuple as constant.

This part of the implementation still has a lot of potential to exploit. Verifying the quality of the heuristics should be done empirically and is part of our future work.

## 5.5. Add new constraints to the theory solver

Let us assume, that we have already received a set of constraints and that we have already applied Algorithm 9 to check its consistency. If it determines consistency for this set of constraints, we achieve a tree of decision tuples, where at least one of them has an empty set of conditions. Now, we want to add some constraints belatedly to the set of the already received constraints and check for consistency once again. A naive approach would be to delete all generated decision tuples of the tree and to apply Algorithm 9 to the set of constraints, including the recently added ones. We would discard all computations done so far and thus repeat already made computations. Our goal is to avoid this, which means that we want to retain previously made computations, if their results can be used again.

The results of already made computations are reflected in the decision tuples the previous consistency check has created. The idea is to extend their set of conditions by the conditions resulting from the constraints to add. It depends on the evaluation approach we use, how we add constraints.

### 5.5.1. Target oriented approach

For this approach it is easy to add new conditions. Indeed we just add constraints incrementally, by adding them to the condition set of each decision tuple. However, we must take care of the indices of the conditions we add. There are two cases we distinguish:

1. We consider the second case of the evaluation of a decision tuple according to Section 5.2. It corresponds to the case that the index of the chosen decision tuple represents a variable and that the selected condition contains a variable, which is mapped by one of the substitutions. The generated children, in which the inherited conditions are all indexed by $\mathrm{True}$ and the new conditions created by the substitution are indexed by $\mathrm{False}$, form the only exception. The new condition must be considered as an inherited one, so it has to be indexed by $\mathrm{True}$ as well. We identify those decision tuples, by checking whether the decision tuple and its father have the same variable as index. It implies, that it resulted from the explained evaluation case.

2. In all other cases we add a constraint to a decision tuple by adding its corresponding condition indexed by $\mathrm{False}$ to the condition set of the decision tuple.

In Algorithm 10 we show a method to add one constraint belatedly to a tree of decision tuples, which represents the result of the previously made consistency check. We store this tree in a global variable of the theory solver.

**Algorithm 10** The algorithm to add a constraint to the constraints in the theory solver using the target oriented approach.

**void add_constraint**$_{TO}$(Constraint $c$)
**begin**
  **for all** $D = (C, S)_v \in V$ **do** (1)
    **if** $D$ has father with same index **then** (2)
      $D \ := \ (C := C \cup \{(c)_{\mathrm{True}}\}, S)_v$; (3)
    **else** (4)
      $D \ := \ (C := C \cup \{(c)_{\mathrm{False}}\}, S)_v$; (5)
    **end if** (6)
  **end for** (7)
**end**

Variable-free consistent conditions get discarded.
Global variables: Tree_of_decision_tuples $T = (V, E)$;

### 5.5.2. Entire substitution approach

Decision tuples created by the evaluation of the entire substitution approach contain either exactly one applicable substitution or all substitutions were entirely executed. The problem, which occurs by adding new constraints to already created decision tuples, is that it possibly

destroys this property and the evaluation cannot be applied anymore, since new cases arise. We do not want to maintain just this property, but also assure, that no substitution gets applied to the same condition twice in order to keep it as efficient as possible. The children created by the generation of test candidates can get split as a result of the execution of the newly created substitution in it. Thus, there could be more than one child having the same substitutions. If we add new constraints as conditions to these decision tuples, the second property cannot be hold. This is why new constraints can only be added to a decision tuple, if we can assure, that no substitution can be applied to it. The root is the only decision tuple, for which this always holds, since it does not contain any substitutions. Hence, we add new constraints by adding them as conditions to the root. The other already generated decision tuples get marked, such that the mechanism to choose the next decision tuple to evaluate does not choose them. Algorithm 11 describes the pseudo code of adding a constraint belatedly.

**Algorithm 11** The algorithm to add a constraint to the constraints in the theory solver using the entire substitution approach.

**void add_constraint**$_{ES}$(Constraint $c$)
**begin**
add $(c)_{\text{False}}$ to the conditions of the root of $T$; (1)
mark the other decision tuples (to avoid choosing them again); (2)
**end**

Variable-free consistent conditions get discarded.
Global variables: Tree_of_decision_tuples $T = (V, E)$;

Nevertheless, we need to extend the evaluation method by another case, where the considered decision tuple contains recently added constraints. This implies, that the conditions must comprise another flag which indicates, if the condition relates to a recently added constraint or not.

In the following we renew the description of the second approach to evaluate a decision tuple $D := (C, S)_v$, which we have already described in Section 5.2.

1. Given: $v = \bot$.

   Procedure: In the same manner as before (see description of the evaluation approach using the entire substitution approach in Section 5.2).

2. Given: $v \neq \bot$ and $D$ contains no condition which relates to a recently added constraint.

   Procedure: In the same manner as in the case of $v \neq \bot$ (see description of the evaluation approach using the entire substitution approach in Section 5.2).

3. Given: $v \neq \bot$ and $D$ contains a condition which relates to a recently added constraint.

   Procedure: The decision tuple $D$ fulfills the required property, that it does not contain a condition, to which we can apply a substitution of $D$. We group the

children of $D$ according to their most recent substitutions as shown in Figure 5.2. We apply each of these substitutions $s_i$ to the recently added conditions $C_{recent}$. Applying one substitution to the constraint of one of the recently added conditions results in a disjunction of conjunctions of real algebraic constraints, which do not contain the variable to substitute anymore. We combine the disjunctions, which result from applying one substitution to all recently added conditions, according to Definition 5.2.1. Hence, we get for each substitution $s_i$ one disjunction of conjunctions of constraints, which we add as conditions with flag $\mathrm{False}$ to each child containing $s_i$. Adding a disjunction, as in our case, splits a decision tuple in the way shown by Figure 5.3. We assumed, that $D$ includes no condition to which a substitution in $D$ is applicable and we extended the children of $D$ by the results of applying the only new substitution in them to $C_{recent}$. So, these children also fulfill the property, that they do not contain conditions to which its substitutions can be applied. Finally, we mark the conditions $C_{recent}$ in $D$ as not recently added and the newly created conditions in the children of $D$ as recently added. Note that the children of $D$ have been marked before to avoid choosing them as next decision tuple to evaluate. Now they are not marked anymore. In this way, the recently added constraints spread out across the already achieved tree of decision tuples.

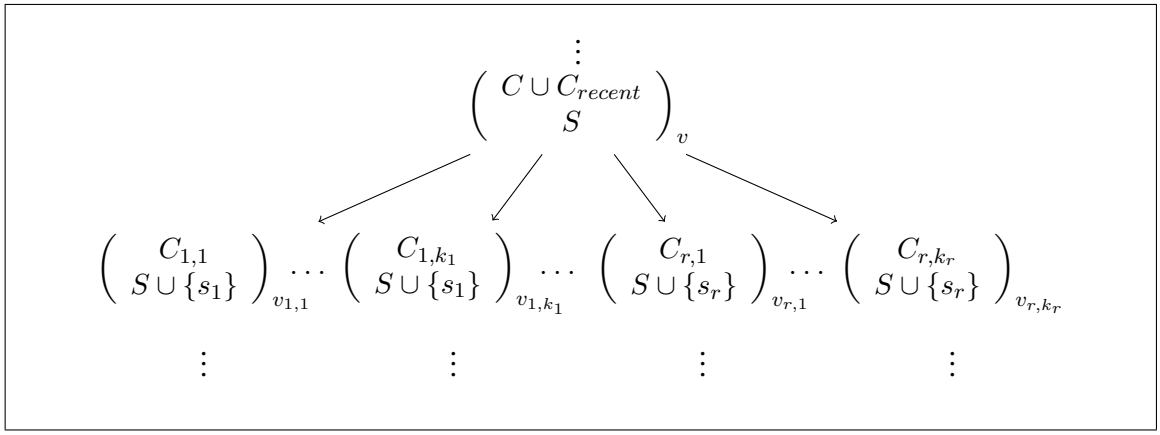


**Figure 5.2.:** ntire substitution approach: Grouping of the children of a decision tuple by their most recent substitution.

In Algorithm 12 we give an incremental version of the evalutation of a decision tuple according to the entire substitution approach. It extends Algorithm 5 by another case, where the considered decision tuple contains a recently added condition.

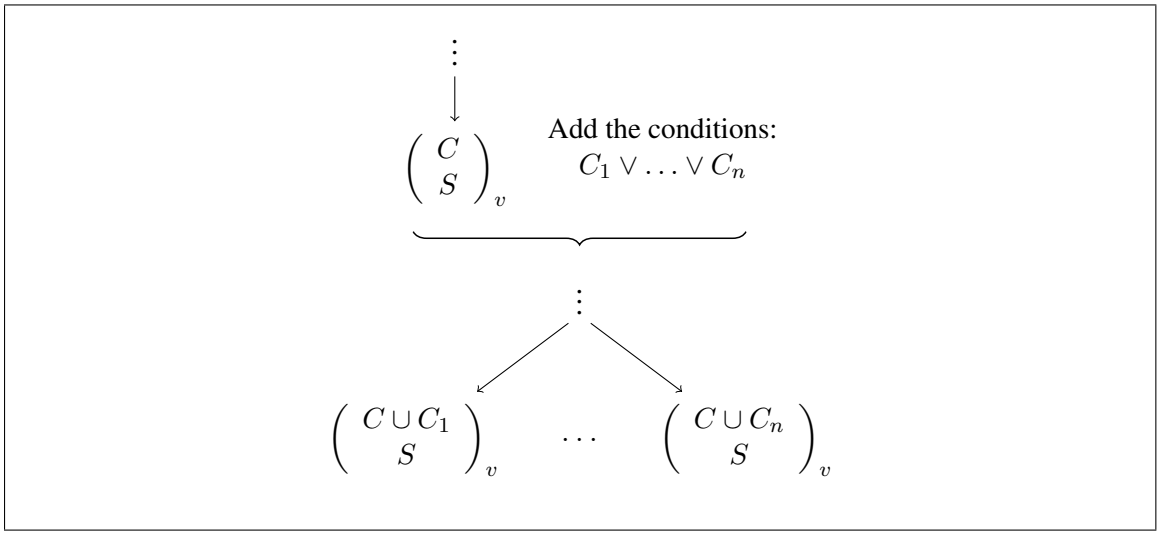


**Figure 5.3.:** Adding a disjunction of $n$ conjunctions of conditions to a decision tuple splits it into $n$ decision tuples.

**Algorithm 12** The algorithm to evaluate a decision tuple supporting incrementality using the entire substitution approach.

```
void evaluate_ES(Decision_tuple D = (C, S)_v)
begin
    if C contains a variable-free inconsistent condition then                    (1)
        V := V \ {D};                                                            (2)
        if D is not the root of T then                                           (3)
            E := E \ {(D_f, D)};        // (D_f father of D)                     (4)
        end if                                                                   (5)
    else if v = ⊥ then                                                           (6)
        substitute_ES(D);                                  // Alg. 6            (7)
    else                                                                         (8)
        if D contains no recently added condition then                           (9)
            generateTestCandidates_ES(D);                  // Alg. 7            (10)
        else                                                                     (11)
            substituteBelated_ES(D);                       // Alg. 13           (12)
        end if                                                                   (13)
    end if                                                                       (14)
end
```

Global variables: Tree_of_decision_tuples $T = (V, E)$;

**Algorithm 13** The algorithm to add all recently added constraints to the children after applying the most recent substitution entirely.

**void substituteBelated**$_{ES}$(Decision_tuple $D = (C, S)_v \in V$)
**begin**

(1) $C = C_{old} \cup C_{recent}$;
(2) $\{s_1, \ldots, s_r\} :=$ the different most recent substitutions in the children of $D$;
(3) **for all** $1 \leq i \leq r$ **do**
(4) $\{D_1 = (C_1, S \cup \{s_i\})_{v_1}, \ldots, D_{k_i} = (C_{k_i}, S \cup \{s_i\})_{v_{k_i}}\} :=$ all children
(5) of $D$ containing $s_i$;
(6) **for all** $(c)_{flag} \in C_{recent}$ **do**
(7) $Disj_c = Conj_1 \vee \ldots \vee Conj_{n_i} \overset{App.\ A}{:=}$ result of $s_i$ applied to $c$;
(8) $\tilde{Disj}_c := \{\{(\hat{c})_{\text{False}} | \ \hat{c} \in Conj_j\} | 1 \leq j \leq n_i\}$;
(9) **end for**
(10) $\{\hat{C}_1, \ldots, \hat{C}_{m_i}\} \overset{Def.\ 5.2.1}{:=}$ combininations of $\{\tilde{Disj}_c | c \in C_{recent}\}$;
(11) **for all** $1 \leq j \leq k_i$ **do**
(12) $T_{D_j} = (V_{D_j}, E_{D_j}) \overset{Def.\ 2.3.3}{:=}$ subtree of $T$ with root $D_j$;
(13) **for all** $1 \leq l \leq m_i$ **do**
(14) $D_{j,l} := (C_j \cup \hat{C}_l, S \cup \{s_i\})_{v_j}$;
(15) $(V_{j,l}, E_{j,l}) :=$ copy of $T_{D_j}$, where $D_j$ is replaced by $D_{j,l}$;
(16) $T := (V := V \cup V_{j,l}, E := E \cup E_{j,l} \cup \{(D, D_{j,l})\})$;
(17) mark conditions in $D_{j,l}$ of $\hat{C}_l$ as recent;
(18) **end for**
(19) $T := (V := V \setminus V_{D_j}, E := E \setminus (E_{D_j} \cup \{(D, D_j)\})$;
(20) **end for**
(21) **end for**
(22) mark conditions in $C_{recent}$ as not recent;

**end**

Variable-free consistent conditions get discarded.
Global variables: Tree_of_decision_tuples $T = (V, E)$;

## 5.6. Conclusion

So far, our theory solver provides two methods for the SMT-solver and maintains a tree of decision tuples hold in a global variable.

**add_constraint$_X$(constraint $c$)** Depending on the underlying evaluation approach we choose $X \in \{TO, ES\}$. The method adds a constraint to the set of constraints the theory solver already received. It uses the previously made computations, which are stored as a global variable in the theory solver forming a tree of decision tuples. The pseudo codes to add a constraint belatedly for both evaluation approaches are shown in Algorithm 10 resp. Algorithm 11.

**is_consistent$_X$()** Depending on the underlying evaluation approach we choose $X \in \{TO, ES\}$. The method checks the consistency of all constraints, which the theory solver has already received. The globally stored tree of decision tuples forms the initial set of decision tuples, from which we choose one according to Section 5.4 in order to evaluate it according to Section 5.2. It repeats this until it considers either a decision tuple, which has an empty set of conditions or there exists no more decision tuple to choose. In the former case the set of already received constraints is consistent, in the latter it is inconsistent. The pseudo code of this method is shown in Algorithm 9.

The following examples illustrate both methods according to the two evaluation approaches we have introduced.

## 5.7. Examples

### 5.7.1. Target oriented approach

We initiate the theory solver with the tree

$$T = \left(\left\{\begin{pmatrix}\emptyset\\ \emptyset\end{pmatrix}_{\perp}\right\}, \emptyset\right)$$

as global variable. We call *add_constraint$_{TO}$* with the argument $c_1 : x^2 - y \geq 0$ to hand over the first constraint to the theory solver. The method *add_constraint$_{TO}$* adds the condition $(c_1)_{\text{False}}$ to the condition sets of all decision tuples in $T$, i.e.

$$\begin{pmatrix}\emptyset\\ \emptyset\end{pmatrix}_{\perp} \text{ gets extended to } \begin{pmatrix}\{(x^2 - y \geq 0)_{\text{False}}\}\\ \emptyset\end{pmatrix}_{\perp}.$$

Afterwards, we perform a consistency check by calling the method *is_consistent$_{TO}$()*. Using a heuristics, an unmarked decision tuple of $T$ is chosen to evaluate next. The marked decision tuples are those, which we do not need to consider anymore, because they contain

no condition indexed by $\mathrm{False}$. In the following the gray drawn decision tuples are marked. There is just one option, namely the root node

$$R = \begin{pmatrix} \{(x^2 - y \geq 0)_{\mathrm{False}}\} \\ \emptyset \end{pmatrix}_{\perp}.$$

We call the method *evaluate$_{TO}$(*$R$, $(x^2 - y \geq 0)_{\mathrm{False}}$*)*, since it is the only condition $R$ has. The index of the decision tuple is $\perp$ and the chosen condition does not contain variables, which can be substituted as there is no substitution in this decision tuple yet. According to Algorithm 4 we generate test candidates, which do not have variable-free inconsistent side conditions, for a variable of $c_1$, let us say $x$, and get

1. $-\sqrt{y}$ with side conditions $1 \neq 0 \wedge 4y \geq 0$,
2. $\sqrt{y}$ with side conditions $1 \neq 0 \wedge 4y \geq 0$,
3. $-\infty$ with no side conditions.

For each test candidate we generate a child of $R$. Its conditions are inherited from $R$ and its substitutions are the substitutions of $R$ plus the substitution, which maps $x$ to the corresponding test candidate. The indices of the new decision tuples are $\perp$ and the index of $R$ is set to $x$. Finally, we set the index of $c_1$ in $R$ to $\mathrm{True}$, since it was used to generate test candidates. The result of this evaluation is shown in Figure 5.4. As a simplification we handle variable-free conditions in this example in the following way: If variable-free conditions, as e.g. $(1 \neq 0)_{\mathrm{False}}$, are consistent we discard them, otherwise, if they are inconsistent, we do not generate the decision tuple, which would contain it. It is indeed the way the algorithm handles these conditions, except in the case, if the variable-free inconsistent conditions are resulting by reason of a substitution. Here it generates the decision tuple, but deletes it after its evaluation.

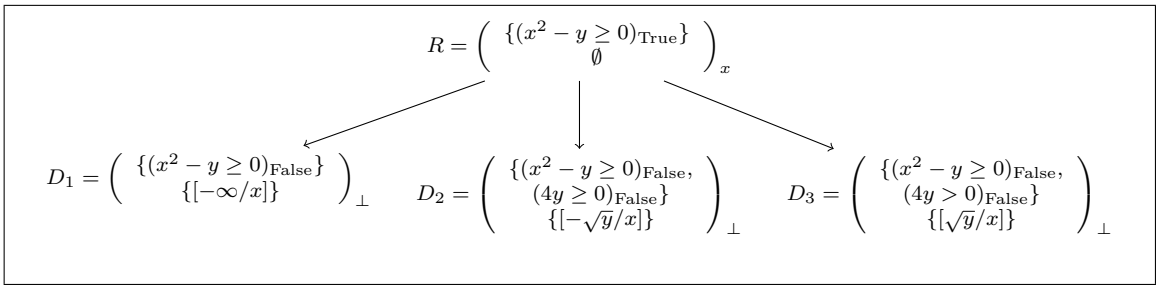


**Figure 5.4.:** Solver state after adding the constraint $c_1 : \; x^2 - y \geq 0$ and the first step of a consistency check.

In the next step we choose an unmarked decision tuple and therein a condition. If we choose e.g. $R$ again, we will observe, that it has no more conditions indexed by $\mathrm{False}$. Thus, we would mark $R$ in order to avoid choosing it again. Let us take the left-most just created decision tuple

$$D_1 = \begin{pmatrix} \{(x^2 - y \geq 0)_{\mathrm{False}}\} \\ \{[-\infty/x]\} \end{pmatrix}_{\perp}.$$

It contains just one condition including the variable $x$, which is mapped by the substitution $[-\infty/x]$. It refers to the first case of the method evaluate($D_1$, $(x^2 - y \geq 0)_{\text{False}}$). The substitution rules of Appendix A consider the coefficients of $x$ (the variable to substitute) in $x^2 - y \geq 0$ (the constraint of the condition to substitute in). Inserting the coefficients in the substitution rule results in:

$$\begin{array}{llllll} & ( & 1 > 0 & & & ) \\ \vee & ( & 1 = 0 & \wedge \quad 0 < 0 & & ) \\ \vee & ( & 1 = 0 & \wedge \quad 0 = 0 & \wedge \quad -y \geq 0 & ). \end{array}$$

The latter two cases are inconsistent, so we do not consider them (see above 5.7.1). The first case is consistent, thus we generate the brother $D_4$ of $D_1$. It has no conditions, since the only condition in $D_1$ got substituted and the resulting case we consider consists of just one variable-free consistent condition, which we discard. The substitutions in $D_4$ are those of $D_1$. Afterwards, we delete $D_1$. The result is shown in Figure 5.5.

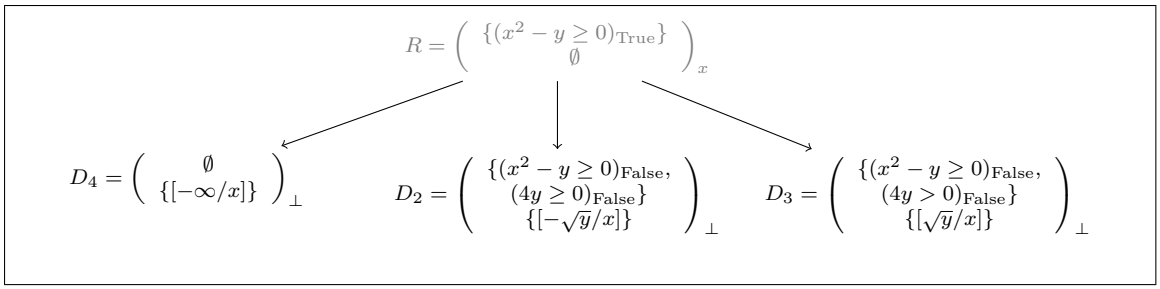


**Figure 5.5.:** Solver state after adding the constraint $c_1 : \; x^2 - y \geq 0$ and a consistency check.

The next decision tuple we choose is

$$D_4 \; = \; \left( \begin{array}{c} \emptyset \\ \{[-\infty/x]\} \end{array} \right)_\perp ,$$

which has no conditions. Hence, the consistency check determines consistency and terminates.

Now we add the constraint $c_2 : \; x^2 - 1 = 0$ by calling *add_constraint$_{TO}$($c_2$)*. To exploit the previously made computation, it takes the tree of decision tuple the consistency check created and adds the corresponding condition of $c_2$ to the condition set of each decision tuple. Algorithm 10 distinguishes between two cases, but in our example only the second case occurs. We add the corresponding condition $(x^2 - 1 = 0)_{\text{False}}$ and get the result which is shown in Figure 5.6.

After adding $c_2$, we can provoke another consistency check, that is calling the method *is_consistent$_{TO}$()* once again. In the same line as in the previous consistency check we choose an unmarked decision tuple of the present tree. In the beginning of a consistency check all are unmarked. Due to a heuristics 5.4 we choose the decision tuple

$$D_4 \; = \; \left( \begin{array}{c} \{(x^2 - 1 = 0)_{\text{False}}\} \\ \{[-\infty/x]\} \end{array} \right)_\perp$$

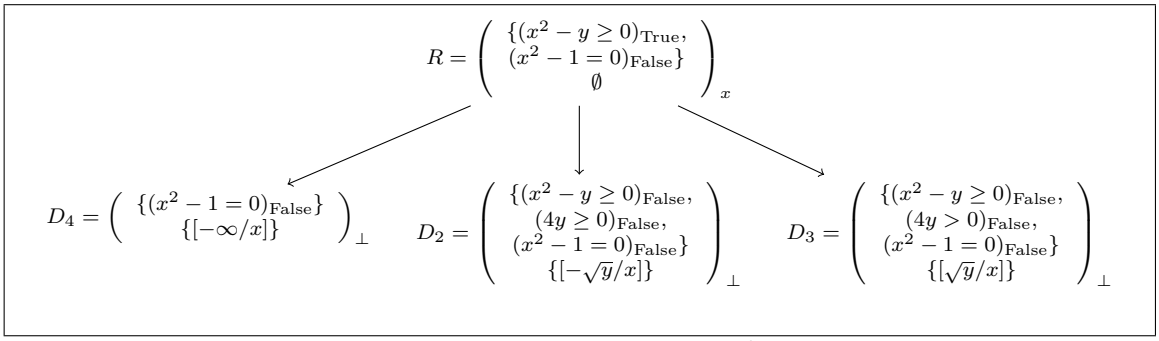


**Figure 5.6.:** Solver state after adding the constraint $c_1 : \ x^2 - y \geq 0$, an entire consistency check, and a belated adding of the constraint $c_2 : \ x^2 - 1 = 0$.

and therein the only condition $(x^2 - 1 = 0)_{\text{False}}$ to proceed. We can apply the substitution $[-\infty/x]$ to it. The corresponding substitution rule says that all coefficients of $x$ in the condition must be zero. This is not fulfilled, thus no decision tuple is created and $D_4$ gets deleted. Figure 5.7 shows the intermediate result of the first step in the second consistency check.

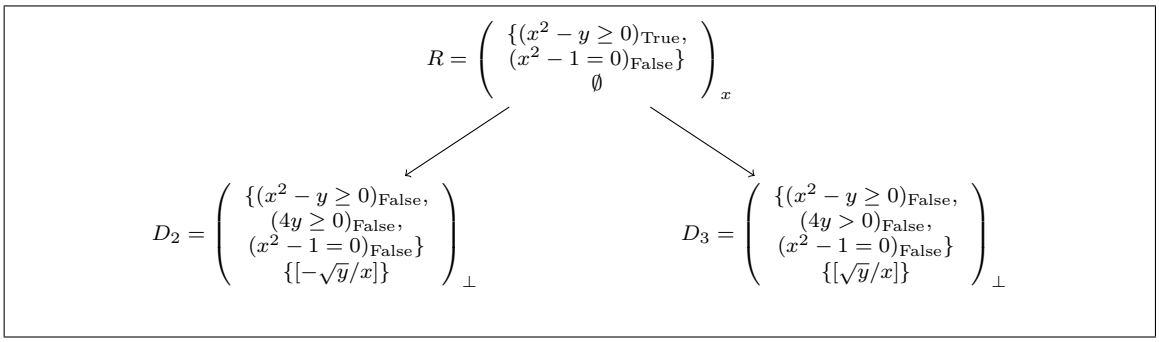


**Figure 5.7.:** Solver state after adding the constraint $c_1 : \ x^2 - y \geq 0$, an entire consistency check, a belated adding of the constraint $c_2 : \ x^2 - 1 = 0$ and another step of the following consistency check.

Now, there are three decision tuples left. We could choose each of them to continue the consistency check. We take the decision tuple

$$D_2 \ = \ \left( \begin{array}{c} \{(x^2 - y \geq 0)_{\text{False}}, \\ (4y \geq 0)_{\text{False}}, \\ (x^2 - 1 = 0)_{\text{False}}\} \\ \{[-\sqrt{y}/x]\} \end{array} \right)_\perp$$

and choose a condition in it, let us say $(x^2 - y \geq 0)_{\text{False}}$. The first case of the evaluation method occurs, where we substitute $x$ in the chosen condition by $-\sqrt{y}$. Considering the rules of Appendix A the result of the common substitution is the term

$$0 \geq 0 \ \Leftrightarrow \ \frac{0}{1} \geq 0$$

and $\delta = 0$, since the degree of $x$ in the considered condition is two. In this case, the result already does not contain any square roots and thus it is a real algebraic constraint. Nevertheless, we apply the substitution rules as the algorithm does in order to demonstrate the idea of the virtual substitution once again. Inserting the parameters in the equivalent formula given by the corresponding substitution rule leads to

$$\begin{array}{rccccc} & ( & 1^0 > 0 & \wedge & 0 \cdot 1^2 \geq 0 & ) \\ \vee & ( & 1^0 < 0 & \wedge & 0 \cdot 1^2 \leq 0 & ) \end{array}$$

All constraints in this formula are real algebraic. The second conjunction is inconsistent and the first conjunction leads to the expected result, which we also would obtain by substituting in the common way. The result is shown in Figure 5.8.

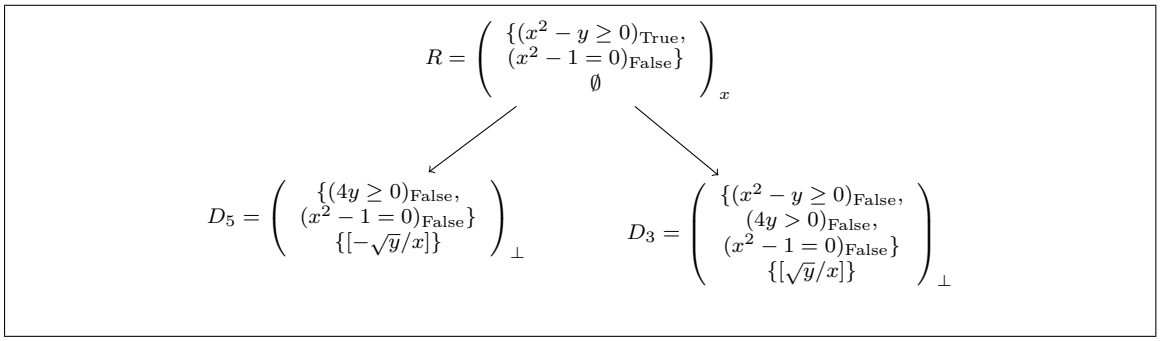


**Figure 5.8.:** Solver state after adding the constraint $c_1 : \ x^2 - y \geq 0$, an entire consistency check, a belated adding of the constraint $c_2 : \ x^2 - 1 = 0$ and two steps of the following consistency check.

The next decision tuple we choose is

$$D_5 \ = \ \left( \begin{array}{c} \{(4y \geq 0)_{\text{False}}, \\ (x^2 - 1 = 0)_{\text{False}}\} \\ \{[-\sqrt{y}/x]\} \end{array} \right)_\perp$$

and let us consider its condition $(x^2 - 1 = 0)_{\text{False}}$. We get a similar case as the previous one, but with an equation. Without further explanation it results in replacing the considered decision tuple by

$$D_6 \ = \ \left( \begin{array}{c} \{(4y \geq 0)_{\text{False}}, \\ (y - 1 = 0)_{\text{False}}\} \\ \{[-\sqrt{y}/x]\} \end{array} \right)_\perp ,$$

which is the next decision tuple we consider. We generate test candidates for the condition $(y - 1 = 0)_{\text{False}}$ resulting in the decision tuples

$$D_7 \ = \ \left( \begin{array}{c} \{(4y \geq 0)_{\text{False}}, \\ (y - 1 = 0)_{\text{False}}\} \\ \{[-\sqrt{y}/x], [-\infty/y]\} \end{array} \right)_\perp$$

and

$$D_8 \;=\; \left( \begin{array}{c} \{(4y \geq 0)_{\mathrm{False}}, \\ (y-1=0)_{\mathrm{False}}\} \\ \{[-\sqrt{y}/x], [1/y]\} \end{array} \right)_{\bot} .$$

The index of $D_6$ gets set to $y$ and the flag of the considered condition gets set to $\mathrm{True}$. In order to finish this example, we take $D_8$ to continue. Two more substitution steps lead to the tree of decision tuples shown in Figure 5.9. It includes a condition-free decision tuple, so the consistency check determines consistency for the given set of constraints and terminates. The condition-free decision tuple also defines a satisfying assignment for $x$ and $y$.

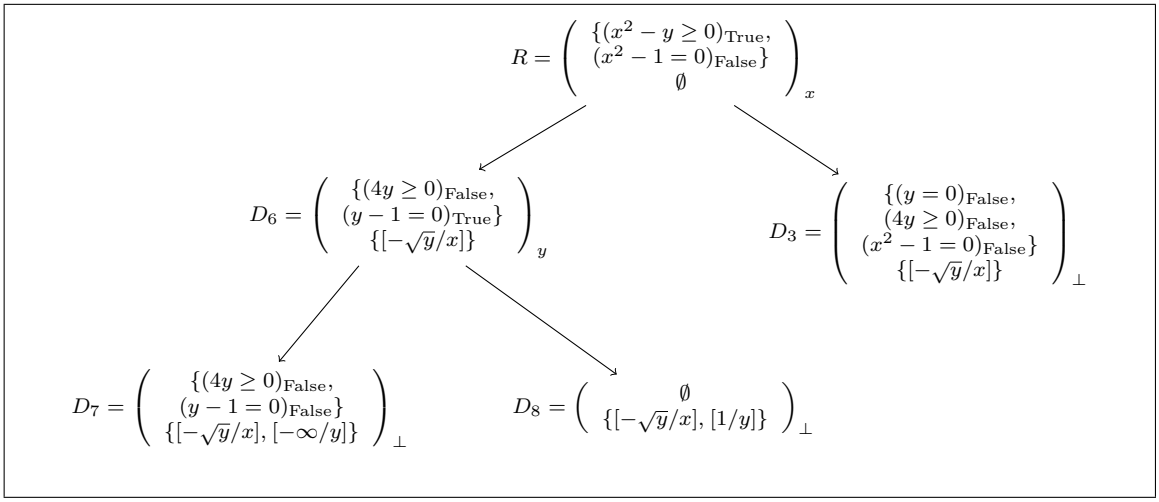


**Figure 5.9.:** Solver state after adding the constraint $c_1 : \; x^2 - y \geq 0$, an entire consistency check, a belated adding of the constraint $c_2 : \; x^2 - 1 = 0$ and the following entire consistency check.

### 5.7.2. Entire substitution approach

We initiate the theory solver with the tree

$$T = \left(\{\begin{pmatrix}\emptyset\\ \emptyset\end{pmatrix}_{\perp}\}, \emptyset\right)$$

kept in a global variable. We call *add_constraint*$_{ES}$ with the argument $c_1 : x^2 - y \geq 0$ to hand over the first constraint to the theory solver. The method *add_constraint*$_{ES}$ adds the constraint $c_1$ to the condition set of the root in $T$, i.e.

$$\begin{pmatrix}\emptyset\\ \emptyset\end{pmatrix}_{\perp} \text{ gets extended to } \begin{pmatrix}\{(x^2 - y \geq 0)_{\underline{\text{False}}}\}\\ \emptyset\end{pmatrix}_{\perp}.$$

Until here, only one difference has occured compared to the other approach, namely, that we underlined the flag of the condition. Conditions with underlined flags represent the recently added ones. Afterwards, we perform a consistency check by calling the method *is_consistent*$_{ES}$*()*. We start with the only existing decision tuple

$$R = \begin{pmatrix}\{(x^2 - y \geq 0)_{\underline{\text{False}}}\}\\ \emptyset\end{pmatrix}_{\perp}.$$

It contains recently added conditions, so we perform the third case of the evaluation method given by Algorithm 12. The decision tuple $R$ has no children, thus we just mark its only condition as not recently added, i.e.

$$R = \begin{pmatrix}\{(x^2 - y \geq 0)_{\text{False}}\}\\ \emptyset\end{pmatrix}_{\perp}.$$

In the next steps the evaluation behaves in the same way as in the previous example, since there is always just one condition to substitute in. So, the final result of the consistency check is the result shown by Figure 5.5. It is the same as we had for the other approach after adding the constraint $c_1$ and performing a consistency check.

Now, we add the constraint $c_3 : \ x - 1 > 0$ by calling *add_constraint*$_{ES}$*(*$c_3$*)*. We changed the constraint compared to the previous example to get cases, where we have to substitute test candidates including the infinitesimal value $\epsilon$. We add $c_3$ as a recently added condition to the root of the tree of decision tuples achieved by the previous consistency check. All the other decision tuples get marked in order to avoid choosing them for evaluation. The result is shown in Figure 5.10.

We have just one opportunity to continue, namely the root node, and have to substitute belatedly as Algorithm 12 describes in its last case [2]. Firstly, we group the children of the

[2]Note that we also could add recently added conditions to children without applying the substitution before, if they have not yet been considered for an entire substitution step.

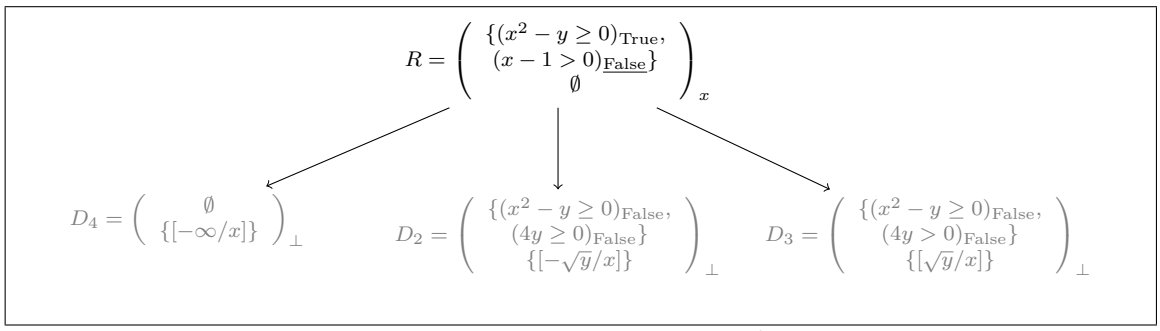


**Figure 5.10.:** Solver state after adding the constraint $c_1 : \ x^2 - y \geq 0$, an entire consistency check, and a belated adding of the constraint $c_3 : \ x - 1 > 0$.

considered decision tuple by their most recent substitutions resulting in:

$$[-\infty/x] \qquad D_4 \ = \ \left( \begin{array}{c} \emptyset \\ \{[-\infty/x]\} \end{array} \right)_\perp$$

$$[-\sqrt{y}/x] \qquad D_2 \ = \ \left( \begin{array}{c} \{(x^2 - y \geq 0)_{\text{False}}, \\ (4y \geq 0)_{\text{False}}\} \\ \{[-\sqrt{y}/x]\} \end{array} \right)_\perp$$

$$[\sqrt{y}/x] \qquad D_3 \ = \ \left( \begin{array}{c} \{(x^2 - y \geq 0)_{\text{False}}, \\ (4y \geq 0)_{\text{False}}\} \\ \{[\sqrt{y}/x]\} \end{array} \right)_\perp$$

We apply each substitution to the recently added condition bearing in mind the substitution rules of Appendix A. By applying $(x - 1 > 0)[-\infty/x]$ we get the equivalent formula

$$\begin{array}{cccc} & ( & 1 < 0 & ) \\ \vee & ( & 1 = 0 \ \wedge \ -1 > 0 & ), \end{array}$$

which is inconsistent. Hence, we delete all children, that concern this substitution, in this case just $D_4$. By applying the second substitution $(x - 1 > 0)[-\sqrt{y}/x]$ we first consider the result of the common substitution:

$$-\sqrt{y} - 1 > 0 \ \Leftrightarrow \ \frac{-1 + (-1) \cdot \sqrt{y}}{1} \hat{=} \frac{\hat{q} + \hat{r} * \sqrt{t}}{\hat{s}}.$$

The right hand side shows the result of the common substitution in the so-called *square root normal form*. The substitution rules consider this form to construct an equivalent real algebraic formula. Appendix **??** shows, that we can transform the result of substituting a variable in a real algebraic constraint by an extended real algebraic constraint in square root normal form always to square root normal form. The degree of $x$ in $x - 1 > 0$ is odd, so $\delta$ is set to 1. Now, we can construct the equivalent real algebraic formula of the term we

got by the common substitution:

$$\begin{array}{llllll}
 & ( & -1>0 & \wedge & 1^1>0 & \wedge & (-1)^2-(-1)^2*y>0 & ) \\
\vee & ( & -1<0 & \wedge & 1^1<0 & \wedge & (-1)^2-(-1)^2*y>0 & ) \\
\vee & ( & -1\leq 0 & \wedge & -1<0 & \wedge & 1^1<0 & ) \\
\vee & ( & -1\geq 0 & \wedge & -1>0 & \wedge & 1^1>0 & ) \\
\vee & ( & -1>0 & \wedge & 1^1>0 & \wedge & (-1)^2-(-1)^2*y<0 & ) \\
\vee & ( & -1<0 & \wedge & 1^1<0 & \wedge & (-1)^2-(-1)^2*y<0 & ).
\end{array}$$

It contains just inconsistent cases. This means, that we delete the decision tuple $D_2$. The result is based upon the fact, that $-\sqrt{y}$ must be negative and so does $-\sqrt{y}-1$. Therefore, $-\sqrt{y}-1>0$ cannot be fulfilled. Applying the last substitution $(x-1>0)[\sqrt{y}/x]$ leads to the equivalent real algebraic formula in a similar manner:

$$\begin{array}{llllll}
 & ( & -1>0 & \wedge & 1^1>0 & \wedge & (-1)^2-1^2*y>0 & ) \\
\vee & ( & -1<0 & \wedge & 1^1<0 & \wedge & (-1)^2-1^2*y>0 & ) \\
\vee & ( & 1\leq 0 & \wedge & -1<0 & \wedge & 1^1<0 & ) \\
\vee & ( & 1\geq 0 & \wedge & -1>0 & \wedge & 1^1>0 & ) \\
\vee & ( & 1>0 & \wedge & 1^1>0 & \wedge & (-1)^2-1^2*y<0 & ) \\
\vee & ( & 1<0 & \wedge & 1^1<0 & \wedge & (-1)^2-1^2*y<0 & ).
\end{array}$$

The only case without a variable-free inconsistent constraint is the second last one. It contains just one constraint having variables, which we add as a recently added constraint to $D_3$. Then, we unmark $D_3$ in order to make its choice as the next decision tuple to evaluate possible again. Finally, we mark the considered conditions, in our case just $(x-1>0)_{\underline{\text{False}}}$, as not recently added. The result of the belated substitution step is shown in Figure 5.11.

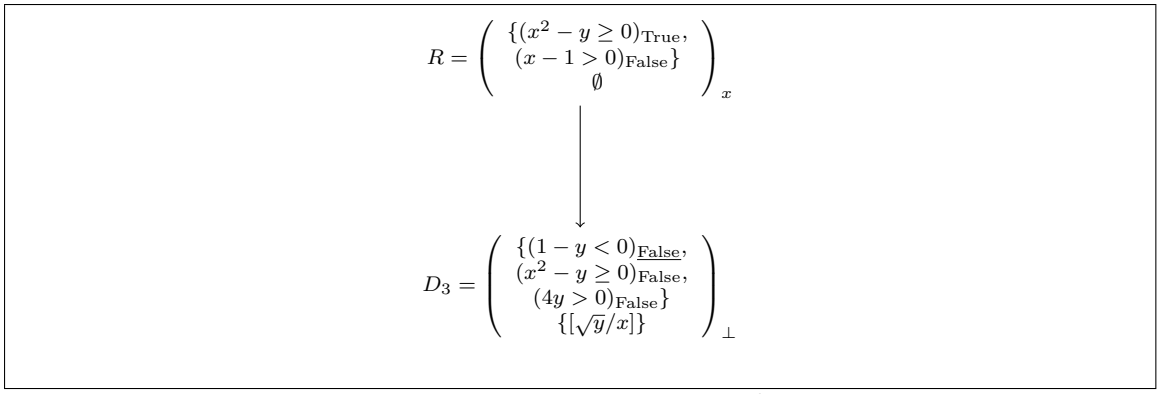


**Figure 5.11.:** Solver state after adding the constraint $c_1: \ x^2-y\geq 0$, an entire consistency check, and a belated adding of the constraint $c_3: \ x-1>0$ and one step of another consistency check.

We continue with the decision tuple

$$D_3 \ = \ \left( \begin{array}{c} \{(1-y<0)_{\underline{\text{False}}}, \\ (x^2-y\geq 0)_{\text{False}}, \\ (4y>0)_{\text{False}}\} \\ \{[\sqrt{y}/x]\} \end{array} \right)_\perp .$$

It includes a recently added condition, but no children. We had this case in the beginning of this example and know that this just leads to marking the condition as not recently added. We choose again $D_3$ and have to substitute all occurrences of $x$ by $\sqrt{y}$. There is just one in the condition $(x^2 - y \geq 0)_{\text{False}}$, hence the substitution step is performed in the same way as in the other evaluation approach. The last example performed the equivalent substitution $(x^2 - y \geq 0)[-\sqrt{y}/x]$, so we know that the only resulting decision tuple, which replaced $D_3$, is

$$D_5 = \begin{pmatrix} \{(1 - y < 0)_{\text{False}}, \\ (4y > 0)_{\text{False}}\} \\ \{[\sqrt{y}/x]\} \end{pmatrix}_{\perp}.$$

The created decision tuple fulfills all requirements to perform another test candidate generation. There is only one variable, namely $y$, and we choose the condition $(1 - y < 0)_{\text{False}}$ as test candidate provider. We generate two test candidates: $-\infty$, since it has not yet been generated in this decision tuple and $1 + \epsilon$ with the consistent side condition $-1 \neq 0$. After adding the corresponding children and setting the flag of the considered condition to $\text{True}$, we get the tree of decision tuples of Figure 5.12.

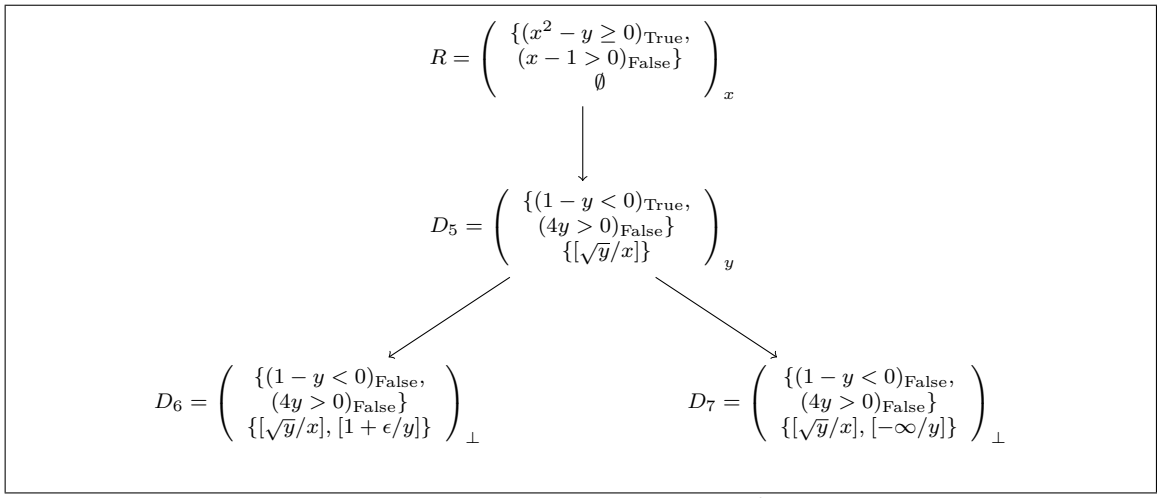


**Figure 5.12.:** Solver state after adding the constraint $c_1 : \ x^2 - y \geq 0$, an entire consistency check, and a belated adding of the constraint $c_3 : \ x - 1 > 0$ and some steps of another consistency check.

We continue with the decision tuple

$$D_6 = \begin{pmatrix} \{(1 - y < 0)_{\text{False}}, \\ (4y > 0)_{\text{False}}\} \\ \{[\sqrt{y}/x], [1 + \epsilon/y]\} \end{pmatrix}_{\perp}$$

and have to substitute all occurrences of the variable $y$ by $1 + \epsilon$. A substitution by a term containing $\epsilon$, requires the derivates of the condition to substitute in, in order to form an equivalent formula of real algebraic constraints. For further explanation you can study the second example we gave in Section 4. The first substitution $(1 - y < 0)[1 + \epsilon/y]$ leads to

the equivalent formula

$$\begin{array}{rlllll} & ( & (1-y<0)[1/y] & & & ) \\ \vee & ( & (1-y=0)[1/y] & \wedge & (-1<0)[1/y] & ) \end{array}$$

$$\begin{array}{rlllll} = & ( & (0<0) & & & ) \\ \vee & ( & (0=0) & \wedge & (-1<0) & ). \end{array}$$

The second substitution $(4y>0)[1+\epsilon/y]$ leads to the equivalent formula

$$\begin{array}{rlllll} & ( & (4y>0)[1/y] & & & ) \\ \vee & ( & (4y=0)[1/y] & \wedge & (4>0)[1/y] & ) \end{array}$$

$$\begin{array}{rlllll} = & ( & (4>0) & & & ) \\ \vee & ( & (4=0) & \wedge & (4>0) & ). \end{array}$$

Both have just one case, which does not contain inconsistent constraints. In both cases all constraints are variable-free and consistent, so we discard them. Thus we get two constraintless cases, whose combination is a constraintless case as well. This entire substitution step results in replacing the considered $D_6$ by a conditionless decision tuple. This implies, that the consistency check determines consistency and terminates with the tree of decision tuples shown in Figure 5.13.

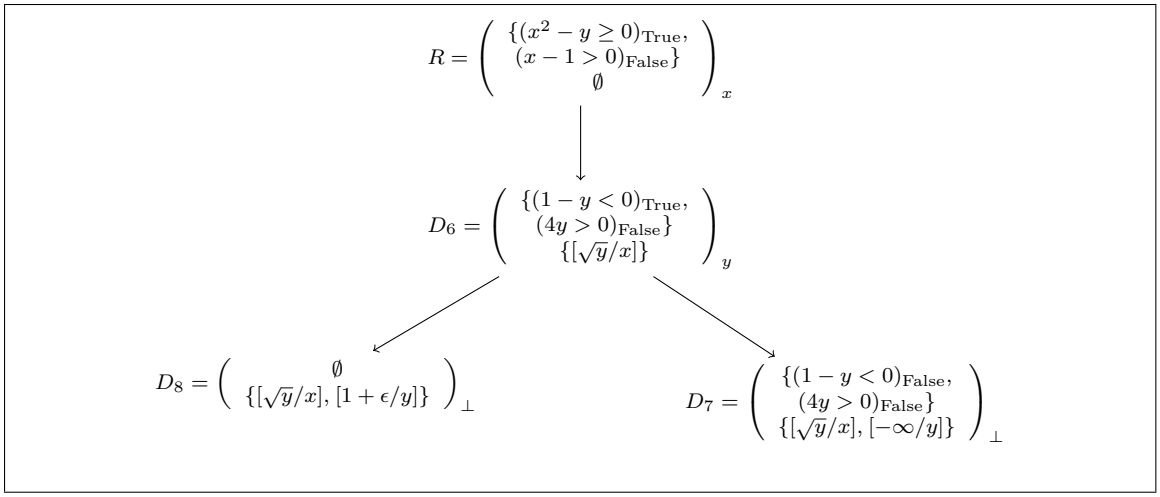


**Figure 5.13.:** Solver state after adding the constraint $c_1: \; x^2-y\geq 0$, an entire consistency check, and a belated adding of the constraint $c_3: \; x-1>0$ and another entire consistency check.

The examples covered all significant cases of Algorithm 2 and Algorithm 12 and almost all variants we can have during a substitution. It also pointed out how much influence the choice of the next decision tuple as well as the choice of the next condition in it has. Our choices were motivated by the goal to step in almost all possible cases. Another important observation is, that the incrementality turned to account, since we used in the second consistency check a lot of the results made by the first consistency check. We also could see how the two evaluation approaches differ and what their intersection is.

# 6. Minimal infeasible subset generation

We have already introduced how to add constraints incrementally and how to check for consistency. The interface we explain in this section comes into play, when the consistency check fails, that is, the algorithm determines inconsistency. The SMT-solver should then be able to extract a reason for the inconsistency from the theory solver, i.e., an inconsistent subset of the set of contraints the SMT-solver passed to the theory solver. The best reason we could achieve would be a minimal infeasible subset, an infeasible subset $R$, which does not contain a infeasible proper subset $R' \subset R$.

In the previous sections we have given two different approaches for the evaluation of a decision tuple. Using an evaluation method, a consistency check creates a tree of decision tuples, which fulfills certain properties. These properties differ between the two approaches. We also need different versions for the minimal subset generation. We concentrate on the second approach, as we are not convinced of the performance, that the first approach achieves. A sketchy analysis exposed, that it would even be more difficult to generate (minimal) infeasible subsets considering the first approach. So, the following abandons the target oriented approach for the evaluation of a decision tuple and just considers the entire substitution approach.

The remaining of this chapter is structured as follows: We first analyse the requirements to find all minimal infeasible subsets of the set of conditions of a decision tuple. It points out, how much effort is necessary to get the best solution for a minimal infeasible subset, which is the smallest minimal subset of those we have. Then we scale the search for (minimal) infeasible subsets down to a justifiable effort, such that we can embed it into the already achieved incremental implementation of a consistency check using the entire substitution approach. In the last part of this chapter we speed up the consistency check using the (minimal) infeasible subsets we have achieved before.

## 6.1. Generation of all minimal infeasible subsets

In order to achieve all minimal infeasible subsets, we have to modify the evaluation of a decision tuple. Both evaluation approaches assume, that decision tuples containing a variable-free inconsistent condition get deleted. For the generation of all minimal infeasible subsets, we assume, that we do not delete them, but still discard variable-free consistent conditions. Considering this modification, we possibly have to evaluate a decision tuple, whose conditions are all variable-free inconsistent. For these decision tuples the evaluation method using the entire substitution approach given by Algorithm 12 cannot generate test

candidates nor apply a substitution. Therefore, a second modification is, that we mark those decision tuples in order to avoid choosing them again [1]. We call this version of evaluating decision tuples the *exhaustive evaluation (using the entire substitution approach)* and do not specify it further, since it does only serve to see the theoretical effort to generate all minimal infeasible subsets. Compared to the original version, it generates more decision tuples, as it does not prune those subtrees of the globally stored tree of decision tuples, whose root contains a variable-free inconsistent condition. In the following we prove, that the exhaustive evaluation is necessary to generate all minimal infeasible subsets from the conditions of an inconsistent decision tuple, which we define next.

**Definition 6.1.1 (Inconsistent decision tuple)**
*Let $D = (C, S)_v$ be a decision tuple. It is* inconsistent *if the set of constraints occuring in $C$ is inconsistent.*

In order to be able to identify inconsistent decision tuples during Algorithm 9, we show in Theorem 6.1.1, which properties make a decision tuple inconsistent.

**Theorem 6.1.1**
*Let $D = (C, S)_v$ be a decision tuple of a tree $T$, generated by the method is_consistent$_{ES}$() of Algorithm 9. If all decision tuples of the subtree of $T$ with root $D$ are marked by the algorithm in order to avoid choosing them again, then $D$ is inconsistent.*

***Proof:*** *If all decision tuples in the subtree of $T$ with root $D$ are marked, all test candidates of each of the conditions of these decision tuples, in particular $D$, are generated. Furthermore, no decision tuple in this subtree is a solution, viz. none of these decision tuples has an empty condition set. Therefore, none of these decision tuples, including $D$ leads to a satisfying decision tuple. If we consider the constraints, which occur in the conditions of one of these decision tuples, each test candidate they provide fails to fulfill them all. In Chapter 4 we have already shown, that this implies the inconsistency of this set of constraints and thus the inconsistency of the decision tuple containing them.*

Using minimal covering sets introduced in Definition 6.1.2, Theorem 6.1.2 shows how we can construct all minimal infeasible subsets of the conditions of an inconsistent decision tuple.

**Definition 6.1.2 (Minimal covering set)**
*Let $M = \{M_1, \ldots, M_k\}$ be a set of sets of sets. A* minimal covering set $MC$ of $M$ *fulfills the following properties:*

1. $\forall i \in \{1, \ldots, k\} : \exists N \in M_i : N \subseteq MC$
2. $MC' \subset MC \Rightarrow \exists i \in \{1, \ldots, k\} : \forall N \in M_i : N \nsubseteq MC'$

Note that there can exist more than one minimal covering set for a set of sets of sets.

[1] Note that we do not consider the case, that a condition with degree higher than 2 can be created during substitution.

**Theorem 6.1.2**
*Let $D = (C, S)_x$ be an inconsistent decision tuple constructed by a consistency check using exhaustive evaluation.*

1. *If $D$ is a leaf: $\{\{c\} | \ c \in C\}$ are the minimal infeasible subsets of $C$.*

2. *If $D$ is not a leaf: Let $t$ be a test candidate provided by any condition of $C$ (this includes $-\infty$). Let $C_1^t, \ldots, C_{k_t}^t$ be the conditions of the $k_t > 0$ children containing the substitution $[t/x]$ and $MIS(C_i^t)$, $1 \leq i \leq k_t$, be the set of all minimal infeasible subsets of $C_i^t$. Let $origin(c')$ with $c' \in C_i^t$ be defined as follows:*

$$origin(c') \ = \ \begin{cases} \emptyset & \text{, if } c' \text{ side condition of } t \\ c & \text{, if } c' \text{ is a result of } c[t/x] \end{cases}$$

*Applying $origin$ to each element in the sets of $MIS(C_i^t)$ results in $MIS(C_i^t)_o \subseteq 2^C$.*

*The minimal covering sets of*

$$MC = \{MIS(C_i^t)_o | \ t \text{ a test candidate of } C \text{ and } 1 \leq i \leq k_t\}$$

*form the minimal infeasible subsets of $C$.*

***Proof**: 1.) An inconsistent leaf just contains variable-free inconsistent conditions. Thus each subset of these conditions consisting of one condition is infeasible, since this condition is inconsistent, and minimal, since its only proper subset is the feasible set $\emptyset$. 2.) Let $M \in MC$. The test candidates $M$ provides are a subset of the test candidates $C$ provides, since $M \subseteq C$. Let $t$ be such a test candidate. For all $1 \leq i \leq k_t$ there exists a set $M' \in MIS(C_i^t)_o$ with $M' \subseteq M$ by reason of the construction of $MC$ and the fact that $M \in MC$. We know that applying the substitution $[t/x]$ to $C \cup C_t$, with $C_t$ the side conditions of $t$, leads to the disjunction of conjunctions $C_1^t \vee \ldots \vee C_{k_t}^t$, since these conjunctions are the conditions of the children of $D$ containing $[t/x]$ (considering conjunctions as sets). Applying $[t/x]$ to one condition of $C \cup C_t$ results in a disjunction of conjunctions and by combining these disjunctions according to Definition 5.2.1 we achieve $C_1^t \vee \ldots \vee C_{k_t}^t$. If we combine just the resulting disjunctions of conjunctions of applying $[t/x]$ to the conditions in $(M \cup C_t) \subseteq (C \cup C_t)$ we get $M_1^t \vee \ldots \vee M_{k_t}^t$ with the property $M_i^t \subseteq C_i^t$, $1 \leq 1 \leq k_t$ (1). Then it holds that*

$$\forall i \ \in \{1, \ldots, k_t\} : \ \exists N \in MIS(C_i^t) : \ N \subseteq M_i^t. \qquad (2)$$

*We prove (2) by the following: All side conditions of $t$ are elements of $M_i^t$, since they do not contain the variable $x$ and thus applying $[t/x]$ to a side condition results in itself. Combining it, according to Definition 5.2.1, with the results of applying $[t/x]$ to the other conditions in $M \cup C_t$ means that each combination contains the side condition, which implies that all side conditions of $t$ are in $M_i^t$ for all $i \in \{1, \ldots, k_t\}$. Therefore, in particular all conditions of all $N \in MIS(C_i^t)$ being a side condition of $t$ occur in $M_i^t$. It remains to show that there exists a set $N \in MIS(C_i^t)$, such that all its conditions not being*

*a side condition of $t$ are also elements of $M_i^t$. There exists a set $N_o$ in $MIS(C_i^t)_o$ such that $N_o \subseteq M$, since $M \in MC$. Then there exists a $N \in MIS(C_i^t)$ such that applying $origin$ to each of its elements leads to $N_o$. Considering the property explained in (1), it holds that $N \subseteq M_i^t$, since $N_o \subseteq M$.*

*As (2) holds for all test candidates $t$, $[t/x]$ applied to the constraints of $M$ results just in inconsistent sets of constraints. It follows that the constraints of $M$ are inconsistent (for further details see Chapter 4). Hence $M$ is an infeasible subset of $C$.*

*Let $\tilde{M} \subset M$. The second precondition of the definition of a minimal covering set says that if $\tilde{M} \subset M$, there exists a test candidate $t$ and an $i \in \{1, \ldots, k_t\}$ such that for all $N \in MIS(C_i^t)_o$ it holds that $N \nsubseteq \tilde{M}$ (3). We construct $\tilde{M}_i^t \subset C_i^t$ analogous to $M_i^t$ introduced in the first part of this proof. For all minimal infeasible subsets $N$ of $C_i^t$, i.e. $N \in MIS(C_i^t)$, it holds that $N \nsubseteq \tilde{M}_i^t$, since there exists a condition in the corresponding $N_o$, which we get by applying $origin$ to each element of $N$, such that this condition is not in $\tilde{M}$ (by reason of (3)). In particular for all $N \in MIS(C_i^t)$ it holds that $N \neq \tilde{M}_i^t$, hence $\tilde{M}_i^t$ is not a minimal infeasible subset. It also cannot be an infeasible subset, since otherwise there would be a minimal infeasible subset $N \in MIS(C_i^t)$ with $N \subseteq \tilde{M}_i^t$. Therefore $\tilde{M}_i^t$ is consistent and $t$ fulfills all conditions of $\tilde{M}$, which makes $\tilde{M}$ infeasible.*

*The last part remaining to prove is that there exists no minimal infeasible subset $\hat{M}$ of $C$, such that $\hat{M} \notin MC$. Let us assume that there exists a minimal infeasible subset $\hat{M}$ of $C$ with $\hat{M} \notin MC$. Hence $\hat{M}$ must violate at least one of the two precondition of Definition 6.1.2. If it violates the second precondition, then there exists a $M$ fulfilling the first precondition of the definition with $M \subset \hat{M}$. We have already proven, that $M$ must be an infeasible subset (minimal if it also fulfills the second precondition of the definition). Thus $\hat{M}$ cannot be minimal. If $\hat{M}$ violates the first precondition of the definition, there exists a test candidate $t$ and an $i \in \{1, \ldots, k_t\}$ such that for all $N_o \in MIS(C_i^t)_o$ it holds that $N_o \nsubseteq \hat{M}$. In (3) we have already proven for an analogous constellation, that it implies consistency. Hence $\hat{M}$ is not infeasible, so it cannot be a minimal infeasible subset.*

Theorem 6.1.2 states that, assuming the exhaustive evaluation, we can achieve not just one minimal infeasible subset but all. This also includes a smallest one, which is an optimal solution for a minimal infeasible subset. Unfortunately, this version requires that the whole tree of decision tuples gets evaluated. The method is_consistent$_{ES}$() in Algorithm 9 prunes subtrees, whose root is a decision tuple containing variable-free inconsistent conditions. The smaller an infeasible subset the faster we can solve the satisfiability problem of the given real algebraic SMT-formula, since the SAT-solver has to consider less literals for a conflict resolution. The less literals it has to consider, the more unsatisfiable assigments the SAT-solver can omit. In future tests we want to find a modification of our implementation, which has a good balance between allowing more effort in order to minimize the size of the infeasible subset we achieve and trying to check for consistency as fast as possible.

## 6.2. Infeasible subset generation

In this section we extend Algorithm 12, such that it generates infeasible subsets of the conditions of an inconsistent decision tuple using the idea of Theorem 6.1.2. Compared to the exhaustive evaluation, Algorithm 12 deletes decision tuples containing variable-free inconsistent conditions. Furthermore, we settle for a set of some infeasible subsets, instead of the set of all minimal infeasible subsets. In the remaining, we first construct the necessary extension of a decision tuple, in order to support the generation of infeasible subsets using the idea of Theorem 6.1.2. Afterwards we extend the achieved algorithms for a consistency check of a set of constraints (entire substitution approach), such that they generate an infeasible subset of these constraints, if they are inconsistent. Finally, we prove the infeasibility of the found subsets and show, why we cannot achieve always a minimal infeasible subset.

### 6.2.1. Extension of the data model

The decision tuples we have considered until now do not support to collect the information we need to perform the generation of (minimal) infeasible subsets according to Theorem 6.1.2. Firstly, we need to be able to reconstruct the condition $c$ in the father of an inconsistent decision tuple $D$, if applying a substitution to $c$ leads to a condition $c'$ of an infeasible subset in $D$. We call $c$ the *original condition of* $c'$. The original conditions of one infeasible subset of $D$ form a subset of the conditions in $D$'s father $D_f$. This set is an element of the *conflict set of* $D$ *in* $D_f$. We cannot reconstruct an infeasible subset of $D$ after deleting it. Without infeasible subsets of $D$, we cannot generate infeasible subsets of $D_f$, since we need infeasible subsets of $D$ to construct the conflict sets of $D$ in $D_f$. If we consider Theorem 6.1.2, the minimal covering sets of $D_f$'s conflict sets form infeasible subsets of $D_f$. Thus, we either do not delete $D$ or store the conflict set of $D$ in $D_f$. We decided for the latter way and extend decision tuples, such that they store a conflict set of each deleted child.

**Definition 6.2.1 (Decision triples)**
*A* decision triple *has the following structure:*

$$\begin{pmatrix} Conditions \\ Substitutions \\ Conflict\_sets \end{pmatrix}_{index}$$

*where:*

- $Conditions$ *is a set of indexed constraints*

  $$cond \;=\; (p \sim 0)^{origin}_{flag}$$

  *where flag is defined as follows:*

  $$flag \;=\; \begin{cases} \text{True} & \text{, if the constraint was used} \\ & \text{to generate test candidates} \\ \text{False} & \text{, otherwise.} \end{cases}$$

*and $origin$ is defined by:*

$$origin = \begin{cases} \bot & \text{, if it is a side condition} \\ oCond & \text{, if } oCond \text{ is the original condition of } cond \end{cases}$$

- $Substitutions$ *is a set of substitutions of variables by test candidates:*

$$[test\ candidate\ /\ variable].$$

*The variables of all substitutions in this set must be pairwise different.*

- *For each deleted child $D_c$ $Conflict\_sets$ contains a set of subsets of $Conditions$. These subsets are the original conditions of an infeasible subset of $D_c$.*

- *The $index$ has the following definition:*

$$index = \begin{cases} var & \text{, the variable for which the test candidates in this decision triple are generated} \\ \bot & \text{, if still no variable is determined to generate test candidates for.} \end{cases}$$

Note that in practicethe original condition of a condition and the conflict sets do not contain conditions, but references to conditions of the corresponding decision tuple. An *inconsistent decision triple* is defined analogously to an inconsistent decision tuple. It can be identified in an analog manner to Theorem 6.1.1 considering that we use Algorithm 15 (see below) instead of Algorithm 9.

## 6.2.2. Embedding in the theory solver

Until now we have achieved two interfaces of the theory solver to the SMT-solver and we want to extend them by another interface, which generates an infeasible subset of the set of constraints, which failed to be consistent according to a previous consistency check. Considering the idea of Theorem 6.1.2, it is not sufficient to add a new method, but we also need to modify the method, which checks for consistency in order to collect the conflict sets in each decision tuple. Furthermore, all existing methods now have to support decision triples instead of decision tuples. Summing up, we have to establish the following:

1. Instead of generating the decision tuple $D = (C, S)_v$, we generate a decision triple $\tilde{D} = (\tilde{C}, S, M)_v$, where $M$ is initially empty and for each $(c)_{flag} \in C$ there exists a $(c)_{flag}^{oCond} \in \tilde{C}$, such that $oCond$ is the condition's original condition according to Definition 6.2.1. We adapt the methods using the entire substitution approach, such

that they deal with and generate decision triples instead of decision tuples:

| **Method** | **Using decision tuples** | **Using decision triples** |
|---|---|---|
| addConstraint | Algorithm 11 | Algorithm 14 |
| isConsistent | Algorithm 9 | Algorithm 15 |
| evaluate | Algorithm 12 | Algorithm 16 |
| generateTestCandidates | Algorithm 7 | Algorithm 17 |
| substitute | Algorithm 6 | Algorithm 18 |
| substituteBelated | Algorithm 13 | Algorithm 19 |

2. Considering Algorithm 12, it deletes a decision tuple $D$, if it contains a variable-free inconsistent condition. We extend this operation, such that it is responsible for the generation of the conflict sets. In the following we delete a decision triple not just, if it contains variable-free inconsistent conditions, but if we know, that it is inconsistent. We assure to delete just inconsistent decision triples recursively, starting with a decision triple, which contains variable-free inconsistent conditions (leaf). We delete a decision triple $D$ in the following way:

   a) Generate infeasible subsets $I_D$ of $D$:
   If $D$ has a variable-free inconsistent condition, $I$ contains all sets consisting of one of the variable-free inconsistent conditions; Otherwise $I_D$ consists of the minimal covering sets of the conflict sets in $D$.

   b) If $D$ has a father $D_f$, add a set $M^D_{D_f}$ to $D_f$'s conflict sets $M_{D_f}$. For each infeasible subset $\tilde{I_D} \in I_D$ extend $M^D_{D_f}$ by a set consisting of the original conditions of $\tilde{I_D}$. If $D$ is the root, store one of its smallest infeasible subsets as an approximation of a minimal infeasible subset of the constraints the theory solver checked.

   c) Delete $D$.

   d) If $D$ has a father $D_f$, which is marked to avoid choosing it again, and does not have children anymore, we delete the $D_f$ in the just described manner. Note that $D_f$ fulfills the requirements to be an inconsistent decision triple according to Theorem 6.1.1.

   The pseudo code of deleting a decision triple is shown in Algorithm 20. We call this method in three situations: Firstly, if we evaluate (Algorithm 16) a decision triple containing a variable-free inconsistent condition, secondly, if we delete the last child of a decision triple, which is marked in order to avoid choosing it again (in Algorithm 20 itself), and, thirdly, if the main method to check for consistency (Algorithm 15) considers a decision triple, which has just conditions with flag $\mathrm{True}$ (at least one), and no children.

We extend the theory solver by a global variable representing an infeasible subset of the set of constraints the theory solver checked. Initially it is empty.

Note that we have to extend all algorithms to be able to deal with decision triples, which has not changed their underlying procedure. The most important change takes place in the method to delete an inconsistent decision triple given by Algorithm 20.

Theorem 6.2.1 states, that the minimal covering sets of the conflict sets generated for an inconsistent decision triple are indeed infeasible subsets of the conditions of this decision triple. In particular the set of constraints represented by the theory solver's global variable $I$, is infeasible, if it was filled up by Algorithm 20.

**Theorem 6.2.1**
*Let $D = (C, S, M)_v$ be an inconsistent decision triple constructed by a consistency check using Algorithm 15. Then each minimal covering set of $M$ is an infeasible subset of $C$.*

***Proof**: In the following we consider the whole tree that Algorithm 15 generates until it determines the inconsistency of the checked constraints. This includes the decision triples, which get deleted during the consistency check. We prove the theorem by induction:*

1. *Base case: If $D$ contains variable-free inconsistent conditions, each set of $M$ consists of one of these conditions. These sets obviously form infeasible subsets of $C$. Note that $D$ is a leaf, since Algorithm 16 does not generate children for it.*

2. *Inductive step: We assume, that for each inconsistent child $D_c = (C_c, S_c, M_c)_{v_c}$ of $D$ it holds, that each minimal covering set of $M_c$ is an infeasible subset of $C_c$. Hence, each conflict set in $M$ consists of the original conditions of an infeasible subset of an inconsistent child of $D$. Algorithm 20 assures, that all children of $D$ got deleted and all conditions of $D$ served as a test candidate provider, since $D$ is marked in order to avoid choosing it again. Let $N$ be a minimal covering set of $M$. The set of test candidates provided by the conditions in $N$ are a subset of those provided by $C$, since $N \subseteq C$. Let $t$ be such a test candidate and $D_i^t = (C_i^t, S \cup \{[t/v]\}, M_i^t)_{v_i^t}$, $1 \leq i \leq k_t$, the children of $D$ containing the substitution $[t/v]$. By the definition of a minimal covering set, $N$ covers a set in each conflict set in $M$. We know that applying the substitution $[t/v]$ to $C \cup C_t$, with $C_t$ the side conditions of $t$, leads to the disjunction of conjunctions $C_1^t \vee \ldots \vee C_{k_t}^t$, since these conjunctions are the conditions of the children of $D$ containing $[t/x]$ (considering conjunctions as sets). Applying $[t/x]$ to one condition of $C \cup C_t$ results in a disjunction of conjunctions and by combining these disjunctions according to Definition 5.2.1 we achieve $C_1^t \vee \ldots \vee C_{k_t}^t$. If we combine just the resulting disjunctions of conjunctions of applying $[t/x]$ to the conditions in $(N \cup C_t) \subseteq (C \cup C_t)$ we get $N_1^t \vee \ldots \vee N_{k_t}^t$ with the property $N_i^t \subseteq C_i^t$, $1 \leq 1 \leq k_t$ (*). Then it holds that*

$$\forall i \ \in \{1, \ldots, k_t\} : \ \exists \text{ infeasible subset } I_i^t \subseteq C_i^t : \ I_i^t \subseteq N_i^t. \qquad (**)$$

*We prove $(**)$ by the following: All side conditions of $t$ are elements of $N_i^t$, since they do not contain the variable $x$ and thus applying $[t/x]$ to a side condition results in itself. Combining it, according to Definition 5.2.1, with the results of applying $[t/x]$ to the other conditions in $N \cup C_t$ means that each combination contains the side condition, which implies that all side conditions of $t$ are in $N_i^t$ for all $i \in \{1, \ldots, k_t\}$.*

*Therefore in particular all conditions of an infeasible subset of $C_i^t$ being a side condition of $t$ do also occur in $N_i^t$. It remains to show that there exists an infeasible subset $I_i^t \subseteq C_i^t$, such that all its conditions not being a side condition of $t$ are also elements of $N_i^t$. As $N$ is a minimal covering set, it covers a set $\tilde{I}_i^t \subseteq N$ in the conflict set of $M$ we have generated after deleting $D_i^t$, such that its conditions are the original conditions of $I_i^t$. Considering the property explained in (*), it holds that $I_i^t \subseteq N_i^t$, since $\tilde{I}_i^t \subseteq N$.*

*As $(**)$ holds for all test candidates $t$, $[t/x]$ applied to the constraints of $N$ results just in inconsistent sets of constraints. It follows that the constraints of $N$ are inconsistent (for further details see Chapter 4). Hence $N$ is an infeasible subset of $C$.*

Theorem 6.2.1 states the infeasibility of the subsets of the conditions of an inconsistent decision triple $D$, that Algorithm 15 finds. According to Theorem 6.1.2, these subsets form all minimal infeasible subsets of $D$'s conditions, if we also have all minimal infeasible subsets of each of its children. If $D$ contains variable-free inconsistent conditions, Algorithm 20 generates just the infeasible subsets consisting of one of these conditions. They form obviously minimal infeasible subsets, but not necessarily all. They form all minimal infeasible subsets, iff the set of those conditions in $D$, which are not variable-free, is consistent. Unfortunately, we cannot assure this without an exhaustive evaluation. We can build an infeasible subset of a decision triple $D$, if all children are inconsistent and thus have their proper infeasible subsets. We do not suppose that this are all infeasible subsets, but at least one. So it is sufficient to find just one infeasible subset of each child to form an infeasible subset of $D$. However, the more infeasible subsets we have for each child, the more elements its conflict set in $D$ has. Having more opportunities to cover a conflict set minimizes the size of the minimal covering sets of them, which are the infeasible subsets of $D$'s conditions. Furthermore, the smaller the infeasible subsets of an inconsistent child of $D$, the smaller the sets in the conflict set of the child in $D$. Again, this leads to smaller minimal covering sets and hence smaller infeasible subsets of the conditions in $D$.

**Algorithm 14** The algorithm to add a constraint to the constraints in the theory solver ussing the entire substitution approach and decision triples.

**void add_constraint**$_{ES}^{IS}$(Constraint $c$)
**begin**
add $(c)^{\perp}_{\text{False}}$ to the conditions of the root of $T$; (1)
mark the other decision triples (to avoid choosing them); (2)
**end**

Variable-free consistent conditions get discarded.
Global variables: Tree_of_decision_triples $T = (V, E)$, Infeasible_subset $I$;

**Algorithm 15** The algorithm to determine consistency of a tree of decision triples using the entire substitution approach.

```
bool is_consistent^IS_ES()
begin
    while exists an unmarked Decision_triple D = (C, S, M)_v ∈ V do                (1)
        if exists c^cond_flag ∈ C with flag = False then                           (2)
            evaluate^IS_ES(D);                                  // Alg. 16         (3)
        else if C ≠ ∅ then                                                         (4)
            if D has children then                                                 (5)
                mark D (to avoid choosing it again);                               (6)
            else                                                                   (7)
                delete^IS(D);                                   // Alg. 20         (8)
            end if                                                                 (9)
        else                                                                       (10)
            return True;                                                           (11)
        end if                                                                     (12)
    end while                                                                      (13)
    return False;                                                                  (14)
end
```

Global variables: Tree_of_decision_triples $T = (V, E)$, Infeasible_subset $I$;

**Algorithm 16** The algorithm to evaluate a decision triple supporting incrementality using the entire substitution approach and decision triples.

```
void evaluate^IS_ES(Decision_triple D = (C, S, M)_v ∈ V)
begin
    if C contains a variable-free inconsistent condition then                      (1)
        delete^IS(D);                                       // Alg. 20             (2)
    else if v = ⊥ then                                                             (3)
        substitute^IS_ES(D);                                // Alg. 18             (4)
    else                                                                           (5)
        if D contains no recently added condition then                             (6)
            generateTestCandidates^IS_ES(D);                // Alg. 17             (7)
        else                                                                       (8)
            substituteBelated^IS_ES(D);                     // Alg. 19             (9)
        end if                                                                     (10)
    end if                                                                         (11)
end
```

Variable-free consistent conditions get discarded.
Global variables: Tree_of_decision_triples $T = (V, E)$, Infeasible_subset $I$;

**Algorithm 17** The algorithm to generate all test candidates for a variable of a condition using the entire substitution approach and decision triples.

**void generateTestCandidates**$^{IS}_{ES}$(Decision_triple $D = (C, S, M)_v \in V$)
**begin**
choose a condition $(c)^{oCond}_{\mathit{flag}} \in C$ with *flag* $=$ False; (1)
*flag* $:=$ True; (2)
**if** test canidate $-\infty$ not yet considered in $D$ **then** (3)
$D_{inf} := (\{(\hat{c})^{cond}_{\text{False}} | cond = (\hat{c})^{o\hat{C}ond}_{\hat{\mathit{flag}}} \in C\}, S \cup \{[-\infty/v]\}, \emptyset)_\perp$; (4)
$T := (V := V \cup \{D_{inf}\}, E := E \cup \{(D, D_{inf})\})$; (5)
**end if** (6)
$\{(t_1,C_1), \ldots, (t_k,C_k)\} \overset{Chap.\ 4}{:=}$ test candidates with side conditions of $c$ for $v$; (7)
**for all** $1 \leq i \leq k$ **do** (8)
**if** $C_i$ has no variable-free inconsistent constraints **then** (9)
$C_{D_i} := \{(\hat{c})^{cond}_{\text{False}} | cond = (\hat{c})^{o\hat{C}ond}_{\hat{\mathit{flag}}} \in C\} \cup \{(\hat{c})^{\perp}_{\text{False}} | \hat{c} \in C_i\}$; (10)
$D_i := (C_{D_i}, S \cup \{[t_i/v']\}, \emptyset)_\perp$; (11)
$T := (V := V \cup \{D_i\}, E := E \cup \{(D, D_i)\})$; (12)
**end for** (13)
**end**

Variable-free consistent conditions get discarded.
Global variables: Tree_of_decision_triples $T = (V, E)$, Infeasible_subset $I$;

**Algorithm 18** The algorithm to apply all substitutions in a decision triple using the entire substitution approach and decision triples.

**void substitute**$^{IS}_{ES}$(Decision_triple $D = (C, S, \emptyset)_\perp \in V$)
**begin**
$s :=$ most recent substitution in $S$; (1)
**for all** $(c_i)^{oCond_i}_{\text{False}} \in C = \{(c_1)^{oCond_1}_{\text{False}}, \ldots, (c_n)^{oCond_n}_{\text{False}}\}$ **do** (2)
$Disj_i = C_{i,1} \vee \ldots \vee C_{i,k_i} \overset{App.\ A}{:=}$ result of $s$ applied to $c_i$; (3)
$\tilde{Disj}_i := \{\{(c)^{oCond_i}_{\text{False}} |\ c \in C_{i,j}\} | 1 \leq j \leq k_i\}$; (4)
**end for** (5)
$\{C_1, \ldots, C_k\} \overset{Def.\ 5.2.1}{:=}$ combininations of $\{\tilde{Disj}_1, \ldots, \tilde{Disj}_l\}$; (6)
**for all** $1 \leq i \leq k$ **do** (7)
$v_i :=$ any variable occuring in $C_i$; (8)
$D_i := (C_i, S, \emptyset)_{v_i}$; (9)
$T := (V := V \cup \{D_i\}, E := E \cup \{(D_f, D_i)\})$; // ($D_f$ father of $D$) (10)
**end for** (11)
$T := (V := V \setminus \{D\}, E := E \setminus \{(D_f, D)\}$; // ($D_f$ father of $D$) (13)
**end**

Variable-free consistent conditions get discarded.
Global variables: Tree_of_decision_triples $T = (V, E)$, Infeasible_subset $I$;

**Algorithm 19** The algorithm to add all recently added constraints to the children after applying the most recent substitution entirely.

**void substituteBelated**$^{IS}_{ES}$(Decision_triple $D = (C, S, M)_v \in V$)
**begin**

$C = C_{old} \cup C_{recent}$; (1)
$\{s_1, \ldots, s_r\}$ := the different most recent substitutions in the children of $D$; (2)
**for all** $1 \leq i \leq r$ **do** (3)
$\{D_1 = (C_1, S \cup \{s_i\}, M_1)_{v_1}, \ldots, D_{k_i} = (C_{k_i}, S \cup \{s_i\}, M_{k_i})_{v_{k_i}}\}$ := all (4)
children of $D$ containing $s_i$; (5)
**for all** $cond = (c)^{oCond}_{flag} \in C_{recent}$ **do** (6)
$Disj_c = Conj_1 \vee \ldots \vee Conj_{n_i} \overset{App.\ A}{:=}$ result of $s_i$ applied to $c$; (7)
$\tilde{Disj}_c := \{\{(\hat{c})^{cond}_{\text{False}} | \ \hat{c} \in Conj_j\} | 1 \leq j \leq n_i\}$; (8)
**end for** (9)
$\{\hat{C}_1, \ldots, \hat{C}_{m_i}\} \overset{Def.\ 5.2.1}{:=}$ combininations of $\{\tilde{Disj}_c | c \in C_{recent}\}$; (10)
**for all** $1 \leq j \leq k_i$ **do** (11)
$T_{D_j} = (V_{D_j}, E_{D_j}) \overset{Def.\ 2.3.3}{:=}$ subtree of $T$ with root $D_j$; (12)
**for all** $1 \leq l \leq m_i$ **do** (13)
$D_{j,l} := (C_j \cup \hat{C}_l, S \cup \{s_i\}, M_j)_{v_j}$; (14)
$(V_{j,l}, E_{j,l}) :=$ copy of $T_{D_j}$, where $D_j$ is replaced by $D_{j,l}$; (15)
$T := (V := V \cup V_{j,l}, E := E \cup E_{j,l} \cup \{(D, D_{j,l})\})$; (16)
mark conditions in $D_{j,l}$ of $\hat{C}_l$ as recent; (17)
**end for** (18)
$T := (V := V \setminus V_{D_j}, E := E \setminus (E_{D_j} \cup \{(D, D_j)\})$; (19)
**end for** (20)
**end for** (21)
mark conditions in $C_{recent}$ as not recent; (22)
**end**

Variable-free consistent conditions get discarded.
Global variables: Tree_of_decision_triples $T = (V, E)$, Infeasible_subset $I$;

**Algorithm 20** The algorithm to delete a decision triple, such that it creates the conflict sets as well.

**void delete**$^{IS}$(Decision_triple $D = (C, S, M)_v \in V$)
**begin**
$I_D := \emptyset$; // (infeasible subsets of $C$) (1)
**if** $C$ contains a variable-free inconsistent conditions **then** (2)
**for all** variable-free inconsistent $c_{flag}^{cond} \in C$ **do** (3)
$I_D := I_D \cup \{\{c_{flag}^{cond}\}\}$; (4)
**end for** (5)
**else** (6)
$I_D \overset{Def.6.1.2}{:=}$ minimal cover sets of $M$; (7)
**end if** (8)
**if** $D$ is not root in $T$ **then** (9)
$D_f = (C_f, S_f, M_f)_{v_f} :=$ father of $D$; (10)
$M_f := M_f \cup \{\{\{cond | c_{flag}^{cond} \in J\} | J \in I_D\}\}$; (11)
$T := (V := V \setminus \{D\}, E := E \setminus \{(D_f, D)\})$; (12)
**if** $D_f$ is marked (to avoid choosing it again) and has no children **then** (13)
delete$^{IS}(D_f)$; (14)
**end if** (15)
**else** (16)
$I := \{c | (c)^{\perp}_{\text{True}} \in J\}$ with $J \in I_D,\ \forall \tilde{J} \in I_D : |J| \leq |\tilde{J}|$; (17)
$T := (\emptyset, \emptyset)$; (18)
**end if** (19)
**end**

Global variables: Tree_of_decision_triples $T = (V, E)$, Infeasible_subset $I$;

## 6.3. Backjumping using infeasible subsets

We do a lot of additional effort to generate an infeasible subset of the constraints the theory solver checked, in the case that it determines inconsistency. Certainly it is reasonable, since it can save many calls to the theory solver, which we consider as the bottleneck of an SMT-solver. Nevertheless, we search for more benefits we can achieve of the additional information we generate in each decision triple. This information can extend the properties, which influence a heuristics to find the next decision triple to evaluate introduced in Section 5.4. In this subsection we introduce an acceleration of the consistency check given by Algorithm 15. Theorem 6.3.1 states that, if we find an infeasible subset of the conditions in an inconsistent decision triple, such that the constraints of this conditions also occur in the conditions of the father of this decision triple, the father is also inconsistent.

**Theorem 6.3.1**
*Let $D = (C, S, M)_v$ be a decision triple. Let $M_i \in M$ be a conflict set of a deleted child of $D$. If $N \in M_i$, such that none of its conditions contains the variable $v$, $N$ is an infeasible subset of $C$, which makes in particular $D$ inconsistent.*

***Proof**: Let $cond_1 = (c_1)^{oCond_1}_{flag_1}, \ldots, cond_k = (c_k)^{oCond_k}_{flag_k}$ be the conditions of $N$. Applying a substitution $[t/v]$ to $\tilde{N} := \{c_1, \ldots, c_k\}$ results in $\tilde{N}$, since no condition in $N$ contains $v$. Hence, this also holds for those substitutions, which we create in the children of $D$. Thus, if we create a child $D_c = (C_c, S \cup \{[t_c/v]\}, M_c)$ of $D$, its set of conditions $C_c$ has a subset $N_c = \{(c_i)^{cond_i}_{flag^i_c} | 1 \leq i \leq k\}$. Let $D_c$ be the child, whose deletion led to the conflict set $M_i$. Then $N_c$ is an infeasible subset of $C_c$, since the original conditions of $N_c$ are $N$ and $N \in M_i$. The set of constraints formed by $\tilde{N}$ is inconsistent, since $N_c$ is infeasible. Let $D_c$ be any other child of $D$. Then $N_c$ is infeasible, since $\tilde{N}$ is inconsistent. Thus, all children we can create for $D$ contain an infeasible subset, which makes them, as well as $D$, inconsistent.*

We modify the method to delete a decision triple, given by Algorithm 20, such that it also considers the property Theorem 6.3.1 deals with. It still has a decision triple $D = (C, S, M)_v$ as argument and creates the infeasible subsets $I_D$ of $D$ in the same way as in Algorithm 20. If $D$ is not the root, the algorithm distinguishes now between two cases, when it adds a conflict set $M^D_{D_f}$ to the father $D_f = (C_f, S_f, M_f)_{v_f}$ of $D$:

1. If there exists an infeasible subset of $C$, such that its original conditions do not contain the variable $v_f$: For each of these infeasible subsets $M^D_{D_f}$ contains a set consisting of their original conditions. We discard all already achieved conflict sets in $M_f$, which assures that the following deletion of the father $D_f$ considers each set of $M^D_{D_f}$ as an infeasible subset.

2. Otherwise: For each infeasible subset $J \in I_D$ extend $M^D_{D_f}$ by a set consisting of the original conditions of $J$. Then delete the subtree of the so far created tree of decision triples $T$ with root $D$ entirely. If the father $D_f$ is marked to avoid choosing it again and has no children anymore, we delete it by calling this method recursively.

If $D$ is the root, the algorithm copies the constraints of one of the smallest infeasible subsets of $C$ to the globally stored infeasible subset $I$. This subset serves as an approximation of a minimal infeasible subset of the constraints the theory solver checked. Algorithm 21 shows the pseudo code of the just explained procedure.

## 6.4. Conclusion

The generation of (minimal) infeasible subsets requires, that we collect more information, which we achieve by the use of decision triples instead of decision tuples. In order to find always a minimal infeasible subset, we have to evaluate all decision triples, even if they contain a variable-free inconsistent condition. Considering e.g. a decision triple with hundreds of conditions of which just one is variable-free and inconsistent, it means, that we evaluate all decision triples of the subtree with root in this decision triple. That is why we do not insist on a minimal infeasible subset, but want to achieve an infeasible subset. This chapter showed an embedding of the generation of infeasible subsets in the algorithms we have developed in the previous chapter. As a side effect we generate infeasible subsets for each inconsistent decision triple of the globally stored tree of decision triples the theory solver maintains. This additional information allows us to speed up the consistency check using backjumping.

The additional interface to get an infeasible subset of the constraints the theory solver already checked is given by the method getInfSubSet() in Algorithm 22.

**Algorithm 21** The algorithm to delete a decision triple, such that it creates the conflict sets as well and uses backjumping.

**void delete**$^{IS}_{+}$(Decision_triple $D = (C, S, M)_v \in V$)
**begin**
$I_D := \emptyset$; // (infeasible subsets of $C$) (1)
**if** $C$ contains a variable-free inconsistent conditions **then** (2)
**for all** variable-free inconsistent $c^{cond}_{flag} \in C$ **do** (3)
$I_D := I_D \cup \{\{c^{cond}_{flag}\}\}$; (4)
**end for** (5)
**else** (6)
$I_D \overset{Def.6.1.2}{:=}$ minimal cover sets of $M$; (7)
**end if** (8)
**if** $D$ is not root of $T$ **then** (9)
$D_f = (C_f, S_f, M_f)_{v_f} :=$ father of $D$; (10)
$\tilde{I}_D := \{\{cond | c^{cond}_{flag} \in J\} | J \in I_D\} \subseteq 2^{C_f}$; (11)
**if** $\tilde{I}'_D := \{J \in \tilde{I}_D | v_f$ does not occur in $J\} \neq \emptyset$ **then** (12)
$M_f := \{\tilde{I}'_D\}$; (13)
delete$^{IS}(D_f)$; (14)
**else** (15)
$M_f := M_f \cup \{\tilde{I}_D\}$; (16)
$T_D = (V_D, E_D) \overset{Def.\ 2.3.3}{:=}$ subtree of $T$ with root $D$; (17)
$T := (V := V \setminus V_D, E := E \setminus (E_D \cup \{(D_f, D)\})$; (18)
**if** $D_f$ is marked (to avoid choosing it again) and has no children **then** (19)
delete$^{IS}(D_f)$; (20)
**end if** (21)
**end if** (22)
**else** (23)
$I := \{c | (c)^{\perp}_{\mathrm{True}} \in J\}$ with $J \in I_D, \forall \tilde{J} \in I_D : |J| \leq |\tilde{J}|$; (24)
$T := (\emptyset, \emptyset)$; (25)
**end if** (26)
**end**

Global variables: Tree_of_decision_triples $T = (V, E)$, Infeasible_subset $I$;

**Algorithm 22** The algorithm to return an infeasible subset of the constraints the theory solver already checked.

**Infeasible_subset getInfSubSet**()
**begin**
return $I$; (1)
**end**

Global variables: Tree_of_decision_triples $T = (V, E)$, Infeasible_subset $I$;

## 6.5. Example

In this example we check the consistency of the set of constraints

$$C \;=\; \{x = 0,\, x - z = 0,\, z \neq 0,\, y^2 < 0\}$$

considering Algorithm 15, which generates infeasible subsets of inconsistent decision triples and uses backjumping. We initialize the theory solver's global variables by setting the tree of decision triples

$$T := (\{D_0\}, \emptyset)$$

where

$$D_0 = \left( \begin{array}{ll} \{ & c_{0,1} : \; (x = 0)^{\perp}_{\mathrm{False}} \;\; , \\ & c_{0,2} : \; (x - z = 0)^{\perp}_{\mathrm{False}} \;\; , \\ & c_{0,3} : \; (z \neq 0)^{\perp}_{\mathrm{False}} \;\; , \\ & c_{0,4} : \; (y^2 < 0)^{\perp}_{\mathrm{False}} \;\; \} \\ & \\ & \emptyset \\ & \\ & \emptyset \end{array} \right)_{\perp}$$

and the infeasible subset of $C$

$$I \;:=\; \emptyset.$$

The algorithm starts to apply all substitutions in $D_0$, since its index is $\perp$, and chooses afterwards a variable to which the index of $D_0$ gets set. It has no substitution and chooses $x$ as index. Afterwards we take a condition, let us say $c_1$, set its flag to $\mathrm{True}$ and generate the test candidates of $c_1$ for $x$, i.e. $\infty$ and one finite test candidate, whose side conditions do not contain a variable-free inconsistent constraint, i.e. $0$ with the side conditions $0 = 0$, $1 \neq 0$. The algorithm creates two children of $D_0$, namely

$$D_1 = \left( \begin{array}{ll} \{ & c_{1,1} : \; (x = 0)^{c_{0,1}}_{\mathrm{False}} \;\; , \\ & c_{1,2} : \; (x - z = 0)^{c_{0,2}}_{\mathrm{False}} \;\; , \\ & c_{1,3} : \; (z \neq 0)^{c_{0,3}}_{\mathrm{False}} \;\; , \\ & c_{1,4} : \; (y^2 < 0)^{c_{0,4}}_{\mathrm{False}} \;\; \} \\ & \\ & \{[-\infty/x]\} \\ & \\ & \emptyset \end{array} \right)_{\perp}$$

and

$$D_2 = \left( \begin{array}{lll} \{ & c_{2,1} : & (x = 0)^{c_{0,1}}_{\text{False}} \quad , \\ & c_{2,2} : & (x - z = 0)^{c_{0,2}}_{\text{False}} \quad , \\ & c_{2,3} : & (z \neq 0)^{c_{0,3}}_{\text{False}} \quad , \\ & c_{2,4} : & (y^2 < 0)^{c_{0,4}}_{\text{False}} \\ & c_{2,5} : & (0 = 0)^{\perp}_{\text{False}} \\ & c_{2,6} : & (1 \neq 0)^{\perp}_{\text{False}} \quad \} \\ & & \\ & & \{[0/x]\} \\ & & \\ & & \emptyset \end{array} \right)_{\perp} .$$

Note that the conditions we added by reason of the side conditions get $\perp$ as original condition. We choose $D_1$ to continue and have to apply the substitution $[-\infty/x]$ to all conditions in $D_1$. Using the substitution rules of Appendix A, the algorithm replaces $D_1$ by the decision triple

$$D_3 = \left( \begin{array}{lll} \{ & c_{3,1} : & (1 \stackrel{\not}{=} 0)^{c_{0,1}}_{\text{False}} \quad , \\ & c_{3,2} : & (0 = 0)^{c_{0,1}}_{\text{False}} \quad , \\ & c_{3,3} : & (1 \stackrel{\not}{=} 0)^{c_{0,2}}_{\text{False}} \quad , \\ & c_{3,4} : & (-z = 0)^{c_{0,2}}_{\text{False}} \quad , \\ & c_{3,5} : & (z \neq 0)^{c_{0,3}}_{\text{False}} \quad , \\ & c_{3,6} : & (y^2 < 0)^{c_{0,4}}_{\text{False}} \quad \} \\ & & \\ & & \{[-\infty/x]\} \\ & & \\ & & \emptyset \end{array} \right)_{y} .$$

Note that $D_3$ contains variable-free conditions. We discard those, which are consistent. As $D_3$ has also variable-free inconsistent conditions, it is inconsistent as well and we apply the method to delete it given by Algorithm 21. The method creates the infeasible subsets consisting of one of the variable-free inconsistent conditions, i.e. $\{c_{3,1}\}$ and $\{c_{3,3}\}$. Then we fill up the conflict sets of the father of $D_3$, namely $D_0$, by the original conditions of these infeasible subsets, i.e. $\{c_{0,1}\}$ and $\{c_{0,2}\}$. We delete $D_3$, thus there are only two decision triples left, $D_2$ and

$$D_0 = \left( \begin{array}{lll} \{ & c_{0,1} : & (x = 0)^{\perp}_{\text{True}} \quad , \\ & c_{0,2} : & (x - z = 0)^{\perp}_{\text{False}} \quad , \\ & c_{0,3} : & (z \neq 0)^{\perp}_{\text{False}} \quad , \\ & c_{0,4} : & (y^2 < 0)^{\perp}_{\text{False}} \quad \} \\ & & \\ & & \emptyset \\ & & \\ & \{ & \{\{c_{0,1}\}, \{c_{0,2}\}\} \quad \} \end{array} \right)_{x} .$$

We choose $D_2$ to continue and substitute all occurrences of $x$ in its conditions by $0$ according to the substitution rules of Appendix A. Afterwards we replace $D_2$ by the only created decision triple,

$$D_4 = \left( \begin{array}{c} \{ \quad c_{4,1} : \ (0=0)^{c_{0,1}}_{\text{False}} \quad , \\ \quad\ c_{4,2} : \ (-z=0)^{c_{0,2}}_{\text{False}} \quad , \\ \quad\ c_{4,3} : \ (z\neq 0)^{c_{0,3}}_{\text{False}} \quad , \\ \quad\ c_{4,4} : \ (y^2<0)^{c_{0,4}}_{\text{False}} \quad \} \\ \\ \{[0/x]\} \\ \\ \emptyset \end{array} \right)_y .$$

The index is set to one of the variables occuring in the conditions of $D_4$. In the remaining of this example we first consider what happens by taking $y$ and later consider what happens by taking $z$.

So, let us say, the index is $y$: We continue with the evaluation of the just created decision triple $D_4$. We generate the test candidates of the condition $c_{4,4}$ for the variable $y$ and set the flag of $c_{4,4}$ to $\mathrm{True}$. In addition to $-\infty$, there is only one finite test candidate, whose side conditions are not variable-free and inconsistent, i.e. $0$ with side conditions $1 \neq 0, 0 \geq 0$ (see Chapter 4). Hence, we get two children of $D_4$:

$$D_5 = \left( \begin{array}{c} \{ \quad c_{5,1} : \ (1\neq 0)^{\bot}_{\text{False}} \quad , \\ \quad\ c_{5,2} : \ (0\geq 0)^{\bot}_{\text{False}} \quad , \\ \quad\ c_{5,3} : \ (-z=0)^{c_{4,2}}_{\text{False}} \quad , \\ \quad\ c_{5,4} : \ (z\neq 0)^{c_{4,3}}_{\text{False}} \quad , \\ \quad\ c_{5,5} : \ (y^2<0)^{c_{4,4}}_{\text{False}} \quad \} \\ \\ \{[0/x], [-\infty/y]\} \\ \\ \emptyset \end{array} \right)_{\bot}$$

$$D_6 = \left( \begin{array}{c} \{ \quad c_{6,1} : \ (1\neq 0)^{\bot}_{\text{False}} \quad , \\ \quad\ c_{6,2} : \ (0\geq 0)^{\bot}_{\text{False}} \quad , \\ \quad\ c_{6,3} : \ (-z=0)^{c_{4,2}}_{\text{False}} \quad , \\ \quad\ c_{6,4} : \ (z\neq 0)^{c_{4,3}}_{\text{False}} \quad , \\ \quad\ c_{6,5} : \ (y^2<0)^{c_{4,4}}_{\text{False}} \quad \} \\ \\ \{[0/x], [0/y]\} \\ \\ \emptyset \end{array} \right)_{\bot}$$

We continue with $D_5$ and apply an entire substitution step considering the substitution $[-\infty/y]$. There is just one condition containing $y$, namely $c_{5,5}$. The resulting cases of applying $[-\infty/y]$ to $y^2 < 0$ contain all variable-free inconsistent constraints, hence the

decision triples which replace $D_5$ have all variable-free inconsistent conditions. Each of these conditions has the original condition $c_{4,4}$, since it was the only condition to which we applied the substitution. Thus, we delete all these inconsistent decision triples and add for all of them a conflict set containing the set $\{c_{4,4}\}$ to $D_4$. The evaluation of $D_6$ goes similarly and results also in the addition of conflict sets containing the set $\{c_{4,4}\}$ to $D_4$. The conditions of $D_4$, which have not yet provided a test candidate for $y$, do not contain $y$, thus we set their flag to $\mathrm{True}$. Finally, we delete $D_4$ according to Algorithm 21, because it has just conditions with the flag $\mathrm{True}$ and no more children. All its conflict sets just consist of one set, namely $\{c_{4,4}\}$, hence the only minimal covering set is $\{c_{4,4}\}$ as well. It forms the only infeasible subset of the conditions of $D_4$, thus we add just one conflict set of $D_4$ in its father $D_0$ consisting of its original condition $c_{0,4}$. This condition does not contain the variable of the index of $D_0$, i.e. $x$, thus we apply backjumping: It discards all conflict sets in $D_0$ except the just added one $\{c_{0,4}\}$ and deletes $D_0$ according to Algorithm 21. We form all minimal covering sets of $D_0$'s conflict sets to achieve infeasible subsets of $D_0$'s conditions, resulting in one set, i.e. $\{c_{0,4}\}$. As $D_0$ is the root, we add the constraint in $c_{0,4}$ to the globally stored variable $I$. It forms an approximation of a minimal infeasible subset of $C$ and in this case even the smallest minimal infeasible subset. We delete all decision triples, which are part of the subtree of $T$ with root $D_0$, that is the entire tree $T$. Hence, the consistency check determines the inconsistency of $C$.

Let us now consider what happens, if we take $z$ as index: We generate the test candidates of $c_{4,2}$ for $z$ and set the flag of $c_{4,2}$ to $\mathrm{True}$. The algorithm creates the following two children of $D_4$:

$$D_7 = \left( \begin{array}{c} \{\ c_{7,1} : \ (-z = 0)^{c_{4,2}}_{\mathrm{False}}\ , \\ \quad c_{7,2} : \ (z \neq 0)^{c_{4,3}}_{\mathrm{False}}\ , \\ \quad c_{7,3} : \ (y^2 < 0)^{c_{4,4}}_{\mathrm{False}}\ \} \\ \\ \{[0/x], [-\infty/z]\} \\ \\ \emptyset \end{array} \right)_{\perp}$$

$$D_8 = \left( \begin{array}{c} \{\ c_{8,1} : \ (0 = 0)^{\perp}_{\mathrm{False}}\ , \\ \quad c_{8,2} : \ (1 \neq 0)^{\perp}_{\mathrm{False}}\ , \\ \quad c_{8,3} : \ (-z = 0)^{c_{4,2}}_{\mathrm{False}}\ , \\ \quad c_{8,4} : \ (z \neq 0)^{c_{4,3}}_{\mathrm{False}}\ , \\ \quad c_{8,5} : \ (y^2 < 0)^{c_{4,4}}_{\mathrm{False}}\ \} \\ \\ \{[0/x], [0/z]\} \\ \\ \emptyset \end{array} \right)_{\perp}$$

We continue with $D_7$ and apply its most recent substitution $[-\infty/z]$ entirely to its condi-

tions resulting in the decision tuples

$$D_9 = \left( \begin{array}{l} \{ \quad c_{9,1} : \ (-1 \overset{\lightning}{=} 0)^{c_{4,2}}_{\text{False}} \quad , \\ \quad\ \ c_{9,2} : \ (0 = 0)^{c_{4,2}}_{\text{False}} \quad , \\ \quad\ \ c_{9,3} : \ (1 \neq 0)^{c_{4,3}}_{\text{False}} \quad , \\ \quad\ \ c_{9,4} : \ (y^2 < 0)^{c_{4,4}}_{\text{False}} \quad \} \\ \\ \quad \{[0/x], [-\infty/z]\} \\ \\ \quad \emptyset \end{array} \right)_y$$

and

$$D_{10} = \left( \begin{array}{l} \{ \quad c_{10,1} : \ (-1 \overset{\lightning}{=} 0)^{c_{4,2}}_{\text{False}} \quad , \\ \quad\ \ c_{10,2} : \ (0 = 0)^{c_{4,2}}_{\text{False}} \quad , \\ \quad\ \ c_{10,3} : \ (0 \overset{\lightning}{\neq} 0)^{c_{4,3}}_{\text{False}} \quad , \\ \quad\ \ c_{10,4} : \ (y^2 < 0)^{c_{4,4}}_{\text{False}} \quad \} \\ \\ \quad \{[0/x], [-\infty/z]\} \\ \\ \quad \emptyset \end{array} \right)_y$$

which replace $D_7$. Both created decision triples are inconsistent and we delete them according to Algorithm 21, which fills up the conflict sets of $D_4$:

$$D_4 = \left( \begin{array}{l} \{ \quad c_{4,2} : \ (-z = 0)^{c_{0,2}}_{\text{True}} \quad , \\ \quad\ \ c_{4,3} : \ (z \neq 0)^{c_{0,3}}_{\text{False}} \quad , \\ \quad\ \ c_{4,4} : \ (y^2 < 0)^{c_{0,4}}_{\text{False}} \quad \} \\ \\ \quad \{[0/x]\} \\ \\ \{ \quad \{ \quad \{c_{4,2}\} \qquad\quad \} \\ \quad\ \ \{ \quad \{c_{4,2}\}, \ \{c_{4,3}\} \quad \} \quad \} \end{array} \right)_z .$$

We continue with the other child of $D_4$, namely $D_8$, and apply again the most recent substitution entirely. We replace $D_8$ by the resulting decision triple

$$D_{11} = \left( \begin{array}{l} \{ \quad c_{11,1} : \ (-0 = 0)^{c_{4,2}}_{\text{False}} \quad , \\ \quad\ \ c_{11,2} : \ (0 \overset{\lightning}{\neq} 0)^{c_{4,3}}_{\text{False}} \quad , \\ \quad\ \ c_{11,3} : \ (y^2 < 0)^{c_{4,4}}_{\text{False}} \quad \} \\ \\ \quad \{[0/x], [0/z]\} \\ \\ \quad \emptyset \end{array} \right)_y$$

and delete it afterwards, since it is inconsistent. It finally adds another conflict set to $D_4$, i.e. $\{\{c_{4,3}\}\}$. We have deleted all children of $D_4$, but there are still conditions in it, which can provide test candidates, namely $c_{4,3}$. The test candidate $-\infty$ has already been considered, thus we just generate the finite ones. There is only one with consistent side conditions, i.e. $0+\epsilon$ with the side conditions $0=0,\ -1\neq 0$. The resulting new children of $D_4$ is

$$D_{12} = \left( \begin{array}{c} \{\ c_{12,1}: \ (0=0)^{\perp}_{\mathrm{False}}\ , \\ \quad c_{12,2}: \ (-1\neq 0)^{\perp}_{\mathrm{False}}\ , \\ \quad c_{12,3}: \ (-z=0)^{c_{4,2}}_{\mathrm{False}}\ , \\ \quad c_{12,4}: \ (z\neq 0)^{c_{4,3}}_{\mathrm{False}}\ , \\ \quad c_{12,5}: \ (y^2<0)^{c_{4,4}}_{\mathrm{False}}\ \} \\ \\ \{[0/x],[0+\epsilon/z]\} \\ \\ \emptyset \end{array} \right)_{\perp}$$

Applying the substitution $[0+\epsilon/z]$ replaces $D_{12}$ by the decision triples

$$D_{13} = \left( \begin{array}{c} \{\ c_{13,1}: \ (-1 \overset{\lightning}{=} 0)^{c_{4,2}}_{\mathrm{False}}\ , \\ \quad c_{13,2}: \ (0=0)^{c_{4,2}}_{\mathrm{False}}\ , \\ \quad c_{13,3}: \ (1\neq 0)^{c_{4,3}}_{\mathrm{False}}\ , \\ \quad c_{13,4}: \ (y^2<0)^{c_{4,4}}_{\mathrm{False}}\ \} \\ \\ \{[0/x],[0+\epsilon/z]\} \\ \\ \emptyset \end{array} \right)_{y}$$

$$D_{14} = \left( \begin{array}{c} \{\ c_{14,1}: \ (-1 = 0)^{\lightning\ c_{4,2}}_{\mathrm{False}}\ , \\ \quad c_{14,2}: \ (0=0)^{c_{4,2}}_{\mathrm{False}}\ , \\ \quad c_{14,3}: \ (0 \overset{\lightning}{\neq} 0)^{c_{4,3}}_{\mathrm{False}}\ , \\ \quad c_{14,4}: \ (y^2<0)^{c_{4,4}}_{\mathrm{False}}\ \} \\ \\ \{[0/x],[0+\epsilon/z]\} \\ \\ \emptyset \end{array} \right)_{y}$$

which are both inconsistent and thus get deleted. The conflict sets we add by this to $D_4$ are $\{\{c_{4,2}\}\}$ and $\{\{c_{4,2}\},\{c_{4,3}\}\}$. Finally, $D_4$ has no children anymore. Its last condition having the flag $\mathrm{False}$ does not provide test candidates, hence we set its flag to $\mathrm{True}$ and

achieve

$$D_4 = \left( \begin{array}{l} \{\ c_{4,2} : \ (-z = 0)^{c_{0,2}}_{\text{True}}\ , \\ \quad\ c_{4,3} : \ (z \neq 0)^{c_{0,3}}_{\text{True}}\ , \\ \quad\ c_{4,4} : \ (y^2 < 0)^{c_{0,4}}_{\text{True}}\ \} \\ \\ \qquad \{[0/x]\} \\ \\ \{\ \{\ \{c_{4,2}\} \qquad\ \} \\ \quad\ \{\ \{c_{4,2}\},\ \{c_{4,3}\}\ \} \\ \quad\ \{\ \{c_{4,3}\} \qquad\ \} \\ \quad\ \{\ \{c_{4,2}\} \qquad\ \} \\ \quad\ \{\ \{c_{4,2}\},\ \{c_{4,3}\}\ \}\ \} \end{array} \right)_z .$$

The only minimal covering set of the conflict sets of $D_4$ is $\{c_{4,2}, c_{4,3}\}$. We delete $D_4$, which effects the conflict sets of $D_0$, such that it gets extended by the set containing the set consisting of the original conditions of $\{c_{4,2}, c_{4,3}\}$, i.e. $\{\{c_{0,2}, c_{0,3}\}\}$:

$$D_0 = \left( \begin{array}{l} \{\ c_{0,1} : \ (x = 0)^{\perp}_{\text{True}}\ , \\ \quad\ c_{0,2} : \ (x - z = 0)^{\perp}_{\text{False}}\ , \\ \quad\ c_{0,3} : \ (z \neq 0)^{\perp}_{\text{False}}\ , \\ \quad\ c_{0,4} : \ (y^2 < 0)^{\perp}_{\text{False}}\ \} \\ \\ \qquad \emptyset \\ \\ \{\ \{\ \{c_{0,1}\},\ \{c_{0,2}\}\ \} \\ \quad\ \{\ \{c_{0,2}, c_{0,3}\} \qquad\ \}\ \} \end{array} \right)_x$$

The conditions $c_{0,3}$ and $c_{0,4}$ do not contain the variable $x$ and thus do not provide further test candidates for it, except of $-\infty$, which we have already considered. We set their flags to $\text{True}$ and consider the last condition of $D_0$, which can provide test candidates for $x$, namely $c_{0,2}$. We set its flag to $\text{True}$ and create children of $D_0$ for its finite test candidates. There is only one finite test candidate with consistent side conditions, i.e. $z$ with the side conditions $0 = 0$, $1 \neq 0$. The created child is

$$D_{15} = \left( \begin{array}{l} \{\ c_{15,1} : \ (x = 0)^{c_{0,1}}_{\text{False}}\ , \\ \quad\ c_{15,2} : \ (x - z = 0)^{c_{0,2}}_{\text{False}}\ , \\ \quad\ c_{15,3} : \ (z \neq 0)^{c_{0,3}}_{\text{False}}\ , \\ \quad\ c_{15,4} : \ (y^2 < 0)^{c_{0,4}}_{\text{False}} \\ \quad\ c_{15,5} : \ (0 = 0)^{\perp}_{\text{False}} \\ \quad\ c_{15,6} : \ (1 \neq 0)^{\perp}_{\text{False}}\ \} \\ \\ \qquad \{[z/x]\} \\ \\ \qquad \emptyset \end{array} \right)_{\perp} .$$

We are not going to explain the remaining of the consistency check in detail, since it is similar to the evaluation of the subtree of $T$ with root $D_4$ (resp. $D_2$ before substitution),

if we also take $z$ as next variable to generate test candidates for. Finally, we add another conflict set to $D_0$, i.e. $\{c_{0,1}, c_{0,3}\}$, and the globally stored tree of decision triples $T$ now consists of only one decision triple, namely

$$D_0 = \left( \begin{array}{l} \{\ c_{0,1}: \ (x=0)^{\perp}_{\mathrm{True}}\ , \\ \quad\ c_{0,2}: \ (x-z=0)^{\perp}_{\mathrm{True}}\ , \\ \quad\ c_{0,3}: \ (z\neq 0)^{\perp}_{\mathrm{True}}\ , \\ \quad\ c_{0,4}: \ (y^2<0)^{\perp}_{\mathrm{True}}\ \} \\ \\ \qquad\qquad \emptyset \\ \\ \{\ \{\ \{c_{0,1}\},\ \{c_{0,2}\}\ \} \\ \quad\ \{\ \{c_{0,2}, c_{0,3}\}\ \} \\ \quad\ \{\ \{c_{0,1}, c_{0,3}\}\ \}\ \} \end{array} \right)_x .$$

All conditions of $D_0$ have the flag $\mathrm{True}$ and it has no children anymore. Hence, we delete $D$ according to Algorithm 21, which results in the termination of the consistency check. The infeasible subset of the constraints in $C$ contains the constraints occuring in one of the smallest minimal covering sets of the conflict sets $D_0$ had before deleting it. The only minimal covering set is $\{c_{0,1}, c_{0,2}, c_{0,3}\}$, thus $I = \{x = 0, x - z = 0, z \neq 0\}$, which is indeed a minimal infeasible subset of $C$.

In this example, we alternated the order of the variables to eliminate. As we could observe, it influences the consistency check, in particular the generation of the infeasible subsets. The first order provoked backjumping, which accelerated the consistency check significantly. It also achieved a smaller infeasible subset, i.e. $\{y^2 < 0\}$, than the second order achieved, i.e. $\{x = 0, x - z = 0, z \neq 0\}$. An analysis of the conditions and conflict sets of a decision triple can influence this order. This is one of the reasons, why we want to investigate the information stored in a decision triple further in order to speed up the consistency check.

# 7. Backtracking

The last interface the theory solver shall provide, is the ability to remove a subset of the constraints, which the theory solver has already checked, and to undo all results, which are effects of these constraints. We call this *backtracking*.

We already achieved a theory solver, which provides three interfaces: one to add constraints incrementally, one to check for consistency of all already added constraints, and one to get an infeasible subset of these constraints, if the consistency check determines inconsistency. This theory solver bases on the entire substitution approach and uses decision triples as data model. We extend this theory solver by an interface to perform backtracking. The resulting theory solver fulfills all requirements we asked for in Chapter 1. We do not consider the target oriented approach for backtracking, as we have already abandoned it before.

Until now, we delete a decision triple, if we know that it is inconsistent. This is reasonable for a consistency check and for later consistency checks after adding further constraints. However, it is not reasonable, if we also consider backtracking. Undoing the effects of the constraints we want to remove, results in removing conditions and test candidates. Removing a condition of an inconsistent decision triple can make it consistent. Thus, we have to remember inconsistent decision triples as well. From now on we consider deleting a decision triple as marking it as deleted instead of erasing it definitively. It assures that we can access its content and reconstruct it after removing some of its conditions.

## 7.1. Preconditions

Let us consider, that a consistency check of the set of constraints $Cons := \{c_1, \ldots, c_n\}$ according to Algorithm 15 results in its inconsistency. The SAT-solver receives an infeasible subset of $Cons$ by calling the method getInfSubSet() given by Algorithm 22. Then it analyzes the conflict for the clause this infeasible subset forms, which results in a conflict clause and a decision level $dl$ (see Algorithm 1). Afterwards, the SAT-solver performs a backtrack, which erases all decision levels higher than $dl$. As a result of this we undo the assignments, which belong to these decision levels. Those constraints, which have been assigned to $\mathrm{True}$ and are now unassigned by reason of this backtrack, form the subset of $Cons_b \subseteq Cons$ we want to remove from the theory solver.

Let the tree of decision triples $T = (V, E)$ be the result of the previous consistency check. All decision triples in $T$ are marked as deleted, since $Cons$ is inconsistent. Considering a decision triple $D = (C, S, M)_v \in V$, we want to identify its contents which are effects of the constraints in $Cons_b$.

## 7.2. Removing the effects of a constraint

In the following we collect the data being effects of the constraints in $Cons_b$ and remove them afterwards definitively. We find these effects recursively, starting with the root $R$ of $T$. The conditions in $R$, which correspond to the constraints of $Cons_b$ are obviously their effects. If we have the conditions of a decision triple $D = (C, S, M)_v \in V$ which are effects of $Cons_b$, i.e. $C_b \subseteq C$, we can find all effects of $Cons_b$ in the substree of $T$ with root $D$:

1. The conditions of $C_b$.
2. The test candidates except $-\infty$, which $C_b$ has provided. Let $T_b$ these test canidates.
3. The children of $D$, we created for a test candidate of $T_b$[1].
4. The conflict set of a decision triple, we created for a test candidate of $T_b$.
5. The sets in a conflict set, which contain a condition of $C_b$. These sets are the original conditions of infeasible subsets of $D$'s children. Deleting conditions from them, means that we also delete conditions in the corresponding infeasible subset, which can make it feasible. Thus we cannot assure that they are infeasible subsets and therefore their original conditions cannot be considered as part of a conflict set.
6. The conflict sets, whose sets got all deleted according the previous point.
7. The effects in the remaining children, if we consider their conditions, whose original conditions are effects of the constraints in $Cons$, as effects of the constraints in $Cons$.

Note that each child has at most one conflict set, but due to backjumping (see Section 6.3), there exist children, which do not concern any conflict set. Furthermore, we can mark all decision triples of a substree of $T$ with root in a decision triple, whose conflict set still exists in its father, as deleted. In point 4. we delete a conflict set, if in 3. all its elements got deleted. If any is left, it means, that the corresponding child is still inconsistent and that is why we can mark these decision triples.

We need to extend the definition of a decision triple, such that it provides the necessary information to reconstruct the effects of a subset of the constraints, which the theory solver checked. Firstly, we have to store the original condition of a substitution in order to be able to detect decision triples, which contain a substitution mapping to a test candidate provided by a condition we remove (see the third point above). Secondly, we must know which children of a decision triple correspond to its conflict sets. Each conflict set has one child it correponds to. This information helps us to identify a child, if we do not delete its conflict set and thus have to mark the child and all decision triples, which are reachable from it, as deleted. The entire redefinition of a decision triple is given by Definition 7.2.1.

[1] Note that until now we did not consider, that possibly the same test candidate not being $-\infty$ is generated more than once. Avoiding this will be one of the improvements we are going to embed in our implementation in future work.

**Definition 7.2.1 (Decision triples)**
*A* decision triple *has the following structure:*

$$\begin{pmatrix} Conditions \\ Substitutions \\ Conflict\_sets \end{pmatrix}_{index}$$

*where:*

- $Conditions$ *is a set of indexed constraints*

  $$cond \;=\; (p \sim 0)^{origin}_{flag}$$

  *where flag is defined as follows:*

  $$flag \;=\; \begin{cases} \text{True} & \text{, if the constraint was used} \\ & \text{to generate test candidates} \\ \text{False} & \text{, otherwise.} \end{cases}$$

  *and* $origin$ *is defined by:*

  $$origin \;=\; \begin{cases} \bot & \text{, if it is a side condition} \\ oCond & \text{, if } oCond \text{ is the original} \\ & \text{condition of } cond \end{cases}$$

- $Substitutions$ *is a set of substitutions of variables by test candidates indexed by its original condition:*

  $$[test\ candidate\ /\ variable]_{origin}.$$

  *where* $origin$ *is defined by:*

  $$origin \;=\; \begin{cases} \bot & \text{, if the test candidate is } -\infty \\ cond & \text{, if } cond \text{ is the condition} \\ & \text{providing the test candidate} \end{cases}$$

  *The variables of all substitutions in this set must be pairwise different.*

- *For each deleted child* $D_c$ $Conflict\_sets$ *is a set of subsets of* $Conditions$ *indexed by* $D_c$*. The subsets form the original conditions of an infeasible subset of* $D_c$*.*

- *The* $index$ *has the following definition:*

  $$index \;=\; \begin{cases} var & \text{, the variable for which the test candidates} \\ & \text{in this decision triple are generated} \\ \bot & \text{, if still no variable is determined to generate} \\ & \text{test candidates for.} \end{cases}$$

All algorithms of Chapter 6 have to be modified, such that they generate this new version of decision triples and that all operations, which delete nodes and edges of the global variable $T$, are replaced by operations, which mark those data structures as deleted. Definition 7.2.1 shows how to generate these decision triples, that is why we do not give further details in the form of modified versions of the already introduced algorithms. So it just remains the interface of the theory solver to provoke a backtracking. The method backtrack(Set_of_constraints $Cons_b$) forms this interface to the SMT-solver, where $Cons_b$ is a subset of the constraints, the theory solver checked. It calls another method with the root and its effected conditions as argument. This method propagates recursively from the root to the leaves and deletes along the way all effects of the constraints in $Cons_b$, which we detect in the way described above.

Algorithm 23 shows the pseudo code of this method and the submethod it involves.

## 7.3. Conclusion

This chapter has completed the interfaces we want a theory solver to provide. The previous chapters have shown that our theory solver maintains a tree of decision triples. All results we achieve during a consistency check are stored in it. Backtracking means, that we want to remove those results from it, which are effects of some of the constraints, the theory solver checked so far. Owing to the information a decision triple provides, we can retain many achieved results.

We remove exactly one information, which we could actually reuse in some cases. We refer to the sets in a conflict set, which contain at least one condition we remove. A set in a conflict set of a decision triple in its father, contains the original conditions of one of the infeasible subsets of the decision triple. If we remove a condition in such a set, we also have to remove conditions in the corresponding infeasible subset. Removing conditions of an infeasible subset possibly makes it feasible. That is why we remove the corresponding set in the conflict set. If it keeps being infeasible, we removed it unnecessarily. The only way to assure that we remove a set in a conflict set always rightly, is to assure that the corresponding infeasible subset is minimal. This leads us back to the generation of minimal infeasible subsets, which we cannot achieve with a justifiable effort.

The following example illustrates the redefined version of decision triples as well as the method to perform backtracking in all its facets.

**Algorithm 23** The algorithm to remove a given subset of the constraints the theory solver already checked. It removes not only the constraints, but also all their effects during previous consistency checks.

**void backtrack**(Constraints $Cons_b$)
**begin**
$I := \emptyset$; (1)
mark all $D \in V$ and $e \in E$ as not deleted; (2)
$D_r = (C_r, \emptyset, M_r)_{v_r}$ := the root of $T$; (3)
$C_b := \{(c)^{oCond}_{flag} \in C_r | c \in Cons_b\}$; (4)
backtrack($D_r$, $C_b$); (5)
**end**

**void backtrack**(Decision_triple $D = (C, S, M)_v \in V$, Conditions $C_b \subseteq C$)
**begin**
**for** all children $D_i = (C_i, S_i, M_i)_{v_i}$ of $D$ **do** (1)
$[t_i/v_i]$ := most recent substitution in $S_i$; (2)
**if** ($t_i$ provided by a constraint in $C_b$) **and** ($t_i \neq -\infty$) **then** (3)
$M := M \setminus \{N_{D_i}\}$; (4)
$T_{D_i} = (V_{D_i}, E_{D_i}) \overset{Def.\ 2.3.3}{:=}$ subtree of $T$ with root $D_i$; (5)
$T := (V := V \setminus V_{D_i}, E := E \setminus (E_{D_i} \cup \{(D, D_i)\}))$; (6)
**else if** $N_{D_i} \in M$ **then** (7)
**for** $J \in N_i$ **do** (8)
**if** $J \cap C_b \neq \emptyset$ **then** (9)
$N_i := N_i \setminus \{J\}$; (10)
**end if** (11)
**end for** (12)
**if** $N_i \neq \emptyset$ **then** (13)
$T_{D_i} = (V_{D_i}, E_{D_i}) \overset{Def.\ 2.3.3}{:=}$ subtree of $T$ with root $D_i$; (14)
mark all $\tilde{D} \in V_{D_i}$ and $e \in E$ as deleted; (15)
**else** (16)
$M := M \setminus \{(N_i, D_i)\}$; (17)
**end if** (18)
**end if** (19)
$C_{b,i} := \{(c)^{oCond}_{flag} \in C_i | oCond \in C_b\}$; (20)
backtrack($D_i$, $C_{b,i}$); (21)
**end for** (22)
$C := C \setminus C_b$; (23)
**end**

Global variables: Tree_of_decision_triples $T = (V, E)$, Infeasible_subset $I$;

## 7.4. Example

We illustrate backtracking by a small example. Let us say we checked consistency for the set of constraints $\{x = 0, x \neq 0\}$ using Algorithm 15 and the new version of decision triples introduced in Definition 7.2.1. It determines inconsistency and creates the tree of decision triples $T$ showed in Figure 7.1. All decision triples of $T$ are marked as deleted, which we indicate by crossing them out. Compared to the decision triples of the last chapter the conflict sets and the substitutions have indices. The index of a conflict set refers to the child, for which it was created. The index of a substitution refers to the condition, which has provided the test candidate in the substitution. The test candidate $-\infty$ is provided by each condition, that is why we index substitutions involving $-\infty$ by $\perp$.

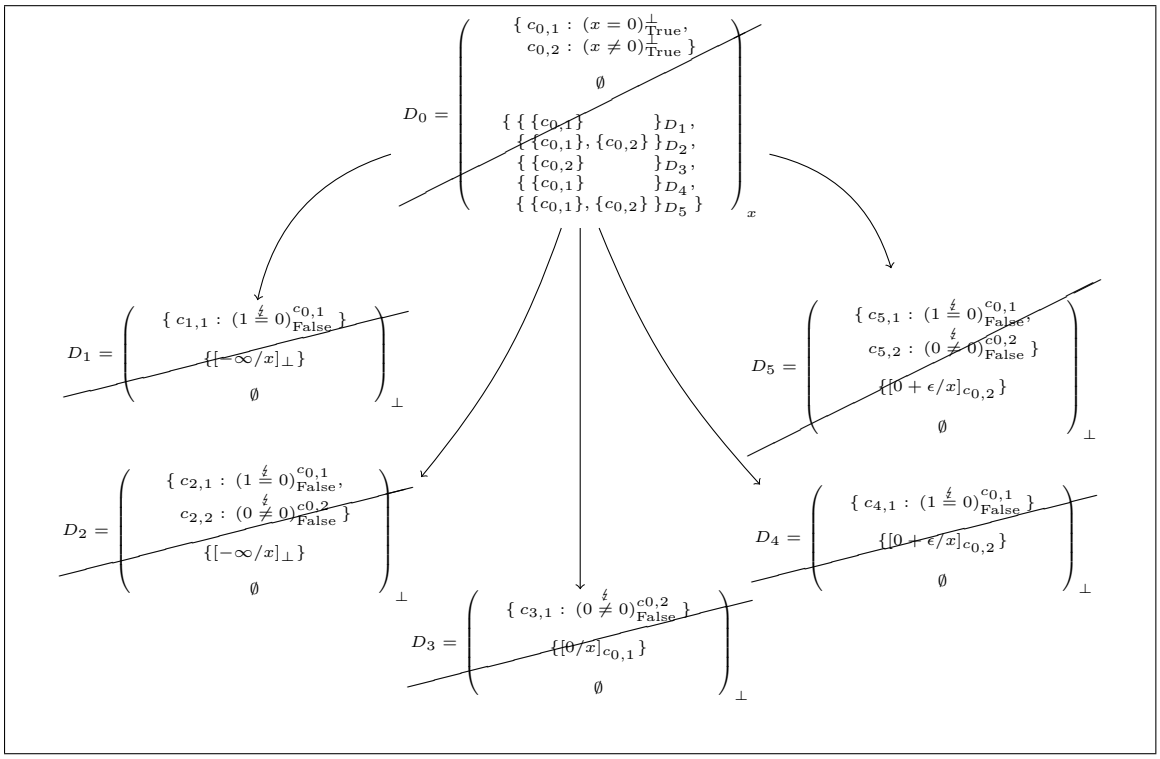


**Figure 7.1.:**

Now we call backtrack($\{x \neq 0\}$) in order to remove all effects the constraint $x \neq 0$ has in $T$. Algorithm 23 gives the pseudo code of the method we consider. We reset the globally stored infeasible subset $I$, mark all decision triples in $T$ as not deleted and call backtrack($\{c_{0,2}\}$, $D_0$), since $c_{0,2}$ is the only condition in the root $D_0$ of $T$, which $x \neq 0$ effects. For each child of $D_0$, we check, if its substitution refers to $c_{0,2}$. If so, we delete the child and its conflict set in $D_0$ definitively. Hence, we delete $D_4$ and $D_5$ together with

their conflict sets in $D_0$ definitively. The conflict sets in $D_0$ change in the following way:

$$\begin{array}{llllll}\{ & \{ & \{c_{0,1}\} & & \}_{D_1} & , \\ & \{ & \{c_{0,1}\}, & \{c_{0,2}\} & \}_{D_2} & , \\ & \{ & \{c_{0,2}\} & & \}_{D_3} & , \\ & \{ & \{c_{0,1}\} & & \}_{D_4} & , \\ & \{ & \{c_{0,1}\}, & \{c_{0,2}\} & \}_{D_5} & \}\end{array} \quad \rightarrow \quad \begin{array}{llllll}\{ & \{ & \{c_{0,1}\} & & \}_{D_1} & , \\ & \{ & \{c_{0,1}\}, & \{c_{0,2}\} & \}_{D_2} & , \\ & \{ & \{c_{0,2}\} & & \}_{D_3} & \}\end{array}$$

For each remaining conflict set in $D_0$ we delete a set in it definitively, if it contains $c_{0,2}$:

$$\begin{array}{llllll}\{ & \{ & \{c_{0,1}\} & & \}_{D_1} & , \\ & \{ & \{c_{0,1}\}, & \{c_{0,2}\} & \}_{D_2} & , \\ & \{ & \{c_{0,2}\} & & \}_{D_3} & \}\end{array} \quad \rightarrow \quad \begin{array}{llll}\{ & \{ & \{c_{0,1}\} & \}_{D_1} & , \\ & \{ & \{c_{0,1}\} & \}_{D_2} & , \\ & \{ & & \}_{D_3} & \}\end{array}$$

Afterwards we delete those conflict sets definitively, which are empty by reason of the previous step:

$$\begin{array}{llll}\{ & \{ & \{c_{0,1}\} & \}_{D_1} & , \\ & \{ & \{c_{0,1}\} & \}_{D_2} & , \\ & \{ & & \}_{D_3} & \}\end{array} \quad \rightarrow \quad \begin{array}{llll}\{ & \{ & \{c_{0,1}\} & \}_{D_1} & , \\ & \{ & \{c_{0,1}\} & \}_{D_2} & \}\end{array}$$

For all remaining children $D_i$, $1 \leq i \leq 3$, we call backtrack($C_b^i$, $D_i$), where $C_b^i$ are the conditions in $D_i$ with original condition $c_{0,2}$:

- $C_b^1 = \emptyset$
- $C_b^2 = \{c_{2,2}\}$
- $C_b^3 = \{c_{3,1}\}$

Each of these decision triples have neither children nor conflict sets. We just delete their conditions in $C_b^i$. Afterwards, we mark all decision triples as deleted, which occur in a the subtree of $T$, whose root is a decision triple, which has a conflict set in its father. In our case there are two decision tuples, which fulfill this property, namely $D_1$ and $D_2$. The subtree of $T$ with root $D_1$ resp. $D_2$, consists of $D_1$ resp. $D_2$, so we mark them as deleted. Finally, we delete $c_{0,2}$ from $D_0$ and achieve the final result shown in Figure 7.2.

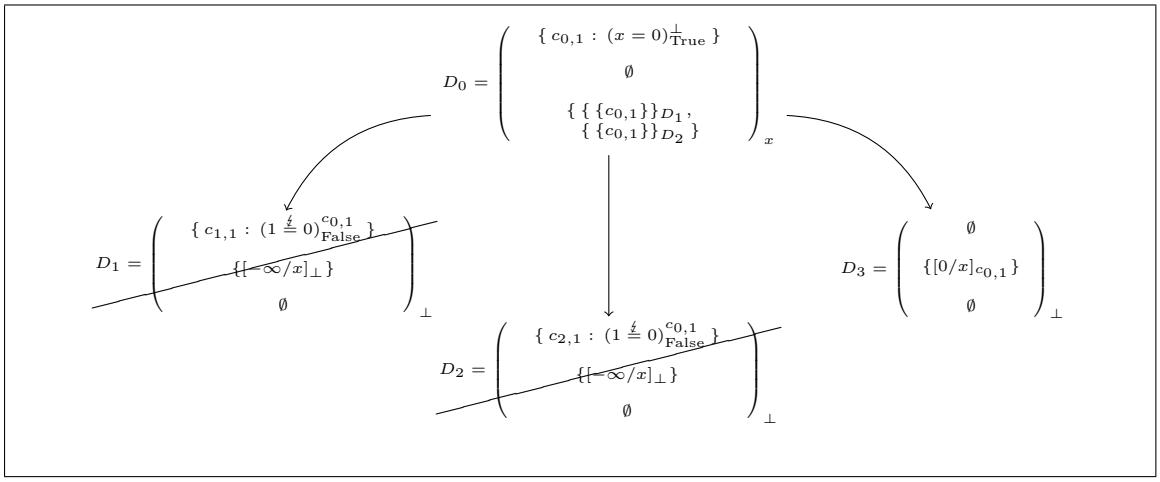
$D_0 = \left( \begin{array}{c} \{\, c_{0,1} : (x = 0)^{\bot}_{\mathrm{True}} \,\} \\ \emptyset \\ \{\, \{\, \{c_{0,1}\}\}_{D_1}, \\ \{\, \{c_{0,1}\}\}_{D_2} \,\} \end{array} \right)_x$
$D_1 = \left( \begin{array}{c} \{\, c_{1,1} : (1 \neq 0)^{c_{0,1}}_{\mathrm{False}} \,\} \\ \{[-\infty/x]_{\bot}\} \\ \emptyset \end{array} \right)_{\bot}$
$D_2 = \left( \begin{array}{c} \{\, c_{2,1} : (1 \neq 0)^{c_{0,1}}_{\mathrm{False}} \,\} \\ \{[-\infty/x]_{\bot}\} \\ \emptyset \end{array} \right)_{\bot}$
$D_3 = \left( \begin{array}{c} \emptyset \\ \{[0/x]_{c_{0,1}}\} \\ \emptyset \end{array} \right)_{\bot}$

**Figure 7.2.:**

# 8. Experimental results

We are currently building a prototype implementation using the proposed algorithms. Until now we have already completed an incremental theory solver using the entire substitution approach of Chapter 5. To get first results for the performance of an SMT-solver involving an incremental theory solver we embedded our implementation into an existing SMT-solver.

We want to construct an example with certain characteristics:

1. No condition with degree higher than 2 may be created during the consistency check.
2. The theory solver shall be called often.

To assure the first characteristic, we construct our example such that the set of constraints the SMT-solver passes to the theory solver does not contain more than one constraint, which is not linear. To assure the second characteristic, we take a CNF formula for our example, whose clauses consist of constraints. Only a few of the sets of constraints corresponding to the satisfying assignments of the boolean skeleton of this formula, shall be consistent.

We have created a random set of test formulas of the form

$$( x_a x_b = d ) \wedge \bigwedge_{(j_0,j_1,j_2,j_3,j_4,c)\in M} \bigvee_{i=0}^{4} ( x_{j_0} + x_{j_1} + x_{j_2} + x_{j_3} + x_{j_4} = c )$$

where $V = \{0,\ldots,19\}$, $V_c = \{1,\ldots,50\}$, $M \subseteq V^5 \times V_c$ with $|M| = 10$, the $x_l$ are variables for all $l \in V$, and $a$,$b$,$d \in V$ with $a \neq b$. The position of the clause $( x_a x_b = d )$ in this CNF-formula is determined randomly, i.e., it is not always the first clause. Table 8.1 shows results, which are characteristic for this kind of input formula. All listed example formulas are satisfiable.

The running times show the expected result. Less lazy solving is superior to the full lazy approach. Note that the example formulas are relatively small, thus less lazy solving has its main advantage in the facts that (1) in case a partial assignment already leads to a conflict, the theory solver needs to check smaller sets of constraints only, and (2) less lazy setting yields smaller conflicts. (As we do not yet support minimal infeasible subset generation, we take the disjunction of the negated unsatisfiable constraints as conflicting clause.)

Since we invoke the theory solver after each decision level handing over new constraints, the incremental version has to do less work than the non-incremental one. The running times show that the book-keeping effort pays off.

**Table 8.1.** First experimental results (times depicted in seconds) of the prototype implementation of the less lazy SMT-solver using an incremental virtual substitution solver.

| | | | | |
|---|---|---|---|---|
| Less lazy/ incremental: | 23.879 | 43.200 | 20.103 | 47.124 |
| Less lazy/ non-incremental: | 49.217 | 99.714 | 109.582 | 68.955 |
| Full lazy/ non-incremental: | 550.988 | 144.826 | 162.805 | 102.458 |
| `Redlog`: | 1147,889 | 2104.883 | 1850.559 | 251.474 |

# 9. Conclusion

In this thesis we have constructed step by step a theory solver based on the virtual substitution method and which fulfills the requirements

- to receive constraints and check their consistency incrementally,
- to generate an infeasible subset of them, if the consistency check fails, and
- to remove some of the received constraints, such that we benefit from results of previous consistency checks (backtracking).

During the development of such a theory solver, we recognized that there are a lot of similarities between our approach and state-of-the-art SAT-solving. The recent development of SAT-solving and its achievements foreshadow opportunities to similar achievements for our virtual substitution approach. We discovered such similarities in decision making, propagation, and conflict-driven non-chronological backtracking:

1. The choice of the next decision variable in SAT corresponds in our approach to the choice of the variable to eliminate next;
2. The choice of the decision variable's value in SAT corresponds to the choice of the test candidate we use for substitution next;
3. Propagation in SAT is analogous to the execution of the substitution;
4. We developed a backjumping mechanism, which is analogous to the conflict-driven backtracking in SAT.

There are further similarities we have not yet explored, e.g.,

5. *heuristics* to select (a) the next variable to eliminate, (b) the next condition to consider, and (c) the next test candidate for substitution,
6. *learning* from conflicts, and
7. *restarts*.

Investigating these approaches could lead to improvements of our virtual substitution method comparable to their impact on SAT-solving.

## 9.1. Theory solver

The data model, which serves to store intermediate results in the theory solver, forms the basis of our implementation. It provides the information we need in order to fulfill the above requirements. However, it provides a lot of more interesting features, which we want to understand and exploit. The most obvious features are the heuristics to decide, which decision triple to choose next, which variable to eliminate next and which condition to handle next. We also want to exploit the information we pass to the SAT-solver. Beside infeasible subsets, further information provided by the theory solver could influence the SAT-solver's decision heuristics.

Our implementation is restricted in the degree of the input constraints. Currently we cannot handle constraints with a degree higher than two. However, even if we cannot generate test candidates for these constraints, it would be possible to apply substitutions to constraints of arbitrary degree (see [Wei97]). The next step will be to extend the degree of the constraints for which we can generate test candidates to its theoritical maximum of $4$. Furthermore, we want to develop an incremental adaptation of the CAD method and to call this complete[1] but less efficient decision procedure in the case that the incremental implementation of the virtual substitution method cannot continue. In this case no further operations can be done for any decision tuples/triples, since the only conditions left for the generation of test candidates are of degree higher than $4$. We will also analyze if it is better to call the CAD implementation earlier and to continue the consistency check parallelly.

## 9.2. SMT-Solver

Our implementation still does not support the theoretically developed generation of infeasible subset nor backtracking. Nevertheless, the embedding of our incremental theory solver in an SMT-environment has already achieved promising results compared to the decision procedure `Redlog`, which does not involve a SAT-solver. However, the main benefits are situated in the generation of infeasible subsets.

Our implementation is embedded in a prototype SMT-solver, without any optimization. We plan a reimplementation using a state-of-the-art SAT-solver, such that it dovetails the single elements mentioned above. This includes not only the infeasible subset the theory solver passes to the SAT-solver, but also other influences the theory solver can have over the decision heuristics of the SAT-solver. It is also important, that we do not violate the modularity of these elements, such that we can replace or supplement them later. In addition, modularity provides the opportunity to parallelize e.g. the calls of the theory solver, which form the bottleneck of an SMT-solver.

[1] The CAD method can handle full real algebra.

# A. Substitution rules

This chapter shows all cases occuring when a substitution

$$[e/x]$$

is applied to a constraint

$$p(x) \ \sim \ 0 \quad \text{, with } p(x) \text{ a polynomial in } x.$$

The maximum degree of $x$ in $p(x)$ is $k$ and

$$\delta \ := \begin{cases} 1 & \text{, k is odd} \\ 0 & \text{, k is even} \end{cases} .$$

Transforming the substitution rules given by [Wei98] serves to avoid multiplying terms, which could increase the degrees of the resulting terms in the new created constraints. We achieve this at the price of increasing the number of resulting clauses.

## A.1. Substitution by a fraction

$$e \ = \ \frac{q}{r} \quad \text{with } q, \ r \ \text{ polynomials}$$

$p(x) \ = \ 0$**:**

$$p(e) * r^k = 0$$

$p(x) \ \neq \ 0$**:**

$$p(e) * r^k \neq 0$$

$p(x) \ < \ 0$**:**

$$\begin{array}{rl} & ( \quad r^\delta > 0 \quad \wedge \quad p(e) * r^k < 0 \quad ) \\ \vee & ( \quad r^\delta < 0 \quad \wedge \quad p(e) * r^k > 0 \quad ) \end{array}$$

$p(x) \ > \ 0$**:**

$$\begin{array}{rl} & ( \quad r^\delta > 0 \quad \wedge \quad p(e) * r^k > 0 \quad ) \\ \vee & ( \quad r^\delta < 0 \quad \wedge \quad p(e) * r^k < 0 \quad ) \end{array}$$

$p(x) \ \leq \ 0$**:**

$$\begin{array}{rl} & ( \quad r^\delta > 0 \quad \wedge \quad p(e) * r^k \leq 0 \quad ) \\ \vee & ( \quad r^\delta < 0 \quad \wedge \quad p(e) * r^k \geq 0 \quad ) \end{array}$$

$p(x) \ \geq \ 0$**:**

$$\begin{array}{rl} & ( \quad r^\delta > 0 \quad \wedge \quad p(e) * r^k \geq 0 \quad ) \\ \vee & ( \quad r^\delta < 0 \quad \wedge \quad p(e) * r^k \leq 0 \quad ) \end{array}$$

## A.2. Substitution by a square root term

Considering $e$ as a square root term, it has the form

$$e \;=\; \frac{q + r * \sqrt{t}}{s} \qquad \text{with } q,\ r,\ s,\ t \text{ polynomials.}$$

**Theorem A.2.1**
*Given are a polynomial $f(x)$ and an expression $e$ of the form*

$$e \;:=\; \frac{q + r\sqrt{t}}{s} \quad (*).$$

*Then $f(e)$ is of the form $(*)$.*

***Proof:*** *Polynomials have just the two operators plus and times. We show that both operations will map two expressions of the form $(*)$ to another expression, which again has this form. Keep in mind, that the radicand of both operands must be the same.*

1. *Addition of two expressions of the form $(*)$:*

$$\begin{aligned}
&\tfrac{q_1+r_1\sqrt{t}}{s_1} + \tfrac{q_2+r_2\sqrt{t}}{s_2} \\
=\;& \tfrac{s_2(q_1+r_1\sqrt{t})+s_1(q_2+r_2\sqrt{t})}{s_1 s_2} \\
=\;& \tfrac{s_2 q_1+s_2 r_1\sqrt{t}+s_1 q_2+s_1 r_2\sqrt{t}}{s_1 s_2} \\
=\;& \tfrac{(s_2 q_1+s_1 q_2)+(s_2 r_1+s_1 r_2)\sqrt{t}}{(s_1 s_2)}
\end{aligned}$$

2. *Multiplication of two expressions of the form $(*)$:*

$$\begin{aligned}
&\tfrac{q_1+r_1\sqrt{t}}{s_1} * \tfrac{q_2+r_2\sqrt{t}}{s_2} \\
=\;& \tfrac{(q_1+r_1\sqrt{t})(q_2+r_2\sqrt{t})}{s_1 s_2} \\
=\;& \tfrac{q_1 q_2+r_1\sqrt{t}q_2+q_1 r_2\sqrt{t}+r_1\sqrt{t}r_2\sqrt{t}}{s_1 s_2} \\
=\;& \tfrac{(q_1 q_2+r_1 r_2 t)+(r_1 q_2+q_1 r_2)\sqrt{t}}{(s_1 s_2)}
\end{aligned}$$

Hence, substituting all $x$ in $p(x)$ by $e$ leads according Proof A.2.1 to a square root term

$$p(e) \;=\; \frac{\hat{q} + \hat{r} * \sqrt{t}}{\hat{s}} \qquad \text{with } \hat{q},\ \hat{r},\ \hat{s} \text{ polynomials}$$

or, if $\hat{r} = 0$, to a fraction

$$p(e) \;=\; \frac{\hat{q}}{\hat{s}} \qquad \text{with } \hat{q},\ \hat{s} \text{ polynomials.}$$

In the latter case the substitution rules of Section A.1 hold; Otherwise the following rules define an equivalent real algebraic formula:

$p(x) \;=\; 0$**:**

$$\begin{array}{rcl} & & \hat{q} * \hat{r} \leq 0 \;\wedge\; \hat{q}^2 - \hat{r}^2 * t = 0 \\ \\ = & & \left(\begin{array}{lll} \hat{r} = 0 & \wedge & \hat{q} = 0 \end{array}\right) \\ & \vee & \left(\begin{array}{lll} \hat{q} = 0 & \wedge & t = 0 \end{array}\right) \\ & \vee & \left(\begin{array}{llll} \hat{q} < 0 & \wedge & \hat{r} > 0 & \wedge\; \hat{q}^2 - \hat{r}^2 * t = 0 \end{array}\right) \\ & \vee & \left(\begin{array}{llll} \hat{q} > 0 & \wedge & \hat{r} < 0 & \wedge\; \hat{q}^2 - \hat{r}^2 * t = 0 \end{array}\right) \end{array}$$

$p(x) \;\neq\; 0$**:**

$$\begin{array}{rcl} & & \left(\begin{array}{l} \hat{q} * \hat{r} > 0 \end{array}\right) \\ & \vee & \left(\begin{array}{l} \hat{q}^2 - \hat{r}^2 * t \neq 0 \end{array}\right) \\ \\ = & & \left(\begin{array}{lll} \hat{r} > 0 & \wedge & \hat{q} > 0 \end{array}\right) \\ & \vee & \left(\begin{array}{lll} \hat{r} < 0 & \wedge & \hat{q} < 0 \end{array}\right) \\ & \vee & \left(\begin{array}{l} \hat{q}^2 - \hat{r}^2 * t \neq 0 \end{array}\right) \end{array}$$

$p(x) \;<\; 0$**:**

$$\begin{array}{rcl} & & \left(\begin{array}{lll} \hat{q} * \hat{s}^\delta < 0 & \wedge & \hat{q}^2 - \hat{r}^2 * t > 0 \end{array}\right) \\ & \vee & \left(\begin{array}{lll} \hat{r} * \hat{s}^\delta \leq 0 & \wedge & \hat{q} * \hat{s}^\delta < 0 \end{array}\right) \\ & \vee & \left(\begin{array}{lll} \hat{r} * \hat{s}^\delta \leq 0 & \wedge & \hat{q}^2 - \hat{r}^2 * t < 0 \end{array}\right) \\ \\ = & & \left(\begin{array}{llll} \hat{q} < 0 & \wedge & \hat{s}^\delta > 0 & \wedge\; \hat{q}^2 - \hat{r}^2 * t > 0 \end{array}\right) \\ & \vee & \left(\begin{array}{llll} \hat{q} > 0 & \wedge & \hat{s}^\delta < 0 & \wedge\; \hat{q}^2 - \hat{r}^2 * t > 0 \end{array}\right) \\ & \vee & \left(\begin{array}{llll} \hat{r} \geq 0 & \wedge & \hat{q} < 0 & \wedge\; \hat{s}^\delta > 0 \end{array}\right) \\ & \vee & \left(\begin{array}{llll} \hat{r} \leq 0 & \wedge & \hat{q} > 0 & \wedge\; \hat{s}^\delta < 0 \end{array}\right) \\ & \vee & \left(\begin{array}{llll} \hat{r} > 0 & \wedge & \hat{s}^\delta < 0 & \wedge\; \hat{q}^2 - \hat{r}^2 * t < 0 \end{array}\right) \\ & \vee & \left(\begin{array}{llll} \hat{r} < 0 & \wedge & \hat{s}^\delta > 0 & \wedge\; \hat{q}^2 - \hat{r}^2 * t < 0 \end{array}\right) \end{array}$$

$p(x) > 0$**:**

$$\begin{array}{rcl}
 & ( & \hat{q} * \hat{s}^\delta > 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t > 0 \ ) \\
\vee & ( & \hat{r} * \hat{s}^\delta \geq 0 \ \wedge \ \hat{q} * \hat{s}^\delta > 0 \ ) \\
\vee & ( & \hat{r} * \hat{s}^\delta \geq 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t < 0 \ )
\end{array}$$

$$\begin{array}{rcl}
= & ( & \hat{q} > 0 \ \wedge \ \hat{s}^\delta > 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t > 0 \ ) \\
\vee & ( & \hat{q} < 0 \ \wedge \ \hat{s}^\delta < 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t > 0 \ ) \\
\vee & ( & \hat{r} \leq 0 \ \wedge \ \hat{q} < 0 \ \wedge \ \hat{s}^\delta < 0 \ ) \\
\vee & ( & \hat{r} \geq 0 \ \wedge \ \hat{q} > 0 \ \wedge \ \hat{s}^\delta > 0 \ ) \\
\vee & ( & \hat{r} > 0 \ \wedge \ \hat{s}^\delta > 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t < 0 \ ) \\
\vee & ( & \hat{r} < 0 \ \wedge \ \hat{s}^\delta < 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t < 0 \ )
\end{array}$$

$p(x) \leq 0$**:**

$$\begin{array}{rcl}
 & ( & \hat{q} * \hat{s}^\delta \leq 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t \geq 0 \ ) \\
\vee & ( & \hat{r} * \hat{s}^\delta \leq 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t \leq 0 \ )
\end{array}$$

$$\begin{array}{rcl}
= & ( & \hat{q} < 0 \ \wedge \ \hat{s}^\delta > 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t \geq 0 \ ) \\
\vee & ( & \hat{q} > 0 \ \wedge \ \hat{s}^\delta < 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t \geq 0 \ ) \\
\vee & ( & \hat{r} = 0 \ \wedge \ \hat{q} = 0 \ ) \\
\vee & ( & \hat{q} = 0 \ \wedge \ t = 0 \ ) \\
\vee & ( & \hat{r} > 0 \ \wedge \ \hat{s}^\delta < 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t \leq 0 \ ) \\
\vee & ( & \hat{r} < 0 \ \wedge \ \hat{s}^\delta > 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t \leq 0 \ )
\end{array}$$

$p(x) \geq 0$**:**

$$\begin{array}{rcl}
 & ( & \hat{q} * \hat{s}^\delta \geq 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t \geq 0 \ ) \\
\vee & ( & \hat{r} * \hat{s}^\delta \geq 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t \leq 0 \ )
\end{array}$$

$$\begin{array}{rcl}
= & ( & \hat{q} > 0 \ \wedge \ \hat{s}^\delta > 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t \geq 0 \ ) \\
\vee & ( & \hat{q} < 0 \ \wedge \ \hat{s}^\delta < 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t \geq 0 \ ) \\
\vee & ( & \hat{r} = 0 \ \wedge \ \hat{q} = 0 \ ) \\
\vee & ( & \hat{q} = 0 \ \wedge \ t = 0 \ ) \\
\vee & ( & \hat{r} > 0 \ \wedge \ \hat{s}^\delta > 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t \leq 0 \ ) \\
\vee & ( & \hat{r} < 0 \ \wedge \ \hat{s}^\delta < 0 \ \wedge \ \hat{q}^2 - \hat{r}^2 * t \leq 0 \ )
\end{array}$$

## A.3. Substitution by a term plus an infinitesimal

Substitution by $[e + \epsilon/x]$:

$bx + c = 0$**:**

$$b = 0 \ \wedge \ c = 0$$

$bx + c \neq 0$**:**

$$\begin{array}{ll} & b \neq 0 \\ \vee & c \neq 0 \end{array}$$

$bx + c < 0$**:**

$$\begin{array}{lllll} & ( & (bx + c < 0)[e/x] & & ) \\ \vee & ( & (bx + c = 0)[e/x] & \wedge \; (b < 0)[e/x] & ) \end{array}$$

$bx + c > 0$**:**

$$\begin{array}{lllll} & ( & (bx + c > 0)[e/x] & & ) \\ \vee & ( & (bx + c = 0)[e/x] & \wedge \; (b > 0)[e/x] & ) \end{array}$$

$bx + c \leq 0$**:**

$$\begin{array}{lllll} & ( & (bx + c < 0)[e/x] & & ) \\ \vee & ( & (bx + c = 0)[e/x] & \wedge \; (b < 0)[e/x] & ) \\ \vee & ( & b = 0 & \wedge \; c = 0 & ) \end{array}$$

$bx + c \geq 0$**:**

$$\begin{array}{lllll} & ( & (bx + c > 0)[e/x] & & ) \\ \vee & ( & (bx + c = 0)[e/x] & \wedge \; (b > 0)[e/x] & ) \\ \vee & ( & b = 0 & \wedge \; c = 0 & ) \end{array}$$

$ax^2 + bx + c = 0$**:**

$$a = 0 \;\wedge\; b = 0 \;\wedge\; c = 0$$

$ax^2 + bx + c \neq 0$**:**

$$\begin{array}{ll} & a \neq 0 \\ & b \neq 0 \\ \vee & c \neq 0 \end{array}$$

$ax^2 + bx + c < 0$**:**

$$\begin{array}{lllllll} & ( & (ax^2 + bx + c < 0)[e/x] & & & & ) \\ \vee & ( & (ax^2 + bx + c = 0)[e/x] & \wedge & (2ax + b < 0)[e/x] & & ) \\ \vee & ( & (ax^2 + bx + c = 0)[e/x] & \wedge & (2ax + b = 0)[e/x] & \wedge \; (2a < 0)[e/x] & ) \end{array}$$

$ax^2 + bx + c > 0$**:**

$$\begin{array}{lllllll} & ( & (ax^2 + bx + c > 0)[e/x] & & & & ) \\ \vee & ( & (ax^2 + bx + c = 0)[e/x] & \wedge & (2ax + b > 0)[e/x] & & ) \\ \vee & ( & (ax^2 + bx + c = 0)[e/x] & \wedge & (2ax + b = 0)[e/x] & \wedge \; (2a > 0)[e/x] & ) \end{array}$$

$ax^2 + bx + c \leq 0$**:**

$$\begin{array}{lllllll} & ( & (ax^2 + bx + c < 0)[e/x] & & & & ) \\ \vee & ( & (ax^2 + bx + c = 0)[e/x] & \wedge & (2ax + b < 0)[e/x] & & ) \\ \vee & ( & (ax^2 + bx + c = 0)[e/x] & \wedge & (2ax + b = 0)[e/x] & \wedge \; (2a < 0)[e/x] & ) \\ \vee & ( & a = 0 & \wedge & b = 0 & \wedge \; c = 0 & ) \end{array}$$

$ax^2 + bx + c \geq 0$:

$$\begin{array}{llllllll} & ( & (ax^2+bx+c>0)[e/x] & & & & & ) \\ \vee & ( & (ax^2+bx+c=0)[e/x] & \wedge & (2ax+b>0)[e/x] & & & ) \\ \vee & ( & (ax^2+bx+c=0)[e/x] & \wedge & (2ax+b=0)[e/x] & \wedge & (2a>0)[e/x] & ) \\ \vee & ( & a=0 & \wedge & b=0 & \wedge & c=0 & ) \end{array}$$

## A.4. Substitution by minus infinity

Substitution by $[-\infty/x]$:

$bx + c = 0$:

$$b = 0 \quad \wedge \quad c = 0$$

$bx + c \neq 0$:

$$\begin{array}{ll} & b \neq 0 \\ \vee & c \neq 0 \end{array}$$

$bx + c < 0$:

$$\begin{array}{lllll} & ( & b>0 & & ) \\ \vee & ( & b=0 & \wedge \quad c<0 & ) \end{array}$$

$bx + c > 0$:

$$\begin{array}{lllll} & ( & b<0 & & ) \\ \vee & ( & b=0 & \wedge \quad c>0 & ) \end{array}$$

$bx + c \leq 0$:

$$\begin{array}{lllll} & ( & b>0 & & ) \\ \vee & ( & b=0 & \wedge \quad c\leq 0 & ) \end{array}$$

$bx + c \geq 0$:

$$\begin{array}{lllll} & ( & b<0 & & ) \\ \vee & ( & b=0 & \wedge \quad c\geq 0 & ) \end{array}$$

$ax^2 + bx + c = 0$:

$$a = 0 \quad \wedge \quad b = 0 \quad \wedge \quad c = 0$$

$ax^2 + bx + c \neq 0$:

$$\begin{array}{ll} & a \neq 0 \\ \vee & b \neq 0 \\ \vee & c \neq 0 \end{array}$$

$ax^2 + bx + c < 0$:

$$\begin{array}{llllll} & ( & a<0 & & & ) \\ \vee & ( & a=0 & \wedge \quad b>0 & & ) \\ \vee & ( & a=0 & \wedge \quad b=0 & \wedge \quad c<0 & ) \end{array}$$

$ax^2 + bx + c \ > \ 0$**:**

$$\begin{array}{lllllll} & ( & a > 0 & & & & ) \\ \vee & ( & a = 0 & \wedge & b < 0 & & ) \\ \vee & ( & a = 0 & \wedge & b = 0 & \wedge \quad c > 0 & ) \end{array}$$

$ax^2 + bx + c \ \leq \ 0$**:**

$$\begin{array}{lllllll} & ( & a < 0 & & & & ) \\ \vee & ( & a = 0 & \wedge & b > 0 & & ) \\ \vee & ( & a = 0 & \wedge & b = 0 & \wedge \quad c \leq 0 & ) \end{array}$$

$ax^2 + bx + c \ \geq \ 0$**:**

$$\begin{array}{lllllll} & ( & a > 0 & & & & ) \\ \vee & ( & a = 0 & \wedge & b < 0 & & ) \\ \vee & ( & a = 0 & \wedge & b = 0 & \wedge \quad c \geq 0 & ) \end{array}$$

# Bibliography


[BD07] C. W. Brown and J. H. Davenport. The complexity of quantifier elimination and cylindrical algebraic decomposition. In *ISSAC '07: Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 54–60. ACM, 2007.

[Boo] *Decision Procedures - An Algorithmic Point of View*. Springer-Verlag.

[BPT07] A. Bauer, M. Pister, and M. Tautschnig. Tool-support for the analysis of hybrid systems and models. In *DATE '07: Proceedings of the Conference on Design, Automation and Test*, pages 924–929. European Design and Automation Association, 2007.

[Bro03] C. W. Brown. QEPCAD B: A program for computing with semi-algebraic sets using CADs. *SIGSAM Bull.*, 37:97–108, 2003.

[CJ98] B. F. Caviness and J. R. Johnson, editors. *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Texts and Monographs in Symbolic Computation. Springer-Verlag, 1998.

[DH88] J. H. Davenport and J. Heinz. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5:29–35, 1988.

[dMB08] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08: Tools and Algorithms for the Construction and Analysis*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer-Verlag, 2008.

[DS97] A. Dolzmann and T. Sturm. REDLOG: Computer algebra meets computer logic. *SIGSAM'97: Bulletin Special Interest Group on Symbolic and Algebraic Manipulation*, 31:2–9, 1997.

[DSW97] A. Dolzmann, T. Sturm, and V. Weispfenning. Real quantifier elimination in practice, 1997.

[FHT+07] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *JSAT*, 1:209–236, 2007.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC'01: Annual ACM IEEE Design Automation Conference*, pages 530–535. ACM, 2001.

[Tar48] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1948.

[Tse83] G. Tseitin. On the complexity of proofs in propositional logics. In *Automation of Reasoning: Classical Papers in Computational Logics 1967-1970*, volume 2. Springer-Verlag, 1983.

[Wei88] V. Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5:3–27, 1988.

[Wei97] V. Weispfenning. Quantifier elimination for real algebra - the quadratic case and beyond. *Appl. Algebra Eng. Commun. Comput*, 8:85–101, 1997.

[Wei98] V. Weispfenning. A new approach to quantifier elimination for real algebra. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Texts and Monographs in Symbolic Computation, pages 376–392. Springer-Verlag, 1998.