Universiteit Utrecht

Faculty of Science

Techniques behind SMT solvers

Bachelor thesis

Tomas Ehrencron

Mathematics

Supervisors:

Prof. Dr. J.T. Jeuring
Department of Information and Computing Sciences

Prof. Dr. J. van Oosten
Department of Mathematics

Abstract

SMT problems form an extensive collection of satisfiability problems. Determining the satisfiability of a Boolean expression lies at the basis of this set and forms the backbone of the P versus NP problem. We study several existing algorithms that solve satisfiability. Another common SMT problem are systems of linear integer inequalities. We look at a few ways these can be solved. Finally, we show how Liquid Haskell uses an SMT solver to verify refinement types.

July 11, 2018

Contents

1 Introduction

In this thesis we look at a two type of problems of the satisfiability modulo theories (SMT). The first one is the set of Boolean satisfiability problems. In general, these problems all express whether we can find values for the variables in a Boolean expression such that it evaluates to true. The second type consists of problems involving linear arithmetic. We discuss a few algorithms that solve these types of problems and we look at an application of solvers in Liquid Haskell.

In Section 3 we will do some housekeeping regarding satisfiability. We give a definition of the exact problem and define some tools and theorems to measure the length of expressions. Then we look at an important conversion of expressions that is essential for solving the satisfiability problem. We discuss two ways this conversion can be done and argue which one is used more commonly.

Section 4 discusses the advancements made in solving the satisfiability problem. First we focus on some general rules to simplify expressions. Then we look at the Davis-Putnam-Logemann-Loveland algorithm (DPLL) that takes an important place in solving satisfiability. Over the years several algorithms have been developed that are based on this algorithm [10, 14]. We will focus on the GRASP algorithm [13] that also heavily relies on DPLL and some improvements on it.

In Section 5 we introduce SMT-LIB2, a standard that all current SMT solvers follow, and we explain the overal structure of it. We then study two algorithms that solve systems of (integer) linear inequality. The first one, Gomory's cut, extends the simplex algorithm to integers. The second algorithm uses the Fourier-Motzkin elimination to resolve variables from our system, but does not work for integer systems (with integer variables). The Omega test extends the elimination to integer.

Lastly, we look at an application of SMT solvers in Liquid Haskell, a tool to verify properties of functions in Haskell. We will use a concept called refinement types to describe these properties and see in an example how Liquid Haskell translates them to code that an SMT solver can verify.

2 Preliminaries

In this thesis we will often use *Boolean expression*, an expression that is composed of Boolean operations and variables. This includes the operations $\neg$ (negation), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implication) and $\leftrightarrow$ (equivalence) as well as the quantifiers $\forall$ (for all) and $\exists$ (there exists). Furthermore, since $\wedge$ and $\vee$ are both commutative and associative, we use the iterated binary operations $\bigwedge$ and $\bigvee$ respectively. We denote the set of variables in an expression as $X = \{x_1, \ldots, x_n\}$. Furthermore, we call an expression ψ a *subexpression* of an expression ϕ if ψ occurs somewhere in ϕ. As an example we take $\psi = x_1 \wedge x_2$ and see that this is a subexpression of $\phi = (x_1 \wedge x_2) \leftrightarrow (x_2 \rightarrow x_3)$.

We will also use the symbol ($\vdash$) to denote inference. That is, $a \vdash b$ means that b can be derived from a. For example, the rule of contraposition can be described as $p \rightarrow q \vdash \neg q \rightarrow \neg p$.

For convenience, we define a Boolean variable x as element of $\{0, 1\}$ instead of the usual {FALSE, TRUE}. Since we will often use Boolean variables in this thesis, we will not define them every time. Similarly, an expression is a Boolean expression unless specified otherwise.

We use the Big O notation to denote asymptotic upper bounds for functions. We use the following definition for this:

Definition 2.0.1. We write $f(n) = \mathcal{O}(g(n))$ to denote that there exist constants C and N such that for all $n \geq N$, $f(n) \leq Cg(n)$. Intuitively, this means that g is an upperbound of f for large values of n.

In some places, we use the SMT-LIB2 syntax of SMT solvers. Although the syntax will be explained and is not too complicated, the syntax uses s-expressions. This means that all functions, including binary operators, are exclusively in prefix notation. This means that the conjunction between two Boolean variables `x` and `y` is written as `'and x y'` instead of `'x and y'`. Although the semantics is exactly the same, it may decrease the readability of an expression. Moreover, there is no difference between normal and iterated binary operators. A conjunction of three variables is simply written as `'and x y z'`.

3 Satisfiability

The satisfiability problem (SAT) is a well known problem in computer science. Given a Boolean expression, it describes whether the expression is satisfiable. The problem can be formulated formally as follows:

Definition 3.0.1. (Satisfiability). Given a Boolean expression with X as the set of variables, we say that the expression is *satisfiable*, if there exist assignments to the variables such that the expression evaluates to 1.

Remark 3.0.2. We only allow expressions of propositional logic as opposed to predicate logic. This means that we do not include the universal and existential quantifiers $\forall$ and $\exists$. If we would introduce them in an expression, it would greatly increase the complexity of the satisfiability problem. Therefore expressions with quantifiers are often classified as quantified Boolean formulas (QBF). Evaluating if such an expression is true (a decision problem that is called TQBF), is a problem that is PSPACE complete [12]. This complexity class contains all problems that can be solved with a polynomial amount of space and unlimited time.

3.1 Definitions and conventions

Before we look at ways to solve SAT, we need to introduce a few definitions.

Definition 3.1.1. (Literal). Let x be a Boolean variable. A literal in an expression is defined as a variable x or its negation $\neg x$.

Definition 3.1.2. (Size) We define the size of an expression ϕ as the number of binary operators in ϕ. Notation: $|\phi|$. For example, $|\neg(x_1 \wedge x_2)| = 1$ and $|(x_1 \leftrightarrow (\neg x_2 \vee x_3))| = 2$. We also define the extended size as the total number of operations in ϕ notated as $|\phi|_2$.

Theorem 3.1.3. *For an expression ϕ, we define $f_L(\phi)$ as the number of literals in ϕ. Note that the same literal can occur multiple times in an expression, but we count them separately. It holds that $|\phi| + 1 = f_L(\phi)$.*

Proof. We present a proof by induction. For a single literal l, it is evident that $|l|+1 = 1 = f_L(l)$. We already saw that adding negation does not increase the size and the number of literals. We now look at an expression of the form $\psi_1 \diamond \psi_2$ where $\diamond$ is an arbitrary binary operator. We presume that the theorem holds for ψ_1 and ψ_2. We see that $|\psi_1| + 1 = n_1 + 1 = f_L(\psi_1)$ and $|\psi_2| + 1 = n_2 + 1 = f_L(\psi_2)$. Then it holds that $|\psi_1 \diamond \psi_2| + 1 = (n_1 + n_2 + 1) + 1 = (n_1 + 1) + (n_2 + 1) = f_L(\psi_1) + f_L(\psi_2) = f_L(\psi_1 \diamond \psi_2)$. We can write this as $f_L(\phi) = \mathcal{O}(|\phi|)$. □

There are several arguments for why we do not include negation in the size. The first one is that we want to measure the complexity of an expression. Negation does not combine multiple subexpressions together. Also the number of literals in the expression does not increase with a negation. Secondly, we could theoretically define an arbitrary number of negations after another like this:

$$\underbrace{\neg \dots \neg}_{2n} x \equiv x$$

Since we can clearly remove these negations, including them has a counterintuitive consequence for the definition of the size. Since we can only add one negation to every subexpression and variable, it follows from Theorem 3.1.3 that the number of negations that we can add, is linear in the size of an expression.

Lastly, most SAT solvers work with CNF (Section 3.3). We will see that the only negations that occur in such an expression are in literals.

We defined the extended size, because in some cases we want to identify the number of subexpressions in an expression. Nevertheless, we need a relation between the two sizes. Therefore, we now prove that the extended size is at most a constant factor larger than the normal size (the extended size of an expression grows linear in its normal size).

Theorem 3.1.4. *For every expression ϕ without double negations it holds that $|\phi|_2 \leq 3\,|\phi| + 1$.*

Proof. By definition, we know that $|\phi|_2 = |\phi| + f_N(\phi)$ where $f_N(\phi)$ is the number of negations in ϕ. Since we assumed that there are no multiple negations after another, there is at most one negation for every variable and subexpression of binary variables. From Theorem 3.1.3 it follows that $f_N(\phi) \leq |\phi| + (|\phi| + 1)$. Therefore, we see that

$$|\phi|_2 = |\phi| + f_N(\phi) \leq 3\,|\phi| + 1$$

Thus we see that $|\phi|_2 = \mathcal{O}(|\phi|)$ □

It is fairly obvious that checking whether a given assignment to the variables yields an expression that evaluates to 1 is an easy task. The difficulty of SAT however becomes clear in the next example.

Example 3.1.5. We consider the following expression:

$$((x_1 \to x_2) \vee (\neg x_2 \wedge x_3)) \leftrightarrow (x_1 \wedge \neg x_2 \wedge \neg x_3). \tag{1}$$

It is not immediately evident that there do not exist any values for x_1, x_2 and x_3 to satisfy 1. The most simple way is to generate the truth table to check all possibilities. For expressions that are easily satisfiable, this method will probably suffice. But unsatisfiable expressions with n variables require us to look at all 2^n combinations. △

3.2 Relevance

The satisfiability problem takes a crucial place in the P versus NP problem. Intuitively, if this last problem is true, it means that many problems that now only have exponential algorithms, could be solved in polynomial time. In 1971 Stephen Cook proved that any such problem can be reduced to the satisfiability problem [5]. As a consequence, all these problems can be solved in polynomial time, if we were to prove that SAT is. This makes it a key element in a major problem in computer science. Although it is currently not known whether there exist a polynomial algorithm to solve this problem, we can improve substantially on the brute-force algorithm. In the next section we will discuss a few algorithms that solve SAT much faster than the brute-force method. That is, in practice it is much more efficient, but worst-case the algorithm still has an exponential time performance.

3.3 CNF

The first step in all algorithms is converting the expression to conjunctive normal form (CNF). The problem that remains is checking satisfiability for an expression in CNF, a problem that is called CNF-SAT.

Definition 3.3.1. (Clause). An expression is called a *clause*, if it is disjunction ($\vee$) of literals.

Definition 3.3.2. (Conjunctive Normal Form, CNF) We say that an expression is in *CNF*, if it is a conjunction ($\wedge$) of clauses.

Example 3.3.3. Here are some examples of expressions in CNF:

- $x_1 \wedge x_2 \wedge \neg x_3$
- $(x_1 \vee \neg x_2) \wedge x_3$
- $\neg x_1$

and a few examples that are not in CNF:

- $\neg(x_1 \wedge x_2)$
- $x_1 \wedge (x_2 \vee (x_3 \wedge x_4))$

△

The reason behind this transformation is that we can make more assumptions about an expression, when it is in CNF. It is also easier to determine the value of an expression. When one of the clauses evaluates to 0, the whole expression is 0. The algorithms discussed in Section 4 highly depend on this property.

There are other forms in which satisfiability can be solved much quicker than CNF. For example, the disjunctive normal form (DNF), a disjunction of conjunctions of literals. If one of the conjuctions is 1, then the whole expression is 1. Checking of a conjunction of literals can be satisfiable is also a very easy task. As long as the conjunction does not contain a variable x and its negation, it is satisfiable. Thus we can verify linearly whether an expression in DNF is satisfiable. The problem here is that there is no (known) algorithm to convert an arbitrary expression to DNF where the size of the expression grows polynomially.

3.3.1 Conversion to CNF

But how do we convert an arbitrary expression to CNF? The first step we need is to remove the implication and the equivalence, since they are not allowed in CNF. We can replace an implication as follows:

$$x_1 \to x_2 \equiv \neg x_1 \vee x_2 \tag{2}$$

and we can replace an equivalence like this:

$$\begin{aligned} x_1 \leftrightarrow x_2 &\equiv (x_1 \to x_2) \wedge (x_2 \to x_1) \\ &\equiv (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \end{aligned} \tag{3}$$

The next step is moving all negations inward, until the only negations are in literals. This can easily be done using De Morgan's laws and double negations:

$$\begin{aligned} \neg(x_1 \wedge x_2) &\equiv \neg x_1 \vee \neg x_2 \\ \neg(x_1 \vee x_2) &\equiv \neg x_1 \wedge \neg x_2 \\ \neg\neg x_1 &\equiv x_1 \end{aligned} \tag{4}$$

At this point we only have conjunctions and disjunctions of subexpressions. The only difference with CNF is a conjunction in a disjunction. Therefore we need to move all conjunctions outward. This can be done using the distributive property of disjunction over conjunction:

$$(x_1 \wedge x_2) \vee x_3 \equiv (x_1 \vee x_3) \wedge (x_2 \vee x_3) \tag{5}$$

If we apply this step iteratively, we eventually obtain an expression in CNF that is equivalent to the original statement. For an expression ϕ we define $C(\phi)$ as the expression in CNF after the transformation described above.

Remark 3.3.4. We notice that the size of an expression remains the same during steps 2 and 4 and increases in steps 3 and 5. Thus we see that the size of an expression does not decrease at any point during the transformation.

We can now transform every Boolean expression to CNF. If we have an algorithm that can efficiently solve the satisfiability problem for expressions in CNF, we should be able to solve every Boolean expression efficiently. This is however not the case. We already saw that the size of an expression can increase during a transformation. In fact, there exist expressions that increase exponentially in size when transformed to CNF.

Theorem 3.3.5. *Let $\phi_n = (\dots((x_1 \leftrightarrow x_2) \leftrightarrow x_3) \dots \leftrightarrow x_n)$. Then $|C(\phi_n)|$ grows exponentially in terms of n. That is, we can present an exponential function $f : \mathbb{N} \to \mathbb{N}$, for which there exists an $N \in \mathbb{N}$ such that $f(n) \leq |C(\phi_n)|$ for all $n \geq N$.*

Proof. We present a proof by induction. Define the function $f(n) = 2^{n-1}$ and let $N = 2$. We already saw that $|C(\phi_2)| = 3 \geq 2 = f(2)$. Now we take an arbitrary $k \geq N$ and assume that the theorem holds for this value. Thus, it holds that $2^{k-1} \leq |C(\phi_k)|$. Now we look at $n = k+1$. The conversion steps taken are the following:

$$\begin{aligned}
\phi_{k+1} &\equiv \phi_k \leftrightarrow x_{k+1} \\
&\equiv C(\phi_k) \leftrightarrow x_{k+1} \\
&\equiv (\neg C(\phi_k) \vee x_{k+1}) \wedge (C(\phi_k) \vee \neg x_{k+1}) && (6) \\
&\equiv C(\phi_{k+1}) && (7)
\end{aligned}$$

We know that the size 7 is not smaller than that of 6. Therefore, we can conclude the following:

$$\begin{aligned}
|C(\phi_{k+1})| &\geq |(\neg C(\phi_k) \vee x_{k+1}) \wedge (C(\phi_k) \vee \neg x_{k+1})| \\
&= 2\,|C(\phi_k)| + 3 \\
&\geq 2 \cdot 2^{k-1} + 3 \\
&\geq 2^{(k+1)-1}
\end{aligned}$$

By induction, this conludes the proof. □

3.3.2 Tseitin transformation

An exponential growth in size is a big problem, if we wish to solve SAT via this transformation. Even if we find a linear algorithm to solve this for an expression in CNF, there exist expressions in which case this algorithm is not better than checking all possibilities. This is the point where the Tseitin transformation gives us a solution. This transformation does not preserve equivalence, but that property is not necessary. It is sufficient if we preserve equisatisfiability. Two expressions ϕ and ψ are equisatisfiable if ϕ is satisfiable if and only if ψ is. The following definition of the Tseitin transformation is taken from [9]:

Definition 3.3.6. (Tseitin transformation). Given an expression ϕ, for every subexpression ψ we define a variable p_ψ. The Tseitin transformation of ϕ begins with a conjunction of:

- the single literal p_ϕ;
- the CNF of $p_\psi \leftrightarrow (p_{\psi_1} \diamond p_{\psi_2})$ for every subexpression ψ of ϕ of the form $\psi = \psi_1 \diamond \psi_2$ with $\diamond$ a binary operator;

- the CNF of $p_\psi \leftrightarrow \neg p_{\psi_1}$ for every subexpression ψ of ϕ of the form $\psi = \neg\psi_1$.

Then we convert all equivalences to CNF with the same process as described in 3.3.1. We define $T(\phi)$ as the Tseitin transformation of an expression ϕ.

Example 3.3.7. Let $\phi = (x_1 \wedge x_2) \rightarrow (x_3 \vee x_4)$. We have the following subexpressions:

- $x_1 \wedge x_2$
- $x_3 \vee x_4$
- $(x_1 \wedge x_2) \rightarrow (x_3 \vee x_4)$

We introduce new variables and the following equivalences:

- $p_1 \leftrightarrow (x_1 \wedge x_2)$
- $p_2 \leftrightarrow (x_3 \vee x_4)$
- $p_3 \leftrightarrow (p_1 \rightarrow p_2)$

The conjunction is then $p_3 \wedge (p_3 \leftrightarrow (p_1 \rightarrow p_2)) \wedge (p_2 \leftrightarrow (x_3 \vee x_4)) \wedge (p_1 \leftrightarrow (x_1 \wedge x_2))$. Now we only need to convert every subexpression to CNF. This does become a rather large expression, especially when there exists a much shorter equivalent expression in CNF: $\phi \equiv \neg x_1 \vee \neg x_2 \vee x_3 \vee x_4$. △

It seems we have found an even less efficient transformation, but this is not true. In fact, we can prove that this transformation grows linear in the size of an expression.

Theorem 3.3.8. *The Tseitin transformation of an expression ϕ grows linear in the size of ϕ. Thus, it holds that $|T(\phi)| = \mathcal{O}(|\phi|)$.*

Proof. We need to show that there exists a C such that for every expression ϕ the following holds:

$$|T(\phi)| \leq C\,|\phi|$$

We define $A_\phi = \{\psi_i \mid i \in I\}$ as the set of newly introduced equivalence subexpressions for some index set I in the Tseitin transformation of ϕ. Since we introduce an equivalence for every subexpression, we see that $|A_\phi|$ is equal to the number of operations in ϕ. From Theorem 3.1.4 it follows that $|A_\phi| = |\phi|_2 \leq K_1\,|\phi|$ for some constant K_1. Then the size of the conjunction

$$p_\phi \wedge \bigwedge_{a \in A_\phi} a$$

is also equal to $|\phi|$.

Next we look at the size of an arbitrary equivalence subexpression a. As shown in Example 3.3.7, this subexpression consists of an equivalence between a variable and either a subexpression with two variables or a negation. We conclude that $|a| \leq 2$. Since there is only a finite number of subexpressions with size 2, it is obvious that there exists some K_2 such that $|C(a)| \leq K_2$. Now the following holds:

$$\begin{aligned}
|T(\phi)| &= \left| p_\phi \wedge \bigwedge_{a \in A_\phi} a \right| + \sum_{a \in A_\phi} |C(a)| \\
&\leq K_1\,|\phi| + \sum_{a \in A_\phi} K_2 \\
&\leq K_1\,|\phi| + K_1\,|\phi| \cdot K_2 \\
&= K_1(K_2 + 1)\,|\phi|
\end{aligned}$$

This concludes the proof. □

A drawback of an equisatisfiable transformation is that it is not always possible to reconstruct the original solution, if we find a solution for the transformation. With Tseitin, we do know the original solution, because we only need to remove all p_ψ.

4 Solving the satisfiability problem

We can now convert every expression to CNF, while still maintaining the same order of size as the original expression. In this chapter we describe two algorithms that use CNF, the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and conflict-driven clause learning (CDCL). For both algorithms we will assume that an expression is in CNF. However, although Tseitin gives us an expression in CNF with a rather specific structure, we will not make any assumptions about the form of the expression, other than it being in CNF. It is possible that our original expression was already in CNF or that we used an algorithm different from Tseitin.

4.1 Resolution

Before we look at algorithms that can solve SAT, we want to simplify an expression if possible. This can potentially reduce the number of variables and clauses significantly. The easiest simplification is removing duplicate clauses. However, this will not happen often, especially after the Tseitin transformation, since we introduce a new variable in every clause.

Next we look at the clauses individually. It is possible that some of them contain the same variable multiple times. If a clause contains both the variable and its negation, it will always evaluate to 1. Therefore, we can safely remove this clause, while still preserving equisatisfiability. If a literal appears multiple times in a clause, we can remove the duplicates.

Furthermore, we call a literal *pure* if its negation does not occur in the expression. Every pure literal can be assigned the value 1. Since every clause that contains this literal will be 1, they can all be removed.

Next, if an expression contains two clauses of the form:

$$\psi_1 \vee x, \psi_2 \vee \neg x$$

with ψ_1 and ψ_2 subexpressions and these are the only two instances of x in the expression, we can reduce these two clauses to $\psi_1 \vee \psi_2$. If only one of ψ_1 and ψ_2 has the value 1, we can choose x accordingly to ensure that both clauses are 1. Note that the assertion that x does not appear anywhere else in the expression, is essential, since we need to choose x without restrictions.

Lastly, single literal clauses can be assigned immediately, since their value must be 1. This step allows us to assume that our first step in every algorithm is a decision. This will be a useful property.

We can do all these simplifications, because we only try to solve the decision problem. As long as the expressions are equisatisfiable, this method works fine. However, a lot of SAT solvers can also produce the solution of the original expression (if it is satisfiable of course). In that case we need to remember all the variables that we have removed. This is generally not very difficult, yet noteworthy.

4.2 DPLL

One of the earlier algorithms for CNF-SAT is the Davis–Putnam–Logemann–Loveland (DPLL) algorithm, published in 1962 [7]. Intuitively, this algorithm checks for all variables if it can

derive its value directly. Otherwise, it makes a decision for a random variable and looks if it can derive more variables. If it finds a contradiction (or *conflict*), it backtracks to the last decision and chooses the opposite value. In the end we either have a solution (satisfiable) or a conflict without decisions (unsatisfiable).

During the algorithm we want to make a distinction between a variable x_i and its value, so we define $\nu(x_i) \in \{0, u, 1\}$ as the assigned value of variable x_i, where u means that x_i is unassigned. We denote an assignment of x_i with value $\nu(x_i)$ as $(x_i, \nu(x_i))$. We also define a stack S that keeps track of our assignments, both for decisions and derived variables. Whenever we change the stack, we write this as $S \to T$, where S is the old stack and T the new one. For example, when assigning a value $\nu(x_i)$ to a variable x_i, we write: $S \to S(x_i, \nu(x_i))$.

Given an expression ϕ, we define an empty initial stack $S = \varepsilon$. The algorithm takes the following steps:

1. Check whether it holds that $S \vdash \phi$. Then ϕ is satisfiable.
2. Check whether there exists a clause ψ such that $S \vdash \neg\psi$ and S does not contain any chosen variables. In that case ϕ is unsatisfiable.

3a. Check iteratively if there exists a literal $l \notin S$ such that there is a clause of the form $l \vee \psi$ for which $S \vdash \neg\psi$. Then we know $\{S, \phi\} \vdash l$. If $l = x$, we define $\nu(x) = 1$ and if $l = \neg x$, we define $\nu(x) = 0$. We call this step *unit propagation* and the corresponding variable an *implied variable*. Notation: $S \to S(x, \nu(x))$.

3b. If no such literal exists, we choose an unassigned variable x randomly and give it a value $\nu(x) \in \{0, 1\}$. To remember that this assignment was a decision we mark it. Notation: $S \to S(\hat{x}, \nu(x))$. Go back to step 1.

4. If the stack is of the form $S = T(\hat{x}, \nu(x))N$ such that N does not contain any chosen variables and there exists a clause ψ such that $S \vdash \neg\psi$, we then have encountered a contradiction. We remove N from the stack and replace $(\hat{x}, \nu(x))$ with $(x, \neg\nu(x))$. We no longer have to mark x, since we know already that $x = \nu(x)$ yields a contradiction. Notation: $S \to T(x, \neg\nu(x))$.
5. Go back to step 1.

Example 4.2.1. Let us look at an example. Consider the following expression:

$$\begin{aligned}\phi &= C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6 \\ &= (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_4 \vee x_5) \wedge \\ &\quad (\neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_5) \wedge (x_2 \vee \neg x_4)\end{aligned}$$

Although we could resolve a few clauses, we just want to look at the algorithm directly. We start with an empty stack ε. It is clear that both $\varepsilon \vdash \phi$ and $\varepsilon \vdash \neg\phi$ do not hold. Therefore, we make a decision: $x_1 = 0$. Clause C_5 implies that $x_5 = 0$. Now we must make a decision again. We could make the not so smart decision $x_2 = 1$. From C_2 and C_3 we conclude that $x_3 = 1$ and $x_4 = 0$ respectively. We see now that $S \vdash \neg C_4$, so we apply step 4 on x_2. From C_6 it follows that $x_4 = 0$ and then we see that $x_3 = 0$ to satisfy C_4. We have taken the following steps in our stack:

$$\begin{aligned}\varepsilon \to \varepsilon(\hat{x}_1, 0) \to \varepsilon(\hat{x}_1, 0)(x_5, 0) \to \varepsilon(\hat{x}_1, 0)(x_5, 0)(\hat{x}_2, 1) \to \\ \varepsilon(\hat{x}_1, 0)(x_5, 0)(\hat{x}_2, 1)(x_3, 1)(x_4, 0) \to \varepsilon(\hat{x}_1, 0)(x_5, 0)(x_2, 0) \to \\ \varepsilon(\hat{x}_1, 0)(x_5, 0)(x_2, 0)(x_4, 0)(x_3, 0)\end{aligned}$$

We now go back to step 1 and see that all clauses are true. We have found a solution and ϕ is satisfiable. $\triangle$

4.2.1 Performance

In many instances the DPLL algorithm will perform better than determining the truth table. If a choice leads to a contradiction and there are still m variables unassigned, we safe ourselves 2^m possibilities to check. But of course, this algorithm is in the worst-case not polynomial. Otherwise we would have solved the P versus NP problem. Unfortunately, worst case the run time of DPLL is still exponential [1].

4.3 CDCL

The DPLL algorithm has already an average performance that exceeds the truth table method. However, that does not mean that there are no improvements possible. From the mid-90's a new technique based on DPLL was developed, called conflict-driven clause learning (CDCL) [4]. It works similar to DPLL, but it allows us to add new clauses to prevent us from encountering the same conflict. It also uses non-chronological backtracking, backtracking to variables other than the last decision.

4.3.1 Housekeeping

Before we begin we must do some housekeeping. That way we can refer more easily to certain notions. We use the following definitions from [4], since they use the same concepts. We define a decision level $\delta(x)$ as the iteration index at which we assign a value to x, either by unit propagation or decision. For unassigned variables, we define $\delta(x) = -1$. If a variable is assigned via unit propagation in clause C, we define

$$\delta(x) = \max(\{0\} \cup \{\delta(y) \mid y \in C, y \neq x\}).$$

For a decision variable the decision level is given by

$$\delta(x) = \max(\{\delta(y) \mid y \in X, y \neq x\}) + 1.$$

As a consequence, all decision variables have a different decision level and all implied variables have a decision level equal to that of the decision variable that implied them.

The second addition to DPLL is a decision graph G, a directed graph that we use to keep track of our decisions. Whenever we make a decision for a literal, we add a *decision node* without incoming arrows. If we apply unit propagation on a literal l, we have found a clause $C = l \vee l_1 \vee \ldots \vee l_n$ such that $S \vdash \neg(l_1 \vee \ldots \vee l_n)$. We then add an *implied node* for l and an arrow from $l_1, \ldots, l_n$ to l to remember the dependencies of l. The result is a directed acyclic graph. Because we assumed previously that we cannot apply unit propagation at the beginning, all nodes without incoming arrows (*roots*) are decision nodes.

When we encounter a conflict, we apparently found a clause $C = l_1 \vee \ldots \vee l_n$ that is false under the current variable assignment. We add an extra conflict node κ and draw arrows from $l_1, \ldots, l_n$ to κ. The graph of Example 4.2.1 in the moment of conflict can be seen in Figure 1. With this node we will define an additional clause to prevent us from encountering the same conflict again. We will discuss how this is done using the GRASP algorithm and we look at some improvements.

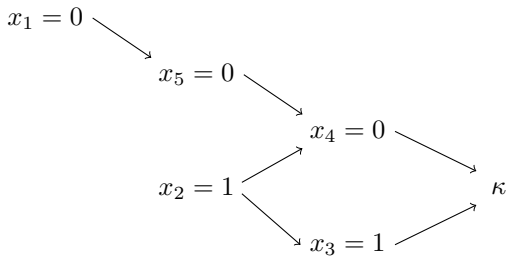

Figure 1: Implication graph of Example 4.2.1

4.3.2 GRASP

The GRASP algorithm (Generic seaRch Algorithm for the Satisfiability Problem) was introduced in the late 90's [13]. We start in the conflict node and determine the nodes of all incoming arrows. We execute this step iteratively, until we have a set of nodes without incoming arrows. Since we know that these are all decision nodes, we come to the conclusion that this set of decisions will eventually lead to a contradiction. We define the *implication set* $D_1(\kappa)$ as the set of these assignments. It is sufficient if we negate at least one of the assignments. We can write this in an additional clause:

$$C_1' = \bigvee_{(x,\nu(x))\in D_1(\kappa)} x^{\nu(x)}$$

where $x^0 = x$ and $x^1 = \neg x$. We add the *conflict-induced clause*, while retaining equisatisfiability. By adding this clause, we prevent reassigning the same values to the variables in our implication set. In Example 4.2.1 we get the clause $C_1' = (x_1 \vee \neg x_2)$. We determine the variable x with the highest decision level in the implication set $D_1(\kappa)$. In our implication graph we add arrows from the other decision nodes in $D_1(\kappa)$ to x. If it were to happen that $D_1(\kappa) = \{(x, \nu(x))\}$ (see Example 4.3.1), we know the value of x for certain. In that case we resolve x by substituting its value in the expression. We remove all decision nodes with a larger decision level, because it is possible that one of these decisions is implied by changing the value of x. Then we remove all implied nodes. On our stack we also jump back to x and take the negation. An important notice is that x is not necessarily the last decision made, in contrary to the DPLL algorithm. It is true that the implication set of the first conflict we encounter contains the last decision, but this does not hold for later conflicts.

Example 4.3.1. We perform GRASP and have made the decisions $x_1 = 1$, $x_2 = 0$ and $x_3 = 0$ consecutively. This causes a conflict κ with $D_1(\kappa) = \{(x_1, 1), (x_3, 0)\}$. We then assign $x_3 = 1$. If this also leads to another conflict κ' with only $D_1(\kappa') = \{(x_1, 1)\}$, we see the assignment $x_1 = 1$ will always lead to a conflict. The DPLL algorithm now backtracks to x_2 even though the conflict and x_2 have no connection. If it makes a decision for x_3, we encounter one of the conflicts again. It then negates the value of x_3 and we encounter the other conflict. Only then will we change the value of x_1 (assuming that we have not made any other decisions in the meantime). △

The GRASP algorithm does give us some overhead. We introduce new clauses that we need to keep track of and that we need to check during our algorithm. We also generate a clause with every backtrack. This potentially increases the number of clauses exponentially. A solution, proposed in [13], is the deletion of large conflict-induced clauses. A clause with 20 literals will not cause a conflict as fast as one with only 5. We define a maximum clause length k. Every time we introduce a new clause with a size larger than k, we mark it. Initially, the clause will evaluate

to 1, but it is possible that after a few more backtracks the clause has some unassigned literals and the other literals are 0. We call a clause then *unresolved*. We delete a marked clause, if it becomes unresolved with at least two unassigned literals. By deleting conflict-induced clauses, we lose known information, but in the worst case the number of clauses grows polynomial [13].

4.3.3 Unique implication points

The last improvement that we will discuss is the use of unique implication points (UIPs). With these UIPs we can decrease the size of the conflict-induced clauses. As stated earlier, a smaller clause gives us more information.

We define a set $U(x) = \{(x_1, \nu(x_1)), \ldots, (x_n, \nu(x_n))\}$ as the set of dominators of κ with respect to a decision variable x. In graph theory a dominator of w with respect to u is a node v such that every path from u to w passes v. Note that it holds that $(x, \nu(x)) \in U(x)$. Finding dominators can be done in almost linear time in terms of the number of edges with the algorithm of Thomas Lengauer and Robert Endre Tarjan [11]. This algorithm runs in $\mathcal{O}(m \cdot \alpha(m, n))$ where m is the number of edges in a graph, n is the number of vertices and $\alpha(m, n)$ is the inverse Ackermann's function with two parameters. This function grows extremely slowly and in practice we can consider it a constant.

Now we define the set of UIPs as

$$U = \bigcup_{(x,\nu(x)) \in D_1(\kappa)} U(x).$$

If two decisions have a dominator in common, we can add that to the clause instead of the decision variables. We can determine the new clause as follows. We begin again in the conflict node and determine all nodes of incoming arrows. But rather than walking all the way back until we found a decision variable, we also stop, when we encounter a UIP. We define the set of these variables as $D_2(\kappa)$ and the conflict-induced clause as

$$C_2' = \bigvee_{(x,\nu(x)) \in D_2(\kappa)} x^{\nu(x)}.$$

Theorem 4.3.2. *For a conflict κ in the implication graph, it holds that $|D_2(\kappa)| \leq |D_1(\kappa)|$.*

Proof. For every $(x, \nu(x)) \in D_2(\kappa)$ there is at least one decision variable y_x in $D_1(\kappa)$ such that there is a path from y_x to x, because we have an acyclic directed graph and by definition there exist no roots $(z, \nu(z)) \notin D_1(\kappa)$ such that there is a path from z to κ. Furthermore, all y_x are distinct. If we have x_1 and x_2 such that $y := y_{x_1} = y_{x_2}$, then we see that x_1 and x_2 are elements of $U(y)$. Then there must be a path from x_1 to x_2 or the other way around. Otherwise they cannot be dominators of κ, because we have found two paths from y to κ. Without loss of generality we assume that there is a path from x_1 to x_2. But that means that there is another path from x_1 to κ that does not pass through x_2, because x_1 is an element of $D_2(\kappa)$. This is a contradiction, because we have found a path from y to κ that does not pass through x_2. Because all y_x are distinct, we know that $D_1(\kappa)$ is at least as big as $D_2(\kappa)$. □

We now have a clause that is not bigger than the clause of $D_1(\kappa)$. In fact, it is quite likely that there exists an x in $D_2(\kappa)$ that is the dominator of multiple decisions. In that case C_2' is strictly smaller than C_1'.

Example 4.3.3. Let us slightly modify Example 4.2.1:

$$\begin{aligned}\phi &= C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6 \wedge C_7 \\ &= (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_4 \vee x_5) \wedge \\ &\quad (\neg x_3 \vee x_4 \vee x_7) \wedge (x_1 \vee \neg x_5 \vee x_6) \wedge (x_2 \vee \neg x_4) \wedge (x_5 \vee \neg x_7)\end{aligned}$$

We also assume that we made the decision $x_6 = 0$ at the beginning. In Figure 2 the new implication graph can be seen. Without UIPs we would get the clause $(x_1 \vee \neg x_2 \vee x_6)$. If we look at the UIPs, we see that $x_5 \in U(x_1) \subset U$. It also obvious that x_3, x_4 and x_7 are no dominators of any decision variable. Thus we conclude that $C_2' = (x_5 \vee \neg x_2)$. This is a smaller clause and still prevents us from encountering the same conflict. △

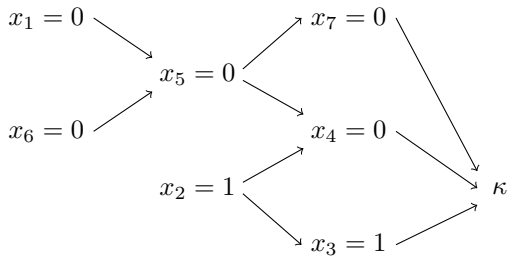

Figure 2: Implication graph of Example 4.3.3

Now we must determine to which point we backtrack. We cannot just backtrack to the variable in $D_2(\kappa)$ with the highest decision level and negate its value. If this is an implied node, we would immediately generate a new conflict. This would happen in Example 4.3.3 if we define $x_5 = 1$. If we jump to the decision variable with the highest decision level, we might miss a part of our search space, possibly getting the wrong answer. It is even possible that there are no decision variables in the conflict-induced clause. Instead, we find the variable x in $D_2(\kappa)$ with the highest decision level and determine the decision variable y with the same decision level. This was the decision that eventually leads to the conflict. We remove again a part of the implication graph in the same way as before. We do know however that we can negate the value y. In the implication graph we add arrows from the other nodes in C_2' to y.

5 Satisfiability Modulo Theories

In the last section we saw SAT and algorithms to solve it. Now it is time to look at satisfiability modulo theories (SMT). As the name suggests it is closely related to SAT. SMT can be looked at as a generalization of SAT. It allows expressions with data types other than Booleans such as integers, real numbers, bitvectors, uninterpreted functions and functional arrays [3]. For this thesis, we will only look at a few types of expressions with integers and real numbers, since SMT is an extensive problem set with many different theories and logics.

5.1 SMT solvers

To standardize all the different types listed above, an international initiative came to be, the SMT-LIB [2]. It is a standardization with many different types, separated in theories and logics. It also defines a standard syntax that all SMT solvers should follow.

5.1.1 Theories

A theory describes the behavior of a specific data type. Examples are the *Core* theory, which defines Booleans and all operators used in the last section, the *Ints* theory, which defines the integers and a few arithmetic operators as well as the comparison operators, and the *Reals*, which defines the real numbers and the same operators as the Ints theory apart from the modulo and absolute value. It also defines a different division than the Ints. A theory does not define any kind of allowed expressions. It only formulates some axioms for the operations that must hold. As an example, the commutativity axiom of the multiplication is defined as

```
(forall ((x Real) (y Real)) (= (* x y) (* y x)))
```

in the theory of real numbers [17].

5.1.2 Logics

A logic tells us which types and expressions should be supported by a solver. The list of logics is still growing. An incomplete overview of the most common logics is shown in Figure 3. An arrow from L1 to L2 means that all expressions of L1 are included in L2. Many logics build on other logics and extend the formulas included.

For example, the logic `QF_LIA` describes all quantifier-free linear integer arithmetic. This includes expressions like $3x + 4y < 20 \wedge 4x - 6y > 3$ with $x, y \in \mathbb{N}$. This expression is easily satisfiable with $x = 3, y = 1$. An extension on this logic is `LIA` that does allow quantifiers. This logic lets us define expressions like

$$\forall x \in \mathbb{N}, \exists y \in \mathbb{N}, 3x + 4y < 20 \wedge 4x - 6y > 3. \tag{8}$$

This statement is obviously not true, because when $x = 7$, it follows that $3x + 4y > 20$ for every $y \in \mathbb{N}$. Statements like the one in 8 are much more complicated to verify than quantifier-free expressions.

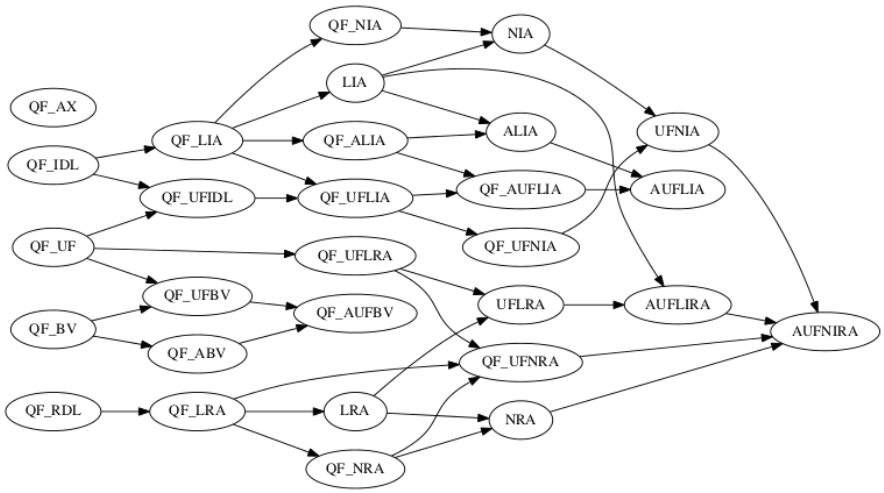

Figure 3: An overview of logics

In this section we will focus on `QF_LIA` and `QF_LRA`, quantifier free linear real arithmetic. More precisely, we look at systems of (integer) linear inequalities (SO(I)LI). There are many

ways to solve these kind of equations. We will first discuss the so called 'eager' method for solving integer linear equations.

5.2 The eager method

The eager method is a relatively easy way of solving SOILIs. It is based on the idea of translating the problem to SAT. We translate the problem to an expression the same way a computer does. That is, we write every integer as a bit string where every bit is represented by a Boolean in the expression. An advantage of this method is that it does not limit us to linear equations. We could in fact extend this method to polynomial, exponential and even modular equations. This method works in fact for a lot of data types, including floating point numbers. We will analyze solving a SOILI. We follow the steps taken from [19] partially. Here they solve linear equations with binary variables. We will extend this to non-negative integer variables. This method can be extended further to include negative integers by adding a sign bit to each number, but we limit ourselves to non-negative integers for the sake of simplicity.

5.2.1 Integer linear inequalities

The first step is defining the exact problem. A system of non-negative integer linear inequalities is a set of inequalities of the form:

$$Ax \leq b$$

Here A is a matrix in $\mathbb{Z}_{\geq 0}^{m \times n}$, x is vector in $\mathbb{Z}_{\geq 0}^{n}$ and b is a vector in $\mathbb{Z}_{\geq 0}^{m}$. We can also write the system as follows:

$$\begin{array}{ccc} a_{11}x_1 + \ldots + a_{1n}x_n \leq b_1 \\ \vdots \qquad\qquad \vdots \qquad \vdots \\ a_{m1}x_1 + \ldots + a_{mn}x_n \leq b_m \end{array} \tag{9}$$

Note that this is an integer problem, so we can assume that all inequalities are not strict. Now we need to define a few things. Since we represent all expressions as bit strings, the size of these bit strings is not necessarily a fixed number. However, all expressions will be finite. Thus we can bound the number of bits by a large enough M. We will define the i-th bit of an expression f as p_i^f with $0 \leq i \leq M-1$. Furthermore, for a constant a, we define B_a as the set of bit indices that have the value 1. For example, it holds that $B_{13} = \{0, 2, 3\}$.

The first step is transforming a constant a. We have defined the variables, but we need to enforce that they all have the right value. We know that $p_i^a = 1$ only if $i \in B_a$. Otherwise, it should be 0. This can be done like this [19]:

$$\text{trans}(a) = \bigwedge_{k \in B_a} p_k^a \wedge \bigwedge_{k \notin B_a} \neg p_k^a \tag{10}$$

We can only satisfy this expression by choosing the right value for each bit.

5.2.2 Addition

The second step is adding two expressions f and g together. This is not too complicated. We do need to introduce an extra variable for each carry. We define the carry from the i-th to the $(i+1)$-th bit as $c_{i,i+1}$. Since we do not have a carry to the 0-th bit, we see that p_0^{f+g} is 1 if and only if exactly one of p_0^f and p_0^g has the value 1. We can express this as $p_0^{f+g} \leftrightarrow (p_0^f \leftrightarrow p_0^g)$. For

the other bits of $f+g$, it shows that p_i^{f+g} is 1 if an odd number of the p_i^f, p_i^g and $c_{i-1,i}$ has the value 1. An interesting property of the bi-implication is that we can express this as

$$p_i^{f+g} \leftrightarrow (p_i^f \leftrightarrow p_i^g \leftrightarrow c_{i-1,i}).$$

For the carries the first one is again slightly different. We see that $c_{01} \leftrightarrow p_0^f \wedge p_0^g$. For the other carry bits $c_{i,i+1}$ is 1 if at least two of p_i^f, p_i^g and $c_{i-1,i}$ have the value 1. We can express this simply as

$$c_{i,i+1} \leftrightarrow \Big((p_i^f \wedge p_i^g) \vee (p_i^f \wedge c_{i-1,i}) \vee (p_i^g \wedge c_{i-1,i})\Big).^1$$

Since we want to reuse the sum of f and g, we say that $h = f + g$ and we define the total transformation of the addition as follows [19]:

$$\begin{aligned} T(h,f,g) = &\Big(p_0^h \leftrightarrow (p_0^f \leftrightarrow p_0^g)\Big) \wedge \\ &\Big(c_{01} \leftrightarrow p_0^f \wedge p_0^g\Big) \wedge \\ &\bigwedge_{i=1}^{M-1} \Big(p_i^h \leftrightarrow (p_i^f \leftrightarrow p_i^g \leftrightarrow c_{i-1,i})\Big) \wedge \\ &\bigwedge_{i=1}^{M-1} \Big(c_{i,i+1} \leftrightarrow \Big((p_i^f \wedge p_i^g) \vee (p_i^f \wedge c_{i-1,i}) \vee (p_i^g \wedge c_{i-1,i})\Big)\Big) \end{aligned} \tag{11}$$

Remark 5.2.1. Note that we defined M large enough to represent all expressions in the system. Therefore, we can assume that $c_{M-1,M} = 0$. Otherwise the consequence is that $p_M^h = 1$, in which case h needs $M+1$ bits to represent. This is of course a contradiction.

5.2.3 Multiplication

The next step is multiplying a number a with a variable x. Since this will be a rather complicated transformation, we start with a multiplication with a binary variable x_i. If $x_i = 0$, then $ax_i = 0$. In that case all bits of ax_i will be 0. If $x_i = 1$, then $ax_i = a$. So we see that the same bits of ax_i are 1 as the bits in a. It follows easily that $p_k^{ax_i}$ only has the value 1 if and only if both x_i and p_k^a are equal to 1. Thus we can write this transformation like this [19]:

$$\text{trans}(ax_i) = \bigwedge_{k=0}^{M-1} p_k^{ax_i} \leftrightarrow (x_i \wedge p_k^a) \tag{12}$$

But in this case we only use a binary variable. To illustrate the complexity that we introduce when multiplying two integers, we present the multiplication of 5 and 7.

$$\begin{array}{cccccccc} & & & 1 & 0 & 1 & & \\ & & & 1 & 1 & 1 & \times & \\ \hline & & {\scriptstyle 1} & 1 & 0 & 1 & & \\ & {\scriptstyle 1} & 1 & 0 & 1 & & & \\ {\scriptstyle 1} & 1 & 0 & 1 & & & + & \\ \hline 1 & 0 & 0 & 0 & 1 & 1 & & \end{array}$$

[1] This is wrongly formulated in [19], where this expression is based on. In this paper the last two conjunctions are the same.

Not only do we perform multiple binary multiplications, we also shift the results and add them together. This process is too complicated to write it in one expression like 12. We define new variables p_{jk}^{ax} as shown below. The row p_{j0}^{ax} is the multiplication of x and p_0^a and so on.

				p_2^x	p_1^x	p_0^x	
				p_2^a	p_1^a	p_0^a	$\times$
$t_0^{ax} =$	p_{50}^{ax}	p_{40}^{ax}	p_{30}^{ax}	p_{20}^{ax}	p_{10}^{ax}	p_{00}^{ax}	
$t_1^{ax} =$	p_{51}^{ax}	p_{41}^{ax}	p_{31}^{ax}	p_{21}^{ax}	p_{11}^{ax}	p_{01}^{ax}	
$t_2^{ax} =$	p_{52}^{ax}	p_{42}^{ax}	p_{32}^{ax}	p_{22}^{ax}	p_{12}^{ax}	p_{02}^{ax}	$+$
	p_5^{ax}	p_4^{ax}	p_3^{ax}	p_2^{ax}	p_1^{ax}	p_0^{ax}	

Remark 5.2.2. Note that, just like with the addition, the length of ax is larger than both a en x. Again this will not be a problem for the same reason. But since a most likely starts with a number of leading zeros, we define the length of a without leading zeros as M_a.

Now we need to enforce that these newly introduced variables have the right values according to our table. For example, we see that $p_{00}^{ax} \leftrightarrow (p_0^x \wedge p_0^a)$ enforces the correct value of p_{00}^{ax}. If we also take the shifts in account, we get the following general expression:

$$p_{jk}^{ax} \leftrightarrow (p_{j-k}^x \wedge p_k^a)$$

This is of course only a correct definition if $j \geq k$. But we already saw in the example of 5 and 7 that $p_{jk}^{ax} = 0$ if $j < k$. We can combine this expression with 10 and get a total expression:

$$\text{trans}_1(ax) = \bigwedge_{k \in B_a} \left(\bigwedge_{j=k}^{M-1} (p_{jk}^{ax} \leftrightarrow p_{j-k}^x) \wedge \bigwedge_{j=0}^{k-1} \neg p_{jk}^{ax} \right) \wedge \bigwedge_{\substack{k \notin B_a \\ k < M_a}} \bigwedge_{j=0}^{M-1} \neg p_{jk}^{ax} \tag{13}$$

Lastly we need to add the t_i^{ax} together. This can be done using 11. We define the partial sum $f_k^{ax} = t_0^{ax} + \ldots + t_k^{ax}$. We get the following expression:

$$\text{trans}_2(ax) = \bigwedge_{k=1}^{M_a - 1} T(f_k^{ax}, f_{k-1}^{ax}, t_k^{ax})$$

The total transformation of multiplication is then simply:

$$\text{trans}(ax) = \text{trans}_1(ax) \wedge \text{trans}_2(ax)$$

5.2.4 Inequality

The last part is the inequality. Again we define a partial sum $f_{ij} = a_{i1}x_1 + \ldots + a_{ij}x_j$. We want to find the transformation of $f_{in} \leq b_i$. What happens if $f_{in} \neq b_i$? Then there exists a largest (most significant) k such that $p_k^{f_{in}} \neq p_k^{b_i}$. If $p_k^{f_{in}} = 1$ and $p_k^{b_i} = 0$, then we see that $f_{in} > b_i$. Thus if we want to enforce that $f_{in} \leq b_i$, no such k may exist. Therefore we need to check this property for every k such that $p_k^{b_i} = 0$. This is of course if $k \notin B_{b_i}$. The property we need to check is that if $p_k^{f_{in}} = 1$, then for all $j > k$ it holds that $p_j^{f_{in}} = p_j^{b_i}$. However, it is sufficient to only check the $j \in B_{b_i}$, since $p_j^{b_i} = 0 \wedge p_j^{f_{in}} = 1$ implies that we have found a larger k with the property. We can now transform the inequality in the following expression [19]:

$$\text{trans}(f_{in} \leq b_i) = \bigwedge_{k \notin B_{b_i}} \left(p_k^{f_{in}} \rightarrow \neg \bigwedge_{j > k, j \in B_{b_i}} p_j^{f_{in}} \right)$$

5.2.5 The full equation

We now have a transformation for all operations separately. The only thing left to do is combining them into one expression. Since we want all inequalities to hold, we take the conjunction of the individual transformation:

$$\text{trans}(Ax \leq b) = \bigwedge_{i=1}^{m} \text{trans}(y_i)$$

Here we define y_i as the i-th inequality. The transformation of y_i is the conjunction of all multiplications, the sum and the inequality. We no longer have to include 10, since this is already combined in the multiplication. We can write the transformation of y_i as follows:

$$\begin{aligned}\text{trans}(y_i) = &\bigwedge_{j=1}^{n} \text{trans}\left(a_{ij}x_j\right) \wedge \\ &\bigwedge_{j=2}^{n} T\left(f_{ij}, f_{i,j-1}, a_{ij}x_j\right) \wedge \\ &\text{trans}(f_{in} \leq b_i)\end{aligned}$$

5.2.6 Result

This method gives us a SAT problem that is equisatisfiable to the original system of inequalities. If we have a satisfiable system and we asked the SMT solver to give us the corresponding assignments, we could calculate the values of all x_i's.

5.3 The lazy method

Although the eager method is direct and to the point, we introduce many redundant variables. Many terms $a_{ij}x_j$ will be quite small, so a lot of the leading bits will be zero. The lazy method does less superfluous work as opposed to the eager method, hence the name. In fact, the lazy method is not a name for one specific algorithm, but rather a collective name for all methods that use integer logic to solve the problem instead of a translation to SAT. In general, they are much more efficient than the eager method. We look at two algorithms, Gomory's cut and the Fourier-Motzkin elimination method and how they can be used for both SOLIs and SOILIs.

5.3.1 Gomory's cut

One method to solve SOILIs is to look at algorithms that solve integer linear programming (ILP) problems. Not only do we need to find a vector that satisfies the constraints, we also have an objective function $c^T \cdot x$ that we want to minimize (or maximize). However, ILPs require us to add an additional constraint, which is that all variables are non-negative (the constants can still be negative). One of the earliest methods was Gomory's cut, proposed by Ralph Gomory in 1958 [8]. The general idea is that we solve the LP relaxation with the simplex method [15]. This means that we drop the integrality constraint. If our solution is not integer, we add an extra linear constraint to reduce our search space. We will not go into detail how this algorithm generates new constraints, but the constraint removes a part of the search space that does not contain any integer solutions. Interestingly, this algorithm uses the same principle as CDCL by adding constraints during the process of solving, even though CDCL was developed in the mid-90's.

Assuming that our SOILI defines a non-empty region, this n-dimensional space is a convex polytope. The simplex method uses the fact that a minimum (if it exists) always occurs in a vertex of our space. Initially, we search for one of the vertices, called the *initial feasible solution*. From this point we walk over one of the connecting edges that decreases our objective function the most. The algorithm states that if we execute this step iteratively, we find a minimal solution or we come to the conclusion that the minimum is unbounded.

However, Gomory's cut is used to solve ILPs. But since we are only interested if our SOILI has a solution, we can make some shortcuts. For example, if we choose an objective function randomly, we need to invest time in solving the LP relaxation. But for determining an initial feasible solution and Gomory's cut, we do not need the objective function. Therefore, we can apply Gomory's cut immediately after we have found a feasible solution.

5.3.2 Fourier-Motzkin elimination

The second method we discuss is the Fourier-Motzkin elimination (FME) [6]. With this method we can remove a variable of our system. If we excute this elimination repeatedly, we eventually have only one variable left. This can easily be solved.

We assume that we have SOLI of the form 9. We partition the system based on whether the coefficients of x_1 are positive, negative or zero. Then we can write get three sets of linear equations:

$$\begin{cases} x_1 \geq D_1(\bar{x}) \\ \vdots \\ x_1 \geq D_p(\bar{x}) \end{cases}, \quad \begin{cases} x_1 \leq E_1(\bar{x}) \\ \vdots \\ x_1 \leq E_q(\bar{x}) \end{cases}, \quad \begin{cases} 0 \leq F_1(\bar{x}) \\ \vdots \\ 0 \leq F_r(\bar{x}) \end{cases}$$

where $D_i(\bar{x})$, $E_j(\bar{x})$ and $F_k(\bar{x})$ are linear functions in $\bar{x} = (x_2, \ldots, x_n)$ [6]. If we solve $\bar{x}$ such that $D_i(\bar{x}) \leq E_j(\bar{x})$ and $0 \leq F_k(\bar{x})$ for all values of i,j and k, we have found a solution for our original system. For the value of x_1 we know that

$$\max_{1 \leq i \leq p} D_i(\bar{x}) \leq x_1 \leq \min_{1 \leq j \leq q} E_j(\bar{x}).$$

Since we know that $D_i(\bar{x}) \leq E_j(\bar{x})$, we can always find a real x_1 that suffices. One major drawback of this method is the increase in inequalities. Since we pair up at most $n/2$ inequalities, we get an upperbound of $n^2/4$ new inequalities.

Although this algorithm can greatly increase the number of inequalities, it will always produce a correct solution if all x_i's can be real numbers. This is no longer the case if we look at SOILIs. Even if we find a $\bar{x} \in \mathbb{Z}^{n-1}$ such that all inequalities hold, it is still possible that there is no integer between $\max_{1 \leq i \leq p} D_i(\bar{x})$ and $\min_{1 \leq j \leq q} E_j(\bar{x})$. But that does not automatically mean that there is no integer solution to our SOILI. We could find a different $\bar{x}$ such that there is a corresponding integer x_1.

In 1991 William Pugh extended the FME method to integers with the Omega test [16]. With the FME method we get a new set of inequalities P' from our original problem P. The Omega test gives us a second set of inequalities P'' that is stricter than P'. If we can find an integer solution $\bar{x}$ to P'', it guarantees that there is an integer solution for P. We already know that there is no integer solution to P, if there is no integer solution to P'. The only case left is when we find a solution for P' and not for P''. Pugh presents a rather complicated method to check this case, so we will not explain it any further. Pugh acknowledges that this last case is expensive and that there are SOILIs that are a "nightmare" for the Omega test. But he also states that this last case does not occur very often in practice.

6 Liquid Haskell

In this section we look at an application of SMT solvers, Liquid Haskell. It is a tool that can be used to prove certain properties of Haskell functions by using refinement types.

6.1 Refinement types

Refinement types form a concept in the area of type theory. They endow a certain type with a predicate. For example, we can define the even numbers as a refinement type of the set of integers: $\{x : \mathbb{Z} \mid x \equiv 0 \pmod 2\}$. The set of even numbers is a refinement of the original type. In mathematical terms, a type can be seen as a set T, and the refinement as a subset R of T. We now present a formal definition of a refinement.

Definition 6.1.1. (Refinement). Given a set T and a propositional function $P : T \to B$ where B is the Boolean domain, we define a refinement:

$$R := \{t \in T \mid P(t)\}$$

Remark 6.1.2. This definition also allows us to define a refinement on a refinement itself.

We will only use this definition in a Liquid Haskell context. Therefore we give an equivalent definition of a refinement type for Haskell.

Definition 6.1.3. Given a `type T` and a propositional function `P :: T -> Bool`, we define a refinement type or subtype as

```
type R = {t:T | P t}.
```

Example 6.1.4. We will use the following refinement types in this section:

```
type TRUE = {v:Bool | v }
type FALSE = {v:Bool | not v }
type NAT = {v:Int | v >= 0 }
type POS = {v:Int | v > 0 }
```

It may seem odd to define `TRUE` and `FALSE` as a separate refinement type, since `True` and `False` are already defined in Haskell. However, these keywords are values of the type `Bool` and not types. △

6.2 Purpose

One of the applications of refinement types is checking pre- and postconditions of functions. They allow us to refine the parameters and return value of functions. We call this a *contract*. We can ask Liquid Haskell to verify these types. It then checks whether the return value always satisfies the refinement type and it verifies if the parameters of all calls to this function also satisfy their refinement type. The next example was taken from [18].

Example 6.2.1. We declare the function

```
div :: Int -> Int -> Int
```

as the Euclidean (integer) division. This function is undefined if the second argument equals 0. Therefore we want it to be impossible to call this function with such an argument. This is

where we use a contract. The main advantage with a contract is that we can check in advance, if there is possibility of dividing by zero instead of getting a run-time error. This does not mean that our program would have crashed. It only means that we cannot guarantee that the call is valid.

Say, in this example we not only want to prevent dividing by zero, we also want the two parameters to be non-negative and positive respectively. With the earlier defined refinement types, the contract becomes:

```
div ::  n:NAT -> d:POS -> {v:NAT | v <= n}
```

Note that a refinement type does not necessarily need to be defined separately. It can also be defined in a contract, when using the parameters of the function. We require that `div` returns a value that is less than or equal to the dividend, which is a trivial property with non-negative integer division. This is of course a necessary but not sufficient condition for the division function. However, we guarantee at least this one property. △

An important aspect of contracts is that every argument in the function must have a refinement type that is a subtype of its original type. In the example, we see that both `NAT` and `POS` are subtypes of `Int` and the return type is subtype of `NAT`, so also a subtype of `Int`.

6.3 Example

Now let us study an example and how Liquid Haskell verifies it using an SMT solver. We are going to look at one of De Morgan's laws:

$$\neg(x_1 \wedge x_2) \leftrightarrow (\neg x_1 \vee \neg x_2)$$

We want to verify that this law always holds, indepently of x_1 and x_2. In (Liquid) Haskell this becomes the following statement[2]:

```
{-@ DeMorgan ::  Bool -> Bool -> TRUE @-}
DeMorgan ::  Bool -> Bool -> Bool                                  (14)
DeMorgan x1 x2 = not (x1 && x2) <=> not x1 || not x2
```

To distinguish between Haskell and Liquid Haskell, we write Liquid Haskell code between `{-@ ... @-}`. We see here the use of the refinement `TRUE`. If we ask Liquid Haskell to verify this statement, it must check if our expression is true one way or another. The problem here is that we do not have a normal satisfiability problem as in Section 3. We want to check if the expression is a tautology. That is, the expression is true for all `x1` en `x2`. So this is in fact the quantified expression:

$$\forall\, x_1, x_2 : \neg(x_1 \wedge x_2) \leftrightarrow (\neg x_1 \vee \neg x_2). \tag{15}$$

Although most SMT solvers are able to verify quantified expressions, they still form a harder set of problems than unquantified expressions. Therefore, we attempt to solve this problem without quantifiers. In this case, we cannot check 15, but we can check its negation. If we were to prove that the negation is unsatisfiable, we know that the original expression is a tautology. This can be done as follows:

$$\begin{aligned} \neg\left(\forall\, x_1, x_2 : \neg(x_1 \wedge x_2) \leftrightarrow (\neg x_1 \vee \neg x_2)\right) &\equiv \exists\, x_1, x_2 : \neg\left(\neg(x_1 \wedge x_2) \leftrightarrow (\neg x_1 \vee \neg x_2)\right) \\ &\equiv \exists\, x_1, x_2 : (x_1 \wedge x_2) \leftrightarrow (\neg x_1 \vee \neg x_2) \end{aligned}$$

[2]We still have to define `(<=>)`, since this is not a standard function in Haskell.

We will not write this completely to CNF, but it is at this point already trivial that this expression is unsatisfiable. A SAT or SMT solver will of course perform the algorithm completely. We now have given a possibility how to tackle this problem. This does not necessarily mean that LIQUID HASKELL also takes these steps.

```
(assert (and (= x1_and_x2 (and x1 x2))
             (= not__x1_and_x2 (not x1_and_x2))
             (= not_x1 (not x1))
             (= not_x2 (not x2))
             (= not_x1_or_not_x2 (or  not_x1 not_x2))
             (= phi (and (=> not__x1_and_x2 not_x1_or_not_x2) (=> not_x1_or_not_x2 not__x1_and_x2)))
             (not phi)))

(check-sat)
; SMT Says: Unsat
```

Figure 4: Part of the *.smt2* file

Although it is very difficult to look at the exact steps before and after the use of the SMT solver, we can look at the *.smt2* file that LIQUID HASKELL generates. A part of this file can be seen in Figure 4[3]. The command `'assert'` presents an expression to the solver, but it does not verify it directly. Only after the command `'check-sat'` is executed, checks the solver all assertions. If the expressions are satisfiable, it returns `'sat'`, otherwise it returns `'unsat'`. The next line is a comment to tell us that LIQUID HASKELL expects that the solver gives us `'unsat'`.

We see that LIQUID HASKELL introduces new variables for every subexpression and enforces the right values in the same way we did in Section 5.2. If we combine all subexpressions into one, we get

$$\neg\left((\neg(x_1 \wedge x_2) \rightarrow (\neg x_1 \vee \neg x_2)) \wedge ((\neg x_1 \vee \neg x_2) \rightarrow \neg(x_1 \wedge x_2))\right)$$

which is equivalent to

$$\neg(\neg(x_1 \wedge x_2) \leftrightarrow (\neg x_1 \vee \neg x_2)).$$

This expression occurred in our own method, but we see that LIQUID HASKELL also expanded the equivalence. It also left the outer negation, instead of simplifying the expression. But apart from these minor details it took the same approach that we proposed instead of using quantifiers.

[3]For readability, all variables have been renamed, all subexpressions of the conjunction have been put on separate lines, and variable declarations have been omitted. LIQUID HASKELL also generates many other variables and unrelated expressions. These also have been omitted.

7 Conclusion

In conclusion, SMT solvers can be used for many types of problems. One of them is the satisfiability problem, which is more or less the basis of SMT, hence its inclusion in the `Core` logic. We studied a few algorithms that have been developed over the years that solve the problem. In the early 60's the DPLL algorithm was published [7]. The general idea of backtracking when encountering a conflict still forms the basis of many newer algorithms that have been developed. One of them was the concept of conflict-driven clause learning. This allowed us to add clauses and to backtrack further on the stack to save time. GRASP applies this idea of adding clauses and with UIPs we could keep the size of new clauses limited.

Another type of problem of SMT were system of (integer) linear equations. The eager method provided a rather direct and extensive way of solving these systems by converting integers to a sequence of bits and translating the operators into expressions. This gave a lot of overhead and we had to introduce many new variables to keep track of intermediate values. With Gomory's cut and the Fourier-Motzkin elimination method we could solve this problem with less additional work. The first algorithm one used a shortened version of Gomory's cut, an algorithm that extends simplex to integer linear programming problems. The second algorithm gradually decreased the number of variables in our SOLI with the FME method, until we were left with a trivial problem. Then we briefly saw the Omega test that extended this to integer inequalities.

Lastly, SMT solvers can be used as application in Liquid Haskell. With Liquid Haskell we can use refinement types to check properties of functions. As an example, we took one of De Morgan's laws and saw how Liquid Haskell translated the function and the refinement types to SMT code and how it utilizes an SMT solver to verify the statement.

References

[1] Michael Alekhnovich, Edward A Hirsch, and Dmitry Itsykson. Exponential lower bounds for the running time of dpll algorithms on satisfiable formulas. In *SAT 2005*, pages 51–72. Springer, 2006.

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

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[4] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Conflict-driven clause learning sat solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153, 2009.

[5] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[6] George B Dantzig. Fourier-motzkin elimination and its dual. Technical report, STANFORD UNIV CA DEPT OF OPERATIONS RESEARCH, 1972.

[7] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[8] Ralph E Gomory et al. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical society*, 64(5):275–278, 1958.

[9] Jan Friso Groote and Hans Zantema. Resolution and binary decision diagrams cannot simulate each other polynomially. *Discrete Applied Mathematics*, 130(2):157–171, 2003.

[10] Marijn JH Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In *Haifa Verification Conference*, pages 50–65. Springer, 2011.

[11] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[12] David Lichtenstein and Michael Sipser. Go is polynomial-space hard. *Journal of the ACM (JACM)*, 27(2):393–401, 1980.

[13] João P Marques-Silva and Karem A Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[14] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[15] John C Nash. The (dantzig) simplex method for linear programming. *Computing in Science & Engineering*, 2(1):29–31, 2000.

[16] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 4–13. IEEE, 1991.

[17] Cesare Tinelli. Reals. `www.smtlib.cs.uiowa.edu/theories-Reals.shtml`, 2017.

[18] Niki Vazou. Liquid haskell: Haskell as a theorem prover. volume 4 of *5*, chapter 1, page 5. University of California, San Diego, 2016.

[19] Joost P Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, 1998.