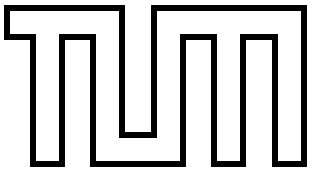


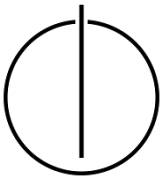
FAKULTÄT FÜR INFORMATIK

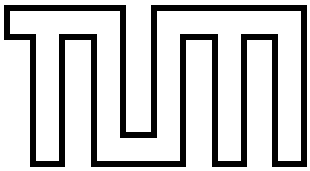
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Functional Implementation of an Optimized UNSAT Proof-Checker

Maximilian Kirchmeier





FAKULTÄT FÜR INFORMATIK

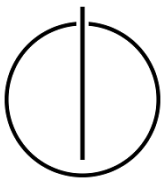
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Functional Implementation of an Optimized UNSAT Proof-Checker

Funktionale Implementierung eines Optimierten UNSAT-Zertifikatsprüfers

Author:	Maximilian Kirchmeier
Supervisor:	Prof. Tobias Nipkow
Advisor:	Dr. Peter Lammich
Date:	May 15, 2017



I assure the single-handed composition of this thesis only supported by declared resources.

Munich, May 22, 2017 Maximilian Kirchmeier

Abstract

This thesis presents funk-DRAT, an optimized implementation of an unsatisfiability proof checker, written in the functional programming language OCaml. An unsatisfiability (UNSAT) proof can be output by a SAT solver in case it finds a formula to be unsatisfiable, so the result can be verified later. The presently most common format for such proofs is the DRAT format, which was introduced together with the associated checker DRAT-trim.

funk-DRATis also based on this proof format. However, funk-DRAT's primary goal was to serve as a kind of "experimental platform" to explore, what an efficient functional design of such a proof checker would look like, in order to make it easier to develop a *formally verified* DRAT proof checker in the future.

To achieve this efficiency, funk-DRAT builds on the methods DRAT-trim employs, but also incorporates a few new optimizations that were conceived of while reconsidering these methods for a functional style. Thanks to these optimizations, funk-DRAT is capable of checking proofs about half as fast as DRAT-trim, which can be rated as fairly successful for a functional approach competing with a C program.

The thesis first introduces some background theory of proof checking, before exploring DRAT-trim's implementation. It then discusses the challenges of a functional implementation and presents the new optimizations, which allowed funk-DRAT to reach the results that are shown at the end of the thesis.

Zusammenfassung

Diese Bachelorarbeit stellt funk-DRAT vor, eine optimierte Implementierung eines Prüfers für "UNSAT-Zertifikate" (UNSAT für "Unsatisfiability" $\hat{=}$ Unerfüllbarkeit), geschrieben in der funktionalen Programmiersprache OCaml. Ein UNSAT-Zertifikat kann von einem "SAT Solver" ausgegeben werden, wenn dieser eine Formel für unerfüllbar erklärt, um dieses Ergebnis im Nachhinein belegen zu können. Das momentan weitverbreitetste Format für solche Zertifikate ist das DRAT Format mit dem zugehörigen Prüfer DRAT-trim.

funk-DRAT wurde ebenfalls zum Prüfen von DRAT Zertifikaten entwickelt. Allerdings war es vor allem das Ziel von funk-DRAT, eine "Experimentierplattform" zu bieten, mithilfe welcher festgestellt werden konnte, wie ein effizientes funktionales Design eines solchen Zertifikatsprüfers aussehen könnte, um die spätere Entwicklung eines *formal verifizierten* Prüfers zu erleichtern.

Um diese Effizienz zu erreichen, baut funk-DRAT auf den Methoden von DRAT-trim auf. Allerdings enthält es zusätzlich noch einige Optimierungen, die während des Überdenkens dieser Methoden zum Zweck der Neuimplementierung entstanden. Dank dieser Optimierungen kann funk-DRAT Zertifikate etwa halb so schnell prüfen wie DRAT-trim, was angesichts der inhärenten Geschwindigkeitsnachteile einer funktionalen Implementierung im Vergleich zu einem optimierten C-Programm ein durchaus annehmbares Ergebnis darstellt.

Die Arbeit stellt zuerst Hintergründe des Zertifikatsprüfens vor und widmet sich dann einer konkreten Implementierung im Stil von DRAT-trim. Anschließend diskutiert sie die Herausforderungen einer funktionalen Implementierung und stellt die neuen Optimierungen vor, die es funk-DRAT erlaubten, die Resultate zu erreichen, die am Ende der Arbeit gezeigt werden.

Acknowledgements

First and foremost I would like to thank Prof. Tobias Nipkow and my advisor Dr. Peter Lammich for giving me the opportunity to write my Bachelor's thesis on a topic that rewarded the thorough exploration of some interesting new concepts and the subsequent "playing around" with these concepts quite a lot. In retrospect I am all the more glad I took the time to look for a topic that really intrigued me, as the many hours working on this thesis would have been a lot more tedious and less enjoyable otherwise.

But that's not all I have to thank my advisor for, as without his clear explanations of proof checking when the topic was still completely new to me, it's easily imaginable that understanding the code of DRAT-trim could have taken a few more months...

Though it has been a while since I took part in it, thanks to Jasmin Blanchette and the rest of the "MCs" behind the old Haskell-Wettbewerb [6] are also in order, who are more than anyone else responsible for my interest in functional programming and probably to blame for my penchant for spending ages optimizing functional programs.

And lastly I would like to thank my friends and family for being supportive of me, while the answer "sorry, still writing my thesis" just wouldn't change.

Contents

1. Introduction

The "Boolean Satisfiability Problem", often simply abbreviated as *SAT*, concerns itself with determining whether there is a "solution" to a Boolean formula, in the sense of an assignment of values (`1` or `0`) to the set of boolean variables the formula refers to, such that the whole formula evaluates to `1`. The algorithms proving the existence of these assignments (or reporting their non-existence) are called *SAT solvers* and have become quite an active field of research in Computer Science. This is on one hand owed to the fact, that SAT is still the prototypical NP-complete problem [10] and thus considered the "reference problem for an enormous variety of complexity statements" [4]. On the other hand, the generality of propositional formulas allows problems from a large number of different areas (such as circuit design, artificial intelligence, software and hardware verification or cryptanalysis [29, 35]) to be expressed through boolean constraints and therefore solved through SAT solvers, which makes these highly optimized programs a very attractive multi-purpose tool for tasks that do not (yet) have a specialized solution.

This high level of optimization comes at a price of increased complexity however, which means that in recent years, SAT solvers have themselves been found to contain a number of bugs [7, 8]. To combat this issue, state-of-the-art solvers can be instructed to not only report the result of their findings, but also output a proof confirming them. The idea behind these proofs is, that the logic required to verify them can be much simpler than the algorithms needed to produce them, which amounts to a marked reduction in the size of the code-base one has to trust in order to put faith in the result.

For satisfiable formulas, a proof can simply consist of the found variable assignment making the formula evaluate to `1`, which is trivial to check. In the unsatisfiable (UNSAT) case however, these proofs have to somehow show the non-existence of a satisfying assignment, requiring a more complicated approach. The UNSAT proofs this thesis will explore in detail consist of a record of the deduction steps taken by the SAT solver, ultimately showing the formula's unsatisfiability. It's then the purpose of a proof checker to verify such proofs by retracing the steps and making sure each one is a valid inference, ending with verifying the deduction of unsatisfiability, which proves the formula UNSAT.

In this thesis, the proof checker funk-DRAT will be presented. It was written in the functional programming language OCaml and is capable of verifying proofs in the most widely adopted UNSAT-proof format called DRAT. The motivation for implementing a performance critical program (verification of very large proofs can easily take hours) in a functional style despite the inherent performance drawbacks is based in the intended future application of this research.

Since *mechanically verifying* a whole SAT solver is rather complicated because of their complexity, there have instead been an increasing number of efforts to develop verified proof checkers [12, 11, 39, 13, 40], in order to further minimize the trusted code base to that of the well-tested theorem prover. None of these verified proof checkers so far directly operates on the DRAT proof format however, which is the only one natively supported by most

commonly used SAT solvers (e.g. CryptoMiniSat [34], Glucose [33], Riss [27]). Instead, they require a separate program as an interim stage, which takes care of pre-processing the DRAT proofs and translating them into more verbose custom proof formats that are then so simple to check, that the verified checker can accomplish it in a reasonable time. This interim stage itself does not need to be certified (as the resulting proof still has to hold up against the original formula) and can therefore be reasonably fast, but still introduces some overhead. Ideally, a mechanically verified checker would be directly capable of checking DRAT proofs, which is precisely the motivation behind this thesis. funk-DRAT, in its form as an optimized functional implementation of a DRAT-proof checker, is meant as a kind of "stepping stone" towards a mechanically verified DRAT checker, by exploring what an efficient architecture for a functional formulation of this problem might look like.

In the following, the theory behind proof checking will be presented in chapter 2, before moving on to a short overview of previous works in the same field of research in chapter 3. Afterwards, the "reference" implementation of a DRAT proof checker in the form of DRAT-trim will be discussed in chapter 4. Finally in chapter 5, the challenges that come with the functional implementation of an efficient proof checker will be explored and the methods and optimizations funk-DRAT used to combat them shown, as well as the results that these optimizations allowed it to achieve.

2. Background

Checking of unsatisfiability proofs is deeply connected to a number of topics related to SAT, which is a fairly complex and densely packed subject. In order for the reader to be able to understand the workings of a proof checker without in-depth knowledge of this field, the following sections will contain a (hopefully just detailed enough) overview of the necessary concepts.

2.1. Definitions and Notation

First of all, it is essential to introduce some definitions which will be used throughout the thesis.

2.1.1. Conjunctive Normal Form (CNF)

A propositional formula is said to be in conjunctive normal form if it is a "conjunction of disjunctive clauses", i.e. if it connects a sequence of *clauses* with the `AND` operator, which themselves consist of a number of *literals* connected with the `OR` operator. Each literal can either refer to a Boolean variable (a) or its negation ($\overline{a}$). One such formula could for example be $F = (a \vee b) \wedge (\overline{b} \vee c) \wedge \overline{d}$. Mathematically a formula can also be seen as a set of clauses, with each clause a set of literals: $F = \{\{a, b\}, \{\overline{b}, c\}, \{\overline{d}\}\}$

CNF is not a limitation on the expressiveness of Boolean formulas however, since any propositional formula can be converted to CNF, as shown by Tseitin's construction [36]. Because of this versatility and its role as the starting point for the important DPLL-algorithm, which will be described later, CNF has been established as the de-facto standard form for input to SAT solvers [18, pg. 91].

2.1.2. Assignments

An *assignment* is a mapping $\tau : L \to \{\mathtt{0}, \mathtt{1}\}$ from a set of literals (L) to truth values. It can also be seen as a set of literal-value tuples: $(\tau(a) = \nu) \Leftrightarrow ((a, \nu) \in \tau)$

The set of literals L contains two literals for every variable x in the set of variables V, the positive literal x, which is equivalent in truth value to that of the variable x, and the negative literal $\overline{x}$, which is equivalent to the negation of variable x. Since the two literals are actually referring to the same variable, assigning a value to one literal always implies assigning the inverted value to its negation as well: $(\tau(a) = \mathtt{1}) \Leftrightarrow (\tau(\neg a) = \mathtt{0})$, with $\neg a = \overline{a}$ and $\neg\overline{a} = a$. If an assignment τ assigns a value to every literal in L, it is called a *complete assignment*. Otherwise, it is a *partial assignment* [5].

Literals, variables, clauses and formulas are all logical expressions that either evaluate to a truth value under an assignment or remain *unresolved*, because some of the literals

involved are still *unassigned*. If they can be evaluated to a value, they are either *satisfied* (for `1`) or *falsified* (for `0`).

SAT is then the problem of determining whether there is an assignment, under which a formula F is satisfied. Such an assignment is called a *satisfying assignment*. In practice however, one is usually not only interested in whether there *is* a satisfying assignment, but also in what it looks like. All practical SAT solvers therefore also output the satisfying assignment they educed, if they proof a formula to be SAT [18].

Note that an assignment can also be interpreted as a simplification of the Boolean formula, as is done in some algorithms in the literature (e.g. [18, pg. 93]). In this scheme, assigning `1` to a literal is equivalent to removing all clauses that contain the literal, while assigning `0` corresponds to the removal of the literal from all clauses.

2.1.3. Clauses

Since clauses represent the main "entity" that both SAT solvers and proof checkers operate on, it is important to introduce some often used terminology for them. First of all, the *empty clause* ϵ, that without any literals, is always unsatisfiable and its occurrence in a CNF formula implies its unsatisfiability as well. Proving the unsatisfiability of a formula is then equivalent to showing the formula implies the empty clause.

$$UNSAT(F) \iff (F \to \epsilon)$$

It's therefore the goal of any UNSAT proof to provide verifiable steps that lead to the deduction of the empty clause, which shows that unsatisfiability was already implied by the original formula.

If, on the other hand, a clause has literals, but all of them are falsified by an assignment, it is called a *conflict clause* and indicates that the assignment that caused the conflict can not be a satisfying one.

Another important concept is that of the *unit clause*, which has one unassigned literal and only falsified literals otherwise. Logically speaking, this implies the literal to be `1` and usually leads to the corresponding assignment immediately (which is referred to as unit propagation and will be explained in detail in 2.3.1). When a literal is assigned because of a unit clause, that clause is said to be the *reason* for the assignment. The set of all clauses that served as reasons (even transitively) in falsifying a clause and making it a conflict clause, will be referred to as its *dependencies*.

A clause that already has at least one literal assigned to `1` is *satisfied*. And finally, when a clause contains both a literal and its negation, it is called a *tautology* and is satisfied per definition.

2.2. Resolution

One very important logical operation for proving the unsatisfiability of propositional formulas is *resolution*.

Proposition 1. *The resolution rule states, that for a CNF formula with the two clauses C_1 and C_2 containing the conjugated literals l and $\bar{l}$ respectively, a new clause $R = C_1 \diamond C_2$ (the* resolvent *on* pivot literal *l) with all the literals in the two* antecedents *C_1 and C_2*

(except for l and $\bar{l}$) is also implied by the formula and can be added to it while preserving (un)satisfiability [16, pg. 352]:

$$\underbrace{(a_1 \vee ... \vee a_n \vee l)}_{C_1 \in F} \wedge \underbrace{(\bar{l} \vee b_1 \vee ... \vee b_n)}_{C_2 \in F} \rightarrow \underbrace{(a_1 \vee ... \vee a_n \vee b_1 \vee ... \vee b_n)}_{R}$$
$$\Rightarrow F \rightarrow F \cup \{R\}$$

Proof. That this has to hold true can be easily verified, as every assignment that satisfies F (and thus $C_1 \wedge C_2$) also has to assign `1` to at least one literal in R. Otherwise either C_1 or C_2 would be falsified, as l and $\bar{l}$ can't both be assigned `1`. Conversely, this means that if R is unsatisfiable, F and more specifically the term $C_1 \wedge C_2$ is also unsatisfiable. □

Resolution is thus deeply connected to unsatisfiability. In fact, it is always possible to deduce the empty clause from a CNF formula in a series of resolutions if and only if the formula is UNSAT [18, pg. 101], which will become important later.

2.3. DPLL-based SAT solvers

In the following we will focus on a particular class of SAT solvers, namely those based on the "Davis-Putnam-Logemann-Loveland" (*DPLL*) algorithm [15]. Before explaining the algorithm, it is important to look at how *unit propagation* functions, however.

2.3.1. Boolean Constraint Propagation (BCP)

Boolean constraint propagation, also known as unit propagation, is a vital part of both SAT solving and proof checking. It refers to the propagation of assignments resulting from clauses that have become unit, i.e. those clauses whose literals are almost completely falsified except for one unassigned literal. To propagate the unit, that literal is assigned `1`, which in turn might cause other clauses to become unit or lead to a conflict. In case a conflict is derived, propagation can abort directly and report the conflict, because the overlying algorithm will either have to "backtrack" to an earlier decision point or declare the formula UNSAT. Further propagations would be discarded in either case. If there is no conflict, BCP will "absorb" all unit clauses into the current assignment, including the newly produced ones. Since all former unit clauses are then satisfied, the formula at this point will only contain satisfied clauses or unresolved clauses with at least two unassigned literals.

2.3.2. The DPLL algorithm

Published in 1962, DPLL is still quite relevant and continues to serves as the basis for the most efficient complete (in the sense of always terminating, whether the formula is satisfiable or not) SAT solvers available today [18, pg. 92]. At its core, the idea behind DPLL is that of an exhaustive search of the whole space of possible assignments, looking for a satisfying one. See listing 1 for pseudo-code describing the recursive implementation of the algorithm. The two functions `satisfiesFormula` and `getUnassignedLiteral` should be self-explanatory, while `propagateUnits` was explained above.

```
type resultDPLL = SAT of assignment_type | UNSAT
type resultUnitPropagation = Success of assignment_type | Conflict

let recursiveDPLL assignment =
  propagationResult ← propagateUnits assignment
  match propagationResult with
  | Success propagatedAssignment :
      τ ← propagatedAssignment
      if satisfiesFormula τ then
        return (SAT τ)
      else
        l ← getUnassignedLiteral τ
        match recursiveDPLL (τ ∪ {(l, 1)}) with
        | UNSAT : return (recursiveDPLL (τ ∪ {(l, 0)}))
        | SAT x : return (SAT x)
  | Conflict : return UNSAT
```

Listing 1: Pseudo-code showing the recursive DPLL algorithm [18, pg. 93]

DPLL works by repeatedly choosing an unassigned variable and "trying out" the two possible assignments for it, until there either is a conflict or the whole formula is satisfied. The literals which are assigned values simply because the algorithm has "decided" to try them are called *decision literals*.

As a side-note, the pseudo-code style used in this thesis is basically imperative, but borrows some of OCmal's syntax especially for type definitions and case analysis for reasons of clarity.

2.3.3. DPLL's predecessor

Though DPLL was the algorithm that made SAT solving practical, it was only a refinement of a previously proposed algorithm called the "Davis-Putnam" (*DP*) algorithm [14]. Though DP also employs unit propagation, the main difference lies in the way unassigned literals are removed from the formula. Where DPLL leaves the formula untouched and merely assigns a value to the selected literal l at each branching step, DP modifies the formula "in place" and eliminates l from it. It does this by first gathering all the possible resolvents on l and adding them to the formula, then removing all clauses still containing l or $\bar{l}$ afterwards. Once the empty clause is deduced (implying unsatisfiability) or a fix-point is reached with no more possible resolutions (signaling satisfiability), the algorithm stops. Though it served as the basis for the derivative DPLL, this algorithm was never itself used extensively because of the obvious drawbacks of continuously storing new resolvents compared to only assigning values to literals [16, pg. 356].

2.3.4. Conflict-Driven Clause Learning

In contrast to DP, DPLL does not normally produce new clauses and add them to the formula. As it turns out, bringing DPLL a bit closer to its predecessor again and giving it the ability to do exactly that in some few cases can improve its performance considerably

however [32]. This is where the concept of "conflict-driven clause learning" (CDCL) comes from, which constitutes the foundation of the most efficient modern Boolean SAT solvers [5]. The basic idea behind it is, that by "learning" clauses whenever a conflict arises and extending the formula with them, the "cause" of the conflict can be made available to later assignment attempts more directly. This can save time by causing a conflict earlier when the algorithm tries a different assignment still containing the combination of decision literals that was responsible for the conflict.

Example 1. Consider the formula $F = ... \wedge (a \vee b \vee c \vee d) \wedge (a \vee c \vee \overline{d}) \wedge (b \vee \overline{c})$ that a DPLL solver is trying to satisfy using the assignment $\tau = \{..., (a, \mathtt{0}), (x, \mathtt{1}), (y, \mathtt{1}), (b, \mathtt{0})\}$. After assigning b to $\mathtt{0}$, unit propagation will lead to a conflict, as $(b \vee \overline{c})$ will become unit and lead to the assignment of $\overline{c}$. This in turn will make both clauses $(a \vee b \vee c \vee d)$ and $(a \vee c \vee \overline{d})$ unit with conflicting unit literals, which means that the propagation of one of them will result in the other one becoming a conflict clause. We will assume w.l.o.g. $(a \vee c \vee \overline{d})$ to be the conflict clause.

To save work and reach a conflict faster when a similar assignment is attempted later, the solver can now learn from this conflict by analyzing it and deducing a clause that "captures" the conflict at an earlier point. In order to see how this might be done, it helps to look at the causal chain leading to the conflict. Figure 2.1 illustrates the causality with the example from above.

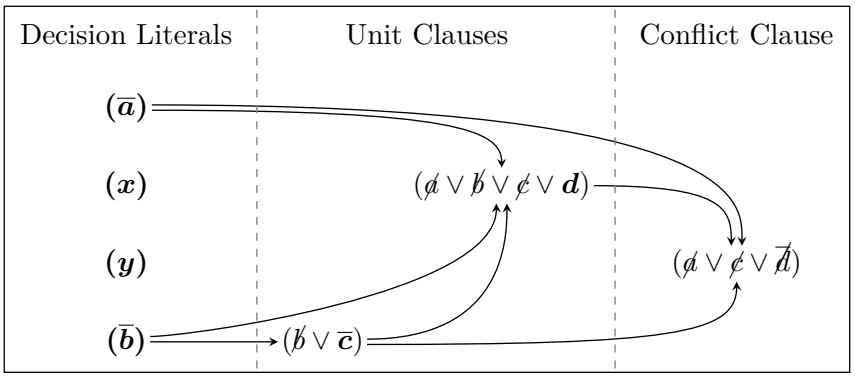


Figure 2.1.: Implication graph showing the cause of a conflict. Each edge corresponds to a unit propagation. Falsified literals are struck through and propagated literals are in bold

It can be noted that the events leading to a conflict always start with a newly assigned decision literal (which can be interpreted as an artificial unit clause), triggering at least one unit propagation and ending with a falsified conflict clause. Especially the fact that it's always a propagated unit clause falsifying the last unfalsified literal in the conflict clause is of importance here, because this means that the conflict clause and that last unit clause always have exactly one conflicting literal-pair. Resolution can therefore be used to resolve these two clauses, which results in a "new" clause with all falsified literals. In this case the resolution would yield $(a \vee b \vee c \vee d) \diamond (a \vee c \vee \overline{d}) = (a \vee b \vee c)$, which already contains more accurate information about the conflict than the clauses in the formula and is one candidate for a clause learned from this conflict.

However, since this clause itself can also be seen as a conflict clause that was falsified by a unit propagation, the same procedure can be repeated by resolving it against $(b \vee \overline{c})$. This results in the even more condensed new conflict clause $(a \vee b)$, which is the second (and probably better) candidate for a learned clause. For both of these possible learned clauses, adding them to the formula would have resulted in a conflict at an earlier time, which is precisely the benefit of clause learning. If the clause $(a \vee b)$ was added to the formula for example, then the algorithm wouldn't even have to propagate any units to see that the assignment $\tau = \{..., (a, \mathtt{0}), (x, \mathtt{0}), (y, \mathtt{1}), (b, \mathtt{0})\}$, which it might try later, can't be a satisfying one and therefore would be able to discard it instantly.

This very basic example shows already that clause learning is a powerful, but complicated technique. The choice of resolvents (which isn't usually as obvious as here) and the right cut-off length for the resolution chain have spawned numerous different learning strategies, which are the subject of more detailed works already [5, 1] and would go beyond the scope of this thesis to explore.

The principle behind CDCL shown here is important to understand though, because it is this mechanism that lets SAT solvers emit proofs of unsatisfiability easily. Since every learned clause is a product of resolution between already existing clauses, it is obviously already implied by the original formula. Furthermore, a CDCL solver learns new clauses from every conflict, only stopping (in the unsatisfiable case) when it can "learn" the empty clause, which means it encountered two conflicting unit clauses. The whole set of learned clauses for a given formula is thus one possible proof of unsatisfiability [1], a concept that will be explored further in the following.

2.4. Unsatisfiability Proofs

As mentioned before, it is difficult to prove the non-existence of a satisfying assignment directly. To show that a formula is unsatisfiable, it suffices however to prove that it implies the empty clause. There are a few different forms that these proofs can come in, but at their core, they always represent the introduction of a number of new clauses $\{C_0, C_1, ..., C_i\}$ (commonly called *lemmas*) to the formula, each one demonstrably preserving the satisfiability of the formula and culminating with the empty clause $C_i = \epsilon$.

It's a proof checker's task to verify that each lemma maintains *satisfiability equivalence*. That is, it has to make sure that if (and only if) there was a satisfying assignment for F_i, there is one for $F_{i+1} = F_i \wedge C_i$, with F_0 containing only the clauses of the original formula F. A lemma C_i that satisfies this condition is called *redundant* relative to the formula F_i.

In other words, the checker has to verify that $(F_i \rightarrow \epsilon) \Leftrightarrow (F_{i+1} \rightarrow \epsilon)$ holds for each step. This makes it clear that it is a proof's end-goal to add the empty clause as a lemma, because $F_i \wedge \epsilon \rightarrow \epsilon$ is obviously true and would by induction also imply $(F_0 \rightarrow \epsilon)$ and $UNSAT(F)$.

Proofs of unsatisfiability therefore consist of a series of lemmas easily verifiable as redundant, that in sum show the formula can't be satisfied.

Since this is the fundamental idea behind all unsatisfiability proofs, the main difference between the various proof formats lies instead in how much information they store for each proof step and how much of it the checker has to infer by itself. This results in a trade-off between verbosity (making for easier verification, but larger proofs and more complicated information gathering on the SAT solver side) and compactness (with smaller proofs and an

easier proof generation for the SAT solver, but more work for the checker). Generally, there are two "families" of proofs with different approaches to this trade-off, namely resolution proofs and clausal proofs.

2.4.1. Resolution Proofs

Resolution proofs are clearly on the "verbose and easy to verify, but harder to generate" side of the divide. As the name might imply, each lemma in a resolution proof is a direct product of a number of input clauses that are resolved in a specified order. As an example, the TraceCheck format will be discussed here as one typical resolution proof format [19]. With this format, every clause in the original formula is assigned an ID, based on its occurrence in the input file defining the CNF formula. Every proof step then contains a list of the IDs of the clauses that are supposed to be resolved against each other (the antecedents), but does not necessarily contain the new lemma itself. The new lemma only has to be stated explicitly, if the solver had problems with listing the antecedents in exactly the order the resolutions need to take place and wants to delegate the task of determining the correct order to the checker. In either case, the product of these resolutions is then added to the formula and also referenced by an ID, to make it possible for further resolutions to refer to it.

The beauty of resolution proofs from the perspective of a proof checker lies in the fact, that no more work is necessary to verify each proof step after the lemma it postulated was derived through resolution. That's because resolution by definition can't result in a clause that was not already implied by the resolvents (see section 2.2). If a proof manages to produce the empty clause through resolution only, its validity therefore follows implicitly, which is why resolution proofs are so easy to verify. Furthermore, they are also easy to produce for *basic* CDCL solvers, because the already discussed learned clauses are perfectly suitable to be recorded in form of a resolution proof.

They have their drawbacks, though. The main problem is, that *modern* CDCL solvers first preprocess the formula using a few techniques that make it "nicer" to solve. Some of these techniques are not expressible solely through resolution steps [19, pg. 10]. This is why pure resolution proof formats have become less common in favor of the more flexible clausal proof format.

2.4.2. Clausal Proofs

Clausal proofs at their most basic just consist of the lemmas that the formula should be extended with. Compared to resolution proofs, they are easier to produce for solvers and usually more compact, because they carry less redundant information (such as antecedent lists) with them. On the other hand, they are a lot more complicated to verify for the proof checker, because many parts of the logic required to produce the clause in the solver are also required in the checker to verify its validity.

2.5. Redundancy Properties

The main challenge with verifying the proof steps of a clausal proof is to make sure that each lemma is redundant in regard to the formula before it. There is a number of different

redundancy properties that a lemma might have, some stronger than others. Consequently, a clausal proof format's ability to retrace steps taken by the SAT solver depends solely on the redundancy it allows its lemmas to have. Though there are a lot of these different "kinds" of redundancy [24], the four most important ones will be presented here.

2.5.1. Tautology

The most trivial kind of redundancy is tautology. A clause that contains both a literal and its negation is always satisfied and therefore doesn't constrict any formula it is added to in satisfiability.

$$T(C) = \exists_l : (l \in C) \wedge (\neg l \in C)$$

2.5.2. Asymmetric Tautology

Asymmetric tautology (AT) on the other hand is already a lot more useful. It is defined through the so-called *Asymmetric Literal Addition*, which describes the process of extending a given clause C with literals from other clauses in the CNF formula F that have exactly one literal not contained in C. For each of these clauses, the negation of the additional literal is added to C. This process is repeated until C has reached a fixpoint. C is said to have AT in relation to F, if the clause produced by $ALA(F, C)$ is a tautology:

$$L_{\Delta C} = \{\, \neg l \mid X \in F \wedge l \in (X \setminus C) \wedge |X \setminus C| = 1 \,\}$$

$$ALA(C, F) = \begin{cases} ALA(C \cup L_{\Delta C}, F) & \text{if } L_{\Delta C} \neq \emptyset \\ C & \text{otherwise} \end{cases}$$

$$AT(C, F) = T(ALA(C, F))$$

Proposition 2. *To determine whether a clause has AT, it suffices to assign* `0` *to all its literals and perform a unit propagation afterwards. If and only if the propagation causes a conflict, the clause has AT [21].*

Proof. To see why this works, consider the contents of $L_{\Delta C}$. It contains exactly the literals that will be falsified by the unit propagation after C is falsified. Therefore, if and only if $ALA(C, F)$ is a tautology, the unit propagation will try to assign conflicting values to at least one literal, which leads to a conflict. □

Because of this property, clauses with AT are also called *Reverse Unit Propagation (RUP)* clauses.

Proposition 3. *Adding a RUP clause to a formula maintains logical equivalence, i.e. the set of satisfying assignments does not change.*

Proof. Since an assignment that would falsify C would also falsify the whole formula (as the BCP showed), F obviously already "encodes" the condition, that AT least one of the literals in C needs to be `1`. Adding the clause to the formula only states this condition more explicitly, but does not impose any new restrictions on assignments. The resulting formula is therefore not only satisfiability- but logically equivalent. □

However, a clausal proof that consisted solely out of RUP lemmas would only be as versatile as a pure resolution proof, since every RUP clause can be written as the product of a resolution (see 2.3.4). For clausal proofs to really become useful, stronger redundancy properties are therefore required. This is where resolution-based redundancy comes in.

2.5.3. Resolution (Asymmetric) Tautology

Both of the previous tautologies have resolution-based generalizations in the form of *Resolution Tautology* (RT) and *Resolution Asymmetric Tautology* (RAT). They are defined as follows:

Given a formula F and a clause $C \in F$, C has property RP (with $P \in \{T, AT\}$) if and only if either (i) C has the property P, or (ii) there is a literal $l \in C$ such that for each clause C' in $F_{\neg l}$ (the set of clauses that contain the literal $\neg l$), each resolvent $C \diamond C'$ has P [21, pg. 348]. In the latter case, it can be said that C has RP *on the pivot literal* l.

Proposition 4. *Adding a clause with RT/RAT to a formula maintains satisfiability equivalence, i.e. the existence of satisfying assignments does not change.*

Proof. To show that adding a clause C with RT/RAT to F results in a satisfiability-equivalent formula, consider an assignment τ that satisfies F, but falsifies C. Since τ falsifies C, it must also assign `0` to l in particular. However, for both RT and RAT it is always possible to construct a new assignment τ' that satisfies C and F both, by keeping all assignments from τ, but assigning l to `1`.

- For clauses with RT (so called *blocked clauses*), it is obvious that an assignment falsifying C will assign `1` to at least one literal other than $\neg l$ for every clause C' in $F_{\neg l}$, because $C \diamond C'$ is a tautology.
- For a clause C with RAT, every resolvent in $(C \diamond F_{\neg l})$ has AT. It follows then from the definition of asymmetric tautology, that $F \cup (C \diamond F_{\neg l})$ is logically equivalent to F. Therefore, τ must also satisfy $F \cup (C \diamond F_{\neg l})$ and by extension $(C \diamond F_{\neg l})$. In order for τ to satisfy the clauses in $(C \diamond F_{\neg l})$, it must assign `1` to at least one other literal beside $\neg l$ for every clause in $F_{\neg l}$.

In both cases τ' will still satisfy F (including the clauses in $F_{\neg l}$), but also satisfy C. Ergo, a formula F that was satisfiable before a clause with RT or RAT was added also stays satisfiable afterwards [24]. Since adding a clause can't make an unsatisfiable formula satisfiable either, satisfiability is preserved in both directions. □

Especially RAT is very important for proof checking, as it is the strongest form of redundancy that preserves satisfiability equivalence [21]. That means it covers all other forms of redundancy; a clause with T, AT or RT always has RAT as well. Unlike AT however, it also allows lemmas that express more complex pre- or inprocessing steps of the SAT solver, such as "blocked clause addition, bound[ed] variable addition, extended resolution, and extended learning" [40, pg. 236]. For bounded-variable addition (BVA), which reduces formulas in size by introducing new variables [28], a short example will be shown here.

Example 2 (taken from [19])**.** Consider a CNF formula

$$F = (\overline{a} \vee \overline{b} \vee \overline{c}) \wedge (\overline{d} \vee \overline{e}) \wedge (a \vee d) \wedge (a \vee e) \wedge (b \vee d) \wedge (b \vee e) \wedge (c \vee d) \wedge (c \vee e)$$

An elegant proof of the unsatisfiability of this formula employs BVA. First, five clauses referring to a new variable f are added and can "replace" the last six binary clauses in F:

$$F' = (\overline{a} \vee \overline{b} \vee \overline{c}) \wedge (\overline{d} \vee \overline{e}) \wedge (f \vee a) \wedge (f \vee b) \wedge (f \vee c) \wedge (\overline{f} \vee d) \wedge (\overline{f} \vee e)$$

These new clauses all have RAT on f or $\overline{f}$ respectively, because the possible resolvents between F'_f and $F'_{\overline{f}}$ are all already contained in F and therefore trivially have AT. Adding these clauses thus does not change F's satisfiability, though it would change its set of satisfying assignments (if there were any) by introducing a new variable. Afterwards, the proof becomes easy and consists of only two lemmas with AT: $\{(f), \emptyset\}$. Though there are (shorter) RUP proofs for this problem, it would not have been possible to express this specific "formula simplification" as resolution-based proof steps [19, pg. 10], which makes RAT-based proofs so versatile.

For this reason, the most common UNSAT proofs and proof checkers today are RAT-based. This includes both DRAT-trim, which will be presented in chapter 4, and funk-DRAT, the proof checker that was developed during the course of this thesis.

2.6. Relevant File Formats: DIMACS and DRAT

In practice, the example from above would be communicated to SAT solvers and proof checkers as shown in figure 2.2. For the CNF formula in DIMACS format, the first line defines the dimensions of the problem: It contains 5 variables and 7 clauses. The clauses themselves are encoded similarly in all three formats: Positive literals are stated as positive numbers (a becomes `1`, b becomes `2` etc.) and negative literals as the corresponding negative numbers. Each clause ends with a `0` as delimiter.

CNF Formula

```
p cnf 5 7
-1 -2 -3  0
   -4 -5  0
    1  4  0
    1  5  0
    2  4  0
    2  5  0
    3  4  0
    3  5  0
```

RUP Proof

```
   1 0
   2 0
     0
```

DRAT Proof

```
    6  1  0
    6  2  0
    6  3  0
   -6  4  0
   -6  5  0
d  1  4  0
d  1  5  0
d  2  4  0
d  2  5  0
d  3  4  0
d  3  5  0
      6  0
         0
```

Figure 2.2.: Figure showing the example formula (in DIMACS format) and RAT proof (in DRAT format) from section 2.5.3 alongside one possible RUP proof

Also shown is one simple RUP proof. Proofs in this format simply list the RUP clauses they postulate, which in this case are: $\{(a), (b), \epsilon\}$

The DRAT (for Deletion-RAT) proof represents the proof taken from the example. DRAT proofs differ from the RUP format in three important aspects:

- They can contain variables that the original formula did not. This is useful especially for techniques such as the shown BVA, which introduces new variables.
- The order of literals is no longer arbitrary, as it was with RUP clauses. That's because checking a lemma for RT/RAT redundancy requires knowledge of the "pivot literal", which in the DRAT format is defined to be the first literal in the clause.
- They allow "deleting" clauses, indicated by a `d` as the first character. Deletions can be emitted by a solver as a "hint" to the checker, informing it of the fact that the clause in question will, after this point, not be one of the unit- or conflict clauses necessary to validate further lemmas, and thus can be safely ignored during unit propagation. Though adding this information leads to larger proofs, "streamlining" unit propagation this way improves checking performance [20].

Note that the DRAT format is backwards-compatible with the (D)RUP formats, since every RUP clause also has RAT. The RUP proof here is therefore technically also a (noticeably shorter) DRAT proof, and the DRAT proof shown is so elaborate only to demonstrate the format's capabilities.

3. Related Work

In order to give the reader an overview over the previous works this thesis builds on and to pinpoint the differences to the similarly motivated ones among them, a short chronological summary of the evolution of unsatisfiability proofs will be given here.

First Proof Formats

Unsatisfiability proofs in the context of SAT solvers have come a long way since they were first introduced in 2003, by Zhang and Malik in the form of resolution proofs, which their solver zchaff was capable of outputting [42], and as "conflict clause proofs" (effectively RUP proofs) by Goldberg and Novikov with their Berkmin solver [17]. In 2006, Jussila et al. already first generalized proofs beyond resolution and implemented their format in the EBDDRES solver as a proof of concept [25]. These efforts were mostly limited in scope to the individual solvers though, and not aimed at establishing any kind of standard proof format.

However, a standardized proof format was precisely what the "SAT Competition", the yearly held SAT competition letting the world's best SAT solvers compete against each other, needed to be able to offer their "certified UNSAT"-track. Allen Van Gelder, member of the committee behind the competitions, therefore proposed the resolution proof format "RES" for the SAT Competition 2005 [37]. He also introduced the RUP format for clausal proofs in 2008 [38], though he originally only published a program to convert the RUP format back into RES proofs for actual verification.

He was not the only committee member to try his hand at engineering proof formats however. Armin Biere, also in 2008, published a paper on the PicoSAT solver [3], which introduced another resolution-based format and an accompanying tool called TraceCheck, capable of either verifying the proof directly or converting it to a "pure" resolution proof.

New Formats and the Advent of Verified Proof Checking

At around this time, interest in mechanically verified proof checking started to pick up. The first efforts in this direction did not focus on formally verifying a proof checker however. Instead, in 2009 Weber and Amjad "translated" both CNF formulas and the proofs of their unsatisfiability into a theorem prover's specification logic, and let the prover itself do the checking [39].

Soon afterwards though, Darbari et al. developed the first verified standalone proof-checker "SHRUTI" [13]. It was designed and verified using the Coq proof-assistant [9], but wasn't actually meant to be executed within that environment. Instead, Coq offers the option of "automatically extracting" a more lightweight OCaml program with the same logic. The checker therefore could be seen as somewhat related to funk-DRAT, with two main differences: First, SHRUTI processes proofs in the resolution-based TraceCheck format and is therefore not as versatile as a modern clausal-proof checker. And second, an "automatically

extracted" program can not really be equated to one directly written in the target language, neither performance- nor readability-wise.

The evolution of proof formats and checkers was nowhere near finished at this point, though. Especially 2013 was a busy year, with Heule et al. supplying three new publications: On the one hand, they first introduced a RAT-based proof format, and published a fast checker for it written in C as well as a verified checker written in ACL2 [40, 21]. On the other hand, they developed the RUP format further, extending it with "deletion information" that speeds up verification by allowing checkers to "forget" about no longer needed clauses, which they implemented in their (unverified, but fast) checker DRUP-trim.

Only in the following year 2014 did they unify these two advancements in the form of the DRAT format and their checker DRAT-trim [41]. These are still the most commonly used proof format and checker today, probably in no small part owed to the fact that since they were published, the SAT competition requires its participants to submit proofs for unsatisfiable formulas in the DRAT format [2], which is by now implemented in most commonly used SAT solvers (e.g. CryptoMiniSat [34], Glucose [33], Riss [27]).

2014 also brought another noteworthy event for proof checking: Konev and Lisitsa managed to proof (one part of) the "Erdős Discrepancy Conjecture" using a SAT solver, and verified their result in the form of a DRUP proof [26]. Though there are other examples to show that UNSAT proofs are not only useful as a "safe guard" against incorrect SAT solvers, this particular case serves to demonstrate this quite nicely.

Latest Developments

Because checking proofs in the DRAT/DRUP formats introduces a fair amount of algorithmic complexity and is thus difficult to formally verify, the most recent developments in this direction have instead focused on establishing new, easily checkable "intermediate" proof formats, that DRAT/DRUP proofs can be converted to in reasonable time.

In 2016, Cruz-Filipe et al. proposed the new GRIT format that extends the TraceCheck format with deletion information and makes the proofs easier to check by requiring the "clause dependencies" (the set of clauses that in sum lead to a conflict when the lemma's literals are falsified) of each lemma to be stated in exactly the order that they need to be visited by BCP [12]. They also present a verified checker for this format, also formalized in Coq and extracted to OCaml afterwards. Since GRIT is still resolution-based however, it can only express what is possible with DRUP proofs.

In an article also published in 2016, after work on this thesis had begun, a slightly different format called LRAT was therefore suggested [11]. It can be seen as an extension of the GRIT format, now allowing for RAT lemmas by including the possibility of specifying the clause dependencies for each resolvent on the pivot literal. For checking such proofs, two verified proof checkers were presented, one using Coq and one using ACL2. Furthermore, DRAT-trim was modified to give it the capability of extracting LRAT from DRAT proofs.

As the example of the LRAT format perfectly shows, these verified checkers still have the drawback of requiring a pre-processing step (equivalent to a whole unverified checking run) in order to bring the DRAT proofs in a form that is easy enough to check for verified checking. This thesis therefore tries to explore DRAT proof checking from a more functional perspective, in an attempt to pave the way for a future verified DRAT proof checker.

4. The Reference Implementation: DRAT-trim

Since DRAT-trim still represents the reference for effective clausal-proof checking, the methods it utilizes will be investigated in the following. Note though, that it is not the intention of this chapter to explain DRAT-trim's code, but rather its implementational approach, seen perhaps from a rather "functional" point of view. This way, the differences to funk-DRAT, which is largely based on DRAT-trim's ideas, can be mapped out easier in the next chapter. Pseudo-code similar in idea to DRAT-trim's implementations will therefore be used to demonstrate algorithms, where appropriate.

4.1. Data Structures

Though DRAT-trim's code won't be explained in detail, it is still useful to know the data structures that will hold all the information the algorithms shown operate on. For this reason, the most important ones will be described here.

For each clause and lemma DRAT-trim encounters during parsing of the formula and proof files, it stores the (sorted) literals and some additional information, such as the clause ID, in a dynamically resized integer array. It also adds a reference to each clause to a hash table, indexed by a hash over the literals (as described in [22, pg. 600]). For every clause deletion in the proof, it is then easy to find the corresponding clause and store the corresponding clause ID with the deletion step, so no lookup is necessary during verification. Deletions are also the reason that the literals are sorted at first, because a deletion might quote the literals in a different order than the clause definition. During parsing, DRAT-trim also keeps track of the number of variables that occurred in the formula and proof, since the proof might introduce new variables that the formula's definition did not include.

After it finishes parsing, it can then allocate the necessary data structures with the correct dimensions. Some of these structures are allocated per-literal, while others are allocated per-variable. Assignments, for example, are stored in an array of integers with one entry per literal. This array is called "`false`" (which is only slightly confusing, as it is a keyword in C++, but not in C) and entries in it are set to a non-zero value, if the corresponding literal is *falsified*. While it may seem a little counterintuitive at first, thinking in terms of falsification in this context is actually quite straight-forward, when considering that it's only the falsified literals that lead to new unit- or conflict clauses. Since DRAT-trim internally encodes literals the same way that the DIMACS format does (see 2.6), the references to the per-literal arrays are moved forward to actually point in the middle of the allocated spaces, so that lookups with negative indices (for negative literals) also work as expected. One would therefore check a literal `l` for "satisfiedness" with `false[-l]`, while an unassigned literal is denoted by `!false[-l] && !false[l]`.

```
let assignFalse literal =
  invertedAssignment[literal] ← 1
  pushToEndOfTrail literal

let assignTrue literal =
  assignFalse ¬literal

let undoLastAssignment () =
  lastLiteral ← popFromEndOfTrail ()
  invertedAssignment[lastLiteral] ← 0

let getTrailPosition () =
  return lengthOfTrail

let backtrackAssignmentsTo position =
  while getTrailPosition () != position do
    undoLastAssignment ()
```

Listing 2: Pseudo-code describing assignments and backtracking

Other important structures only need to be dimensioned on a per-variable basis. One is the `reason` array, which contains references to the clauses that became unit and thus served as the "reason" for assigning a value to a variable. The other one is the `falseStack`, which contains the "trail". The trail refers to an ordered sequence of assignments (in this case falsifications). It is important to keep track of for two reasons: On the one hand, unit propagation needs to know which literals were falsified since it was last invoked, so it can update the corresponding clauses. On the other hand, assignments need to be "rolled back" to a previous reference point in some cases (also referred to as *backtracking*), so it is important to know which assignments happened after the given point.

Finally, the arrays required for the *watched literals* structure are allocated. Since the concept of watched literals is not so trivial, it will be explained when DRAT-trim's unit propagation algorithm is discussed.

4.2. Verification Logic

After the data structures are allocated, verification of lemmas can start. In the following sections, it will be explained how this is accomplished.

4.2.1. Assignments and Backtracking

As mentioned above, the "trail" structure makes it very easy to do backtracking of assignments, which is an important aspect of redundancy checks. Listing 2 shows how it can be used to conveniently manage assignments and backtrack to previous assignment-states.

```
let markAsCore clause =
  addToCore clause
  for l in getLiterals clause do
    markLiteral l

let addClauseAndDependenciesToCore clause =
  markAsCore clause
  for l in reversed(trail) do (* newest entries first *)
    if isMarked l && not (isTemporaryAssignment l) then
      markAsCore (getAssignmentReason l)
```

Listing 3: Pseudo-code describing core-marking after a conflict

4.2.2. Core Marking

Trying to verify each lemma one by one, starting at the beginning of the proof, would quickly lead to one important realization: In most cases (unless the proof has been optimized beforehand), only a subset of all lemmas is actually necessary to deduce the empty clause and thus proof the unsatisfiability of the formula. This is because SAT solvers produce clausal proofs by simply recording the clauses they've learnt during the solving process, which means that a possibly substantial fraction of these has been generated while exploring "fruitless branches" of the search space and can be ignored for the purpose of proof verification. Focusing only on the "necessary" part of the whole clause-and-lemma set (called the *core*) permits optimizations with sometimes substantial speed-ups, which is why Goldberg and Novikov already proposed this method of proof checking when introducing the clausal proof [17].

Thus, whenever a clause or lemma is found to be a unit- or conflict clause that was necessary to validate another lemma, DRAT-trim marks it as part of the core. Listing 3 shows pseudo-code describing the procedure to do just that, comparable to the way DRAT-trim implements it.

Note that it's possible for assignments to lack an assignment reason, as the redundancy checks for the lemmas require *temporary* assignments (or *assumed*, as DRAT-trim calls them) without an accompanying unit clause as reason.

4.2.3. Fast Unit Propagation: Watched Literals and Core-First Strategy

As unit propagation is at the core of checking a lemma's redundancy and constitutes the only really computationally expensive step of verifying a clausal proof, a proof checker's performance in large part is determined by that of the BCP algorithm it uses. The main challenge for such an algorithm lies in how to efficiently keep track of all that clauses that might have been affected by a new assignment and could now have become unit or falsified.

It's easy to determine which literals were newly assigned since the propagation was last carried out, as this information is contained in the trail. But for every changed literal, the algorithm still has to have a way of determining which clauses might be impacted. To be able to do this efficiently and save the effort of iterating over every clause for every assignment, a data structure containing back-references to clauses from their literals is needed. The most

```
type resultUnitPropagation = NoConflict | Conflict of clauseType

let assignNewUnitWithReason literal reason =
  assignTrue literal
  setReason literal reason

let propagateUnits () =
  for l_false in getUnprocessedFalsifiedLiterals () do
    watchList ← getWatchListForLiteral l_false
    for clause in (getClausesInWatchList watchList) do
      pos_false ← getLiteralPosition clause l_false
      pos_other ← (1 - pos_false) (* watched literals are kept at positions 0 and 1 *)
      l_other ← getLiteralAt clause pos_other
      if not (isAssignedTrue l_other) then (* if clause not already satisfied *)
        if hasUnfalsifiedUnwatchedLiteral clause then
          l_new ← getUnfalsifiedUnwatchedLiteral clause
          (* move watch from falsified lit to new unfalsified lit *)
          removeClauseFromWatchList l_false clause
          addClauseToWatchList l_new clause
          (* exchange their positions so the watched lits stay at the start *)
          swapLiteralsInClause clause l_false l_new
        else if isAssignedFalse l_other then
          return (Conflict clause)
        else (* l_other is still unassigned -> new unit *)
          assignNewUnitWithReason l_other clause
  return NoConflict
```

Listing 4: Pseudo-code describing unit propagation with the watched literal structure

naïve implementation would, for each literal, simply store a complete lookup list of all the clauses it is contained in. That this would still be far from ideal not only memory- but also runtime-wise should be evident however, as every clause (even those already satisfied or far from being unit/falsified) containing the assigned literal would still have to be iterated over for each assignment.

When thinking about what it means for a clause to become unit or falsified, it becomes apparent that clauses don't actually need to be visited for every assignment to one of their literals. That is because clauses, that still have at least two *unfalsified* (either unassigned or satisfied) literals can never be unit or falsified and thus aren't of interest to BCP. This opens up the possibility of using much more efficient *lazy* data structures. They are lazy in the sense that they contain just enough information for clauses to be checked for a change in state when necessary and not for every assignment to one of their literals.

Watched Literals

The currently most widely implemented such data structure uses the concept of *watched literals*, which was introduced by Moskewicz et al. for their SAT solver Chaff [30]. The basic idea behind it is to choose two unfalsified literals from every clause (provided it even contains more than one literal) and make these two the clause's "watched" literals. At first,

```
type clauseSelector = CoreOnly | NoncoreOnly
type resultUnitPropagation = NoConflict | Conflict of clauseType

(* if a new unit is found in non-core mode, trigger core-only propagation *)
let assignNewUnitWithReason literal reason =
  assignTrue literal
  setReason literal reason
  if getCurrentClauseSelector () = NoncoreOnly then
    propagateUnitsOnClauses CoreOnly

let propagateUnitsOnClauses clauseSelector =
  setCurrentClauseSelector clauseSelector
  (* do normal BCP, but only consider clauses according to 'clauseSelector' *)
  ...

let propagateUnitsCoreFirst () =
  match (propagateUnitsOnClauses CoreOnly) with
  | Conflict conflictClause : return (Conflict conflictClause)
  | NoConflict : return (propagateUnitsOnClauses NoncoreOnly)
```

Listing 5: Pseudo-code describing unit propagation with the Core First strategy

all literals are unassigned (and thus unfalsified), so any two literals can be picked.

Now, instead of checking a clause for every assignment to one of its literals, it suffices to have the BCP visit a clause only when one of its two watched literals becomes falsified by an assignment, ignoring assignments to any of the other literals. That this is enough to still catch all newly formed unit clauses or conflicts should be readily apparent when considering, that as long as these two watched literals both remain unfalsified, a clause simply can't become unit or form a conflict. In this sense, the watched literals could also be called *canary literals*, though the only recorded utterance of this phrase so far seems to stem from a mistranscribed Floridian newspaper article from 1977 [31], which just did not seem right.

To maintain their usefulness, a clause's watched literals are regularly updated to keep them unfalsified for as long as possible. When unit propagation visits a clause because one of its watched literals was falsified, it therefore proceeds as follows: It first checks to see whether the other watched literal is already satisfied, because that would mean it could move on to the next clause right away. If this is not the case, it then tries to find a new unfalsified literal (other than the second watched literal) to replace it with. If such a literal is found, it is made the watched literal in place of the now falsified old literal and unit propagation can also continue with the next clause. Only when the algorithm can't find a new falsified literal does it check whether the other watched literal is still unassigned or already falsified. If it was unassigned, this means that the clause is now unit and as a consequence the unassigned literal is assigned `1`; the unit is propagated. Should the other watched literal also be assigned `0`, then the clause is a conflict clause and unit propagation can end. Listing 4 contains pseudo-code describing the algorithm, similar in approach to DRAT-trim's implementation. It should be noted that DRAT-trim always keeps the two watched literals in the first two literal "slots". This is a common implementation variant,

```
let hasRATAgainstClause pivot clause =
  ignoredLiteral ← ¬pivot
  for l in (getLiterals clause) do
    if isAssignedTrue l then (* blocked clause *)
      if not (isTemporaryAssignment l) then
        addClauseAndDependenciesToCore (getAssignmentReason l)
      return true
    else if not (l = ignoredLiteral) && not (isAssignedFalse l) then
      addTemporaryAssignment (l, 0)
  match propagateUnits () with
  | Conflict conflictClause :
      addClauseAndDependenciesToCore conflictClause
      return true
  | Success : (* no conflict, lemma not redundant *)
      return false

let isRedundant lemma =
  (* try AT redundancy *)
  for l in (getLiterals lemma) do
    addTemporaryAssignment (l, 0)
  match propagateUnits () with
  | Conflict conflictClause : (* BCP led to a conflict, lemma has AT *)
      addClauseAndDependenciesToCore conflictClause
      return true
  | Success : (* no conflict, need to check for RAT *)
      pivot ← getPivotLiteral lemma
      trailbeforeRAT ← getTrailPosition ()
      for clause in getClausesContainingLiteral ¬pivot do
        if not (checkRATAgainstClause pivot clause) then
          return false
        else
          backtrackAssignmentsTo trailBeforeRAT
      return true
```

Listing 6: Pseudo-code describing the redundancy check

because it simplifies some logic, such as finding out the position of the other watched literal.

What's especially convenient about watched literals is that they do not need to be updated at all during backtracking. That's because both watched literals only need to fulfill the invariant of "as long as it could be an unfalsified literal, it is an unfalsified literal" to remain valid. That guarantees that a clause's state as being either unresolved, a unit- or a conflict clause can always be correctly identified just by checking the value of the two watched literals. With the BCP as shown here, these invariants are only broken between the time that a watched literal is falsified and the time that the BCP updates all the corresponding clauses. This means, that as long as one takes care to pick only "coherent" assignment states to backtrack to, i.e. those produced by a completed BCP, backtracking will not invalidate any watched literals. Even if the watches were moved to other literals during the assignment that is being rolled back, then these other literals must have been unfalsified at the time

the watches were moved, which means they necessarily also are unfalsified in the "earlier" assignment state. That's because during backtracking, literals can only become unfalsified, but never falsified.

In the following, the clauses (and lemmas) that are referenced through watched literals will be referred to as the *active* clauses, because they are the ones that can still influence the verification process.

Core-First BCP

Since every lemma that needs to be checked means a lot of additional work, the checker ideally wants to keep the number of lemmas in the core at a minimum. For this reason, it makes sense to prioritize core clauses during unit propagation. That's because it's preferable to find a conflict in one of the clauses already marked as core, as these clauses have played a role in a conflict before and should therefore introduce only a small number of new clauses (and especially lemmas) to the core. Core-first unit propagation does exactly that, by always prioritizing core clauses over non-core clauses during BCP. Listing 5 shows the important differences between normal unit propagation and the core-first variant, but does not restate the whole BCP algorithm from listing 4. It should especially be noted, that after a new unit clause is found during the non-core phase, propagation is interrupted and another core-only propagation is triggered, to also process the newly assigned literal core-first.

4.2.4. Redundancy Check

Verifying a lemma's redundancy is done in a two-stage process. Because checking a lemma for RAT redundancy is a lot more expensive computationally than a RUP check, DRAT-trim first attempts to verify each lemma as a RUP clause. This involves temporarily assigning `0` to all the lemma's literals and doing a unit propagation afterwards (see 2.5.2 for an explanation of why this works). If this propagation finds a conflict, then the lemma's validity as a RUP clause is already verified and verification can go on with the next lemma. If the propagation does not derive a conflict however, the lemma needs to be checked for RAT.

Luckily, the assignments that already resulted from the RUP check can simply be kept for the RAT check, because checking a clause for RAT means falsifying the literals of the clause's possible resolvents on the pivot literal (see 2.5.3). But all of these resolvents again include the lemma's original literals, except for the pivot literal, whose value does not matter for checking RAT (see the proof for proposition 4), which means the already falsified literals would have to be falsified again for each resolvent.

Listing 6 contains the pseudo-code that explains how a lemma's redundancy is checked in practice.

4.2.5. Backward Checking

Now it's time to take a look at the actual proof-verification logic. While one might expect that this simply consists of checking each lemma's redundancy in the order that is given in the proof, this is not the case. In order to reap the benefits of the core/non-core distinction, it is necessary to turn the intuitive approach to verifying lemmas on its head and start the verification process at the end of the proof instead. This is because a lemma's necessity for

the proof comes from some later lemmas depending on it for their validity, which means these later lemmas need to be verified first.

DRAT-trim thus starts out with a *forward pass*, adding all clauses to the set of active clauses (i.e., setting watches on their literals, as described) and doing a unit propagation to process any units already contained in the formula. Afterwards, each lemma is also added to the active clauses. This is done slightly differently compared to the clauses though: For each lemma, its literals are resorted so that the falsified literals are at the very end. This speeds up unit propagation later, as the unfalsified literals will be found faster if they are closer to the front. The lemma then also gets added to the watch lists of its first two literals (if it contains at least two). Now the code checks whether the lemma is unit, i.e., if it contains only one still unfalsified literal. If that is the case, then `1` is assigned to the literal and a unit propagation is triggered.

Lemmas are added in this fashion until these unit propagations lead to a conflict, at which point the conflict clause and its dependencies (of which the just added lemma is one) are marked as part of the core. After marking is done, verification starts with the lemma that caused the conflict and moves backwards from there.

At each proof step, the corresponding lemma is removed from the set of active clauses and, if it was unit, all assignments it was involved in are undone. If the lemma is not marked as core, nothing more is done and the algorithm moves on to the preceding lemma. If it is part of the core, the lemma is verified by checking its redundancy relative to the now smaller set of active clauses and all clauses that were necessary to prove its redundancy are themselves marked as part of the core.

For clause deletions, the procedure is the other way around: During the forward pass, the clauses in question are taken off the watch-lists so BCP does not process them anymore. During verification, each previously "deleted" clause is re-added to the formula and referenced again in the watch-lists of its first two literals. Listing 7 shows pseudo-code for the whole base program logic.

Compared to forward checking, this approach requires a more complicated logic, but has the benefit of potentially saving a lot of work in the case of proofs with many superfluous lemmas. Additionally, it gives the proof checker the ability to emit reduced formulas and proofs, only consisting of the core clauses, almost "for free".

```
type resultUnitPropagation = NoConflict | Conflict of clauseType
type resultVerification = Success | InvalidProof

let addClause clause =
  if literalCount clause = 1 then (* unit clause *)
    l ← (getFirstLiteral clause)
    if not (isAssignedTrue l) then
      assignTrue l
  else
    setWatchesOnFirstTwoLiterals clause

let addLemma lemma =
  bringUnfalsifiedLiteralsToFront lemma (* resort literals *)
  if literalCount clause >= 2 then
    setWatchesOnFirstTwoLiterals lemma
  if unfalsifiedLiteralCount clause = 1 then (* 'pseudo' unit clause *)
    l ← (getFirstLiteral clause)
    if not (isAssignedTrue l) then
      assignTrue l
      return (propagateUnits ())
  return NoConflict

let forwardPass clauses proofSteps =
  for clause in clauses do addClause clause
  propagateUnits ()
  for step in proofSteps do
    match step with
    | Lemma (lemma, pivotLiteral) :
        setTrailBeforeUnit lemma (getTrailPosition ())
        match (addLemma lemma) with
        | Conflict clause : (* proof resulted in conflict, start verification *)
            addClauseAndDependenciesToCore clause
            removeProofStepsAfter step
            return
        | NoConflict : continue
    | Deletion clause :
        removeWatchesFromFirstTwoLiterals clause

let verifyBackward proofSteps =
  for step in reversed(proofSteps) do (* backwards checking *)
    match step with
    | Lemma (lemma, pivotLiteral) :
        removeWatchesFromFirstTwoLiterals lemma
        backtrackAssignmentsTo (getTrailBeforeUnit lemma)
        if isPartOfCore lemma then
          trailBeforeCheck ← getTrailPosition ()
          if not (isRedundant lemma) then return InvalidProof
          backtrackAssignmentsTo trailBeforeCheck
    | Deletion clause :
        setWatchesOnFirstTwoLiterals clause
  return Success
```

Listing 7: Pseudo-code describing backward verification as implemented by DRAT-trim

5. New Implementation: funk-DRAT

As stated already in the title, it was the goal of this thesis to develop a new, functional implementation of a DRAT proof-checker. The reasoning behind writing it in a functional language stems from its intended use as a kind of "precursor" to a mechanically verified checker. This prototyping could, in theory, also have been done in a purely imperative language, by taking care not to rely too heavily on constructs that would later be hard to express in a formalization.

The choice of a functional language such as OCaml (even though it does support a somewhat imperative style) did definitely pay off however, simply because it discourages these hard-to-verify approaches from the outset. Furthermore, it leads to a rather fine-grained modularization of the code almost "automatically", thanks to the functional paradigm of partitioning algorithms into smaller functions. This made it quite easy to test possible optimizations, which played a big role in achieving a reasonable performance.

5.1. Program Architecture

The biggest challenge for a functional implementation of an inherently stateful algorithm operating on a huge number of variables, such as unit propagation, lay undoubtedly in the management of that state in appropriate data structures. OCaml is sadly not so forthcoming in this regard, because it stores anything but the most basic data types (e.g. integers and floating point numbers) in a *boxed* fashion [23]. That means, if one was to create a `clause` data type and store it in a conventional OCaml `Array`, that array would actually contain only a small stub that contains type information and points to a separate area of memory with the actual data. Because of this additional indirection and the overhead it introduces, storing values this way is not acceptable for algorithms which need to iterate over large segments of quite "simple" data (such as the clauses, which are basically a collection of integers) that could be stored more efficiently.

5.1.1. Data Structures

As a resort, OCaml offers a data structure that is better suited for this task and offers linear storage of integers, the `BigArray` structure. All the often-accessed entities in funk-DRAT (most importantly the clauses and their literals) therefore "live" on one-dimensional integer `BigArrays`, called `IntArrays` in the code. The entities are then referenced through their index on the `IntArray`, typed as `array_index` in the code, representing a kind of "pointer".

In an early revision of the program, it was attempted to use OCaml's support of object-oriented programming to build a convenient interface for the entities, by creating objects that encapsulate these "pointers" and offer member functions operating on the entities behind them. This proved not very performant however, since the hope of the compiler being smart enough to optimize away initializations of an essentially "empty" object did not

prove justified. The current version therefore simply uses normal functions to manipulate entities referenced by `array_index` arguments and groups them together in modules to mark them as belonging to an "entity class".

The modular approach was also taken to hide direct access to the `IntArrays`. Encapsulating the `IntArrays` in a managerial data structure has the additional benefit of allowing dynamic resizing when space runs low and storing objects of dynamic size, such as a clause containing a variable number of literals. Adding a new clause, after the literals have been parsed from the proof or CNF file, therefore looks as follows:

```
ClauseArray.add_obj storage_clauses ~var_size:literal_count
↪  ~constructor:(Clause.constructor new_literals)
```

`storage_clauses` is one of these management structures, containing the actual `IntArray` and additional information, such as the currently used-up array space (for resizing) and another `IntArray` with the start-indices of all the entities (for iterating over variably sized entities). The `Clause.constructor` function is responsible for initializing all data fields inside the `IntArray`, e.g. copying over the literals and storing their length.

While nice to work with, this level of abstraction eventually turned out to be a problem for performance, as each access to an element (e.g. a literal) needed to go through one additional level of dereferencing. As a remedy, the `IntArray` that "houses" the most often accessed entities, namely the clauses and literals, is extracted from `storage_clauses` after parsing is done and passed directly as an argument to every function that operates on clauses or literals after that point. This is rather unaesthetic from a programming perspective, but offers the best performance, since these memory accesses are very much the bottleneck compared to an implementation in a lower-level language. As an example, accessing the i-th literal of a clause would therefore look as follows in the code:

```
Clause.get_literal clause_buf clause i
```

Here, `clause_buf` refers to the mentioned bare `IntArray`, while `clause` is actually an `array_index` pointing to the beginning of the clause inside `clause_buf`. Since `get_literal` knows the start offset of the literals relative to the clause's `array_index`, it can retrieve the corresponding literal from the `IntArray`.

5.1.2. Literal Encoding

While DRAT-trim can use the capabilities of pointer-arithmetic to allow lookups with both positive and negative literals as indices into its per-literal arrays (see 4.1), such "magic" is not possible in OCaml. The literal encoding was therefore changed from the DIMACS-based style of using positive and negative integers for the literals to one that stores the literal's "sign" in the least significant bit. The "encoding" therefore looks as follows:

```
let Literal.encode lit = if lit < 0 then 2*(-lit)+1 else 2*lit
```

This translation is already done while parsing the input files, to make all later accesses to arrays indexed by literals straight-forward.

```
type resultUnitPropagation = NoConflict | Conflict of clauseType
type resultClauseAddition = Okay | AlreadyFalsified | ConflictFound of clauseType

let addClause clause =
  match Clause.findTwoUnfalsifiedLiteralsWithPositions clause with
  | (None, None) : return AlreadyFalsified (* all literals are false *)
  | (Some (i, l), None) :
      Clause.setIsConstUnit clause
      if not (isAssignedTrue l) then
        assignNewUnitWithReason l reason
        match propagateUnits () with
        | Conflict clause : return (ConflictFound clause)
        | NoConflict : return Okay
  | (Some (i_1, l_1), Some (i_2, l_2)) :
      if i_1 <> 0 then Clause.swapLiterals clause 0 i_1
      if i_2 <> 1 then Clause.swapLiterals clause 1 i_2
      setWatchesOnFirstTwoLiterals clause
  return Okay

let addLemma lemma =
  addClause lemma
```

Listing 8: Pseudo-code describing clause initialization as implemented by funk-DRAT

5.2. Algorithmic Improvements over DRAT-trim

While funk-DRAT mostly implements the algorithms as presented in chapter 4, achieving reasonable performance under the constraint of quite expensive memory accesses required thinking intensively about possible work-saving optimizations. In the following, the various changes to DRAT-trim's approach, of both the "successfully implemented with big performance gain"- and the "nice in theory, not so great in practice"-kind, will be presented.

5.2.1. Improvements to the Backward Checking Logic

When thinking about the way, clauses and especially lemmas are initially added to the formula, there is one important thing to realize: If any literals are already assigned a value when the clause is added to the formula, they will keep that value as long as the clause remains part of the formula. That's because they could only get become unassigned by removing an *earlier* clause from the formula, at which point they also would have already been removed.

DRAT-trim does seem to consider this in a way, since it optimizes lemmas when they are first added to the formula by moving the already falsified literals (which will never become unfalsified while the lemma is active) to the back of the formula. It does not follow this thought to its conclusion, however: All the clauses that were unit at the time of their addition will never change their state and do not need to be considered by BCP. Therefore they don't even need to be added to any watch-lists, which DRAT-trim still does.

funk-DRAT's approach therefore is as follows: It does not reorder literals (an experiment

done with this strategy proved slower, for reasons so far unknown), but rather tries to be "smarter" about adding watches and only does so, if the clause still has at least two unfalsified literals at the time of addition. If there is only one unfalsified literal on the other hand, then the clause is unit and the corresponding assignment is propagated. Such a clause is then marked as "constant-unit", which means it will never need to have watches set on its literals and can be ignored when it comes to clause deletions/re-additions. Listing 8 shows the relevant part of funk-DRAT's approach compared to DRAT-trim's implementation.

Not shown is the handling of the `AlreadyFalsified` return value in the forward pass, but it is easily explained: If it is a clause in the formula that is already falsified, then the formula is trivially UNSAT and proof checking is not necessary. If it is a lemma though, then the proof needs to be declared invalid, as a falsified clause containing any literals can never be implied by the formula. Only the empty clause can be implied, but the algorithm should never reach the point of trying to add the empty clause to the formula, because for a valid proof, unit propagation would derive a conflict as soon as the empty clause is implied.

5.2.2. Improvements to the Watched Literals Implementation

One optimization with a big impact on run-time was the change from DRAT-trim's watch-list implementation in the form of lists containing clause IDs to a doubly-linked list scheme directly connecting the clauses. For this purpose, the clause entities were outfitted with two `WatchListEntries` for their two watched literals, each with "forward" and "backward" references to other `WatchListEntries`. A literal's watch-list is then simply defined by a reference to the `WatchListEntry` of the first clause in the list.

This approach has the main advantage, that removing clauses from watch-lists (which happens whenever a watched literal is replaced because it became falsified) is sped up noticeably, since it doesn't involve scanning over the watch-list anymore but is instead a simple unlinking operation that can be done in constant time.

5.2.3. Improvements to BCP

Early Checking

DRAT-trim's unit propagation also offers opportunity for optimization. When its BCP visits a clause because a watched literal l_{false} was falsified, it does not actually check whether the other watched literal l_{other} is also falsified, so long as it finds another unfalsified literal to replace l_{false} with.

But this "lazy checking" approach can have some quite negative consequences, as can easily be shown. Assume for example that l_{other} had also been falsified since a propagation was last carried out but was still ahead of l_{false} in the trail, and the clause in question had only one more literal, l_3, which was unassigned. In that case, BCP would first visit the clause because of l_{false}, but would find the unfalsified literal l_3 to replace it. It would therefore move the clause from the watch-list of l_{false} to the watch-list of l_3, then carry on with other clauses. But while moving through the trail, it would eventually encounter the falsified l_{other} and realize, that the clause was actually unit and l_3 needs to be assigned `1`.

This may be a worst case scenario, but it can be easily averted by making BCP check for the other watched literal's value more eagerly, even when it finds an unfalsified literal to replace l_{false} with. If the other watched literal also turns out to be `0`, then the algorithm

can attempt to replace both watched literals by unfalsified ones at once. This saves BCP from a second visit to the same clause and speeds up the search for a replacement for the other watched literal slightly, because the search for the second unfalsified literal can start at the point of the first found unfalsified literal, instead of re-scanning from the front.

Note that this change has no bearing on the validity of the watched literals in the face of backtracking. The reasoning from 4.2.3 still applies without any adjustment, as early checking only "compresses" two visits by the BCP to the same clause into one, but does not change the net outcome of a whole unit propagation run.

Separate Core/Non-Core Watch-Lists

DRAT-trim's core-first unit propagation iterates over all the clauses in a watch-list, but only updates the ones it is targeting at the given time (either core or non-core). This process can be made a lot more efficient by simply storing two watch-lists per literal, for the core and non-core clauses respectively. So long as one takes care to move clauses from the non-core to the core watch-lists upon their becoming part of the core, this is a fairly simple optimization with a very noticeable improvement in performance.

5.2.4. Experiments with Clause Prioritizing

There were also some attempts at developing a more fine-grained clause-prioritizing strategy for BCP, compared to only processing the core before the non-core clause set.

The first idea was based on the realization, that it should be advantageous to prioritize clauses from the formula over proof lemmas, since a conflict in a clause always introduces at least one lemma fewer to the core than a conflict in a lemma, which should result in less work for the checker. To test this theory, literals were given three watch-lists, for the core clauses/lemmas, the non-core clauses and the non-core lemmas respectively. Then, the BCP was updated to always check the core set before the non-core set and the non-core clauses before the non-core lemmas. The results, however, were not so convincing, with a marked loss in performance. It was difficult to pinpoint a reason for this, but since unit propagation is executed so often, it is quite possible that the more complex and thus slower logic required to assure clauses of higher "importance" are always checked first simply outweighs the benefits of having to check fewer lemmas.

The second idea was in a similar vein, but only realizable with a more profound change in architecture. The insight in this case was, that an ideal unit propagation would derive a conflict as fast as possible, skipping any unnecessary assignments that would be undone after the conflict anyway. With watched literals, it is indeed possible to build a rough heuristic of clauses that are more likely to result in a conflict and should therefore be prioritized. If you look at BCP as a function taking a list of newly falsified literals and returning a conflict clause, then the eventually found conflict clause will always have had both its watched literals contained within this list of newly falsified literals. In other words, if one were to think in terms of BCP processing "clause update lists" instead, which contain all the clauses from the watch-lists of the newly falsified literals, then a clause could only be a conflict clause if it occurs in this list twice.

To test this idea, BCP was indeed adjusted to work with such clause update lists instead of trail segments. For this purpose, the `assign_false` function taking care of a new assignment

Checker	Total Time	∅ by Time	∅ by Prob.	Minimum	Maximum
DRAT-trim	1306 s	100%	100%	100%	100%
funk-DRAT	2636 s	50.0%	50.0%	25.5%	101.5%
funk-DRAT, no const-units	2651 s	49.7%	49.8%	24.8%	100.6%
funk-DRAT, no early check	2717 s	48.5%	48.4%	25.5%	102.3%
funk-DRAT, no separate WLs	4549 s	29.0%	35.0%	15.3%	84.1%

Figure 5.1.: Benchmark results comparing different proof checkers, performance is measured relative to DRAT-trim's checking speed

was given the task of directly inserting all the clauses from the falsified literal's watch-lists into two core/non-core clause update lists, which were also implemented as doubly-linked lists. With this architecture, it was easy to check whether a given clause was already contained in an update list and move it to the front of the list, instead of inserting it a second time. This guaranteed that all potential conflict clauses were always checked before the clauses that could only lead to new units.

After implementing this approach, it sadly turned out to be substantially slower than the normal watch-list variant, mostly because moving through the watch-lists for every literal assignment and adding the clauses inside them to the update lists proved to cause too much overhead given the already expensive memory accesses. This idea will be investigated further however, as it seems to hold some potential.

5.3. Performance Results

To compare the performance of funk-DRAT with that of DRAT-trim, a benchmarking suite with proofs for unsatisfiable problems from the "SAT Competition 2016" was assembled. The problems were chosen according to two criteria: They had to be proven UNSAT by the SAT solver CryptoMiniSat in less than 20,000 seconds, and DRAT-trim's checking time for the resulting DRAT proof had to be between 0.1 and 300 seconds. That left a suite of 40 proofs, 8 of which contained RAT lemmas. Benchmarking was done on a machine with an Intel Core i7-3820QM processor and 8 GBs of RAM, running Lubuntu 17.04. funk-DRAT was compiled with the `4.04.1+flambda` version of the OCaml compiler using the additional `-O3 -unboxed-types` flags, which instruct the compiler to do aggressive optimization (carried out by the "flambda" suite of optimizers) and treat record types containing only a single field, such as the `array_index` type, as an unboxed value. With this setup, funk-DRAT's checking times ranged from 406 s to 0.87 s, with an average of about 66 s per proof.

Each proof was checked three times by each checker, and the minimum times were compared against each other. Figure 5.1 shows the results, with performance measured relative to DRAT-trim's checking speed. As can be seen, funk-DRAT with all optimizations activated reaches, on average, about half of DRAT-trim's performance for checking the selected proofs, which means it generally only takes about twice as long to do so. Considering that funk-DRAT is written in a language that is still far from allowing the same level of low-level optimization as C, this seems like a fairly decent result.

To show the benefits of the described optimizations, a few variants of funk-DRAT with

respectively one optimization not implemented are also benchmarked. While this makes it apparent that all optimizations do have a (sometimes only quite small) positive impact on performance, the watch-list separation constitutes the biggest improvement by far. This should not be too surprising, especially given the fact that the core-first strategy always triggers another core-only propagation for each new assignment. The only optimization lacking a distinct comparison here is the doubly-linked watch-list implementation. That's because "undoing" it for the purpose of benchmarking simply proved to be too cumbersome, as it is by now an integral part of the checker.

For further details, table A.1 in the appendix shows a more thorough overview of the respectively "best" times each checker achieved for each problem. Note that a single proof in the suite, namely `test_v3_r8_vr5_c1_s8257.smt2-stp212`, was responsible for all of funk-DRAT's performance results over 100%. It could not easily be determined how this came about, but it might be advisable to investigate this further, seeing as it seems to represent either a worst-case scenario for DRAT-trim, or a kind of "sweet spot" for funk-DRAT's implementation, or a combination of both.

6. Conclusion

It was the stated goal of this thesis to implement an optimized checker for DRAT proofs in a functional programming language. The main motivation behind it was to reexamine the methods implemented by previous, imperatively written proof checkers (most importantly DRAT-trim) from a functional perspective and bring the problem of proof checking in a form, that will be easier to formalize later on for the purpose of developing a verified DRAT proof checker.

funk-DRAT, the proof checker that resulted from this effort, indeed turned out to be quite well-performing, reaching about half the performance of DRAT-trim thanks to a number of newly implemented optimizations. It still remained true to the functional mindset of attempting to make functionality as clear as possible, which means it should also provide a useful base for further development and experimentation. Considering, these results, it could be said that it fulfilled its intended goal quite successfully.

Furthermore, work on this thesis proved to be a great opportunity both for learning OCaml and for gaining some understanding of the fascinating, but complex world of SAT solving. This in itself can also be considered a success, as it seems fairly safe to assume that these new insights won't have been put to use for the last time after this thesis is submitted.

A. Detailed Benchmark Results

Problem	**DRAT-trim**	**funk-DRAT**	..., without early check	..., without separate WLs	..., without const-units
snw_16_8_preOpt_pre	262.40	405.72	429.60	997.96	418.38
stone-width3chain-nmarkers-08_shuffled	136.73	233.90	237.30	535.92	236.40
barman-pfile07-028.sas.ex.15	127.15	355.10	365.58	486.19	352.18
arcfour_initialPermutation_6_14	96.93	183.74	183.58	294.59	184.23
snw_13_8_CCSpreOptnopre	90.94	135.22	138.46	283.24	135.01
snw_13_8_CCSpreOptpre	68.62	105.12	108.35	219.27	104.84
snw_13_8_CCSpreOptEncpre	66.11	100.90	103.66	210.50	101.64
barman-pfile06-022.sas.ex.7	44.63	126.30	127.58	126.44	126.42
barman-pfile08-032.sas.ex.15	40.24	124.60	127.07	128.99	125.34
modgen-n200-m90860q08c40-14424	39.74	62.17	69.36	195.49	62.22
barman-pfile10-037.sas.ex.7	30.94	91.59	93.03	92.30	91.65
barman-pfile08-032.sas.ex.7	29.24	84.11	85.33	84.08	84.04
barman-pfile07-027.sas.cr.37	22.03	52.14	55.04	90.03	52.53
test_v3_r8_vr5_c1_s8257.smt2-stp212	21.24	20.93	20.77	25.26	21.11
barman-pfile09-036.sas.cr.33	20.20	57.90	59.71	82.59	58.68
barman-pfile10-039.sas.ex.15	19.06	54.16	55.07	62.62	53.41
tseitingrid4x185_shuffled	19.06	26.90	28.79	39.10	27.18
tseitingrid4x195_shuffled	18.79	27.24	28.87	40.12	27.35
tseitingrid4x190_shuffled	17.95	25.84	27.60	39.24	26.43
barman-pfile10-038.sas.ex.15	17.83	51.14	51.76	57.66	50.87
barman-pfile10-040.sas.ex.15	17.69	52.17	51.72	59.08	50.35
tseitingrid4x200_shuffled	17.18	24.72	26.38	35.68	25.06
uum8.smt2-stp212	16.50	25.23	30.04	107.76	24.80
add_01_1000_4.smt2-cvc4	16.28	24.66	26.47	42.50	24.51
sokoban-p01.sas.ex.17	9.72	28.37	27.94	32.35	28.58
barman-pfile10-038.sas.cr.25	9.45	33.62	33.96	40.07	34.00
barman-pfile08-030.sas.cr.27	7.82	27.02	27.65	33.36	27.45
sokoban-p04.sas.ex.13	7.08	22.17	22.42	25.16	22.18
barman-pfile10-040.sas.cr.21	5.82	22.86	22.82	24.26	23.44
sokoban-p17.sas.ex.11	4.59	12.47	12.60	12.45	12.52
barman-pfile10-040.sas.cr.17	4.51	17.34	17.14	17.85	17.45
safe027_pso.opt_true-unreach-call.i-cbmc-u2	2.43	4.73	4.82	6.60	4.75
modgen-n200-m90860q08c40-28046	1.61	2.14	2.23	3.83	2.11
sokoban-p10.sas.cr.35	1.60	2.69	2.74	3.97	2.73
sokoban-p09.sas.cr.25	1.14	2.85	2.84	2.86	2.92
safe028_tso.oepc_true-unreach-call.i-cbmc-u2	1.11	2.36	2.41	2.68	2.39
square.2.0.i.smt2-cvc4	1.07	2.02	2.05	2.81	2.01
safe009_pso.oepc_true-unreach-call.i-cbmc-u2	0.92	1.99	1.98	2.10	1.98
sokoban-p18.sas.cr.29	0.73	1.26	1.25	1.47	1.27
sokoban-p19.sas.cr.23	0.47	0.87	0.87	1.00	0.88

Table A.1.: Table showing each checker's respective lowest checking times (in seconds) for all 40 problems in the benchmark suite

References

[1] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22: 319–351, 2004.

[2] Anton Belov, Daniel Diepold, Marijn Heule, and Matti Järvisalo. SAT competition 2014. URL `satcompetition.org/2014/certunsat.shtml`. Accessed on 2017-04-22.

[3] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[4] Armin Biere, Marijn Heule, Hans Van Maaren, and Toby Walsh. *Handbook of Satisfiability: Preface*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, Netherlands, 1 edition, 2009. ISBN 9781586039295.

[5] 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.

[6] Jasmin Christian Blanchette, Lars Hupel, Tobias Nipkow, Lars Noschinski, and Dmitriy Traytel. Experience report: The next 1100 haskell programmers. *SIGPLAN Not.*, 49 (12):25–30, 2014. ISSN 0362-1340. doi: 10.1145/2775050.2633359.

[7] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 1–5. ACM, 2009.

[8] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 44–57. Springer, 2010.

[9] Pierre Castéran and Yves Bertot. *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004. Traduction en chinois parue en 2010. Tsinghua University Press. ISBN 9787302208136.

[10] 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.

[11] Luís Cruz-Filipe, Marijn Heule, Warren Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. *arXiv preprint arXiv:1612.02353*, 2016.

[12] Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. *arXiv preprint arXiv:1610.06984*, 2016.

[13] Ashish Darbari, Bernd Fischer, and João Marques-Silva. *Industrial-Strength Certified SAT Solving through Verified SAT Proof Checking*, pages 260–274. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-14808-8. doi: 10.1007/978-3-642-14808-8_18.

[14] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

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

[16] John Franco and Sean Weaver. Algorithms for the satisfiability problem. In *Handbook of Combinatorial Optimization*, pages 311–454. Springer, 2013.

[17] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, pages 10886–10891. IEEE Computer Society, 2003.

[18] Carla P Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. *Foundations of Artificial Intelligence*, 3:89–134, 2008.

[19] Marijn JH Heule and Armin Biere. Proofs for satisfiability problems. *All about Proofs, Proofs for all*, 2015.

[20] Marijn JH Heule, Warren A Hunt, and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 181–188. IEEE, 2013.

[21] Marijn JH Heule, Warren A Hunt Jr, and Nathan Wetzler. Verifying refutations with extended resolution. In *International Conference on Automated Deduction*, pages 345–359. Springer, 2013.

[22] Marijn JH Heule, Warren A Hunt, and Nathan Wetzler. Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. *Software Testing, Verification and Reliability*, 24(8):593–607, 2014.

[23] Jason Hickey, Anil Madhavapeddy, and Yaron Minsky. Chapter 20. memory representation of values / real world OCaml, 2012. URL `https://realworldocaml.org/v1/en/html/memory-representation-of-values.html`. Accessed on 2017-05-11.

[24] Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. *Automated Reasoning*, pages 355–370, 2012.

[25] Toni Jussila, Carsten Sinz, and Armin Biere. *Extended Resolution Proofs for Symbolic SAT Solving with Quantification*, pages 54–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-37207-3. doi: 10.1007/11814948_8.

[26] Boris Konev and Alexei Lisitsa. *A SAT Attack on the Erdős Discrepancy Conjecture*, pages 219–226. Springer International Publishing, Cham, 2014. ISBN 978-3-319-09284-3. doi: 10.1007/978-3-319-09284-3_17.

[27] Norbert Manthey. CL-tools. URL `http://tools.computational-logic.org/content/riss.php`. Accessed on 2017-04-22.

[28] Norbert Manthey, Marijn J. H. Heule, and Armin Biere. *Automated Reencoding of Boolean Formulas*, pages 102–117. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-39611-3. doi: 10.1007/978-3-642-39611-3_14. URL `http://dx.doi.org/10.1007/978-3-642-39611-3_14`.

[29] Filip Marić. Formalization and implementation of modern SAT solvers. *Journal of Automated Reasoning*, 43(1):81–119, 2009.

[30] 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.

[31] The Palm Beach Post. Edition from april 10th, 1977. URL `https://www.newspapers.com/newspage/133913803/`. Accessed on 2017-04-28.

[32] JP Marques Silva and Karem A Sakallah. Conflict analysis in search algorithms for satisfiability. In *Tools with Artificial Intelligence, 1996., Proceedings Eighth IEEE International Conference on*, pages 467–469. IEEE, 1996.

[33] Laurent Simon. Glucose's home page. URL `http://www.labri.fr/perso/lsimon/glucose/`. Accessed on 2017-04-22.

[34] Mate Soos. Github - msoos/cryptominisat: An advanced SAT solver. URL `https://github.com/msoos/cryptominisat`. Accessed on 2017-04-22.

[35] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 244–257. Springer, 2009.

[36] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983. ISBN 978-3-642-81955-1. doi: 10.1007/978-3-642-81955-1_28. URL `http://dx.doi.org/10.1007/978-3-642-81955-1_28`.

[37] Allen Van Gelder. RES file format, 2005. URL `ftp://ftp.cse.ucsc.edu/pub/avg/ProofChecker-fileformat.txt`. Accessed on 2017-05-12.

[38] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *ISAIM*. Citeseer, 2008.

[39] Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic*, 7(1):26 – 40, 2009. ISSN 1570-8683. doi: https://doi.org/10.1016/j.jal.2007.07.003. Special Issue: Empirically Successful Computerized Reasoning.

[40] Nathan Wetzler, Marijn JH Heule, and Warren A Hunt Jr. Mechanical verification of SAT refutations with extended resolution. In *International Conference on Interactive Theorem Proving*, pages 229–244. Springer, 2013.

[41] Nathan Wetzler, Marijn JH Heule, and Warren A Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 422–429. Springer, 2014.

[42] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10880–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1870-2.