CreuSAT

Using Rust and CREUSOT *to create the world's fastest deductively verified SAT solver*

Sarek Høverstad Skotåm

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2022

CreuSAT

Using Rust and CREUSOT *to create the world's fastest deductively verified SAT solver*

Sarek Høverstad Skotåm

© 2022 Sarek Høverstad Skotåm

CreuSAT

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

This thesis describes CreuSAT, a formally verified SAT solver written in Rust. In addition to implementing the core conflict-driven clause learning (CDCL) algorithm, we also implement a series of crucial optimizations. The most important of these is the two watched literals scheme with blocking literals and circular search, the variable move-to-front (VMTF) decision heuristic, clause deletion, phase saving, and moving average based restarts. The resulting solver is the first deductively verified solver which is able to consistently solve problems from the SAT competition. This is done while maintaining a relatively small code base, amounting to around 4 thousand lines of proof code and program code combined, with a low proof overhead of around three lines of proof code per line of program code.

In addition to presenting CreuSAT, we also present two other verified SAT solvers. The first of these is called Friday, and represents what we consider to be the minimal verified SAT solver. The second of these is called Robinson, and is a SAT solver based on the Davis–Putnam–Logemann–Loveland (DPLL) algorithm. We present Friday and Robinson mostly for pedagogical reasons, though it should be noted that Robinson is, to the best of our knowledge, the fastest verified DPLL SAT solver presented in literature.

To prove the correctness of our solvers, we use the deductive verification tool CREUSOT, which is based on the WHY3 software verification platform. CREUSOT leverages the guarantees given by the Rust type system to efficiently translate the program code to a functional program. This results in verification conditions which require little CPU time to prove, and we demonstrate improvements in verification time when compared to other verified solvers.

Our solvers represent the first major usage of CREUSOT, and the largest published verification effort of Rust code to date. They further mark the first time the direct verification of an imperative program has been able to produce a verified SAT solver capable of solving problems of recent SAT competitions.

Acknowledgements

This thesis has its origin in the fortunate event that my initial thesis topic had been solved by someone else. In part due to chance, and in part due to interest, I ended up reaching out to Xavier Denis, after watching his talk at RustVerify 2021. If I had known the degree to which formal verification is a masochistic endeavor back then, I am not sure I would have sent the initial mail, though I am glad I did. It has been a great deal of fun, and I would like to express my gratitude to Xavier Denis, both for suggesting proving a SAT solver in the first place, and for all the hours spent either on fixing CREUSOT, listening to me talk about SAT solving, reading drafts of this thesis, or on teaching me about various topics.

I would also like to thank my supervisor, Martin Steffen, for having confidence in me and my ideas, as well as for his feedback on my writing.

Before commencing the work on the thesis, I decided to terminate my tenancy and go abroad. I would in that regard like to thank those I've met on my travels, and especially those who have shown me hospitality. I would also like to thank Thomas Hylland Eriksen, Kari Spjeldnæs, and of course Doffen, for my stay in their house. A great share of the main ideas came to be during those months.

Furthermore, to all my friends and foes at the Department of Informatics: thanks. I appreciate all the people who do their tiny bit, be it by being TAs, by being active in one of the student organizations, or just by being curious and engaged in some subject or some topic. In that regard, the cream of the crop: the people in and around the MAPS student organization, and of course, the youngest "old" man I know: Per Magne Kirkhus.

Finally, I would like to thank you, for reading my thesis. I hope you find some of the same joy in reading it as I have had in making it.

Sarek

Contents

1 Introduction **1**

I Background and the verification of a minimal solver **7**

2 Background **9**
2.1 Rust . . . 9
2.2 SAT and SAT solvers . . . 17
2.2.1 SAT . . . 17
2.2.2 CNF . . . 18
2.2.3 Algorithms for solving SAT . . . 18
2.3 Proof of code and CREUSOT . . . 19

3 Verification of a minimal solver **29**
3.1 The algorithm . . . 29
3.2 The proof idea . . . 30
3.3 Implementation of Friday . . . 30
3.4 Proof of Friday . . . 32

II Verification of a DPLL solver **39**

4 Verification of the DPLL algorithm **41**
4.1 The DPLL algorithm . . . 41
4.2 The main ideas of the proof . . . 42
4.3 Proof of Robinson . . . 43

III Verification of a CDCL solver **53**

5 Verification of the CDCL algorithm **55**
5.1 The CDCL algorithm . . . 56
5.1.1 Overview . . . 56
5.1.2 Interlude: resolution and the Davis Putnam procedure 58
5.1.3 Introduction to the trail . . . 59
5.1.4 The conflict analysis algorithm . . . 60
5.2 The main ideas of the proof . . . 62

5.2.1 The suboptimality of the cut of the implication graph 62
5.2.2 CDCL as an extension of DP 62
5.3 Proof of CreuSAT 64
5.3.1 Furthering our understanding of the trail 64
5.3.2 Facilitating clause learning 67
5.3.3 Proving the clause learning 69
5.3.4 Backtracking the trail 71
5.4 Optimizations 71
5.4.1 Two watched literals 71
5.4.2 Variable move-to-front 75
5.4.3 Phase saving 77
5.4.4 Clause database simplification 78
5.4.5 Clause deletion 78
5.4.6 Search restart 79
5.5 The top level contracts 80

IV Evaluation and the road ahead **85**

6 Evaluation **87**
6.1 Setup 87
6.2 Evaluation of Robinson 88
6.3 Evaluation of CreuSAT 90
6.4 Discussion of results 93

7 Conclusion **97**
7.1 Summary of Contributions 97
7.2 Discussion of CREUSOT 97
7.3 Related work and conclusion 98
7.4 Future work 100
7.4.1 Extending Robinson 100
7.4.2 Improving CreuSAT 101

List of Figures

2.1 Illustration of the process from Rust code to proven program . . . 20
2.2 The `incr` function loaded in the WHY3 IDE 21
2.3 The `incr` function in the WHY3 IDE after running a split . . 21
2.4 The `incr` function in the WHY3 IDE with proven safety . . . 22
2.5 The `incr` function with full proven correctness in the WHY3 IDE 23

List of Tables

6.1 Results of running the solvers on random 3SAT benchmarks 89
6.2 Results of running the solvers on pigeonhole problems . . . 89
6.3 Results of running the solvers on the SAT Race 2015 problems 92
6.4 Results of running the solvers on the SAT Race 2015 problems with the manthey_Dimacs* problems removed . . 92

Chapter 1

Introduction

When writing a computer program, it is desired that the program is *safe* — that it does not crash or contain vulnerabilities, and that it is *correct* — that it does what is intended. To achieve this goal, various techniques are employed. Examples of such techniques include code reviews, testing, systematic fuzzing, or model checking. None of these can guarantee the correctness of the software, and thus, our software still has bugs. Another option is to utilize formal verification techniques to *prove* the correctness of the program with regards to some specification. Such techniques can range from the type checking process of various programming languages, which statically proves that the program is *type correct*, to more elaborate processes, such as *deductive program verification*, which can statically prove *functional correctness* of the program.

An example of a type system which can statically ensure a high level of guarantees, is the type system of the Rust programming language. It goes further than for instance Java and C, and can guarantee properties such as *memory safety* and *thread safety*. That being said, while the Rust programming language can guarantee a high degree of safety, it can not provide all possible safety guarantees, nor guarantees of correctness. To ensure this, we have to use an auxiliary tool. One such tool is the CREUSOT [17] deductive verification tool, which grants the user the ability to prove further safety properties, such as that all array accesses are within bounds, and full correctness guarantees, such that a sorting function correctly sorts the given vector.

These guarantees do, however, not come for free, and achieving guarantees of correctness requires the programmer to annotate the code with *contracts* which specify the desired program behaviour. To do this, the programmer uses the PEARLITE *Behavioral Interface Specification Language* [23], and uses satisfiability modulo theories (SMT) solvers to discharge the generated verification conditions (VCs). Whereas previous efforts to verify code in imperative programming languages have had to rely on specialized logics such as *separation logic* [47] or *dynamic frames* [53], CREUSOT is able to verify Rust code by using the simpler logic of WHY3 [19]. To achieve this,

CREUSOT leverages the fact that the *safe* subset of Rust can be efficiently translated to a functional programming language, by utilizing the *prophetic encoding* of mutable pointers of Matsushita et al. [38].

This approach has thus far shown promising results [39], but the viability of CREUSOT for a larger project has yet to be determined. To remedy this, we use CREUSOT to verify programs which solve the Boolean satisfiability problem, also known as SAT solvers. A SAT solver takes as input a Boolean formula in conjunctive normal form (CNF), and determines whether there exists an assignment which satisfies the given formula. The code base of a modern SAT solver can be in the range of tens of thousands of lines of code, and are highly optimized programs where proving safety and correctness is non-trivial. Furthermore, as these solvers are used for problems such as verifying the design of integrated circuits, or to serve as a back-end for a *model checker*, it is also critical that they produce the correct result.

Ensuring correctness while maintaining a solver of high performance is far from easy, and solvers are regularly found to produce the wrong answer[1]. As verifying the solvers themselves has in general been considered to be too difficult, the solvers are instead required to provide a proof that their result is correct. In the case the formula is satisfiable, the solvers are required to produce a satisfying model, and in the case the formula is unsatisfiable, they have to produce a *proof of unsatisfiability*. SAT is in the class NP, and UNSAT is thus in the class coNP. This means that generating and checking the certificates is not a trivial task, often taking longer than solving the problem itself [24].

The non-triviality of proof checking, combined with the fact that the checkers may also be faulty, means that having a solver which is guaranteed to produce the correct answer is of interest. Indeed, some attempts at making verified solvers have been conducted, either in a verification enabled language, or by using a proof assistant and its code generation feature to create a solver which is *correct by construction*. Examples of the latter include the IsaSAT [21] solver, which is written in the Isabelle Proof Assistant [45], Lescuyer and Conchon's solver in Coq [32], or the `versat` [48] solver, which is written in the GURU programming language.

Of the verified solvers, IsaSAT is the only one which has been able to solve a significant amount of problems from the SAT competition, with performance not too far from the well-known MiniSat solver of Eén and Sörensson [18][2]. This is an impressive feat, taking many years of collaborative effort to achieve. The resulting solver requires over 150

[1] See for example Sat Competition 2020, where 4 solvers competing in the parallel track were disqualified for producing the wrong result, or Sat Competition 2018, where 3 solvers competing in the parallel track were disqualified, among them the two best solvers of 2018. As a matter of fact, working on this thesis exposed that the solver MicroSat produces the wrong result on at least 100 instances from the Sat Race 2015. See this GitHub Issue for details.

[2] See Section 6 - Evaluation for benchmarks of IsaSAT and MiniSat.

thousand lines of proof script, and is considered hard to extend, even for experts [20].

This seems to be contrary to solvers which are written in an imperative language. Take for instance the unverified solver MiniSat, which is the ancestor of many of the solvers which have won medals at the Sat Competition. There has also been MiniSat hack tracks at the Sat Competition, which later were replaced by the Glucose [3] hack track. Glucose is itself a MiniSat hack, and has won numerous medals at the Sat Competition. Judging by the amount of solvers which are extensions of MiniSat, and the success of for instance Glucose, we believe that MiniSat achieved its goal of becoming an extensible SAT solver. Though verifying such a solver may reduce its readability and extensibility, it seems likely that one will end up with a solver which is more readable and more extensible than those based on proof assistants and code generation.

That being said, the creation of a verified solver by targeting an imperative language directly has yet to yield a solver which performs as well as even the lowest performing SAT solvers based on using a proof assistant and proof generation. Whether this is due to some inherent limitation of this approach will be seen during the course of this thesis. As such a solver has yet to exist, eventual advantages, such as being extensible, or having a small code base, also remain to be seen.

We thus present the thesis statement, which we will explore during the course of this document:

- To investigate whether it is possible to create a formally verified SAT solver with comparative performance to those based on proof assistants, while targeting an imperative implementation directly.

This thesis

This thesis introduces and describes CreuSAT, a formally verified SAT solver which is written in Rust, and verified with CREUSOT. CreuSAT is verified to be correct: if the solver returns SAT, the solver returns a satisfying assignment, and if the solver returns UNSAT, there is no assignment which satisfies the formula. CreuSAT is also verified to be safe: there is no possibility of a panic during runtime. The source code for CreuSAT can be found at github.com/sarsko/CreuSAT, and, though this thesis is a self-contained document, it is recommended to read the thesis in conjunction with the source code.

CreuSAT is a conflict-driven clause learning (CDCL) SAT solver. In addition to implementing and proving clause analysis and clause learning, we also implement and prove a series of optimizations. The most important of these is the two watched literal (2WL) scheme [42] with blocking literals and cirular search, and the variable move-to-front (VMTF) decision heuristic [10]. We also implement clause deletion, phase saving, backtracking to asserting level, and exponential moving averages based

restarts.

This thesis also describes Friday and Robinson, two other formally verified SAT solvers, both of which are also written in Rust, and verified with CREUSOT. These are proven to be safe, sound, and complete. Friday is a naive, functional solver of less than 200 lines, mostly acting as a stepping stone up to Robinson. Robinson is, to the best of our knowledge, the fastest Davis-Putnam-Logemann–Loveland (DPLL)-based formally verified SAT solver, both in terms of execution speed, and in terms of how long it takes to prove its correctness.

The reason for presenting Robinson and Friday in addition to CreuSAT is threefold. We do it in part for pedagogical reasons: we had to implement Robinson and Friday to be able to implement CreuSAT, and it is likely others will require the same. Second: we believe Robinson is a substantial contribution both to the area of verified SAT solvers, and to the area of formally verified Rust code, and would like for its ideas to not get lost. Robinson was, to the best of our knowledge, the largest deductively verified piece of Rust code at its creation, and is today only beaten by CreuSAT. The last reason is that we would like to aid in the learning of CREUSOT, which we believe the simpler solvers are much better suited for than CreuSAT.

The full benchmarks of CreuSAT, Robinson, and a selection of other solvers, some of which are formally verified, some of which are not, are available in Chapter 6 – Evaluation. We demonstrate that CreuSAT is able to solve a substantial amount of problems from recent SAT competitions, being the first deductively verified solver to ever do so. That being said, there is still some work to be done to become competitive with the state-of-the-art. Considering that the sheer amount of years of work that has went into the best solvers, this is to be expected.

As mentioned, CreuSAT is, to the best of our knowledge, the largest piece of Rust code which is verified to be correct. It is the first major use case of CREUSOT, and we argue that it is a strong first showing. In Section 6 – Evaluation, we compare and discuss the verification effort required to achieve various results. We demonstrate that CREUSOT is able to offer significant improvements in verification time when compared to other verification efforts.

Thesis structure

The thesis is divided into 4 parts. In the first part, we will cover the necessary background material. First, we will briefly cover the Rust programming language, then we will introduce the art of SAT solving, and finally we will look at CREUSOT. We will end this first part by verifying the solver which we call Friday, to reinforce the ideas of the background chapter, and to ease the transition to more complex solvers.

In the second part, we will present Robinson, the DPLL-based solver. We

will cover the main ideas of the proof, with a focus on the proof of unit propagation. This part also exists to explain lemma functions, and the act of building up a proof context. In the third part, we will present CreuSAT, the CDCL-solver. For this part, we will assume familiarity with the previous parts, and will spend most of our effort on the core ideas of the proof, and the concrete implementation. In the fourth and last part, we will do a thorough benchmark of Robinson and CreuSAT, a collection of other verified solvers, as well as a selection of unverified solvers. Afterwards, we will discuss the results, CREUSOT, and look at the future of CreuSAT.

Part I

Background and the verification of a minimal solver

Chapter 2

Background

In this chapter we will present some of the background knowledge which is needed to understand the rest of the thesis. We have structured the thesis such that most concepts are introduced just before they are needed. That being said, there are some preliminaries which should be covered, either because they do not fit anywhere else, or because they serve as a foundation which we will later build upon.

We do not believe that Rust, SAT solving, nor CREUSOT can be considered common knowledge, and do thus not assume much in terms of prior knowledge of these concepts. Though we do explain borrows, being familiar with imperative programming and pointers/references, for instance by having previous experience with programming in C or C++, will be advantageous. Explaining these concepts in any thorough manner would simply be too big a task. That being said, much of the thesis and the associated code should be perfectly understandable in the case that the reader does not know these concepts.

If the reader is familiar with Rust, the section about Rust can safely be skipped. We will not be using any of the features of Rust which could be considered more advanced. This is in part due to them not being supported by CREUSOT, and in part due to the desire of reducing the complexity of the proof work.

2.1 Rust

In this section, we will look at the Rust programming language. We do not have space to teach Rust fully, so the intention of this section is to give a brief overview of the language, and to give an introduction to the constructs which are used throughout the thesis and the associated code base.

We start this section by giving a summary of the Rust programming language, after which we cover the syntax of the language. Rust does not differ significantly from other C-like languages in its syntax, so if

you already know a C-like language, then most concepts should be understandable from their context. To make this understanding easier, and in the case that the reader is not too familiar with a C-like language, we provide a brief introduction to the concepts which we use to implement our solvers.

Summary

Rust is a systems programming language developed by Mozilla which was released in 2015. It has a strong focus on safety, guaranteeing memory-safety and thread-safety through its rich type system. It does not have a garbage collector, and solves the management of memory through its type system and concept of ownership. The language is multi-paradigm and performant, offering features often found in functional languages as zero-cost abstractions.

It is syntactically similar to C and C++, though it also bears resemblance to the ML family of languages, offering features like type inference, pattern matching, algebraic data types and closures. Rust uses the call-by-value evaluation strategy, and does also offers safe pointer types, called *references*. Shared references, which give read-only access to the referenced value, are denoted by the by the `&`-prefix, and mutable references, which give read- and write access to the referenced value, are denoted by the `mut &`-prefix. In C and C++, all references are mutable, whereas in Rust, there can at most be a single mutable reference at a time, or multiple immutable references.

This restriction is enforced by the *borrow checker* at compile time, which in addition to enforcing this so-called *aliasing XOR mutability*, also ensures that the references point to a piece of memory which can be accessed by the program, and which has not been freed. This is done through *lifetimes*, which must hold for the duration of the reference — ensuring freedom from use after free errors. Lifetimes are for the most part inferred, but do sometimes have to be annotated through the `'`-prefix. There exists one reserved lifetime: `'static`, which indicates that the data lives for the entire lifetime of the program.

Syntax

Variables and functions

Variables are declared by using `let`, as in:

```
let variable = true;
```

By default, variables are *immutable* — we cannot change the value of a variable after it is declared. The following is thus illegal, and will result in a compilation error:

```
let variable = true;
variable = false;
```

To be able to change a variables value after declaration, we have to declare it as *mutable*, by using the `mut` keyword. Thus, the following code is perfectly legal:

```
let mut variable = true;
variable = false;
```

All variables in Rust carry a type. Rust types are for the most part inferred, but can also be explicitly annotated. This is done by adding a colon after the variable name, as in:

```
let variable: usize = 0;
```

The code above declares a variable of type `usize`. `usize` is an unsigned integer, and its size is the amount of bits needed to reference any location in memory. This means that on 64-bit platforms, a `usize` is 64 bits, and on 32-bit platforms, it is 32 bits. Other integer types include the `u8`, `u16`, `u32`, `u64`, and `u128` for the unsigned integers of 8, 16, 32, 64 and 128 bits, and correspondingly there is `i8`, `i16`, `i32`, `i64`, and `i128` for signed integers of 8, 16, 32, 64 and 128 bits.

Functions in Rust can be declared with the `fn` keyword, as such:

```
fn equals(a: bool, b: bool) -> bool {
    return a == b;
}
```

Note that we could also omit the `return` keyword and the `;` at the end of the last line of the function to get an equal, but more succinct function:

```
fn equals(a: bool, b: bool) -> bool {
    a == b
}
```

All functions must declare the type of their return value, as we have done above with the `-> bool` annotation. Note also that we have to declare the type of all parameters to a function.

Composite data types

There are a few composite data types in Rust. We start with the tuple:

```
let tuple: (i32, f64, u8) = (123, 3.14, 1);
let a = tuple.0; // I'm the 32 bit signed integer 123
let b = tuple.1; // I'm the double precision float 3.14
let c = tuple.2; // I'm the 8 bit unsigned integer 1
```

A composite data type which is quite similar to the tuple is the struct:

```
struct MyStruct {
    a: i32,
    b: f64,
    c: u8,
}
```

```
let my_struct = MyStruct {a: 123, b: 3.14, c: 1};
let a = my_struct.a; // I'm the 32 bit signed integer 123
let b = my_struct.b; // I'm the double precision float 3.14
let c = my_struct.c; // I'm the 8 bit unsigned integer 1
```

Both the struct and the tuple are fixed length, and consists of a series of typed fields. Their main difference is syntactical, and a struct can be viewed as a tuple where all the fields are named.

Continuing, we have the array, which is a fixed length sequence of values, where every value must be of the same type. Arrays in Rust are always allocated on the stack. They are 0-indexed and can be indexed using `[]`:

```
let a = [1, 2, 3, 4, 5];
let three = a[2];
```

A construct which is quite similar to the array is the vector:

```
let mut v = Vec::new(); // v = []
v.push(123);            // v = [123]
v.push(321);            // v = [123, 321]
v.insert(0, 1000);      // v = [1000, 123, 321]
v.swap(0, 2);           // v = [321, 123, 1000]
let a = v[0];           // v = [321, 123, 1000], a = 321
v.remove(1);            // v = [321, 1000], a = 321
let b = v.pop();        // v = [321], a = 321, b = Some(1000)
```

The vector is a heap-allocated sequence of values, where every value must be of the same type. Whereas arrays are of fixed length, vectors can change length, through for instance `push`, `insert`, `pop`, or `remove`. Vectors are efficient and very practical, and will be used extensively in our solvers.

Enums, generics and matching

The reader may have noticed that we write in the comments that `b` is equal to `Some(1000)`, and not `1000`, as one might expect. This is Rust's way of handling nullability. When calling `pop`, it may be that the vector was empty, which would mean that there would be no element for `pop` to return. In some languages this would be solved by for instance assigning NULL to `b`, or by having `pop` on an empty vector cause an error. Assigning NULL is not possible in Rust, as, with the exception of what is called *raw pointers*, which we do not use, types are not nullable. We thus get an `Option<i32>`, to represent that we either got our desired value, or that we got nothing. `Option` is defined as follows:

```
pub enum Option<T> {
    None,
    Some(T),
}
```

The definition of `Option` begs a few questions: what is an enum, what is this `<T>` syntax, and how does one get the value out of an `Option`?

The answer to the first question is that an enum is a construct to represent a type which can be one of a set of *variants*. This is similar to enums in C, but whereas in C enums are simply a mapping from a name to an integer, in Rust they can also carry arbitrary data, as we see for the `Some(T)` case above. The `T` indicates that `Option` is *generic* — it can take any type. This means that we can have `Some(1000)`, `Some(true)`, and `Some(-123)` without having multiple implementations for `Option`.

As a matter of fact, we have already seen an instance of a generic type: the vector. This allows a single vector implementation to be used to create vectors of `i32`s, `bool`s, `Option<MyStruct>`s, or whatever else we would like to store in a vector. During compilation, the Rust compiler will look for all the usages of `Vec`, and all other generic types for that matter, and *monomorphize* them. This means that if you for instance have `Vec<i32>`, `Vec<bool>` and `Vec<MyStruct>` in your source code, the generated code will have three vector implementations, one for each. These are optimized for their concrete type, and generics thus enable us to write a single implementation, and have the runtime performance as if we wrote one implementation for each type.

Finally, to the question of how to retrieve the value from the `Option`. To do this, we can use a match statement, for instance by writing the following:

```
let b_val = match b {
    Some(val) => val,
    None      => 0,
};
```

Or by doing it directly, without assigning to `b` first:

```
let b = match v.pop() {
    Some(val) => val,
    None      => 0,
};
```

Match statements are similar to if-statements in that they allow our code to do different things depending on a value. They do however offer a feature which is not offered by if-statements in that match statements are *exhaustive* — we have to account for all possible values. As such, the following code will fail to compile, due to us not handling the `None` case:

```
let b = match v.pop() {
    Some(val) => val,
};
```

In the case of `Option`, there are only two possible variants, but other types may have more variants. For instance, if we were to match on an `i32`, we would have to account for $2^{32} - 1$ distinct values. To make it so that we do not have to write $2^{32} - 1$ different arms to our match statements, we have some options. One such option is to match on ranges of values with the `from..=to` syntax. Another option is to give anything which is not a

pattern, and which is a valid variable name. This will match and bind to the given name. For example, the following code will bind whatever a is to to the variable b:

```
match a {
    b => println!("This will always happen"),
}
```

In the code above, we do not use the b variable. In this cases it is better to use use the _ wildcard pattern, which will not bind the matched value to a variable.

We can also group matches together by the usage of |, which corresponds to "or". As an example, a function which matches on a u32, and prints depending on its value:

```
fn match_and_print(a: u32) {
    match a {
        0       => println!("Zero"),
        1 | 2   => println!("Either one or two"),
        3..=10 => println!("Between three and ten"),
        _       => println!("The number is over ten"),
    }
}
```

There are some more possibilities with matches which we will not use, and thus will not cover. We end with the introduction of *guards*, which is simply a further if condition after a pattern. We can make a function which checks if a given number is even or odd as follows:

```
fn even_or_odd(a: i32) {
    match a {
        b if a % 2 == 0  => println!("{} is even", b),
        b if a % 2 == 1  => println!("{} is odd", b),
        _                => println!("This is unreachable"),
    }
}
```

Ownership and borrows

We will now look at the concept of *ownership*, and the *moving* of values, before looking at *borrows*. As an example of ownership and moving, take the following code:

```
fn take_ownership(mut v: Vec<i32>) {
    // I now own v
    v.push(4);
} // v is dropped here

fn main() {
    let mut v = vec![1, 2, 3];
```

```
    take_ownership(v); // v is moved here
}
```

The `main` function instantiates the `v` vector, before giving it away to the `take_ownership` function. `take_ownership` is then free to do whatever it wants to the `v` vector, and could for instance give it away further. At the end of `take_ownership`, `v` goes out of scope, and is thus *dropped*, which means that the backing memory for the vector is deallocated.

This means that the following code is illegal, and will not compile:

```
fn take_ownership(mut v: Vec<i32>) {}

fn main() {
    let mut v = vec![1, 2, 3];
    take_ownership(v); // v is moved here
    take_ownership(v); // This is illegal
}
```

If we would like to keep using `v` in `main` after calling `take_ownership`, we would have to pass a *borrow* to `v`, as in:

```
fn take_borrow(v: &mut Vec<i32>) {
    v.push(123);
} // The borrow is given back here

fn main() {
    let mut v = vec![1, 2, 3];
    take_borrow(&mut v); // v is borrowed here
    take_borrow(&mut v); // v is borrowed again here
    println!("{:?}", v); // [1, 2, 3, 123, 123]
}
```

Borrows can either be mutable, as in the example above, or immutable, which is denoted with `&` instead of `& mut`. To access the value of the borrow, we have to *dereference* it, by the usage of the `*` operator. The reader might notice that there are no occurrences of `*` in the code above. This is because the Rust compiler inserts this for us, by a process of *auto-dereferencing*.

If a function takes an immutable borrow, also known as a *shared* borrow, the type system ensures that no mutable borrows exists, and if a function takes a mutable borrow, then it is ensured that no other borrows exist. This means that the following will fail to compile, as we are giving a mutable borrow to a part of a shared borrow:

```
fn take_borrows(num: &mut i32, v: &Vec<i32>) {}

fn main() {
    let mut v = vec![1, 2, 3];
    take_borrows(&mut v[0], &v);
}
```

Copy semantics and traits

When passing integers, floats, bools, and other primitive types to functions, it would not be ergonomic, nor efficient, if our only two options were to either pass them by reference, or have them moved. Fortunately, this is not the case, as these are subject to *copy semantics*, which means that the following is allowed:

```
fn add(mut a: i32, b: i32) {
    a += b;
}

fn main() {
    let a = 21;
    add(a, 34);
    add(a, a); // We can even use it twice
}
```

Since `i32`s are subject to copy semantics, we can keep using `a` after the call to `add`. As we saw when discussing ownership and borrows, this is not the case for vectors, which are subject to *move semantics*. This is because vectors cannot be duplicated simply by copying bits, which is the criterion for a type to enable copy semantics. If we were to copy all the bits that make up the vector, we would, in addition to duplicating the values of the vector, also duplicate the responsibility of managing the backing memory, which would lead to a double free. Vectors and other types with backing memory do thus not allow copying, but can be cloned by calling `.clone()`.

Types have move semantics by default, and do not enable cloning. Copy semantics and cloning can be enabled for a construct by implementing the `Copy` and `Clone` *trait*. A trait is a collection of methods defined for a type, and traits are thus similar to *interfaces* in Java, *abstract classes* in C++, or the typeclasses of Haskell. Traits are used to implement ad-hoc polymorphism, and, as our solver will be entirely monomorphic, we will for the most part see them when we implement traits for internal types which satisfy external traits.

To implement the `Clone` and `Copy` traits, we can do the following:

```
struct S;

impl Copy for S {}

impl Clone for S {
    fn clone(&self) -> S {
        *self
    }
}
```

Traits such as `Copy` and `Clone` are usually not implemented manually, and are rather *derived*, which means that their implementation is automatically

generated by the Rust compiler:

```
#[derive(Copy, Clone)]
struct S;
```

As another example of traits, the traits for the `enum AssignedState`:

```
#[derive(Copy, Eq)]
pub enum AssignedState {
    Unset,
    Pos,
    Neg,
}

impl PartialEq for AssignedState {
    fn eq(&self, other: &Self) -> bool {
        return match (self, other) {
            (AssignedState::Unset, AssignedState::Unset) => true,
            (AssignedState::Pos,   AssignedState::Pos)   => true,
            (AssignedState::Neg,   AssignedState::Neg)   => true,
            _ => false,
        };
    }
}
```

Here we see three traits — `Copy`, `Eq` and `PartialEq`, where the first two are derived. We implement `PartialEq` as one might expect: instances of the same variant are considered equal, all other combinations are considered not equal. This enables us to compare `AssignedState`s with `==` and `!=`. With `PartialEq` implemented, we can derive `Eq`, which has no extra methods. It simply takes a type implementing `PartialEq`, and assures reflexivity, symmetry and transitivity.

2.2 SAT and SAT solvers

2.2.1 SAT

The *Boolean satisfiability problem* (SAT) is the problem of deciding whether a Boolean formula is satisfiable — whether all variables of the formula can be replaced with true or false such the entirety of the formula evaluates to true. SAT is the first problem to be proven NP-complete, and is considered the canonical NP-complete problem [14]. As it is NP-complete, all other problems in NP can be reduced to it in polynomial time. We can thus make an efficient SAT solver, and use it to solve all other problems in NP. Examples of areas where this is useful include electronic design automation (EDA) and bounded model checking (BMC).

There exists no known efficient algorithm for solving SAT. That being said, through the discovery of new algorithms and techniques, combined with the general improvement in hardware over the period, SAT solvers have

become capable of solving large industrial problems. Since 2002 there has been held yearly SAT solving competitions. These include a selection of problems from various domains, and people submit solvers in an attempt to solve as many problems as possible within the given time and memory limit.

2.2.2 CNF

The standard input format for SAT solvers is called *conjunctive normal form* (CNF). A formula is in CNF if it is a conjunction of clauses where each clause is a disjunction of literals — an AND of ORs. For instance:

$$(a \vee b \vee c) \wedge (\neg a \vee b \vee c) \wedge (a \vee \neg b \vee \neg c)$$

is a formula in CNF-form of three clauses with three literals each, whereas

$$(a \wedge b \vee c) \wedge (c \vee a) \wedge (c \vee b)$$

is not — the latter has $\wedge$ in the first clause. All Boolean formulas can be transformed to be in CNF-form. An example of an algorithm which does this is the *Tseitin transformation* [55], which runs in linear time with regards to the input formula.

The *Center for Discrete Mathematics and Theoretical Computer Science* (DIMACS) has developed a standard for formulas in CNF form to be used for SAT-solvers, known as DIMACS CNF.

It is very simple:

- Lines starting with `c` are comments
- The line starting with `p` indicates problem type, number of clauses and number of variables.
- Clauses are indicated by a series of integers, terminated by a `0`. The `-`-prefix indicates the negation of a variable.

Note that a clause may span multiple lines. The first formula above can be encoded as:

```
c This is a comment line
p cnf 3 3
1 2 3 0
-1 2 3 0
1 -2
-3
0
```

2.2.3 Algorithms for solving SAT

During the course of this thesis, we will look at and verify various algorithms for solving SAT. Instead of presenting them here, we present them right before we discuss their ideas and how we implement and prove

them. It should be noted that there exists more algorithms for solving SAT than are presented in this thesis, and that there exists both complete and incomplete algorithms for solving SAT. All the algorithms presented in this thesis are complete algorithms, and they form what can be considered the "chain" of state-of-the-art algorithms from the start of computing to today.

2.3 Proof of code and CREUSOT

When developing a computer program, it is desired that the program is *correct* — that it does what it is intended to do. To ensure the correctness of programs, various techniques are employed, ranging from testing, which can increase our confidence in the correctness of programs, to full proven correctness[1], which allows us to guarantee that the program is correct with regards to some specification.

There exists various approaches to program verification, with various strengths and weaknesses. Most of the existing work in proving SAT solvers has been done by utilizing what is called *proof assistants* or *interactive theorem provers*. Examples of interactive theorem provers include Coq [12], Isabelle [45], PVS [50], and Agda [46]. Some interactive theorem provers allow for the generation of code which is *correct by construction*. This means that if we for instance have proven a specification of a SAT solver, then we can use the code generation function to generate a functionally correct SAT solver. Another approach is to start with a program implementation, and then prove that the implementation satisfies some specification. This is the approach taken by the tool which we are going to use, CREUSOT, which is what is called a *deductive verification* tool.

Deductive program verification is the act of verifying all possible program behaviors through a process of logical inference. In the early days of computing, it was done as pen-and-paper proofs, as with Alan Turing's 1949 paper *Checking a large routine* [41], or with Hoare's 1971 paper *Proof of program: FIND* [26], but the process has since been mechanized. The ideas are however the same: we express the correctness of the program as a set of mathematical expressions, and then prove those. The specifications may appear alongside the program code, or exist externally, and the proof may be done either manually, or through the usage of either automated provers or interactive provers.

Some languages, such as Ada [5], Eiffel [40], or Dafny [31], support verification as a core part of the language, whereas other languages have support for program verification through external tooling. Examples of the latter include the Java Modelling Language (JML) [30] for Java, the ANSI/ISO C Specification Language (ACSL) [7] for C, or CREUSOT, which offers the PEARLITE specification language for verification of Rust code.

[1]Strictly speaking, there is no such thing as "full proven correctness", since, at some point, we have to trust some abstraction, be it the correctness of the CPU, a compiler, the Coq kernel, the SMT solvers used to verify the proof, that the used RAM is not faulty etc. We thus prove full correctness with regards to a model of the system and the environment.

These annotations occur intertwined with the program code, and the proofs are for the most part done automatically, though one can use manual *tactics* if one desires.

CREUSOT

CREUSOT is a deductive verification tool for Rust which is currently under development. It is based on the WHY3 [19] platform for deductive program verification.

The process is visualized in Figure 2.1.

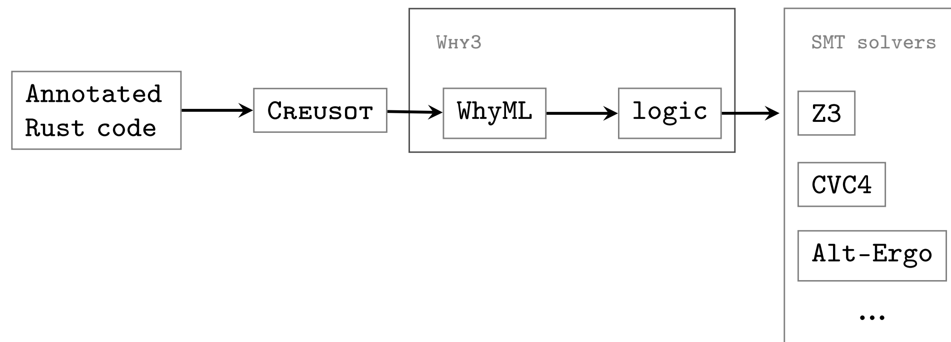

Figure 2.1: Illustration of the process from Rust code to proven program

At a high level, CREUSOT works by taking Rust code annotated with contract expressions, which is then parsed and converted into *MLCFG*, a call flow graph for WhyML. The MLCFG is then fed to WHY3, where a structured program is reconstructed. One can then deploy SMT-solvers such as Z3 [43], CVC4 [6], and Alt-Ergo [13] to prove the goals derived from the CREUSOT contracts.

When doing proofs, we use the WHY3 IDE. The WHY3 IDE allows us to see the proof context and the corresponding goal, the generated verification conditions (VCs), as well as automatic and manual discharging of various transformations. All the proofs presented in this thesis pass when running the default "Auto level 3" strategy, which runs the installed solvers until they timeout, then deploy various transformations such as "split" and the "inline_" transformations, before it increases the timeout limit and tries running the solvers again. Auto level 3 stops whenever a solution is reached, or the solvers timeout on the maximum timeout limit.

As an illustration of how a proof in CREUSOT is conducted, let us prove the `incr` function, which we define as follows:

```
fn incr(x: &mut u32) { *x += 1 }
```

We use CREUSOT to generate the corresponding MLCFG and load the MLCFG in the WHY3 IDE. The resulting WHY3 IDE session can be seen in Figure 2.2.

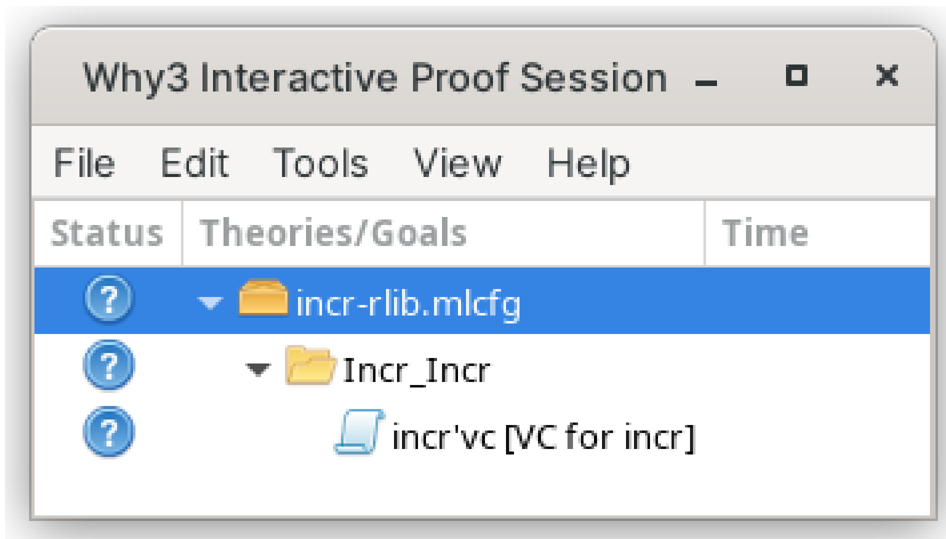

Figure 2.2: The `incr` function loaded in the WHY3 IDE

If we run the "split" strategy, and look at the generated VC and the corresponding "Task" panel, we see that what we have to prove to ensure safety of this function, is that the addition does not overflow. We see the resulting WHY3 IDE session in Figure 2.3.

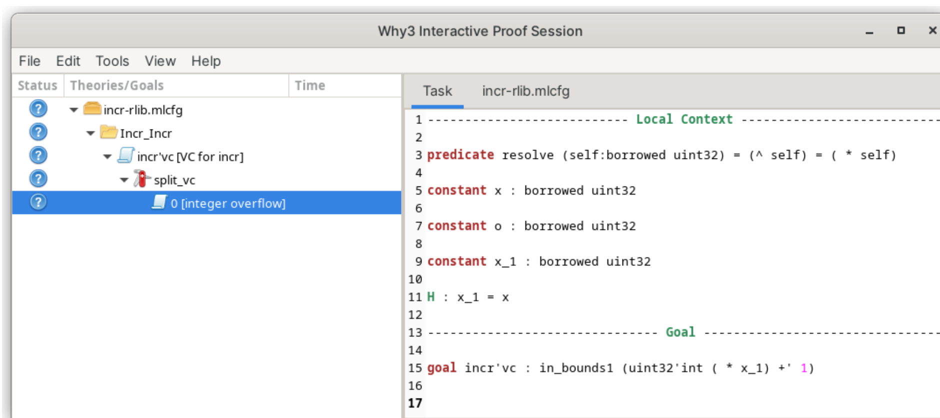

Figure 2.3: The `incr` function in the WHY3 IDE after running a split

There are two ways to make this VC pass. The first is to add a runtime check on whether `x` is the maximum value for an `u32` inside of the `incr` function. The second way is to add a `#[requires()]` contract to the function, which requires the caller to ensure that `x` is less than `u32::MAX` (constant corresponding to $2^{32} - 1$) when calling `incr`. As part of the reason for doing program proofs is to remove runtime checks, we choose the latter, and add `#[requires(*x < u32::MAX)]` as a contract to `incr`. The resulting WHY3 IDE session after splitting and running Alt-Ergo can be seen in Figure 2.4.

In Figure 2.4 we can see that the task context has been updated with the line

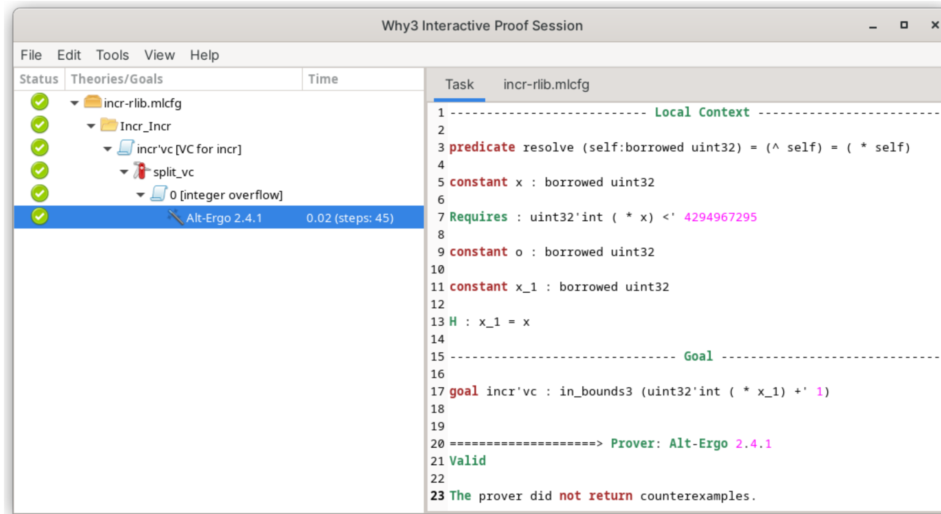

Figure 2.4: The `incr` function in the WHY3 IDE with proven safety

`Requires : uint32'int ( * x) <' 4294967295`, which is the translated version of *`#[requires(*x < u32::MAX)]`*. Due to this requirement being added, Alt-Ergo is able to prove that our goal holds.

Having proven safety, we can look at proving functional correctness. What we want to state is that the value of `x` when the function returns is equal to the value at the function entry plus one. To do this, we add the following contract: *`#[ensures(^x == *x + 1u32)]`*. `^` is what is called the *final* operator, corresponding to the value of a mutable borrow at the end of the borrow, which for `incr` is when the function returns[2]. The `*` operator is the regular dereference operator of Rust, which gets the value of `x` on function entry. The `ensures` contract can thus be read as "ensures that the final value of `x` is equal to the initial value of `x` plus 1". The resulting WHY3 IDE session after splitting and running Alt-Ergo can be seen in Figure 2.5.

Much of the strength of CREUSOT comes from it being able to leverage the WHY3 verification platform, which is a verification platform for the functional language WhyML. WHY3 offers a slew of transformations, where the most important ones are the "split" and "inline_" transformations, as well as integration with multiple SMT solvers. The reason why CREUSOT can leverage WHY3 is due to it being able to efficiently translate Rust code to WhyML. This is enabled by the insight that the *safe* subset of Rust, without what is called *interior mutability*, behaves like a functional language. This is due to the type system enforcing the aforementioned aliasing XOR mutability, which ensures that there can not be more than one mutable reference to a value at a given point. Thus, all mutating operations can safely be modelled as the assignment of a new variable.

[2] All the code shown in the thesis will have end of borrow corresponding to function exit. It is however possible to have borrows which end during the lifetime of the function, for instance when iterating over a data structure by taking mutable borrows to its elements.

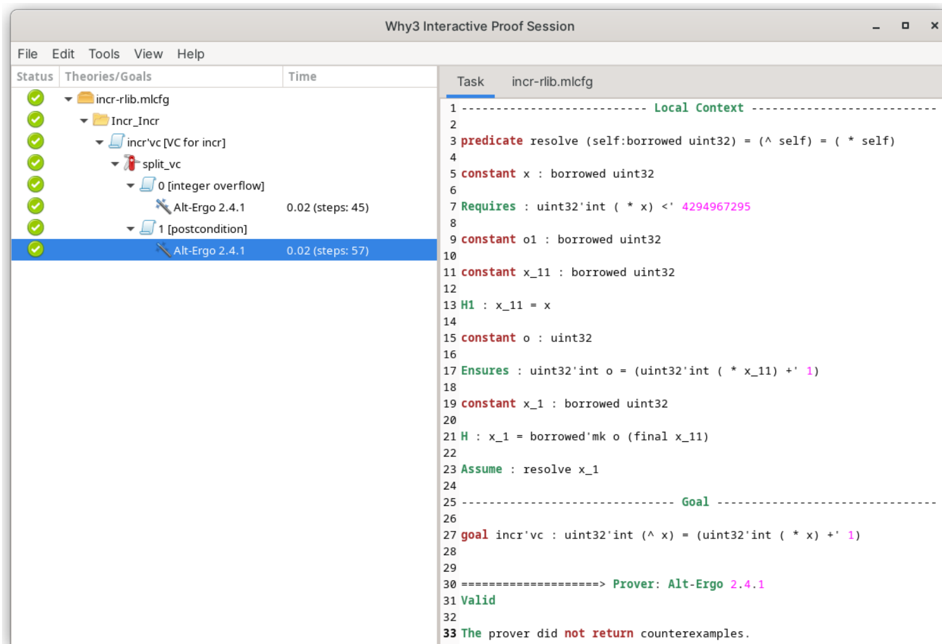

Figure 2.5: The `incr` function with full proven correctness in the WHY3 IDE

PEARLITE

CREUSOT's language for writing specifications is called PEARLITE. It is very similar to Rust, and does in addition to many of the constructs found in Rust, offer some logical operations and connectives. We have already seen parts of it in the example above, and will now explain it in detail.

A function can be marked with `#[predicate]` or `#[logic]`, and then invoke the `pearlite!` macro to gain access to PEARLITE's features. PEARLITE allows access to the constructs found elsewhere in the program, but not mutability or functions with side effects. It is type checked, but not borrow checked, as the idea of ownership does not make sense in a logical context.

The additional annotations offered by PEARLITE:

- `#[requires()]` and `#[ensures()]` to define preconditions and postconditions for functions.
- `#[invariant()]` to annotate invariants which hold for a loop.
- `#[variant(EXPR)]` where `EXPR` implements the `WellFounded` trait to prove termination of functions.
- `#[trusted]` to indicate that a piece of code should not be checked by CREUSOT. This is useful in the case of unsupported features, or for establishing properties CREUSOT cannot reason about. `trusted` can also be used as a temporary annotation while doing the proof work.

The additional constructs offered:

- `@` to gain access to the *model* of an object. See below for a thorough explanation.
- `forall` and `exists` to gain access to the quantifiers of first order logic.
- `==>` for logical implication.
- `^` to gain access to the final value of a mutable borrow.
- `result` for the return value of a function.

The totality of macros offered:

- `pearlite!` to gain access to the extended syntax of PEARLITE.
- `proof_assert!` to provide a proof assertion which must hold at a given point in the program. This is used either to guide the solvers, or during development, similar to regular assertions or the `println!` macro. Note that a `proof_assert!` block must evaluate to a `bool`.
- `#[maintains()]` which gets rewritten to the corresponding `#[requires()]` and `#[ensures()]` contract. The `mut` keyword is used to indicate that a `*` should be inserted in the precondition, and a `^` should be inserted in the postcondition. For instance: `#[maintains((mut self).invariant())]` becomes `#[requires((* self).invariant())]` and `#[ensures((^ self).invariant())]`.

Models

When writing specifications, it is often useful to abstract away implementation details. To do this, we use the *model operator*, denoted `@`, which is syntactic sugar for the `Model` trait. The `Model` trait is defined as follows:

```
pub trait Model {
    type ModelTy;
    #[logic]
    fn model(self) -> Self::ModelTy;
}
```

A *model* is simply a mapping from a Rust type to a logical type.

As a motivating example, take the `compare` function:

```
fn compare(a: u32, b: usize) -> bool {
    (a as usize) < b
}
```

Proving safety for this function would entail proving that a `u32` can be safely casted to a `usize`, which it can. If we wanted to prove functional correctness as well, we would essentially have to redo the type cast in our `#[ensures()]` statements, writing contracts such as `#[ensures(result == (a as usize) < b)]`. Casting a value to a concrete representation does not really make sense in the logic — what we want

to reason about is the values of `a` and `b` in relation to each other, not their types. We therefore have the option to "cast" all integers to a mathematical integer, denoted `Int`. This results in the postcondition of the above function being `#[ensures(result == (@a < @b))]`.

We look at how the process works by looking at the `Model` of `u8`:

```
impl Model for u8 {
    type ModelTy = Int;
    #[logic]
    fn model(self) -> Self::ModelTy {
        Int::from(self)
    }
}
```

It states that the `ModelTy`(pe) of `u8` is `Int`, and that to transform a `u8` to an `Int`, the `Int::from()` function has to be called on the `self` parameter. `Int::from()` for `u8` is defined as:

```
impl From<u8> for Int {
    #[logic]
    #[trusted]
    #[creusot::builtins = "prelude.UInt8.to_int"]
    fn from(_: u8) -> Self {
        absurd
    }
}
```

As we can see, `from` simply returns `absurd`, which denotes an unreachable piece of code. What is really happening is that it gets translated to `prelude.UInt8.to_int`, which is a WhyML function which transforms a `UInt8` to an `Int`. This also highlights a point where we need to use `trusted` — there is no way for Creusot to reason about `prelude.UInt8.to_int` so the transformation has to be proven in some other tool, for instance Coq. This process ensures that both signed and unsigned integers of different sizes can be compared by doing comparisons on their model, `Int`.

The other WhyML type which we will use pervasively, is the `Seq<T>` type, where `T` is a Rust type. `Seq<T>` is the model for vectors, arrays and slices. As an example, a sneak peek of our model for `Formula`. We define `Formula` as a struct with two fields:

```
pub struct Formula {
    pub clauses: Vec<Clause>,
    pub num_vars: usize,
}
```

And model it as a tuple consisting of a `Seq<Clause>` and an `Int`:

```
impl Model for Formula {
    type ModelTy = (Seq<Clause>, Int);
    #[logic]
```

```
    fn model(self) -> Self::ModelTy {
        (self.clauses.model(), self.num_vars.model())
    }
}
```

The reason we do this, in addition to the aforementioned comparison between integers, is that defining and working on models allows us to access WhyML functions on our constructs. This is especially useful when talking about what one might call "hypotheticals" — what happens if one were to do some change to the value. An example of a function which allows us to do that, is the `.set(index, value)` on sequences. It allows us to construct a "hypothetical" sequence, which is like the original sequence, except we have set the value at `index` to the value given by the `value` parameter. Another such function which we will be using, is the `.push(value)` function to talk about a sequence if we were to push some `value` to it.

Ending notes and a proven selection sort

We have now covered all the parts which are needed to understand the rest of the thesis. Though PEARLITE is not a very large language, building up intuitions for how to use it, and how a proof is conducted, is no trivial task. As it is with programming, most of these intuitions are best learned through practice. To aid in this, we end the chapter with a proof of a generic version of selection sort, and show in the next chapter our proof of the minimal solver Friday. A final note is that there is one assumption which all proofs in CREUSOT make: that there is sufficient memory. We do not know beforehand how much memory the target machine has, and thus, if Rust allows the allocation, we allow the allocation.

Example: selection sort

As a somewhat complete example, we prove safety and functional correctness of a generic selection sort. We choose selection sort as the algorithm is both quite simple and fairly well known, and it lets us focus on just showing the features of CREUSOT. It shows all features except for `exists`, `requires` and `proof_assert!`. The code is based on the one found on Rosetta Code, which is an online repository of implementations of common algorithms in various programming languages. It was proven by me and then refactored to use `partition()` by Xavier Denis.

```
#[predicate]
fn sorted_range<T: Ord>(s: Seq<T>, l: Int, u: Int) -> bool {
    pearlite! {
        forall<i: Int, j: Int> l <= i && i < j && j < u ==>
            s[i] <= s[j]
    }
}
```

```
#[predicate]
fn sorted<T: Ord>(s: Seq<T>) -> bool {
    pearlite! {
        sorted_range(s, 0, s.len())
    }
}

#[predicate]
fn partition<T: Ord>(v: Seq<T>, i: Int) -> bool {
    pearlite! { forall<k1: Int, k2: Int>
        0 <= k1 && k1 < i && i <= k2 && k2 < v.len() ==>
            v[k1] <= v[k2]
    }
}

#[ensures(sorted(@^v))]
#[ensures((@^v).permutation_of(@v))]
fn selection_sort<T: Ord>(v: &mut Vec<T>) {
    let mut i: usize = 0;
    let old_v = Ghost::record(&v);
    #[invariant(proph_const, ^v == ^@old_v)]
    #[invariant(permutation, (@v).permutation_of(@*@old_v))]
    #[invariant(i_bound, @i <= (@v).len())]
    #[invariant(sorted, sorted_range(@v, 0, @i))]
    #[invariant(partition, partition(@v, @i))]
    while i < v.len() {
        let mut min = i;
        let mut j = i + 1;
        #[invariant(min_is_min, forall<k: Int> @i <= k && k < @j ==>
            (@v)[@min] <= (@v)[k])]
        #[invariant(j_bound, @i <= @j && @j <= (@v).len())]
        #[invariant(min_bound, @i <= @min && @min < @j)]
        while j < v.len() {
            if v[j].lt(&v[min]) {
                min = j;
            }
            j += 1;
        }
        v.swap(i, min);
        i += 1;
    }
}
```

We first define three `predicate` functions: `sorted_range`, `sorted` and `partition`. `sorted_range` is used to assert that the given Sequence(`Seq`) is sorted in the range from the lower bound, given by the parameter `l` of type `Int`, to the upper bound, which is given by the parameter `u` of type `Int`. `sorted` is simply a wrapper stating that the range from the start of

the Seq to the end of the Seq is sorted. partition is an encoding of the Selection Sort invariant: that all elements in the first partition are smaller than all elements in the second partition, and vice versa.

A couple of things warrant further explanation: the `<T: Ord>` syntax, and the fact that we are operating over Seq and Int, neither of which are regular Rust types. The `<T: Ord>` is what is called a *trait bound*: this function accepts any type T which satisfies the Ord trait – the type forms a total order. The reason for using Seq and Int is that we are operating in the *logic*. If we look at the places where the predicates are used, for instance in `#[invariant(partition, partition(@v, @i))]`, we see that we use the @-operator to access the *models* of v and i. The model of Vec is Seq, and the model of usize is Int.

Moving on to the main selection_sort function, there are a couple things to note. The first is that the function does not have any require statements, meaning that no preconditions have to hold. In other words: if the Rust compiler allows you to call selection_sort on your vector, then your vector will become sorted, as ensured by the two ensures-statements. The other things to note is the usage of ^, and the statement `let old_v = Ghost::record(&v);`.

The ^ operator is, as mentioned, the prophetic operator *final*, which gives us access to the value of a mutable borrow at the end of its lifetime. In this case, this is the end of the call to the selection_sort function. The usage of `let old_v = Ghost::record(&v);` is due to a limitation in CREUSOT/WHY3. We need to maintain a *ghost* version of the vector, and then use the two invariants `^v == ^@old_v` and `(@v).permutation_of(@*@old_v)` to maintain that the code version and the ghost version contain the same elements, and that they are equal at the end of the borrow.

Chapter 3

Verification of a minimal solver

We conclude the first part by implementing and proving a minimal solver, which we call Friday. The purpose of this is to connect the components of the background chapter, as well as to ease the transition into Robinson and CreuSAT. We also believe that a solver of this calibre could be useful for those interested in learning Creusot, for instance by removing the annotations and redoing them, or by extending the solver to mutate the partial assignment instead of returning a mutated clone.

Friday, Robinson and CreuSAT have the same core representation for literals, clauses, the formula and the partial assignment. CreuSAT augment the existing data structures, and has further auxiliary data structures, in addition to further invariants on the existing data structures. We will explain all parts of Friday in detail, but will later omit explaining most of the concrete implementation, rather giving a higher level overview of the ideas of the proof.

3.1 The algorithm

As this is our starting point, the solver is not very complex. It simply tries all possible assignments, and if it ever reaches a satisfiable assignment, it returns `true`. If none of the assignments are satisfiable, that means the formula is not satisfiable, and we return `false`.

In Rust-like pseudocode:

```
fn solve(f: &Formula, pa: Pasn) -> bool {
    if pa.complete() { return f.eval(&pa.assign); }
    solve(f, set_next(&pa, true)) || solve(f, set_next(&pa, false))
}
```

3.2 The proof idea

What we have to prove is that if the formula is satisfiable, then we state that it is satisfiable, and if it is not satisfiable, then we state that it is not satisfiable. To prove this, we have to establish:

1. That we return that the formula is satisfiable if we find an assignment which satisfies it.
2. That we try all possible assignments.

3.3 Implementation of Friday

As we remember from Section 2.2 - SAT and SAT solvers, a SAT solver takes as input a Boolean formula in CNF form, and returns whether the formula is satisfiable or not. A formula consists of a series of clauses, each of which consists of 0 or more literals. We start with deciding on a representation for literals:

```
struct Lit { idx: usize, polarity: bool }
```

A `Lit` consists of an `idx`, which identifies which literal it is, and a `polarity`, which identifies whether the literal is positive or negative. We allow `idx` to be `0`, which means that, since CNF is 1-indexed, we subtract 1 as a part of the parser, and add 1 when printing the result. Note also that we have chosen a somewhat inefficient representation: we could have represented both index and polarity in 64 bits, for instance by keeping the encoding from the CNF. We have chosen not to do this, to make our solvers easier to reason about.

We continue, with our representation for clauses:

```
struct Clause(Vec<Lit>);
```

A `Clause` is simply a vector of literals. We could again have been more efficient: efficient SAT solvers want to avoid cache misses, and therefore store data which is often accessed together with the clause before the clause, as a *clause header*. This is something which we will do later. The other thing which efficient solvers do, is to not store a vector of clauses, which again is a vector of literals, but to have the clauses be a linear buffer with clause headers followed by literals, and then keep track of the clause boundaries as a part of this structure. These changes would remove the need of following pointers, and also improve cache locality, so it would be faster, but it would make our code harder to reason about, so we again choose a simple representation.

Our representation for the formula:

```
struct Formula { clauses: Vec<Clause>,  num_vars: usize }
```

The `num_vars` variable states how many variables we have in our input formula, and `clauses` is a vector of the clauses which make up the input

formula.

Furthermore, we need a representation of the current assignment:

```
#[derive(Clone)]
struct Assignments(Vec<bool>);
```

`Assignments` is simply a vector of Booleans, indicating whether a variable is assigned to true or false. We will later make the assignment vector ternary, to introduce the option of no assignment yet. For now we ensure this with our last construct, and the only construct which is not present for Robinson and CreuSAT, the `pasn`, short for partial assignment:

```
#[derive(Clone)]
struct Pasn { assign: Assignments, ix: usize }
```

It contains our current assignment, and uses `ix` to keep track of how far along we are in assigning to it. This is of course an optimization — we could have made `Assignments` ternary, and then traversed it until we found an index which was not set. This would not be any simpler, and would as well be even slower, so we decided to make the `Pasn` struct. In CreuSAT we will have another construct which contains the current assignments, the `Trail`.

Finally, we present the solver, in its entirety:

```
impl Clause {
    fn eval(&self, a: &Assignments) -> bool {
        let mut i: usize = 0;
        let clause_len = self.0.len();
        while i < clause_len {
            if a.0[self.0[i].var] == self.0[i].value {
                return true;
            }
            i += 1;
        }
        false
    }
}

impl Formula {
    fn eval(&self, a: &Assignments) -> bool {
        let mut i: usize = 0;
        while i < self.clauses.len() {
            if !self.clauses[i].eval(a) { return false; }
            i += 1;
        }
        true
    }
}
```

```
fn set_next(pa: &Pasn, b: bool) -> Pasn {
    let mut new_pa = pa.clone();
    new_pa.assign.0[pa.ix] = b;
    new_pa.ix += 1;
    new_pa
}

fn solve(f: &Formula, pa: Pasn) -> bool {
    if pa.ix == pa.assign.0.len() { return f.eval(&pa.assign); }
    solve(f, set_next(&pa, true)) || solve(f, set_next(&pa, false))
}

pub fn solver(f: &Formula) -> bool {
    solve(f, Pasn { assign: Assignments(
            vec::from_elem(false, f.num_vars)), ix: 0 })
}
```

Starting at the bottom, we find our entry point, `solver`, which simply instantiates a partial assignment and calls `solve`. In `solve` we check if our assignment is complete, and if it is, then we return the valuation of the formula under the current assignment. If we are not complete, then we call `solve` with the current index set to `0`, and if that returned `false`, then we try setting the current index to `1`.

The `set_next` function consists of cloning before setting the current index to the provided value and incrementing the index. `eval` for `Formula` and `eval` for `Clause` both correspond to the intuitive notion of satisfiability: a formula is SAT if all its clauses are SAT, and is UNSAT if at least one clause is UNSAT. A clause is SAT if at least one literal is SAT, otherwise it is UNSAT. As we always call this on a complete assignment, we don't have to account for the undecided case.

3.4 Proof of Friday

The first part of verifying a program with Creusot consists of proving safety, which means that we have to prove that our array indexes are in bounds, and that our arithmetic operations do not over- or underflow. This is achievable by adding loop invariants on our loops, as well as requiring that `pa.ix` is less than `usize::MAX` on the entry of `set()`, which we do by requiring that `pa.ix` is less than the length of the assignment contained in the partial assignment. The proof of the safety properties can be seen in the final code.

With safety proven, we introduce the *invariants* of the data structures. We go in the same order as before, starting with the invariant for `Lit`:

```
impl Lit {
    #[predicate]
    fn var_in_range(self, n: Int) -> bool {
```

```
        pearlite! {
            @self.var < n
        }
    }
}
```

The invariant we have on `Lit`, `var_in_range`, simply states that the index is less than some `n`. The n which we are going to be invariant with regards to is `f.num_vars`. In other words, it is used to state that the index of a literal is in the range [0, `f.num_vars`).

Perhaps not shockingly, the invariant for the `Clause` states that all its literals satisfy their invariant:

```
impl Clause {
    #[predicate]
    fn vars_in_range(self, n: Int) -> bool {
        pearlite! {
            forall<i: Int> 0 <= i && i < (@self.0).len() ==>
                (@self.0)[i].var_in_range(n)
        }
    }
}
```

And the invariant for `Formula` states that all the clauses satisfy their invariant:

```
impl Formula {
    #[predicate]
    fn invariant(self) -> bool {
        pearlite! {
            forall<i: Int> 0 <= i && i < (@self.clauses).len() ==>
                (@self.clauses)[i].vars_in_range(@self.num_vars)
        }
    }
}
```

In the coming solvers we will have invariants which are a fair amount larger, but the general structure of them largely remains.

With the invariants out of the way, we can start focusing on proving the semantics of our solver. For that we need a notion of satisfiability. We present `sat` for `Lit`, `Clause` and `Formula`, none of which should contain any surprises:

```
impl Lit {
    #[predicate]
    fn sat(self, a: Assignments) -> bool {
        pearlite! { (@a.0)[@self.var] == self.value }
    }
}
```

```
impl Clause {
    #[predicate]
    fn sat(self, a: Assignments) -> bool {
        pearlite! {
            exists<i: Int> 0 <= i && i < (@self.0).len() &&
                (@self.0)[i].sat(a)
        }
    }
}

impl Formula {
    #[predicate]
    fn sat(self, a: Assignments) -> bool {
        pearlite! {
            forall<i: Int> 0 <= i && i < (@self.clauses).len() ==>
                (@self.clauses)[i].sat(a)
        }
    }
}
```

We also present the `compatible` predicate. It states that the `Assignments` which is given as the `self` parameter is compatible with the `Pasn` which is given as the `pa` parameter. This means that they have the same assignment for all assignments up to `pa.ix`, or more intuitively, that `self` extends `pa`. We define it as follows:

```
impl Assignments {
    #[predicate]
    fn compatible(self, pa: Pasn) -> bool {
        pearlite! {
            (@pa.assign.0).len() == (@self.0).len() &&
                forall<i: Int> 0 <= i && i < @pa.ix ==>
                    (@pa.assign.0)[i] == (@self.0)[i]
        }
    }
}
```

We are now ready to prove the implementation of `set_next`. The proven version is as follows:

```
#[requires(@pa.ix < (@pa.assign.0).len())]
#[requires((@pa.assign.0).len() <= @usize::MAX)]
#[ensures(result.assign.compatible(*pa))]
#[ensures((@result.assign.0)[@pa.ix] == b)]
#[ensures(@result.ix == @pa.ix + 1)]
fn set_next(pa: &Pasn, b: bool) -> Pasn {
    let mut new_pa = pa.clone();
    new_pa.assign.0[pa.ix] = b;
    new_pa.ix += 1;
    new_pa
```

```
}
```

We require that the `pa.ix` is in bounds, and that the length of the assignment vector is less than $2^{64} - 1$. This latter requirement is purely pro forma: it is not possible to construct a Rust vector with a length remotely close to $2^{64} - 1$. This is in general a limitation with all of the solvers: we do not know how much memory the target computer has, and may run out of memory. This will result in the program crashing, but will not yield the wrong result.

As long as the `set_next` function is called with these requirements met, it will ensure that the resulting assignment is compatible with the given assignment. It will also ensure that the provided Boolean, `b`, is assigned at the index `pa.ix`, and that the returned `Pasn` will have had its `ix` incremented.

We can now look at the proof of the `eval` functions. We start with `eval` for `Clause`:

```
impl Clause {
    #[requires(self.vars_in_range((@a.0).len()))]
    #[ensures(result == self.sat(*a))]
    fn eval(&self, a: &Assignments) -> bool {
        let mut i: usize = 0;
        let clause_len = self.0.len();
        #[invariant(prev_not_sat, forall<j: Int>
            0 <= j && j < @i ==> !(@self.0)[j].sat(*a))]
        #[invariant(loop_invariant, @i <= @clause_len)]
        while i < clause_len {
            if a.0[self.0[i].var] == self.0[i].value {
                return true;
            }
            i += 1;
        }
        false
    }
}
```

It requires that the clause invariant is satisfied with regards to the length of the assignment, and ensures that the result is a correct evaluation of the clause under the given assignment. If we look in the function body, we see that we have added two invariants. The top one of these, `prev_not_sat`, states that the literal at all previous indexes has been unsatisfied under the assignment.

The definition of `eval` for `Formula`:

```
impl Formula {
    #[requires(self.invariant())]
    #[requires((@a.0).len() == @self.num_vars)]
    #[ensures(result == self.sat(*a))]
```

```
    fn eval(&self, a: &Assignments) -> bool {
        let mut i: usize = 0;
        #[invariant(prev_sat, forall<j: Int>
            0 <= j && j < @i ==> (@self.clauses)[j].sat(*a))]
        while i < self.clauses.len() {
            if !self.clauses[i].eval(a) { return false; }
            i += 1;
        }
        true
    }
}
```

It requires that the given formula satisfies the formula invariant, and that the assignment is of the correct length. It ensures that the result is a correct evaluation of the formula under the given assignment. Looking at the function body, we see that it is very similar to `eval` for `clause`. The invariants are almost the same, but now we have `prev_sat` instead of `prev_not_sat`. `prev_sat` maintains the invariant that the clause at all previous indexes has been satisfied under the given assignment.

We continue, with the penultimate function, `solve`:

```
#[variant(@f.num_vars - @pa.ix)]
#[requires(pa.invariant(@f.num_vars))]
#[requires(f.invariant())]
#[ensures(!result ==
    forall<a: Assignments> a.compatible(pa) ==> !f.sat(a))]
fn solve(f: &Formula, pa: Pasn) -> bool {
    if pa.ix == pa.assign.0.len() { return f.eval(&pa.assign); }
    solve(f, set_next(&pa, true)) || solve(f, set_next(&pa, false))
}
```

`solve` requires that its parameters satisfy their invariants. We further see the `variant` which ensures that the function terminates. We do this by proving that the difference between `f.num_vars` and the index of the `Pasn` decreases for each recursive call. `solve` ensures that if there exists a satisfying assignment of the formula in any extension of the partial assignment, then it will return `true`, otherwise, it will return `false`.

Finally, we have the entry point of the solver.

```
#[requires(f.invariant())]
#[ensures(!result ==> forall<a: Assignments>
    (@a.0).len() == @f.num_vars ==> !f.sat(a))]
#[ensures( result ==> exists<a: Assignments> f.sat(a))]
pub fn solver(f: &Formula) -> bool {
    solve(f, Pasn { assign: Assignments(
            vec![false; f.num_vars]), ix: 0 })
}
```

Note that Friday requires that the formula satisfies its invariant on entry.

To remove this requirement, we would have to add code which checks and establishes the variant on function entry. We omit this, as it is not a core part of the algorithm. Do note that we do check and establish the invariant on entry for Robinson and CreuSAT, so these offer an entry point with any requirements.

`solver` ensures that if it returns false, then there does not exist any assignment of the correct length which satisfies the formula, and that if it returns true, then there exists a satisfying assignment.

Part II

Verification of a DPLL solver

Chapter 4

Verification of the DPLL algorithm

In this chapter we present the implementation and verification of a DPLL SAT solver, which we call Robinson. We believe that before CreuSAT, this was the largest piece of deductively verified Rust code, and it is, to the best of our knowledge, the fastest fully verified implementation of the DPLL algorithm. We believe that it may be interesting for those who want to try to prove SAT solvers with CREUSOT. Compared to CreuSAT, it is smaller, easier to understand, and proves much faster, and it does in addition have more possible optimizations which should not be too hard to implement. We have not implemented pure literal elimination, nor done much work in efficiently identifying unit clauses, which, together with doing changes to the decision mechanism, as well as potentially integrating the trail and backtracking mechanism of CreuSAT, should be both interesting and challenging to do.

We implement the solver using efficient data structures, using built in Rust-vectors and machine integers.

4.1 The DPLL algorithm

There are many possible improvements to Friday. A somewhat obvious improvement is to not write a functional solver in an imperative language, and to use for instance mutable vectors instead of cloning vectors every time we set a value. A perhaps less obvious improvement is algorithmic, and is to implement a technique called *unit propagation* (UP), also known as *Boolean Constraint propagation* (BCP). It is based on the observation that if we ever have a situation where a clause only has one unassigned literal, and we falsify this literal, the clause becomes falsified, which results in the formula becoming falsified. We therefore have to make this literal evaluate to true to make the clause evaluate to true, and thus keep the potential for the formula to evaluate to true. In other words: as we have a "forced"

assignment, there is no point in trying both assignments, and we can "shave off" parts of the search tree.

The other algorithmic improvement we can implement, is a technique which is called *pure literal elimination*. It is very simple: if a variable occurs with only a single polarity in the formula, meaning that it is always positive, or that it is always negative, then we can assign its variable to that polarity, and simplify. In simplification, we remove all the clauses which contains the literal, as they have now become satisfied.

Pure literal elimination and unit propagation combined with search form the foundation of the Davis-Putnam-Logemann–Loveland (DPLL) algorithm [15]. DPLL was discovered in 1961, and remained as the state-of-the-art until the introduction of the conflict-driven clause learning (CDCL) algorithm, which is the subject of the next chapter. The CDCL algorithm is often viewed as an extension of DPLL, and ideas from DPLL are present in the current state-of-the-art.

The algorithm in Rust-like pseudocode:

```
fn dpll(clauses: &mut Clauses) -> bool {
    if consistent(clauses) { true }
    if contains_empty(clauses) { false }
    for clause in clauses {
        if clause.len() == 1 {
            clauses = unit_propagate(clause[0]);
        }
    }
    for literal in clauses {
        if occurs_pure(literal) {
            clauses = pure_literal_assign(literal);
        }
    }
    let literal = choose_literal();
    let clauses2 = clauses.clone();
    dpll(clauses.push(literal)) || dpll(clauses2.push(-literal))
}
```

4.2 The main ideas of the proof

Before looking at the details of the proof, we give a high level overview of the core idea, and what we want to do. We are essentially augmenting Friday with mutable vectors and unit propagation. Adding mutable vectors adds a fair amount of tedium, and, though it might be useful to look at for those who want to learn CREUSOT, we will not discuss it, and rather focus on the proof of unit propagation. To prove unit propagation, we have to convince the SMT solvers that when we have a unit clause, then we do not have to try both polarities for the unit literal. We will do this by using *lemma functions* to build up a proof context which convinces the SMT

solvers.

The gist of the proof is as follows:

1. If we have a formula which is not satisfiable from any extension of the current assignment, then it does not, by definition, matter which polarity we choose for any of the unset literals.
2. If we have a clause which is unit, and we satisfy the unit literal, then the clause becomes satisfied. If we don't satisfy the unit literal, the clause becomes unsatisfied.
   - Sublemma: A formula cannot be satisfied by any extension of an assignment which falsified it.
3. Therefore: either our choice did not matter, or it was the only choice which maintained eventual satisfiability of the formula with regards to the assignment.

A thing to note before we present the proof, is that the current proof of Robinson, as it is in the repository, does not require any lemmas. This is in part due to improvements to the code since the initial version, and in part due to improvements to CREUSOT. Having lemmas which are needed when developing a proof, but then later can be removed, has been a common occurrence. Initially, when figuring out the proof, one has many lemmas, proof assertions, and long contracts. Once the initial proof is done, a better understanding of what is needed is established, and both the code and the proof can be simplified. This simplification usually leads to proofs passing much faster, and to some lemmas being unneeded.

The presented proof is thus intended to provide intuitions into how an initial proof is created, rather than to present the most up to date version of Robinson. We are also of the belief that most proofs in most tools will need similar lemmas, at least initially.

4.3 Proof of Robinson

Robinson is a significant step up from Friday in terms of performance, total size, and in how much effort it took to develop, yet it is not conceptually significantly more complex than Friday. We are still search-based, so we can keep our definitions of satisfiability and unsatisfiability, and have to prove that unit propagation is a valid optimization. A note about the implementation is that we are using a vector of `u8`s to manually encode `Option<bool>`, as this seemed to be either marginally faster or the same speed, and it allows us to easily implement a technique which is called *phase saving* for CreuSAT. We have abstracted away false (`0`) behind `neg()`, true (`1`) behind `pos()`, and unset (everything greater than 1) behind `unset()`. In the code, we use the type alias `AssignedState`, but in the lemmas below we use `u8` to save on space.

Something else which should be noted is that some of our lemmas have

the postfix _inner. This is our naming convention for functions which take models of as parameters. As an example, take the definition of unsatisfiability for the formula:

```
impl Formula {
    #[predicate]
    pub fn eventually_sat(self, a: Assignments) -> bool {
        pearlite! { self.eventually_sat_inner(@a) }
    }

    #[predicate]
    pub fn eventually_sat_inner(self, a: Seq<u8>) -> bool {
        pearlite! {
            exists<a2 : Seq<u8>> a2.len() == @self.num_vars
            && compatible_inner(a, a2)
            && self.sat_inner(a2)
        }
    }
}
```

eventually_sat() is just a thin wrapper that calls eventually_sat_inner() with the model of the provided assignment. We have to have an _inner function, as we will at some points only have an Seq<u8>, and not an Assignment, and there is no way to go from the model of a construct back to the original construct. As we will see below, we have a decent amount of functions which work on the model of Assignments and which want to access various predicates on Assignments. This leads to the _inner pattern becoming quite pervasive, and most predicates which are not _inner are wrappers of an _inner function. We will therefore use these interchangeably in discussion.

With that out out the way, let us continue with the proof. In the actual development, the process went back and forth between writing code and developing lemmas. We started with an implementation of unit propagation, which we then massaged into a version which passed with unproven lemmas. As the lemmas are proved, and their preconditions are added, further insight is gained, and the program code changes accordingly.

We shortcut this process, and present the final version of the code, before the lemmas were removed. We then step through it and explain what is happening. The entirety of unit_prop_once is as follows:

```
#[maintains((mut self).invariant(*f))]
#[requires(f.invariant())]
#[requires(0 <= @i && @i < (@f.clauses).len())]
#[ensures((*self).compatible(^self))]
#[ensures(f.eventually_sat_complete(*self)
       == f.eventually_sat_complete(^self))]
#[ensures(match result {
```

```
    ClauseState::Sat => (@f.clauses)[@i].sat(^self) && @self == @^self,
    ClauseState::Unsat => (@f.clauses)[@i].unsat(^self) && @self == @^self,
    ClauseState::Unit => (@f.clauses)[@i].unit(*self) && !self.complete(),
    ClauseState::Unknown => @self == @^self && !(^self).complete(),
})]
#[ensures((self).complete() ==> *self == ^self
    && ((result == ClauseState::Unsat) || (result == ClauseState::Sat)))]
pub fn unit_prop_once(&mut self, i: usize, f: &Formula) -> ClauseState {
    let clause = &f.clauses[i];
    let _old_a = ghost!(self);
    match clause.check_if_unit(self, f) {
        ClauseState::Unit => {
            let lit = clause.get_unit(self, f);

            proof_assert!(lemma_unit_wrong_polarity_unsat_formula(
                *clause, *f, @self, lit.index_logic(),
                bool_to_assignedstate(lit.polarity));
            true);
            proof_assert!(lemma_unit_forces(
                *f, @self, lit.index_logic(),
                bool_to_assignedstate(lit.polariy));
            true);

            if lit.polarity {
                self.0[lit.index()] = 1;
            } else {
                self.0[lit.index()] = 0;
            }

            proof_assert!(lemma_extension_sat_base_sat(
                *f, @_old_a.inner(), lit.index_logic(),
                bool_to_assignedstate(lit.polarity));
            true);
            proof_assert!(lemma_extensions_unsat_base_unsat(
                @_old_a.inner(), lit.index_logic(), *f);
            true);

            proof_assert!(^self == ^_old_a.inner());
            return ClauseState::Unit;
        }
        o => return o,
    }
}
```

The code may initially seem daunting, but there is not all that much going on. If the call to `clause.check_if_unit` returns that the clause is not unit, we simply return this valuation. In the case that the clause on the given index is a unit clause, then we get the unit literal, satisfy it and return

`ClauseState::Unit`. Interspersed are four calls to lemmas, which we will now explain.

The first lemma which gets invoked, is `lemma_unit_wrong_polarity_unsat_formula`. It is defined as follows:

```
#[logic]
#[requires(f.invariant())]
#[requires(@f.num_vars == a.len())]
#[requires(0 <= ix && ix < a.len() && unset(a[ix]))]
#[requires(!unset(v))]
#[requires(c.unit_inner(a))]
#[requires(c.in_formula(f))]
#[requires(c.invariant(a.len()))]
#[requires(exists<j: Int> 0 <= j && j < (@c).len()
    && (@c)[j].index_logic() == ix
    && bool_to_assignedstate(((@c)[j].polarity)) == v)]
#[requires(forall<j: Int> 0 <= j && j < (@c).len() &&
    !((@c)[j].index_logic() == ix) ==> (@c)[j].unsat_inner(a))]
#[ensures(!f.eventually_sat_complete_inner(a.set(ix, flip_v(v))))]
#[ensures(f.unsat_inner(a.set(ix, flip_v(v))))]
pub fn lemma_unit_wrong_polarity_unsat_formula(
        c: Clause, f: Formula, a: Seq<u8>, ix: Int, v: u8) {
    lemma_correct_polarity_makes_clause_sat(c, a, ix, v);
    lemma_incorrect_polarity_makes_clause_unsat(c, a, ix, v);
    lemma_not_sat_clause_implies_unsat_formula(f, c, a.set(ix, flip_v(v)));
    lemma_unsat_implies_not_eventually_sat(f, a.set(ix, flip_v(v)));
}
```

`lemma_unit_wrong_polarity_unsat_formula` essentially states that if we have a unit clause `c`, and we have the index of the unset literal, `ix`, and that index has the polarity `v`, then we are guaranteed to get a formula which is both `unsat`, and not `eventually_sat_complete`. To prove this, it builds up an internal proof context consisting of four lemmas.

We will now present the four lemmas in the order they appear. A thing to note is that that they ensure a bit more than strictly is needed, as they were used as lemmas elsewhere as well.

Building up a proof context with lemmas is conceptually fairly straight forward. Before invoking the lemma, we have a collection of predicates which hold. If we manage to satisfy all the preconditions of the lemma when invoking it, we "gain" its postconditions as well. Take for instance `lemma_correct_polarity_makes_clause_sat`:

```
#[logic]
#[requires(0 <= ix && ix < a.len())]
#[requires(exists<j: Int> 0 <= j && j < (@c).len()
        && @(@c)[j].idx == ix
        && bool_to_assignedstate((@c)[j].polarity) == v)]
#[ensures(c.sat_inner(a.set(ix, v)))]
```

```
pub fn lemma_correct_polarity_makes_clause_sat(
        c: Clause, a: Seq<u8>, ix: Int, v: u8) {}
```

As long as the two preconditions hold, then `c.sat_inner(a.set(ix, v))` can be added to the proof context as well. As we can see above, the two preconditions are contained in the preconditions of `lemma_unit_wrong_polarity_unsat_formula`, and thus we now have the previous proof context with `c.sat_inner(a.set(ix, v))` added.

The next lemma is `lemma_incorrect_polarity_makes_clause_unsat`:

```
#[logic]
#[requires(c.invariant(a.len()))]
#[requires(!unset(v))]
#[requires(0 <= ix && ix < a.len() && unset(a[ix]))]
#[requires(exists<j: Int> 0 <= j && j < (@c).len()
    && @(@c)[j].idx == ix && (@c)[j].sat_inner(a))]
#[requires(forall<j: Int> 0 <= j && j < (@c).len()
    && !(@(@c)[j].idx == ix) ==> (@c)[j].unsat_inner(a))]
#[ensures(forall<j: Int> 0 <= j && j < (@c).len() ==>
    !unset((a.set(ix, v))[@(@c)[j].idx]))]
#[ensures(!(unset(a.set(ix, flip_v(v))[ix])))]
#[ensures(c.unsat_inner(a.set(ix, flip_v(v))))]
#[ensures(!c.sat_inner(a.set(ix, flip_v(v))))]
pub fn lemma_incorrect_polarity_makes_clause_unsat(
    c: Clause, a: Seq<u8>, ix: Int, v: u8) {}
```

We again satisfy all its preconditions, and thus "gain" the four postconditions.

Continuing, we have `lemma_not_sat_clause_implies_unsat_formula`:

```
#[logic]
#[requires(c.unsat_inner(a))]
#[requires(c.in_formula(f))]
#[ensures(f.unsat_inner(a))]
pub fn lemma_not_sat_clause_implies_unsat_formula(
    f: Formula, c: Clause, a: Seq<u8>) {}
```

The observant reader will notice that we have `c.unsat_inner(a.set(ix, flip_v(v))` from the previous lemma, and, as we invoked the lemma with `a.set(ix, flip_v(v))`, we now gain `f.unsat_inner(a))`.

Finally, we have the final lemma, which takes us from `f.unsat_inner(a)` to `!f.eventually_sat_complete_inner(a)`.

```
#[logic]
#[requires(f.invariant())]
#[requires(assignments_invariant(a, f))]
#[requires(f.unsat_inner(a))]
#[ensures(!f.eventually_sat_complete_inner(a))]
pub fn lemma_unsat_implies_not_eventually_sat(f: Formula, a: Seq<u8>) {}
```

We thus have a complete chain of lemmas from the preconditions of `lemma_unit_wrong_polarity_unsat_formula` to the two postconditions: `!f.eventually_sat_complete_inner(a.set(ix, flip_v(v)))` and `f.unsat_inner(a.set(ix, flip_v(v)))`.

The second lemma which gets invoked in the body of `unit_prop_once`, is `lemma_unit_forces`:

```
#[logic]
#[requires(f.invariant())]
#[requires(@f.num_vars == a.len())]
#[requires(0 <= ix && ix < a.len() && unset(a[ix]))]
#[requires(!unset(v))]
#[requires(f.eventually_sat_complete_inner(a))]
#[requires(!f.eventually_sat_complete_inner(a.set(ix, flip_v(v))))]
#[ensures(f.eventually_sat_complete_inner(a.set(ix, v)))]
pub fn lemma_unit_forces(
    c: Clause, f: Formula, a: Seq<u8>, ix: Int, v: u8) {
    lemma_unsat_implies_not_eventually_sat(f, a);
}
```

The two preconditions which are important, are
`f.eventually_sat_complete_inner(a)` and
`!f.eventually_sat_complete_inner(a.set(ix, flip_v(v)))`. The latter of these is, as we just saw, ensured by `lemma_unit_wrong_polarity_unsat_formula`. `f.eventually_sat_complete_inner(a)` is only satisfied for formulas which are eventually sat complete. `lemma_unit_forces` thus ensures that if we have a formula which is eventually sat complete, then setting `v` on `ix`, corresponding to satisfying the unit clause, will maintain this property.

Continuing through the body of `unit_prop_once`, we do the assignment, and thus satisfy the unit clause, before invoking the final two lemmas. We used `lemma_unit_wrong_polarity_unsat_formula` and `lemma_unit_forces` to prove

```
f.eventually_sat_complete(*self) ==> f.eventually_sat_complete(^self)
```

and we are now invoking `lemma_extension_sat_base_sat` and `lemma_extensions_unsat_base_unsat` to prove

```
f.eventually_sat_complete(^self) ==> f.eventually_sat_complete(*self)
```

We define `lemma_extension_sat_base_sat` as follows

```
#[logic]
#[requires(0 <= ix && ix < a.len() && unset(a[ix]))]
#[requires(f.eventually_sat_complete_inner(a.set(ix, v)))]
#[ensures(f.eventually_sat_complete_inner(a))]
pub fn lemma_extension_sat_base_sat(
        f: Formula, a: Seq<u8>, ix: Int, v: u8) {}
```

and `lemma_extensions_unsat_base_unsat` as follows:

```
#[logic]
#[requires(0 <= ix && ix < a.len() && unset(a[ix]))]
#[requires(!f.eventually_sat_complete_inner(a.set(ix, neg())))]
#[requires(!f.eventually_sat_complete_inner(a.set(ix, pos())))]
#[ensures(!f.eventually_sat_complete_inner(a))]
pub fn lemma_extensions_unsat_base_unsat(
    a: Seq<u8>, ix: Int, f: Formula) {}
```

`lemma_extensions_unsat_base_unsat` might initially seem a bit strange, as we are requiring both

```
!f.eventually_sat_complete_inner(a.set(ix, neg()))
```

and

```
!f.eventually_sat_complete_inner(a.set(ix, pos()))
```

The thing to remember here is that we have

```
!f.eventually_sat_complete_inner(a.set(ix, flip_v(v)))
```

from `lemma_unit_wrong_polarity_unsat_formula`. As the current assignment is equal to `a.set(ix, v)`, we will be able to ensure both preconditions, given that the formula was `!f.eventually_sat_complete_inner(a)` before.

With the invocation of these final two lemmas completed, we do the pro forma proof assertion of `^self == ^_old_a.inner()` to state that the final version of the assignments is equal to the ghost version of the assignments, and thus conclude the proof of unit propagation.

The top level specification

We end this chapter with a presentation of the top level specification of Robinson. It is as follows:

```
#[ensures(match result {
    SatResult::Sat(_assn) => { formula.eventually_sat_no_ass() },
    SatResult::Unsat      => { !formula.eventually_sat_complete_no_ass() },
    _                     => true ,
})]
```

where `eventually_sat_no_ass` is defined as follows:

```
impl Formula {
    #[predicate]
    pub fn eventually_sat_no_ass(self) -> bool {
        pearlite! { exists<a2 : Seq<AssignedState>> self.sat_inner(a2) }
    }
}
```

and `eventually_sat_complete_no_ass` as follows:

```
impl Formula {
    #[predicate]
    pub fn eventually_sat_complete_no_ass(self) -> bool {
        pearlite! {
            exists<a2 : Seq<AssignedState>> a2.len() == @self.num_vars
            && complete_inner(a2)
            && self.sat_inner(a2)
        }
    }
}
```

The sat_inner predicates are defined as follows:

```
impl Formula {
    #[predicate]
    pub fn sat_inner(self, a: Seq<AssignedState>) -> bool {
        pearlite! {
            forall<i: Int> 0 <= i && i < (@self.clauses).len() ==>
                (@self.clauses)[i].sat_inner(a)
        }
    }
}

impl Clause {
    #[predicate]
    pub fn sat_inner(self, a: Seq<AssignedState>) -> bool {
        pearlite! {
            exists<i: Int> 0 <= i && i < (@self).len() &&
                (@self)[i].sat_inner(a)
        }
    }
}

impl Lit {
    #[predicate]
    pub fn sat_inner(self, a: Seq<AssignedState>) -> bool {
        pearlite! {
            match self.polarity {
                true  =>  (@a[@self.idx] == 1),
                false =>  (@a[@self.idx] == 0),
            }
        }
    }
}
```

complete_inner is defined as follows:

```
#[predicate]
pub fn complete_inner(a: Seq<AssignedState>) -> bool {
    pearlite! {
```

```
        forall<i: Int> 0 <= i && i < a.len() ==> !unset(a[i])
    }
}
```

Where `unset` is defined as:

```
impl Lit {
    #[predicate]
    pub fn unset(self, a: Assignments) -> bool {
        pearlite! { self.unset_inner(@a) }
    }

    #[predicate]
    pub fn unset_inner(self, a: Seq<AssignedState>) -> bool {
        pearlite! { @(a)[@self.idx] >= 2 }
    }
}
```

A thing to note about the top level specification is that it does not ensure completeness, as marked by `_ => true` . This arm of the match statement is only hit in the case that establishing the formula invariant fails. This will happen if the `f.num_vars` is set to be less than the actual number of variables which occur in the input formula, which may happen when the CNF file is incorrectly formatted, or if there is a bug in the parser. The solver is thus complete with regards to input formulas which have `f.num_vars` set to be higher than or equal to the actual number of variables. Making it complete with regards to all formulas would require the trivial change of setting `f.num_vars` in the case that it is set too low. We have chosen not to do this, as we prefer for the solver to take an immutable borrow to the formula.

Part III

Verification of a CDCL solver

Chapter 5

Verification of the CDCL algorithm

In this chapter we present the main contribution of the thesis: the implementation and verification of CreuSAT, a conflict-driven clause learning SAT solver. We implement and prove the safety and correctness of the following features:

- Clause analysis and learning of clauses
- Unit propagation
- Two watched literals with blocking literals
- The variable move-to-front literal selection heuristic
- Phase saving
- Backtracking to asserting level
- Search restart based on exponential moving averages
- Deletion of learned clauses

We implement the solver using efficient data structures, using built in Rust-vectors and machine integers. As we want to be able to represent literals with 64 bits, we require that the input formula contains less than 2^{63} - 1 variables. This limit is mostly artificial: if your formula actually has 2^{63} - 1 distinct variables, then you would need exabytes of memory to store it, in excess of the amount of memory currently available in any computer on Earth.

Whereas we for Friday and Robinson included a substantial amount of code, this chapter will be quite sparse in code. We choose instead to focus on the overarching ideas of proofs of the various components. We thus believe that this chapter is best read in conjunction with the source code, which is available at github.com/sarsko/CreuSAT.

We choose this way to present our material in part due to space, in part due to the fact that the concrete proofs contains much tedium, and in part due to a shift in focus. The previous parts were intended to help the reader in understanding how program proofs in CREUSOT are conducted, and give an introduction to SAT solving. This part is intended to help the reader understand CreuSAT and the ideas behind a program proof of the CDCL algorithm and some of its optimizations.

Chapter overview

This chapter starts with an introduction to the CDCL algorithm, where we explain the general concepts with regards to our concrete implementation. Those who are familiar with CDCL-based SAT solving may notice that we choose to forego the common explanation of clause analysis as doing a cut on the implication graph, and only mention it in passing. The reason for this is largely pedagogical: we have found that merely having the cut-based understanding is not sufficient to understand the proof, whereas merely having the resolution-based understanding is. This ties in with the fact that the implication graph and the 1-UIP cut are largely invisible both in implementation and in proof, whereas the trail and resolution are center stage.

Following the introduction to CDCL, we will look at the main ideas of the proof, giving the mental framework which we have built the proof around. We will then explain how we prove CreuSAT. We split the explanation into two parts, Section 5.3 – Proof of CreuSAT, which covers what the core of the algorithm, and Section 5.4 – Optimizations, which covers the various optimizations we implement. We explain the core parts in much more detail, as we are of the belief that a solid understanding of the core of the algorithm is paramount in understanding the proof. We have found that, with the exception of two watched literals, most of the optimizations are not all that hard to implement and prove. In the off-chance that they are, this usually harks back to a poor understanding of the core of the algorithm, or the suboptimality of a previous design decision, rather than the intrinsic complexity of the optimization.

5.1 The CDCL algorithm

5.1.1 Overview

The conflict-driven clause learning SAT solver was discovered by Marques-Silva and Sakallah in 1996 [37], and has remained as the state-of-the-art since. It is often viewed as an extension of DPLL, but, as we will discuss in the coming chapter, it is fundamentally different from DPLL. DPLL is *search based* — the notion of unsatisfiability is tied to the complete exploration of the search space. This is not the case for CDCL, where the notion of unsatisfiability is tied to learning the empty clause, which, by definition, there exists no assignment which satisfies.

In short, a CDCL solver needs the following "mechanisms":

- Unit propagation mechanism
- Clause analysis and clause learning mechanism
- Decision mechanism
- Backtracking mechanism

To implement these mechanisms, we need our data structures from before, in addition to a new data structure, which is usually called the *trail*. It is implemented as a a stack which keeps track of the assignments we have made, when we did them, and why we did them. The trail enables us to backtrack efficiently, and it is also a key part of clause learning. The CDCL algorithm in Rust-like pseudocode:

```
fn solve(formula: &mut Formula, t: &mut Trail
) -> Result<SatResult, SatError> {
    loop {
        loop {
            let res = unit_propagate(f, t);
            if res != confl { break; }
            let clause = clause_analysis(res, f, t)?;
            formula.push(clause);
            t.backtrack();
        }
        t.make_decision()?;
    }
}
```

Note that we are (ab)using the `?` notation to return the result of the solver. We have also abstracted away the partial assignment, and have it as a part of the trail. In solvers with a trail, the partial assignment is simply the realization of the trail on an empty assignment, and can thus be viewed as an optimization, rather than a necessity.

`unit_propagate` is for the most part the same as the one for the DPLL algorithm, but may now return a reference to a conflict clause — a clause in the current formula which has become unsatisfied by unit propagation. We then do clause analysis on this clause, which yields a clause which is implied by the current clause database. In the case that this clause is empty, the formula implies the empty clause, and we return that the formula is unsatisfiable. If the clause is of non-zero length, we add it to the clause database, and backtrack at least enough for the clause to not be conflicting. We will later look at how we design the learned clause.

In the case that the call to unit propagation did not yield a conflict, then our formula does not evaluate to false under the current assignment. We then try to make a decision. If there are no more decisions to be made, then we have a complete assignment under which the formula does not evaluate to false, or, in other words: we have a satisfying assignment. If it is possible to

make a decision, we make a decision, and enter the unit propagation loop. We are free to decide on any of the unassigned variables, and we will later look at heuristics for making this choice.

The next part of the algorithm we will look at is how we do the clause analysis mechanism. Clause analysis is done by iteratively doing a process which is called *resolution* on clauses from the formula. To efficiently find the clauses to do resolution on, we use the trail. We will therefore have to explain the resolution rule, and also get a better understanding of the trail, before we are able to to look at the conflict analysis algorithm in detail.

5.1.2 Interlude: resolution and the Davis Putnam procedure

One of most important concepts of the modern CDCL SAT solver is the *resolution rule*. It forms the foundation of clause learning, which consists of a series of resolution steps, it can be done as a part of the inprocessing and preprocessing of the clause database, and it is the technique which both local clause minimization and recursive clause minimization are based on.

The resolution rule states that given two clauses, C and C', such that there exists one, and only one literal $l \in C$ and $\bar{l} \in C'$, then we may create another clause C'' such that for all $l', l' \in C'', l' \in C \vee l' \in C'$ and $l' \neq |l|$. We say that we *resolve* C and C' to create the *resolvent* C''. For example, the clause $(a \vee b)$ can be resolved with $(\bar{b} \vee c)$ to create the clause $(a \vee c)$

Resolution can be shown schematically as:

$$\frac{C \cup \{l\} \quad C' \cup \{\bar{l}\}}{C \cup C'}$$

Resolution can be traced back to the Davis-Putnam (DP) algorithm of 1960 [16]. We will later explain CDCL as an extension of DP, similar to how we explained Robinson as an extension of Friday.

DP is very similar to the DPLL algorithm, and contains both unit propagation and pure literal elimination. In fact, DPLL was introduced to solve the main problem of DP: memory. Whereas DPLL requires a linear amount of memory, DP is in the worst case exponential. They are almost identical, but instead of using search to progress the solver, as is the case for DPLL, resolution is used instead, by doing what is called the Davis-Putnam *procedure*.

We illustrate the DP procedure with an example run on a satisfiable formula:

$$\{ (a \vee b \vee c) \wedge (b \vee \neg c) \wedge (\neg b \vee d) \}$$

If we try to resolve on a, we get the same formula. If we resolve on b, we get the following formula:

$$\{ (a \vee c \vee d) \wedge (\neg c \vee d) \}$$

We then resolve on c, and get:

$$\{ (a \vee d) \}$$

At this point we have no more opportunities to do resolution, and our formula is therefore satisfiable.

An example run on a formula which is unsatisfiable:

$$\{ (a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg c) \}$$

If we resolve on a, we get:

$$\{ (b \vee c) \wedge (b \vee \neg c) \wedge (\neg b \vee c) \wedge (\neg b \vee \neg c) \}$$

Further, resolving on b, we get:

$$\{ (c) \wedge (\neg c) \wedge (c \vee \neg c)\}$$

After removing the trivially satisfied clause $(c \vee \neg c)$, and resolving on c, we get:

$$\{ () \}$$

As we have resolved the empty clause, the formula is unsatisfiable.

5.1.3 Introduction to the trail

The second concept we need to look at to understand the clause analysis algorithm is the trail data structure. The trail is arguably the most important data structure of a modern SAT solver, and a majority of the time spent proving CreuSAT has been spent on predicates and functions which are related to the trail. We will now give an introduction to the trail and how it enables clause learning, and we will in Subsection 5.3.1 - Furthering our understanding of the trail look at how we realize this in CREUSOT.

As mentioned, the trail is simply a stack consisting of the literals which have been assigned, the reason for their assignment, and the time for their assignment. Our encoding for the trail is derived from the SAT solver called Starlit, which is authored by Jannis Harder. We encode the trail as a `Vec<Step>`, where a `Step` is defined as follows:

```
pub struct Step {
    pub lit: Lit,
    pub decision_level: usize,
    pub reason: Reason,
}
```

The `lit` is the literal which was assigned, the `decision_level` is the *decision level* where the assignment was made, and the `reason` is the reason for the assignment. We define `Reason` as follows:

```
pub enum Reason {
    Decision,
    Unit(usize),
    Long(usize),
}
```

`Decision` means that the literal was a decision — the solver was in a state where `unit_propagate` did not return a conflict, and the decision making mechanism produced this literal. The `Long(usize)` reason means that the the given literal was found to be the unit literal of the clause which is found by indexing the clause database on the given `usize`. The assignment was thus implied by a "long" clause, which in our case means any clause with length greater than 1[1]. We call this clause the literal's *antecedent*. `Unit(usize)` is the same as `Long`, but for unit clauses[2].

The decision level of an assignment is simply a count of how many `Steps` that occur "below" the given `Step` on the stack which have `Decision` as their `Reason`. `Steps` which have the same decision level have the same decision as their "root cause".

The key thing to understand about the trail is that it, in addition to keeping track of decisions, and making efficient backtracking possible, it also gives us efficient access to clauses to do resolution on. If we have a clause which is unsatisfied, then, for all its literals which do not have `Decision` as their `Reason`, we are able to use the trail to find a clause which is guaranteed to be resolvable with the clause. The antecedent of a literal will always be a clause which at an earlier point was unit, and then had its unit literal satisfied. We will call this property *post unit* later, meaning that the clause was unit, and is now satisfied, and the fact that the trail efficiently yields clauses which are post unit with regards to some literal is a centerpoint of the proof.

5.1.4 The conflict analysis algorithm

With resolution and the trail explained, we can explain our implementation of the conflict analysis algorithm. For the conflict analysis algorithm, we will be following the algorithm of Chaff [42], as presented in [58], which modern solvers derive their implementation from. We translate it to Rust-like pseudocode:

```
fn conflict_analysis(self) -> Result<(usize, ())> {
    check_ground()?;
    let mut cl = self.find_conflicting_clause();
    loop {
        let lit = choose_literal(cl);
        let var = variable_of_literal(lit);
        let ante = self.get_antecedent(var);
        cl = resolve(cl, ante, var);
        if stop_criterion_met(cl) {
            break;
        }
```

[1] It is called `Long` because we in the future plan to handle binary clauses separately.

[2] Having `Unit` carry a reference to a clause in the formula is due to a weakness of the current proof, where we need to actually instantiate the clause and store it in the clause database.

```
    }
    self.add_clause_to_database(cl);
    Ok(clause_asserting_level(cl))
}
```

On entry, we check whether we have a ground conflict, which means that we have reached a conflict on decision level 0, corresponding to a formula which is conflicting without having made any decisions. If that is the case, then our formula is UNSAT, and we return an error. A thing to note here is that in our implementation, we do a complete derivation of the empty clause. This is simply because we have not prioritized proving that having made no decisions and reaching a conflict means that the formula is UNSAT. Instead we choose to execute a version of conflict analysis without the stopping condition, and produce the empty clause as a witness.

In the case that there was no ground conflict, then we get the conflicting clause from the `Formula`, and start doing iterated resolution. For each loop iteration, we find the conflicting literal, get the reason for its assignment, and then update the clause to be the resolvent of the clause and the antecedent of the conflict literal.

As `cl` is a conflict clause, it starts being unsatisfied. It is then resolved with the antecedent of the conflicting literal. As the antecedent was a unit clause with its only satisfied literal being the one which was just resolved out, the resolvent is unsatisfied as well. At the next iteration, the process repeats itself: we resolve out the only literal which is satisfied in the antecedent of a literal in `cl`, and maintain a clause which is unsatisfied. All of this is due to how we designed the trail: it allows us to efficiently find clauses to do resolution on.

At each point during this process we have a clause which is implied by the current formula. We therefore have do decide on when to stop doing resolution, and learn the clause. This is handled by the `stop_criterion_met` function.

For the stopping criterion, we have many choices. As `cl` is at all times implied by the clause database, we could at any point learn the current clause, potentially learning multiple clauses if we so desire. We decide to learn one clause for each conflict, and to learn the first clause which has exactly one literal assigned at the current decision level. Such a clause has the desirable property that it becomes unit if we backtrack one level, and is what is called an *asserting clause*, giving rise to what is called *assertion based backtracking* [57]. In practice we backtrack "as far as possible", which corresponds to backtracking to the level before the clause would stop being unit.

This scheme of clause learning corresponds to learning a clause which contains the first *unique implication point* (UIP), and is called the *1UIP scheme*. We will not go into all the details of UIPs and various UIP-learning schemes, as this is not needed to understand the algorithm.

5.2 The main ideas of the proof

With the core of the algorithm explained, we can start discussing the main ideas of the proof. Before doing that, we will briefly explain why we have omitted explaining the CDCL algorithm as the process of creating an *implication graph*, and then doing a *cut* of said implication graph. This subsection is intended for those who are already familiar with the explanation of clause learning as doing a UIP-cut of the implication graph, and who are curious as to why we have omitted explaining it.

If you are not familiar with this view, then you may safely skip this subsection, as we will not explain it in detail. It should be noted that the views are equivalent, and there may very well be a CREUSOT proof for which the cut of the implication graph view is suited as an explanation.

5.2.1 The suboptimality of the cut of the implication graph

We have not found this way of explaining CDCL to be well suited for understanding the program proof. The reason for this is that the implication graph and the cuts do not show themselves in the proof, nor in the code. In addition: when explaining the 1-UIP cut, the reason for its optimality — that it yields an asserting clause — is obscured. The implication graph puts emphasis on the fact that one assignment led to another assignment, an aspect which is completely vacant in our proof. In the proof, the relevant part is not that an assignment implied another assignment, but that we can, given an unsatisfied clause and a literal in that clause, use the trail to find a clause to do resolution on. How that clause ended up in the trail is not important, what matters is that all the reasons of the trail carry that property.

Out of this comes our view of the trail, and thus the solver as a whole. Instead of viewing the trail as an implication graph, we view it as what it is: a stack, where there exists a series of invariants on the elements of this stack.

5.2.2 CDCL as an extension of DP

The CDCL algorithm, though often viewed as an extension of the DPLL algorithm, is fundamentally different in a few key ways. The first key difference is that the DPLL algorithm is fundamentally *search based*, whereas the CDCL algorithm is fundamentally *resolution based*. In DPLL unsatisfiability is tied to the complete exploration of the search space, whereas in CDCL it is tied to a valid series of resolution steps, ending in the empty clause. Both DPLL and CDCL have the same notion of satisfiability: a complete assignment which is not unsatisfiable.

In Chapter 4 – Verification of the DPLL algorithm, we presented DPLL as an optimization on the search algorithm of Chapter 3 – Verification of a minimal solver, where the net effect of unit propagation is to reduce the search space. CDCL, however, we view as an improvement on the DP

algorithm which we looked at in Subsection 5.1.2 – Interlude: resolution and the Davis Putnam procedure. DP does all possible resolutions, whereas in CDCL, unit propagation is used to efficiently identify the clauses to do resolution on. Whereas the DP algorithm can be viewed as doing *eager* resolution, CDCL can be viewed as doing *lazy* resolution, only applying the resolution algorithm on the clauses which are antecedents of the literals of the conflict clause.

In DP, satisfiability is tied to reaching a formula which does not contain the empty clause, and where there are no further possibilities for doing resolution. For CreuSAT, we view satisfiability as "happenstance" — we "happened" to reach a complete assignment which did not lead to a conflict, whereas the notion of unsatisfiability is still tied to resolution of the empty clause. In other words: we combine the satisfiability condition of the DPLL algorithm and the unsatisfiability condition of the DP algorithm. Both of these are based on giving a witness, which makes the postconditions for the solver both easy to specify and easy to prove.

The fact that we don't have to maintain and prove a complete search makes it possible to achieve a much more *decoupled* solver, in some ways making the proof of CDCL arguably easier to implement, at least in terms of "peak hardness". Whereas we for the proof of the DPLL algorithm had to prove that we did in fact traverse the whole search space, and therefore had to do the whole proof of unit propagation being a "forced" choice, this obligation is completely lifted for the CDCL algorithm. For the CDCL algorithm, our formula is SAT if we manage to find a complete assignment which does not lead to a conflict, and it is UNSAT if we in the attempt to do so manage to do a series of resolution steps which end in the empty clause. As we have maintained equisatisfiability with the original formula while learning new clauses, the formula is thus equisatisfiable with the empty clause, which by definition is unsatisfiable.

We do not have to prove that we are exploring the entirety of the search space, as the semantics of the solver are not tied to this notion. As long as the satisfiability of the input formula is maintained by the operations done by the solver, with the learning of new clauses being the critical part to get right, the remainder of the proof is for the most part proof of safety properties. This is illustrated by the fact the entirety of the proof of correctness for unit propagation has been removed, and that implementing decision heuristics has been solved by solely proving safety, and then injecting a runtime-check of completeness, which is only run for satisfiable formulas.

That being said, proof of correctness for clause learning is not exactly trivial, and CDCL solvers open up for a lot more optimizations. CDCL solvers also have both more code, and code which is less amenable to verification than DPLL solvers. The algorithm does in general consist of manipulation of pointers and vectors, both of which are notoriously hard to reason about. Though much of the difficulties are lifted by the Rust type system and its modelling in CREUSOT, even what might be considered

"simple" properties, for instance that all permutations of a clause are equivalent, can be cumbersome to prove.

We will not go into all of these challenges, though we will look at the trail in some detail. The correctness of the solver is reliant on the correctness of the trail, and thus specifying the correct invariants and maintaining these throughout the run of the solver is paramount. Maintaining the invariants is what enables us to efficiently use the trail to find clauses to do resolution on, in addition to enabling efficient backtracking.

5.3 Proof of CreuSAT

We will in this section explain what we consider to be the most important parts of proving CreuSAT. Compared to the earlier explanations, we maintain a rather high level description, with the goal of transferring the intuitions of the proofs, though we will present some code snippets throughout.

5.3.1 Furthering our understanding of the trail

As we have already touched on, the trail is arguably the most important part of CreuSAT, and understanding it is key to understanding the proof.

The final version of the trail consists of the previously discussed `trail`, in addition to a few related data structures, which we use for efficiency reasons.

```
pub struct Trail {
    pub assignments: Assignments,
    pub lit_to_level: Vec<usize>, // usize::MAX if unassigned
    pub trail: Vec<Step>,
    pub curr_i: usize,
    pub decisions: Vec<usize>,
}
```

`assignments` is the same `assignments` as known from Robinson: a vector of `u8`s, where `0` corresponds to false, `1` corresponds to true, and all other values correspond to unset. There exists a bijection from the `trail` to the `assignments`. `assignments` corresponds to a realization of all the steps of the `trail` on an empty `Assignments`, and all the entries of the `assignments` which are assigned have a corresponding `Step` in the `trail`. `assignments` is thus strictly speaking an optimization, but we know of no solver which does not keep track of the partial assignment.

`lit_to_level` gives the decision level where a literal was assigned, and `curr_i` is used to keep track of how much of the trail has been processed. During unit propagation, we will enqueue `Steps` to the trail as further unit clauses are identified. Whenever unit propagation returns without a conflict, `curr_i` is equal to the length of the `trail`. `decisions` is used to

keep track of the amount `Steps` at each decision level, and is used to do backtrack more efficiently.

We define the invariant for the `Trail` as follows:

```
impl Trail {
    #[predicate]
    pub fn invariant(self, f: Formula) -> bool {
        pearlite! {
            self.assignments.invariant(f)
            && trail_invariant(@self.trail, f)
            && lit_to_level_invariant(@self.lit_to_level, f)
            && decisions_invariant(@self.decisions, @self.trail)
            && self.trail_entries_are_assigned()
            && self.decisions_are_sorted()
            && self.lit_is_unique()
            && self.lit_not_in_less(f)
            && unit_are_sat(@self.trail, f, self.assignments)
            && long_are_post_unit_inner(@self.trail, f, @self.assignments)
        }}}
```

We will not discuss the invariants which carry the `_invariant` postfix, as they are safety invariants. `trail_entries_are_assigned()` maintains the first part of the aforementioned bijection. We used to maintain an invariant for the second part as well, but found that we did not use it[3], and thus removed it. `decisions_are_sorted()` states that the decision levels of the trail form a monotonically increasing sequence.

`lit_is_unique` states that there does not exist multiple steps which contain a literal with the same index as another literal on the trail. `lit_not_in_less` states that the literal of a given step does not occur in any clause which is referenced at an earlier entry. We carry this property in the trail, but it could most likely be removed, and then proven every time we need it. This is because if the literal were to occur in any previous clause, then either the literal has to be the literal of the clause which is satisfied, or it has to be one of the unsatisfied literals. In the first case, we would have a duplicate entry on the trail, breaking the `lit_is_unique` invariant. In the second case, we would have a literal which is both unsatisfied and satisfied at the same time: a contradiction.

`unit_are_sat` exists solely to help the SMT solvers[4], and it should be possible to remove it. It is essentially a rewording of `trail_entries_are_assigned` with regards to the `Reason` of the `Step`.

This leaves only the aptly named `long_are_post_unit` predicate. It states

[3] Enabling the bijection is a requirement for making unit propagation a complete evaluation function. As we will discuss in Subsection 5.4.1 – Two watched literals, we sacrifice completeness. When we reintroduce completeness, we will have to reestablish the bijection.

[4] As we discuss in https://sarsko.github.io/Creusot, the absolutely most important job of a proof writer is to make the proof easy for the SMT solvers.

that entries on the trail which have Long as their reason are what we call post_unit. post_unit means that a clause was unit and then had its unit literal satisfied, or, in other words: that it has one literal which is satisfied, and that the remainder of its literals are unsatisfied. We define post_unit as follows:

```
impl Clause {
    #[predicate]
    pub fn post_unit_inner(self, a: Seq<AssignedState>) -> bool {
        pearlite! {
            exists<i: Int> 0 <= i && i < (@self).len()
                && (@self)[i].sat_inner(a)
                && forall<j: Int> 0 <= j && j < (@self).len()
                && j != i ==> (@self)[j].unsat_inner(a)
        }
    }
}
```

It states that there exists a satisfied literal, and that all other literals of the clause are unsatisfied.

On top of post_unit, we define the clause_post_with_regard_to_lit predicate. Its definition is somewhat convoluted, but it essentially states that the given clause is post unit, and that the given literal is the literal of the clause which is satisfied. It is defined as follows:

```
#[predicate]
pub fn clause_post_with_regard_to_lit(
    c: Clause, a: Assignments, lit: Lit) -> bool {
    pearlite! {
        c.post_unit(a)
        && exists<i: Int> 0 <= i && i < (@c).len()
        && (@c)[i].polarity == lit.polarity
        && @(@c)[i].idx == @lit.idx && (@c)[i].sat(a)
    }
}
```

With clause_post_with_regard_to explained, we can give the definition of long_are_post_unit. It is as follows:

```
#[predicate]
pub fn long_are_post_unit(trail: Trail, f: Formula) -> bool {
    pearlite! {
        forall<j: Int> 0 <= j && j < (@trail.trail).len() ==>
            match (@trail.trail)[j].reason {
                Reason::Long(k) => clause_post_with_regards_to(
                                    (@f.clauses)[@k],
                                    trail.assignments,
                                    @(@trail.trail)[j].lit.idx),
            _                  => true
```

```
        }
    }
}
```

The maintaining of the `long_are_post_unit` is what ensures the correctness of the clause learning mechanism, which is the topic of the next couple of sections.

5.3.2 Facilitating clause learning

It makes little sense to do clause analysis without learning the clause. We therefore start with the clause learning mechanism, and "move backwards" to the clause analysis mechanism. To prove clause learning, we need to introduce a notion of two formulas being equisatisfiable, such that if the formula was eventually satisfiable before, it is eventually satisfiable after the clause addition. Whereas we before were exploring the search space, and therefore had to take into account the current partial assignment(the formula could be an eventually satisfiable formula, but not in any extension of the current partial assignment), a formulas satisfiability is now completely separated from the current partial assignment. Thus, we introduce the notion of equisatisfiability as follows:

```
#[predicate]
pub fn equisat(
    f: (Seq<Clause>, Int), o: (Seq<Clause>, Int)) -> bool {
    pearlite! { eventually_sat_complete_no_ass(f) ==
                eventually_sat_complete_no_ass(o)
    }
}
```

where the `eventually_sat_complete_no_ass` predicate is defined as follows:

```
#[predicate]
pub fn eventually_sat_complete_no_ass(
    f: (Seq<Clause>, Int)) -> bool {
    pearlite! {
        exists<a2 : Seq<AssignedState>> a2.len() == f.1
        && complete_inner(a2) && formula_sat_inner(f, a2)
    }
}
```

`complete_inner()` and `formula_sat_inner()` are defined as before, that is: a complete assignment is one where all variables are assigned, and a satisfied formula is one where all its clauses are satisfied. Finally: a satisfied clause is a clause where at least one literal is satisfied.

In other words: two formulas are `equisat` if and only if the existence of an assignment which satisfies the first formula implies the existence of an assignment which satisfies the second formula, and vice versa.

We further add this as a postcondition to the functions which have access to a mutable formula, adding loop invariants where necessary, ensuring that our algorithm in its entirety maintains a formula which is equisat to the original formula.

With the equisatisfiability predicate sorted, we have to prove that the resolution function generates clauses which when added to a formula results in an equisatisfiable formula. To do this, we create a predicate, `equisat_extension`, which states that a given formula, were it to be extended with a given clause, would result in a formula which is equisatisfiable with the original formula.

```
#[predicate]
pub fn equisat_extension_inner(
    c: Clause, f: (Seq<Clause>, Int)) -> bool {
    pearlite! {
        eventually_sat_complete_no_ass(f) ==>
        eventually_sat_complete_no_ass((f.0.push(c), f.1))
    }
}
```

We then create a lemma function which states that if we have a clause which is an equisatisfiable extension of the formula, and resolve it with a clause which is in the formula, then we get a new clause which is an equisatisfiable extension of the formula.

```
#[logic]
#[requires(formula_invariant(f))]
#[requires(equisat_extension_inner(c, f))]
#[requires(c2.in_formula_inner(f))]
#[requires(c3.resolvent_of(c, c2, k, m))]
#[ensures(equisat_extension_inner(c3, f))]
pub fn lemma_resolvent_of_equisat_extension_is_equisat(
    f: (Seq<Clause>, Int), c: Clause, c2: Clause, c3: Clause,
    k: Int, m: Int) {
    lemma_eq_formulas(f, (f.0.push(c3), f.1), c3);
}
```

The observant reader may notice that we had to define and access the *model* of Formula to prove these lemmas, as there was no way to talk about a formula where we have added a clause without working on sequences. The observant reader may also notice the internal helper lemma, `lemma_eq_formulas`. It is defined as follows:

```
#[logic]
#[requires(f2.0 == f.0.push(c))]
#[ensures((f.0).len() + 1 == (f2.0).len())]
#[ensures(forall<i: Int> 0 <= i && i < (f.0).len() ==>
          ((f.0)[i]).equals((f2.0)[i]))]
#[ensures(@(f2.0)[(f2.0).len()-1] == @c)]
```

```
pub fn lemma_eq_formulas(
f: (Seq<Clause>, Int), f2: (Seq<Clause>, Int), c: Clause) {}
```

`lemma_eq_formulas` exists to help the SMT solvers understand what the result of pushing to a formula is. `equals()` is defined as follows:

```
impl Clause {
    #[predicate]
    pub fn equals(self, o: Clause) -> bool {
        pearlite! {
            (@self).len() == (@o).len()
            && forall<j: Int> 0 <= j && j < (@self).len() ==>
                (@self)[j] == (@o)[j]
        }
    }
}
```

Finally, we have the `resolvent_of` predicate:

```
impl Clause {
    #[predicate]
    pub fn resolvent_of(
    self, c: Clause, c2: Clause, k: Int, m: Int) -> bool {
        pearlite! {
            (forall<i: Int> 0 <= i && i < (@c ).len() && i != k ==>
            (@c   )[i].lit_in(self)) && // #1
            (forall<i: Int> 0 <= i && i < (@c2).len() && i != m ==>
            (@c2  )[i].lit_in(self)) && // #2
            (forall<i: Int> 0 <= i && i < (@self).len()         ==>
            ((@self)[i].lit_in(c) || (@self)[i].lit_in(c2))) && // #3
            !(@c)[k].lit_in(self) && !(@c2)[m].lit_in(self) && // #4
            (@c2)[m].is_opp((@c)[k]) // #5
        }
    }
}
```

`k` and `m` are the indexes of the resolved out literal of the first and the second clause, respectively. `resolvent_of` is conjunction of 5 different statements. We have commented each of them in the snippet, and go through them in order. `#1` states that all of the literals of `c` which are not the resolved out literal occur in the resolvent. `#2` states the same, but with regards to `c2` instead of `c`. `#3` states that all the literals of the resolvent either exist in `c` or in `c2`. `#4` states that the resolved out literal does not occur in the resolvent. Finally, `#5` states that the literal which has been resolved out occur with opposite polarities in `c` and `c2`.

5.3.3 Proving the clause learning

With the ability to add clauses, as well as the relevant lemmas and the resolution predicate defined, we return to the clause analysis mechanism.

We follow the structure of the proof as presented earlier. On entry to `conflict_analysis`, we get the conflicting clause from the clause database. We then start doing iterative resolution on the antecedents of its literals until we reach a clause which is asserting. We simply maintain the invariant that our clause satisfies the clause invariant with regards to the input formula, that it is an equisatisfiable extension of the input formula, and that it is UNSAT. Whenever we do resolution, we know, due to the invariant on the trail, that the antecedent is what we call `post_unit` — it has one literal which evaluates to true, while the rest evaluates to false.

We also know that the literal which evaluates to true occur in both clauses. As it is the only literal which is true in the antecedent, and the other clause only contains UNSAT literals, we know that we can do resolution. We then do resolution, wherein we invoke the `lemma_resolvent_of_equisat_extension_is_equisat` lemma. We therefore have a new clause which both satisfies the clause invariant with regards to the formula, and which is an equisatisfiable extension. As we resolved out the only literal which was satisfied, this new clause is also UNSAT. If the literal has now become asserting, we are done, and may learn the clause.

The initial implementation and proof ended up being a fair bit of pain. Part of this was lack of understanding, part of it was some earlier suboptimal design decisions, part of it was due to limitations with CREUSOT, and part of it was due to some of the properties actually being hard, especially for SMT solvers. We ended up having to guide the SMT solvers more than desired, and the code became plagued with proof assertions. The same was the case for the trail, which together with clause analysis ended up accounting for over 2/3 of the invocations of `proof_assert!`.

We later redid the proof and implementation of resolution and the clause analysis mechanism. This resulted in a faster implementation with less code and significantly shorter proofs. To make it easier on the SMT solvers, as well as to make it easier to make an efficient implementation, we change `long_are_post_unit` slightly. Whereas we before had that the clause was post unit with regards to *some* literal of the clause, we now have it so that it is always post unit with regards to the first literal of the clause.

It should be noted that we were doing this before as well, but having it statically proven enables us to skip the first literal of all clauses which we do resolution on. This means that resolution becomes the simple procedure of removing the literal which is to be resolved out, and then add all but the first literal of the second clause. Ideally we would not add the literals which are going to be resolved out at all, but we leave this for future work.

A note about the current implementation of conflict analysis is that we maintain a `seen` vector to enable O(1) checking of membership, as well as a variable to keep track of the amount of literals which are assigned on the current decision level.

5.3.4 Backtracking the trail

Implementing backtracking was surprisingly difficult, and it was in fact the last part of the algorithm which was proven correct. We believe the reason for it being challenging is a combination of the author's lack of expertise, the tools offered by CREUSOT, the chosen encoding of the trail, certain patterns which are challenging for SMT solvers, and an inherent hardness of the problem. The weighting of the different factors contributing to the difficulty is not entirely clear.

Our initial representation of the trail encoded it as a tuple consisting of `Vec<Vec<Lit>>` and `Vec<(usize, Reason)>`. The first of these represents the trail, where each decision level is encoded as a vector of assigned literals. The first element in each of these vectors is the decision of that decision level, and the remaining literals are the literals implied by that decision. The 0th vector does not adhere to this invariant, and contains all the literals which are implied at decision level 0, or, in other words: it is a level of units.

The second is simply a mapping between each literal and the decision level where it was assigned, as well as the reason for its assignment. The second vector is exactly `f.num_vars` long, and is indexed on the `idx` of a `Lit`.

This representation worked fine, and it was fairly straightforward to prove all desired properties, except that undoing a single decision level yielded a result where the `trail_are_post_unit` invariant holds. It is not decided whether proving this property is truly infeasible for such a data structure, but it was decided to redo the trail to follow the more traditional "linear" trail representation. Though this was a fairly large refactoring, the provability and general improvements in ergonomics of the new trail were immediately obvious.

The proof of backtracking for the linear trail was done before integrating the trail with the rest of the solver. An attempt was therefore made to port the proof of backtracking for the linear trail to the initial version of the trail, which was already integrated with the rest of the solver. This effort was fruitless, and it was decided more viable to refactor the entirety of the solver, and thus in part redo a substantial amounts of the proofs.

5.4 Optimizations

In this section we describe the various optimizations which have been implemented, and the main ideas behind their proofs.

5.4.1 Two watched literals

The simplest way to figure out whether a clause is unit is to visit it and check whether it is satisfied, and if it is not, check if it has one unassigned literal, at which point we propagate the unit literal. This is the way we do it in Robinson. The step up from this approach is to keep counts of

the amount of unassigned literals for all the clauses, and enqueue those who reach a count of 1. This is an improvement, but we can do better, by employing a technique called two watched literals (TWL/2WL).

2WL explained

The two watched literals scheme dates back to the head/tail list of the solver SATO [56] and was later refined by the solver Chaff [42] in 2001. They noted that 90% of the time of a SAT solver was spent on unit propagation. To remedy this, the two watched literal scheme was introduced. It is based on the observation that a clause cannot become unit as long as at least two of its literals are unset. We therefore "watch" two unset literals of each clause. If any unwatched literal in the clause is assigned, the clause has either become satisfied or its satisfiability is still undecided, but we know that it is not unit: our two watched literals are still unassigned.

In the case that one of the watched literals is assigned, we check to see if there exists another unassigned literal which is not watched. If so, then we remove the previous watch, and watch this literal instead. In the case that the clause is satisfied, for instance by one of the unwatched literals being satisfied at a previous point, we update the watch to watch this literal. Usually the watching of literals is organized by watching for one of the first two literals of a clause being falsified. We then reorder the clause to maintain that the first two literals are watched, and in the case that the clause is satisfied, we move this satisfied literal to the start of the clause and watch it.

As an example, take the following clause:

$$(\neg \underset{\varnothing}{\overset{\downarrow}{a}} \vee \underset{\varnothing}{\overset{\downarrow}{b}} \vee \underset{\varnothing}{c} \vee \neg \underset{\varnothing}{d})$$

The currently watched literals are indicated with an arrow. The currently assigned value is denoted with $\varnothing$ for no current assignment, $\top$ for a positive assignment, and $\bot$ for a negative assignment.

Let us say that c is assigned to $\top$, as in:

$$(\neg \underset{\varnothing}{\overset{\downarrow}{a}} \vee \underset{\varnothing}{\overset{\downarrow}{b}} \vee \underset{\top}{c} \vee \neg \underset{\varnothing}{d})$$

This results in no work to be done for this particular clause.

Let us say we later set a to $\top$ as well, falsifying $\neg a$, as in:

$$(\neg \underset{\top}{\overset{\downarrow}{a}} \vee \underset{\varnothing}{\overset{\downarrow}{b}} \vee \underset{\top}{c} \vee \neg \underset{\varnothing}{d})$$

This will result in a visit to the clause, and we will move the watch to c, which is satisfied:

$$(\neg \underset{\top}{a} \vee \overset{\downarrow}{\underset{\varnothing}{b}} \vee \overset{\downarrow}{\underset{\top}{c}} \vee \neg \underset{\varnothing}{d})$$

In practice we reorder the clause, resulting in the following clause:

$$(\overset{\downarrow}{\underset{\top}{c}} \vee \overset{\downarrow}{\underset{\varnothing}{b}} \vee \neg \underset{\top}{a} \vee \neg \underset{\varnothing}{d})$$

In the case that b were to become falsified, we would visit the clause, see that c is currently satisfied, and return, without updating the watches. In the case that $\neg d$ were to become assigned, we would not have to do any changes to our watches.

As an example of the case where the clause becomes propagating, take the following initial assignment:

$$(\neg \overset{\downarrow}{\underset{\varnothing}{a}} \vee \overset{\downarrow}{\underset{\varnothing}{b}} \vee \underset{\bot}{c} \vee \neg \underset{\top}{d})$$

c and $\neg d$ are currently unsatisfied, but as both our watches are unset, we know the clause is yet to be propagating. Let us say we falsify b as in:

$$(\neg \overset{\downarrow}{\underset{\varnothing}{a}} \vee \overset{\downarrow}{\underset{\bot}{b}} \vee \underset{\bot}{c} \vee \neg \underset{\top}{d})$$

This results in a visit to the clause, where we discover that there is nowhere to place the second watch, resulting in the propagation of $\neg a$ with this clause as its reason.

This leaves only the case where the clause somehow has become falsified. Let us again say we have the previous formula and assignments:

$$(\neg \overset{\downarrow}{\underset{\varnothing}{a}} \vee \overset{\downarrow}{\underset{\bot}{b}} \vee \underset{\bot}{c} \vee \neg \underset{\top}{d})$$

Since b became falsified, we currently have a visit to the clause in queue. If $\neg a$ is falsified as well by the time we visit the clause, the visit will show that all of the literals are falsified, and thus that the clause is falsified. This could for instance happen if the following clause is also in the problem set:

$$(\overset{\downarrow}{\underset{\varnothing}{a}} \vee \overset{\downarrow}{\underset{\bot}{b}})$$

In this case, whichever clause is visited first will result in either a or $\neg a$ being propagated, falsifying the other clause before we get a chance to update the watch on b.

A correctly implemented two watched literal scheme has the advantage that no watches has to be updated during backtracking. When backtracking due to a conflict, we undo at least all the variables which were assigned at the current level, and thus go from a state where the invariant for the two watched literals scheme does not hold, to a state where we know it does hold. When undoing further levels, we are going from a state where the invariant holds, to a new state where we know the invariant holds.

Improvements to 2WL

There are a few further refinements which can be done to the 2WL scheme. The first is to treat binary clauses separately. Binary clauses are essentially implications, and if one of the literals is assigned, we either have a conflict, or we can propagate the other literal. By treating them separately, we can save a likely cache miss, by only loading the associated watch for the binary clause. We did not observe a significant improvement in speed in treating binary clauses separately, and have thus not prioritized adding it to CreuSAT, though it is likely to be included in the future.

The second refinement is to add a blocking literal for each watch. The blocking literal is a literal which is not the watched literal, which also occurs in the watched clause. We store the blocking literal directly in the watch, and then check whether it is satisfied before visiting the clause to find a new literal to watch. If the blocking literal is satisfied, then we know that the clause is not unit, and return without updating the watch. This saves us a likely cache miss on loading the watched clause. We found blocking literals to be a substantial improvement, as well as being easy to prove, and do therefore include this optimization.

The third refinement is to keep track of the current search position for each clause [22]. This is done to avoid accidental quadratic behaviour when trying to find new literals to watch, which can happen if the literals which occur early in a clause are falsified early in the search process. When searching for a new literal to watch, we start from this search position, and loop around when the index goes out of bounds. If we find an unset literal, we update the search position to the next index to start the search from. We observed very solid improvements from this technique, and do therefore include this optimization.

Proving 2WL

Proving two watched literals proved to be quite tricky. Because we implement clauses as a vector of literals, we have to prove that the reordering of the clauses maintains all invariants and notions of satisfiability. The same applies to the popping and pushing of the `Watches`. These goals, in combination with a few others, are the "notoriously sticky" goals of the proof. This seems to be linked to sequences being modeled opaquely in WHY3, though there might exists solutions which could be employed, either in CREUSOT or by the proof writer.

The "stickyness" introduced by the current proof of 2WL, combined with discovering that proving 2WL in Isabelle took 15 months [29], made us decide on foregoing completeness, both in order to save time, and in order to keep fluidity of the code base as a whole. This allows us to treat 2WL as an optimization and as largely separate from the solver as a whole, which again enabled more experimentation in other parts of the solver. The main issue with fully proving 2WL, and thus maintaining completeness, is that it would lead to pervasive obligations, and make some changes which one

might want to make lead to parts of the proof having to be redone.

That being said, the solver has since become more mature, and it is likely that proving 2WL, and thus reintroducing completeness, will be looked at in the not too distant future.

Proving the improvements to 2WL

The two optimizations to 2WL which we prove, are blocking literals and circular search. Adding and proving blocking literals is, as we have sacrificed completeness, not very difficult. We simply check the satisfiability of the blocking literal before entering the body of the `propagate_lit_with_regard_to_clause` function. When updating the blocking literal, we simply have to maintain the safety invariant that the variable of the literal is less than the number of variables for the formula.

Proving circular search, on the other hand, is a bit more cumbersome. We considered adding an invariant that the carried search index is between 2 and the length of the associated clause, but decided that this was not worth it. The main issue is that this invariant became somewhat pervasive, and all functions which do clause reorderings, additions or deletions would have to either reestablish the property, or prove that it is maintained. We therefore opted for the following:

```
let mut search = util::max(util::min(search, clause.len()), 2);
```

which seems to have had negligible performance impact. The loop is then split into two loops, one which iterates from `search` to `clause.len()`, and then one which iterates from `2` to `search`.

5.4.2 Variable move-to-front

Whenever unit propagation does not return a conflict clause or a ground conflict, we either have a complete assignment, in which case we are SAT, or we have an incomplete assignment, and must therefore make a decision. The simplest alternative is to make for instance an either arbitrary or a random decision, neither of which are very good. A step up from not discriminating, is to calculate some metric for each variable, and then choose the variable which has the highest score. Examples of such scores could be the number of occurrences of a variable, or number of occurrences with a metric favoring literals which occur in shorter clauses.

Such metrics are significantly better than arbitrary or random decisions, but they can be further improved upon. During the search, it is possible to gain information about which variables are actually important, and update the decision order. This is done by favoring literals which occur in recently learned clauses. This is what is done by the two decision heuristics which are considered state-of-the-art. The first of these is called (Exponential) Variable State Independent Decaying Sum (EVSIDS/VSIDS), and was also introduced by the Chaff solver. The second of these, and the one we choose to implement, is called variable move-to-front (VMTF).

We choose to implement the VMTF heuristic as it performs at least as well as VSIDS, while in general being considered easier to implement and to reason about, as it does not require a heap, nor floating point numbers. The VMTF heuristic was originally presented by Lawrence Ryan in his Master's thesis [51]. Even though the thesis demonstrated good results for the heuristic, VSIDS has remained as the dominant decision heuristic, likely because Ryan never released the source code to his solver. That being said, largely due to [10] by Biere and Fröhlich in 2015, and the subsequent adoption of VMTF in solvers such as CaDiCaL, Splatz [8], and IsaSAT [21], the VMTF heuristic has gained in popularity in recent years.

We will be implementing the VMTF of Biere et al., as it is presented in [10] and [21]. We will be employing the moving to front of all of the literals which are involved in conflict analysis, rather than the moving of a small constant, e.g. 8, as described in [51]. The VMTF of Biere et al. consists of having a doubly-linked linked list stored in a vector or array, ensuring O(1) enqueue and O(1) dequeue. Furthermore, each node also stores a *timestamp*, indicating the last time it was moved to front. When using the linked list, we maintain the invariant that the linked list is sorted in reverse order with regards to timestamps, or in other words: the high timestamps are towards the start of the linked list, and the low timestamps are towards the end. A note here is that we sort the variables to bump on current timestamp, and thus keep the relative order of the bumped variables.

We also maintain a current search index, such that everything "before" (reachable by following a series of previous-pointers from the search index to the `INVALID` pointer) the search index are assigned. Maintaining these invariants means that finding the next unassigned index is cheap, and backtracking simply requires us to check if the timestamp of the unassigned index is higher than the current search index, and is thus also cheap. Finally, as the timestamp may overflow, we check for this, and eventually do a rescoring of all the nodes in the list, maintaining their order and their invariant.

We implement VMTF with the `Decisions` struct:

```
pub struct Decisions {
    pub linked_list: Vec<Node>,
    timestamp: usize,
    pub start: usize,
    pub search: usize
}
```

where `Node` is defined as:

```
pub struct Node {
    pub next: usize,
    pub prev: usize,
    pub ts: usize,
}
```

`timestamp` stores the current timestamp, `start` stores the index of the first node of the linked list ("head"), and `search` is the index to start the next search for an unassigned index to decide on. Each `Node` stores its timestamp, `ts`, as well as the index to the node before it, and the index to the node after it. We use `usize::MAX` as the value for `None`, as we have bounded the number of variables of the formula to `usize::MAX/2`. We have not tested whether this actually gives any performance or space improvements over using `Option<usize>`, but it may, and it also makes the ergonomics of the specifications marginally better.

Proving decision heuristics is not as difficult as it initially may seem. The key insight is to realize that one can treat the decisions as entirely separate from the rest of the solver, and that one can get away with just proving safety. We therefore implement VMTF, prove its safety, then add a linear check at the end which checks that all variables have been assigned. In the case that there exists an unassigned variable, this variable is returned. It should be noted that we could prove away this check, but we have not prioritized it, as it is a linear pass which will run once for instances which are SAT, and never for instances which are UNSAT. The current check does not introduce incompleteness, nor affect performance, so we believe effort is better spent elsewhere.

The final proof of VMTF is thus a proof of the safety of the linked list, and that all the variables which are returned from the `get_next` function are unassigned and in bounds with regards to the partial assignment.

5.4.3 Phase saving

When making a decision, we have to find an variable which is unassigned, and then decide on which polarity to assign. Previously this was done as a part of the decision heuristic, but what is now considered state-of-the-art is a technique which is called *phase saving*. It is really a quite simple technique: when deciding on a polarity, choose the same polarity as the last time the index was assigned. If the index has not been assigned to before, it is usual to assign it to false, but one could also opt to assign true, choose at random, or base the choice on some calculation. We choose to initially try setting each index to false.

Implementing and proving phase saving are both fairly straight forward, at least with the way we chose to represent the partial assignments. Initially we represented assignments as a `Vec<Option<bool>>`. As `Option<bool>` has three possible values, we are guaranteed not to have a bitarray, most likely ending up with using a byte for each of our assignments. This leaves us in a situation where we either implement two vectors — one which keeps track of which indexes are assigned, and one which contains the actual assignments, or we accept the one byte per value, and represent our assignments as a vector of `u8`s.

The first alternative not only less ergonomic to use and to reason about, it is not even likely to be any faster, especially when one later wants

to add phase saving. We therefore refactored the code to represent the assignments as a `Vec<u8>`, and solve the issue of unassigning in the following way:

```
pub fn unassign(&mut self, idx: usize) {
    self[idx] += 2;
}
```

Proving this then becomes quite easy, we just have to require that the index is assigned on entry.

Similarly, on assigning a decision, we simply decrement the value at the index by 2, and prove safety and correctness by requiring that the index is either `2` or `3` on function entry.

5.4.4 Clause database simplification

As the solver progresses, it may learn clauses which enables the clause database to be simplified. Most of the techniques which are applied by the state-of-the-art are out of scope for this thesis, but we will implement a very simple clause database minimization mechanism. The technique we implement is essentially a simplified version of pure literal elimination. When we learn a unit clause, we unwatch all clauses which have become satisfied by the unit clause, as they will now always be satisfied. Ideally, we would also remove all occurrences of the negation of the literal of the unit clause, which we could model as a resolution step which results in the removal in the unit clause.

For the proof, we simply exploit the incompleteness introduced by the current proof of watched literals, and remove the watches for the clause. If we reintroduce completeness at a later point, this would not work for clauses which are a part of the original formula. We would then either have to prove that a unit clause enables the deletion of all the clauses where it occurs, or make do with only deleting learned clauses. As all learned clauses are implied by the initial formula, their deletion can be done at will.

5.4.5 Clause deletion

As the search progresses, more and more clauses are added to the clause database. As clauses are added to the clause database, they are added to the watch list as well, making the unit propagation mechanism take more and more time. To mitigate this slowdown, clauses are deleted from the clause database, which also has the benefit of using less memory. As observed for instance by Audemard and Laurent in [2], more than half of the learnt clauses do not occur in the final proof of unsatisfiability. Though they may propagate a clause which is not useful, or update the heuristics to lead to a favourable effect, keeping them around is a net negative. The same is the case for clauses which do occur in the final proof, but which will not be used again during the run of the solver.

We don't know for certain which clauses are useful, and which are not, but certain heuristics are better than others. In general, useful clauses are those which are likely to become involved in unit propagation. This is linked to the length of the clause, and thus many solvers keep clauses which are less than some length. It is also linked to what is called the *Literal Block Distance* (LBD) which was introduced by the Glucose solver [3]. The LBD of a clause is initially calculated when learning a clause, and is simply the count of distinct decision levels among the literals of the clause. The intuition for why LBD is useful, is that it captures clauses which have literals which are "linked". A clause with a low LBD has required few decisions to become unsatisfied, even though it might consist of many literals. This means that an assignment to any of the "linked" literals is likely to lead to the propagation of the other "linked" literals, and the clause is thus likely to be involved in unit propagation.

There are multiple designs for clause deletion. The design we chose was to implement it as a two phased design, where the first phase simply unwatches the clauses to be deleted, and the second phase cleans up and frees the deleted clauses. If one does it in one phase, either deletion has to be combined with search restart, or one has to check whether the clause exists as the antecedent for some assignment in the trail when doing the deletion. Splitting it into two phases circumvents this issue, as one can do memory reclamation at a point when one is sure the clauses do not exist as antecedents, such as during search restart.

We have yet to implement the actual garbage collection phase, and are thus wasting memory. This should not be a conceptually very hard proof, as we are maintaining the invariant that all the clauses which have been added to the clause database are equisatisfiable extensions of the initial formula, and can thus be added and removed at will. It will, however, most likely be a somewhat tedious proof, as poping from vectors always leads to having to prove that invariants are maintained. As we have yet to experience issues with memory, we have not prioritized this effort. We thus simply exploit the incompleteness introduced by 2WL to gain clause deletion with minimal required proof work.

5.4.6 Search restart

Search restarts has its origin in the desire to avoid heavy-tailed behaviour for SAT instances when running the DPLL algorithm. The observation was that the solver could get "stuck" in an area of the search space where there was no solution, while if it had done a different decision early on in the search, it would find a solution and terminate. This effect of search restart persists in CDCL solvers. In addition, doing a search restart enables the solver to learn clauses from a different part of the search space, which is helpful both for SAT and UNSAT instances.

A consideration when implementing search restart is how often to restart, and how much to increase the delay between each restart. If one restarts

too often, one might not make any meaningful progress, and miss either a potential solution or the learning of important clauses. Restarting too seldom leads to the aforementioned problem of being stuck. If search restart is combined with clause deletion, either restarts have to become less and less frequent, more and more clauses have to be kept, or both, otherwise the solver may become non-terminating.

There are a few schemes for clause restarts which are popular. The first scheme, which for instance MiniSat uses, is what is called *Luby restarts*, which based on the *Luby sequence* of Luby, Sinclair and Zuckerman [33]. The second scheme, which was popularized by the Glucose [3] solver, is usually called *Glucose (style) restarts*, or *aggressive restarts*.

It may also be called exponential moving average (EMA) based restarts, as it is based on keeping track of the slow-moving and the fast-moving averages of the LBD of learned clauses. In the case that the fast-moving average is substantially higher than the slow-moving average, the search is restarted. This corresponds to recently learned clauses having an on average high LBD, or, in other words: we are learning bad clauses.

Aggressive restarts are very useful, but they have one drawback: they may cause a SAT assignment to be missed. The scheme was therefore later extended to *block* a restart if the current length of the trail is significantly longer than the average length of the trail for the last 5000 conflicts [4].

Recently, combining Luby restarts and Glucose style restarts has become popular. This was introduced in [49] by Chanseok Oh, and has later been refined through a technique which is called *target phases*. Target phases was introduced by CaDiCaL in 2019, and was later described in [9]. It combines a hybrid restart strategy with a refinement of phase saving. Instead of always choosing the last assigned polarity, the solver also has "phases" where it may try for instance always true, always false, random, or the opposite of the saved phase.

As we did not learn about the hybrid techniques until quite recently, we implement a search restart scheme which is similar to the one found in Glucose. It should be noted that it is likely that we will add target phases in the near future.

Implementing and proving search restart is fairly straightforward. Search restart is just the act of backtracking to asserting level, and then *continuing backtracking*. In other words, we get most of the implementation for free.

5.5 The top level contracts

We end this chapter with a presentation of the top level specifications of the entry point to our solver. In other words: if you agree that these specifications capture the correctness of a SAT solver, then you agree that we have, indeed, created a SAT solver which is verified to be correct.

Whereas we previously treated `inner` and non-`inner` predicates as being the same, we will now provide the entirety of the specifications, ad verbatim. This is because we were before presenting the ideas of the proof, whereas we are now providing the *contract* of our solver.

We start with presenting the data structures which will appear throughout our specifications:

```
pub type AssignedState = u8;
pub struct Lit {
    pub idx: usize,
    pub polarity: bool,
}
pub struct Clause {
    pub rest: Vec<Lit>,
}
pub struct Formula {
    pub clauses: Vec<Clause>,
    pub num_vars: usize,
}
```

A note about `Clause` is that its field is called `rest`. This is due to a previous attempt of having all clauses be at least binary, with the fields `first` for the first literal, `second` for the second literal, and `rest` for the remaining literals. This experiment did not pan out proof-wise, but as we would like to resume the experiment in the future, we have kept the naming.

We also present their models:

```
impl Model for Clause {
    type ModelTy = Seq<Lit>;
    #[logic]
    fn model(self) -> Self::ModelTy {
        self.rest.model()
    }
}

impl Model for Formula {
    type ModelTy = (Seq<Clause>, Int);
    #[logic]
    fn model(self) -> Self::ModelTy {
        (self.clauses.model(), self.num_vars.model())
    }
}

impl Model for Assignments {
    type ModelTy = Seq<AssignedState>;
    #[logic]
    fn model(self) -> Self::ModelTy {
        self.0.model()
    }
```

```
}
```

We simply call the `model()` function on their composite parts. We do not define a model for `Lit` .

The entry point of CreuSAT:

```
#[ensures(match result {
    SatResult::Sat(assn) => { formula_sat_inner(@(^formula), @assn)
        && formula.equisat(^formula) },
    SatResult::Unsat     => { (^formula).not_satisfiable()
        && formula.equisat(^formula) },
    _                    => { true },
})]
pub fn solver(formula: &mut Formula) -> SatResult
```

We will start with explaining `SatResult`. Then we will look at `equisat`, before looking at the satisfiability predicates.

`SatResult` is an `enum` which is defined as follows:

```
pub enum SatResult {
    Sat(Vec<AssignedState>),
    Unsat,
    Unknown,
    Err,
}
```

The `Sat` variant gives a satisfying assignment of the formula. The `Unknown` variant exists in case we make the solver incremental in the future, and the `Err` variant exists both to capture errors such as an invalid formula, and to allow for the aforementioned incompleteness.

`equisat` is as previously discussed:

```
impl Formula {
    #[predicate]
    pub fn equisat(self, o: Formula) -> bool {
        pearlite! { self.eventually_sat_complete_no_ass() ==
                              o.eventually_sat_complete_no_ass() }
    }
}
```

The same applies to `eventually_sat_complete_no_ass`:

```
#[predicate]
pub fn eventually_sat_complete_no_ass(
    f: (Seq<Clause>, Int)) -> bool {
    pearlite! { exists<a2: Seq<AssignedState>> a2.len() == f.1
    && complete_inner(a2) && formula_sat_inner(f, a2) }
}
```

`eventually_sat_complete_no_ass` calls the predicate `complete_inner()`, which is the same as before:

```
#[predicate]
pub fn complete_inner(a: Seq<AssignedState>) -> bool {
    pearlite! { forall<i: Int> 0 <= i && i < a.len() ==> !unset(a[i]) }
}
```

It also calls `formula_sat_inner`. It is conceptually the same satisfiability predicate as the one used in Friday and in Robinson. We provide it and its component parts for the sake of completeness:

```
#[predicate]
pub fn formula_sat_inner(f: (Seq<Clause>, Int),
    a: Seq<AssignedState>) -> bool {
    pearlite! { forall<i: Int> 0 <= i && i < f.0.len() ==>
                f.0[i].sat_inner(a) }
}

impl Clause {
    #[predicate]
    pub fn sat_inner(self, a: Seq<AssignedState>) -> bool {
        pearlite! { exists<i: Int> 0 <= i && i < (@self).len() &&
                (@self)[i].sat_inner(a) }
    }
}

impl Lit {
    #[predicate]
    pub fn sat_inner(self, a: Seq<AssignedState>) -> bool {
        pearlite! {
            match self.polarity {
                true  =>  (@a[@self.idx] == 1),
                false =>  (@a[@self.idx] == 0),
            }
        }
    }
}
```

The `SatResult::Sat(assn)` branch of the match statement thus says that our final formula is equisatisfiable with the formula given at entry, and that the final formula is satisfiable with the `Assignments` `assn`.

This leaves `SatResult::Unsat`. It calls the predicate `not_satisfiable()`

```
impl Formula {
    #[predicate]
    pub fn not_satisfiable(self) -> bool {
        pearlite! {
            exists<c: Clause> (@c).len() == 0
            && c.equisat_extension(self)
        }
    }
}
```

`not_satisfiable()` simply states that the empty clause is an equisat extension of the formula — that the formula implies the empty clause. `equisat_extension` is, as we remember, defined as follows:

```
#[predicate]
pub fn equisat_extension_inner(
    c: Clause, f: (Seq<Clause>, Int)) -> bool {
    pearlite! {
        eventually_sat_complete_no_ass(f) ==>
        eventually_sat_complete_no_ass((f.0.push(c), f.1))
    }
}
```

And that is the entirety of specification. Note that we do not put any requirements on the input formula. We thus try to establish the formula invariant on entry, and return an error in the case that the formula is malformed.

Part IV

Evaluation and the road ahead

Chapter 6

Evaluation

In this chapter we will evaluate Robinson and CREUSOT in terms of performance, lines of code, and how long they take to prove. We will compare them to a collection of other solvers, some of which are verified, some of which are not. Finally we will discuss the results.

6.1 Setup

Execution setup

We will be testing the solvers on the StarExec cluster, which is the cluster which the SAT competitions are held on.

At the time of writing, this is the specification of the StarExec cluster, as stated on https://www.starexec.org/starexec/public/machine-specs.txt:

```
# Starexec stats nodes 001 - 192:
Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz (2393 MHZ)
    10240  KB Cache
    263932744 kB main memory

# Software:
OS:       CentOS Linux release 7.7.1908 (Core)
kernel:   3.10.0-1062.4.3.el7.x86_64
glibc:    glibc-2.17-292.el7.x86_64
          gcc-4.8.5-39.el7.x86_64
          glibc-2.17-292.el7.i686
```

We will be building CreuSAT with the following command:

```
RUSTFLAGS='-C relocation-model=static' cargo build --release
 \ --target x86_64-unknown-linux-musl
```

We are building on a 64-bit Manjaro 21.2.6 with the `5.10.117-1-MANJARO` kernel and the following version of `rustc`:

```
    rustc 1.61.0 (fe5b13d68 2022-05-18)
```

We compile with the following flags:

```
-O3
```

Verification setup

We will be using CREUSOT commit #830cec1 with `rustc nightly-2022-05-14` and WHY3 version 1.5. We will be running the default auto level 3 strategy in the WHY3 IDE. We will be using Alt-Ergo 2.4.1, Z3 4.8.12, and CVC4 1.8. We are doing the benchmarks on Manjaro 21.2.3 with the 5.10.98-1-MANJARO kernel. The CPU is an Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz, and the memory is 4 x 8GB of Corsair Vengeance DDR4 3000MHz.

WHY3 allows for specifying a time limit, a memory limit and a process limit for external provers. We will for all configurations use a time limit of 0 seconds, corresponding to no timeout. We will be running once with the configuration of 1 prover and 16 GB of memory, and once with the configuration of 8 provers and 4GB of memory. We will call the first configuration "single-threaded", and the second configuration "multi-threaded". We did attempts with 16 provers with 2 GB each and 32 provers with 1 GB each as well, but they were indistinguishable performance-wise from 8 provers with 4 GB of memory each.

We verify TrueSAT commit `#62f52fd` as found in the repository of Cezar Andrici. The verification is done on the aforementioned Manjaro with Dafny 3.6.0.40511 and the Dafny Visual Studio Code extension version 2.4.0 with
`aspnet-runtime-6.0.2.sdk102-1`.

We build TrueSAT using the provided Makefile. To execute the code, we use the Mono JIT compiler version 6.12.0.

We verify IsaSAT commit `#a09e37b` as found in the IsaFoL repository. The verification is done on the aforementioned Manjaro with Isabelle2021-1, isabelle_llvm commit #547c8a2 and afp-2022-06-07. We run the following command: `isabelle build -d ~/afp-2022-06-07/thys -d ~/isabelle_llvm/thys -d . -b IsaSAT`

We will be benchmarking IsaSAT as submitted to SAT Competition 2022.

We will be benchmarking `versat` as found on the homepage of Aaron Stump per 11.01.22. Built with gcc 8.5.0 and the following command: `gcc -s -m32 -std=gnu89 -O2 versat.c -o versat` on Red Hat Enterprise Linux 8.6, kernel version 4.18.0, and then submitted as built to the StarExec cluster.

6.2 Evaluation of Robinson

We will be comparing Robinson with TrueSAT. TrueSAT is a pure DPLL solver written and verified with Dafny, and it is thus quite similar to

Robinson, both in terms of what the code does, and in how the proof is conducted.

Verifying Robinson

Verifying Robinson by choosing a the root node in WHY3 and running the auto level 3 takes around 1 minute, 46 seconds of wall-clock time when running single-threaded, and around 24 seconds of wall-clock time when running multi-threaded.

Verifying Robinson after having done proof discovery by obsoleting all the proofs, and then replaying all the obsolete proofs in WHY3 takes around 24 seconds of wall-clock time when running single-threaded, and around 10 seconds of wall-clock time when running multi-threaded.

Using the WHY3 proof replay feature by running `time why3 replay -Lprelude mlcfgs/Robinson` in the root directory of the repository yields the following: `42,64s user 1,77s system 275% cpu 16,144 total`

Verifying TrueSAT

Verification of TrueSAT takes 1 minutes 36 seconds when running with 8 cores, and 4 minutes 22 seconds when running single-threaded.

Benchmarking the solvers

Benchmark	Robinson	TrueSAT	CreuSAT
uf100	14.69	72.73	1.18
uuf100	33.80	121.86	2.62
uf125	8.85	26.43	0.44
uuf125	18.38	54.45	0.97
uf150	35.11	110.79	1.38
uuf150	98.34	298.70	3.50
uf175	213.06	613.27	5.21
uuf175	520.56	1924.57	14.05
uf200	950.48	4189.55	23.68
uuf200	2301.75	10491.42	67.40

Table 6.1: Results of running the solvers on random 3SAT benchmarks

Benchmark	Robinson	TrueSAT	CreuSAT
hole6	0.085	0.056	0.088
hole7	0.137	0.153	0.139
hole8	0.949	2.254	3.759
hole9	15.274	45.445	57.162
hole10	281.032	1088.263	491.569

Table 6.2: Results of running the solvers on pigeonhole problems

As we were not sure whether we would get TrueSAT to work on the StarExec clusters, we chose to do the benchmarking locally. DPLL solvers are, except for problems such as those based on the pigeonhole principle, orders of magnitudes slower than CDCL solvers. We therefore believe that running Robinson and TrueSAT on problems from the SAT competitions would not be very insightful. We instead choose to run the solvers on the problems used by Andrici and Ciobâcă in [1], under the assumption that these were chosen due to being suited to evaluate DPLL solvers.

These are random 3SAT problems from the SATLIB problems, as found per 14.05.22. The uf100 and uuf100 problem sets consist of 1000 problems each, whereas the rest consist of 100 problems each. As DPLL solvers are known to outperform CDCL solvers on pigeonhole problems, we also benchmark the solvers against the 5 pigeonhole problems which are found in the SATLIB benbchmarks. The given times are in seconds, and is the time it takes to run the given solver on all the problems in the problem set. We include CreuSAT for comparison reasons.

6.3 Evaluation of CreuSAT

We will be comparing CreuSAT with the verified SAT solvers `versat` and IsaSAT. These are, to the best of our knowledge, the only two verified solvers which have been able to solve a substantial amount of problems from the SAT competitions.

versat

`versat` was introduced as a part of the PhD thesis of Duck Ki Oe [48] in 2012. It is written in the dependently typed programming language GURU, which offers code generation to the C programming language. `versat` was, at the time of its creation, the highest performing verified SAT solver, and remained as this until being surpassed by IsaSAT in 2018.

The source code of `versat` totals 9884 lines of code, of which it is estimated that around 80% is auxiliary code [48]. Oe does not state how long it takes to verify `versat`. We have not prioritized getting GURU to work, largely because the GURU programming language has been discontinued since 2012.

IsaSAT

IsaSAT is a verified SAT solver which was introduced in 2018 [21], and which has since been improved. IsaSAT is based on a refinement technique in Isabelle/HOL, and is a part of the Isabelle Formalization of Logic (IsaFoL) [27] project.

Running the build command results in the following:

- Finished Isabelle_LLVM (0:10:05 elapsed time, 0:34:42 cpu time, factor 3.44)

- Finished Watched_Literals (0:33:00 elapsed time, 1:24:31 cpu time, factor 2.56)
- Finished IsaSAT (1:20:56 elapsed time, 4:33:34 cpu time, factor 3.38)

In terms of lines of code, IsaSAT is a lot larger than `versat`. The CDCL session is around 25 thousand lines of code, the Watched_Literals session is around 44 thousand lines of code, and the IsaSAT session is around 78 thousand lines of code. It should be noted that much of this has been generated automatically, by using the Sledgehammer functionality in Isabelle, and that much of this code is needed to support borth LLVM and SML as compile targets.

We also count the lines of code of the IsaSAT of branch CPP 2019, which be believe corresponds to IsaSAT-30 of [20]. The line count for this version is around 104 thousand lines of code.

CreuSAT

The current version of CreuSAT has a bit over 4 thousand lines of code, with a proof overhead of around 3 to 1.

Most versions of CreuSAT has had its proof pass by running auto level 3 and waiting. The current version needs manual intervention on the `move_to_end` and `remove_from_clause` functions in `Clause`. These do not pass after splitting, but pass when running Alt-Ergo on them for 15 and 7 seconds, respectively. We also have to manually do `inline_all` + split on a few of the subgoals. We thus prove CreuSAT by running Alt-Ergo without a time limit on these, and then run WHY3 auto level 3 on the root node. Once we see that auto level 3 has become stuck, we do the manual steps. Time is measured from the instant the first Alt-Ergo solver is dispatched, and ends when the root node is verified.

Verifying CreuSAT by doing the procedure stated above takes around 53 minutes of wall-clock time when running single-threaded, and around 15 minutes of wall-clock time when running multi-threaded.

Verifying CreuSAT after having done proof discovery by obsoleting all the proofs, and then replaying all the obsolete proofs in WHY3 takes around 9 minutes and 3 seconds of wall-clock time when running single-threaded, and 3 minutes and 14 seconds of wall-clock time when running multi-threaded.

Using the WHY3 proof replay feature by running `time why3 replay -Lprelude mlcfgs/CreuSAT` in the root directory of the repository yields the following: `996,67s user 27,10s system 405% cpu 4:12,31 total`

Performance evaluation

We will, in addition to CreuSAT, IsaSAT and `versat`, also benchmark the following solvers:

- CleanMaple_PriPro as submitted to SAT Competition 2021. Chosen as it was the lowest performing solver of the previous SAT competition, and does thus indicate the "bottom" of the state-of-the-art.
- Kissat_MAB as submitted to SAT Competition 2021. Chosen as it was the highest performing solver of the previous SAT competition.
- Minisat-v2.2.0-106-ge2dd095, simp_proof as submitted to SAT Competition 2018. Chosen as it is the latest version of the well-known MiniSat solver.
- Varisat as submitted to SAT Competition 2018. Chosen as it is, to the best of our knowledge, the best SAT solver written in Rust to ever enter the SAT Competition.
- MicroSat as it exists in the repository of Marijn Heule per commit #04f9625.

SOLVER	SAT	UNKNOWN	UNSAT
Kissat_MAB	230	204	217
MiniSat	282	211	158
Varisat	281	230	140
IsaSAT	175	346	130
CleanMaple	253	290	108
CreuSAT	210	362	79
versat	60	529	62

Table 6.3: Results of running the solvers on the SAT Race 2015 problems

SOLVER	SAT	UNKNOWN	UNSAT
Kissat_MAB	230	104	217
MiniSat	200	193	158
Varisat	200	211	140
IsaSAT	175	246	130
CleanMaple	182	261	108
MicroSat	158	288	105
CreuSAT	145	327	79
versat	60	429	62

Table 6.4: Results of running the solvers on the SAT Race 2015 problems with the manthey_Dimacs* problems removed

We run the solvers on the 651 benchmarks of the SAT Race 2015 with a memory limit of 24 GB and a time limit of 1800 seconds.

We choose these benchmarks as they are the most recent benchmarks which currently exist on the StarExec clusters. This means that we can test on them without using of our storage quota, which we need for storing solvers, results and some other CNF files which we use for testing.

A thing to note about table 6.3 is that IsaSAT and Kissat rejects, due to discrepancies in the CNF-file, 100 benchmarks which are known to be

SAT. These problems also revealed a soundness issue in MicroSat, which reported these as being UNSAT, even though they are SAT. They also cause `versat` to crash. We therefore also present 6.4, which has these benchmarks removed.

6.4 Discussion of results

Discussion of the Robinson results

As we can see from table 6.1, Robinson is consistently around 3 to 5 times faster than TrueSAT. We believe this is in part due to Robinson being written in Rust, which in general is faster than C#, and in part due to the problem set not favouring the improvements which TrueSAT implements, and which we do not implement. The first difference between Robinson and TrueSAT is in terms of variable heuristic. TrueSAT implements the Maximum Occurrences in clauses of Minimum Size (MOMS) variable ordering heuristic, whereas Robinson simply does a count of the number of occurrences of all the variables on solver entry, and uses that as its order. As all the clauses are of the same length, MOMS becomes a dynamic maximum occurrence heuristic.

The second optimization which TrueSAT implements, and we do not, is to keep a count of how many literals are satisfied in each clause at all times. This enables more efficient identification of clauses which are unit than having no scheme, but is more expensive than 2WL. As we went straight to 2WL, we do not know how efficient such a scheme is. We believe much of the gain from this scheme is lost due to the fact that all of the clauses have a length of 3, and thus visiting all of the clauses to check whether they are unit is not very expensive.

We are not all that surprised that Robinson is faster on the pigeonhole problems as well, as these also contain short clauses. We also do not see how the slight difference in decision heuristic would be beneficial, as we have to explore the entirety of the search space anyways. What we however are somewhat surprised by is that CreuSAT performs so well on the pigeonhole problems in general, and the hole10 problem in particular. This is likely in part carried by the fact that CreuSAT in general has slightly faster code than Robinson, for instance by using unsafe indexing methods, which Robinson does not. We also see that both Robinson and TrueSAT are completely outclassed by CreuSAT on the random 3SAT problems — an expected result, which highlights the performance difference between CDCL and DPLL solvers.

Verifying Robinson is about twice as fast as verifying TrueSAT when running single threaded, and about four times as fast when running multi-threaded. This is much less of a difference than expected, as the version described in [1] took 13 minutes to verify. They state in the same paper that it used to take 2 hours, which indicates that they have put in an effort to reduce the verification time. We have not looked into reducing the time

to verify Robinson, but it may be that the same techniques which have benefited TrueSAT may benefit Robinson as well.

Discussion of the CreuSAT results

There are a couple of things to note about the performance results. The first is that, though there is still some more to go to become competitive with the state-of-the-art, CreuSAT is able to solve a substantial amount of the problems. As we can see from table 6.4, CreuSAT solves 85 more SAT instances than the verified solver `versat`, and 30 less than the verified solver IsaSAT. We can also see that it solves 17 more UNSAT instances than `versat`, and 51 less UNSAT instances than IsaSAT. This brings us to the second thing to note: there is a substantial difference between the amount of solved SAT instances and the amount of solved UNSAT instances.

We can see from the table that this is the case for other solvers as well, though the difference is more pronounced for CreuSAT. This suggests that it may in part be explained as an artifact of the problem set, and in part as a defect of CreuSAT. The fact that Kissat exhibits this behaviour to a lesser degree, may suggest that there is substantial gain in doing preprocessing, which most of the other solvers do not do.

We also believe that the difference may be due to two other reasons: lack of clause minimization, and a suboptimal heuristic for restarts and clause deletions. Indeed, an unverified extension of CreuSAT with a better heuristic solves 17 more UNSAT instances, while only solving 2 more SAT instances. Though heuristics in general are considered easy to prove, verifying this better heuristic would require making a design change. As we are still considering whether we want to make this change, we have not prioritized verifying this improvement.

As for the lack of clause minimization: we believe this would yield a substantial improvement. The smaller clauses are beneficial as they, in addition to having fewer literals to visit, also prune the search space more efficiently. This helps for both the SAT and the UNSAT problems, but should boost UNSAT performance more than it boosts SAT performance. A formula being UNSAT corresponds exactly to a series of resolution steps from the input formula to the empty clause, and having shorter clauses results in this proof being shorter. For the SAT case we still have to at some point experience a sufficiently long period without restarts that we manage to reach a satisfying assignment.

We believe these two changes would bring the solver closer to the state-of-the-art in terms of performance, especially with regards to UNSAT problems. That being said, we are very content with the current performance, especially when considering the size of the program and the size of the proof. CreuSAT manages to achieve a performance which is better than `versat` with less than half as many lines of code. Comparing the proof overhead against IsaSAT is a bit more difficult, as IsaSAT has better performance than CreuSAT.

We have not been able to get IsaSAT-30 to work on the StarExec clusters, but, we conjecture, based on the results of [20], where IsaSAT-30 solved 801 problems, and MicroSat solved 1018 problems, that we are around the same performance as IsaSAT-30, potentially slightly above. IsaSAT-30 consisted of a bit more than 104 thousand lines of code, which is 100 thousand lines more than CreuSAT. We have not been able to prove IsaSAT-30, but we do not find it likely that it proves in less than 15 minutes, which is the time CreuSAT takes, when modern IsaSAT requires a little over two hours.

Chapter 7

Conclusion

In this chapter, we summarize the main contributions of the thesis. After that, we discuss CREUSOT, and revisit the belief which lead to this thesis. Then we will look at some of the related work. Finally we discuss the future of CreuSAT.

7.1 Summary of Contributions

We identify the following contributions:

- We present, to the best of our knowledge, the largest proof of Rust code to date.
- We present a positive example showing the efficacy of the prophetic encoding of mutable pointers, and of CREUSOT in particular. Its robustness can be seen both in reduced verification effort and in reduced verification times.
- We present the first verified SAT solver which is able to solve a substantial amount of problems from recent SAT competitions without the use of interactive theorem provers and code synthesis. This is notable both in terms of showing the viability of deductive verification in general, and of Rust as a target for deductive verification efforts.
- We present the second fastest verified SAT solver to date.

7.2 Discussion of CREUSOT

This thesis has its origin in the belief that CREUSOT, or a tool like CREUSOT, could, through leveraging the Rust type system, offer improvements over existing methods. We believe that this thesis provides substantial evidence for this belief, and have thus increased our confidence in this initial belief. We would in that regard like to do a short discussion on our experience with using CREUSOT. This section is based on our experience and is therefore subjective.

The success of this thesis is in large part because of Rust and because of CREUSOT, but it also in part despite them. CREUSOT did not have support for vectors until October of last year, and there are many features which either are not supported, or are just now being supported[1]. There are still many opportunities to achieve nonsensical errors in WHY3, and we still manage to find new ways to get CREUSOT to crash. There is also some to be desired with regards to the user experience, and the resources for learning CREUSOT have until now been close to non-existent.

Though none of this is desirable, they are also the sort of thing which it is both possible to overcome, and possible to solve. It is quite remarkable how much CREUSOT has improved since RustVerify of last year, and it seems to be likely that progress will continue. Most of the faults are at the surface level, and the core is solid. CREUSOT generates VCs which check out quickly, in addition to having access to multiple SMT solvers for the more challenging VCs. For most VCs it is possible to quickly figure out if your proof is correct, or, in the case where it is not, to figure out what is missing. One can then focus on the VCs which take some more time, with the confidence that it is more likely to be an error in the proof, than that the SMT solvers need some more time.

7.3 Related work and conclusion

Verification of Rust code

Much of the verification effort in relation to Rust has been done with regards to verifying the foundation of the language, for instance with RustBelt [28]. There has also been made a fair amount of verification tools, some of which are aimed at program verification. Of these, there are a couple of tools aimed at verifying safe Rust code, and which support a sufficiently large subset of the Rust programming language that they can be used for the same purposes as CREUSOT. The first is the Prusti [54] verification tool, which is based on the Viper [44] verification infrastructure, and the second is the Aeneas [25] verification toolchain.

Prusti is on the surface level quite similar to CREUSOT, being a deductive verification tool where one annotates the source code. It is however based on separation logic, and they do thus model borrows as *pledges*, instead of by using the prophetic *final* (ˆ) operator. What we consider to be a strength of CREUSOT when compared to Prusti, is the access to multiple SMT solvers, as well as manual tactics to aid the SMT solvers when needed. We do not know of any project based on Prusti which is similar in size to this one, though Prusti has a substantial amount of tests in their repository, and has also been used for instance in the VerifyThis Competition,

Aeneas is a new tool which is based on the modelling the termination of a mutable borrow as a backwards function. They do not do program

[1] It has jokingly been said that CreuSAT is probably the world's fastest SAT solver without for-loops. We are not sure IsaSAT has any either, so maybe we are in second place.

verification through the annotation of source code, opting rather for *extrinsic* proofs, for instance in F* or Coq. Another difference is that their approach is independent of the concrete implementation of the borrow checker, meaning that they can verify programs which the current borrow checker rejects, due to it being too conservative. They implement a low-level, resizing hash table as their chief case study. In the case that there exists more work which is based on Aeneas, then we do not know of it.

This means that this thesis represents, to the best of our knowledge, the largest proof of Rust code to date, in part by default. We believe that we make a strong case of the efficacy of verifying Rust code, and hope that this thesis inspires others to use either CREUSOT, Prusti, Aeneas, or some other tool, to verify Rust code.

Formal verification of SAT solvers

The mechanized verification effort of modern SAT solving algorithms dates back to the PhD thesis of Filip Marić [35], which was improved upon in later work [34][36]. The work of Marić et al. is based on a shallow embedding into Isabelle/HOL. In the same period, Lescuyer and Conchon developed a formalization in Coq [32], and Shankar and Vaucher verified a description of a modern DPLL procedure [52]. These early works either did not offer an executable version, as is the case for Shankar and Vaucher's work, or did not prioritize the performance of the solver. They were focused on creating a proof of the DPLL/CDCL algorithm, whereas we have been focused on creating an executable SAT solver with high performance and high correctness guarantees.

A solver which is more similar to CreuSAT in that regard, `versat`, which, to the best of our knowledge, is the first verified solver able to solve a substantial amount of problems from the SAT competitions. As we demonstrated in the previous chapter, we achieve a better performance than `versat`, especially with regards to satisfiable instances, requiring substantially less proof code to achieve this result. We also compare favourably in terms of the properties we prove, as `versat` only gives a static guarantee of the correctness of the UNSAT result, based on the argument that checking whether a given model is SAT can be done in linear time. `versat` also does the same trade-off as CreuSAT in sacrificing completeness for performance, and does thus have the same run-time check to circumvent the proving of 2WL.

A solver which does prove completeness, and which also focuses on performance, is the IsaSAT solver, which we also looked at in the previous chapter. There we saw that it outperforms CreuSAT, and is thus, to the best of our knowledge, currently the fastest verified SAT solver. We also saw that it requires around 150 thousand lines of proof code to achieve this result, and that verifying the solver takes hours, instead of minutes, as is the case for CreuSAT.

It is still too early to say if the reduced code size is beneficial, and if

maintaining a fast proving time is doable while further properties and optimizations are proven. What we do believe is an advantage, though, is being "code first". Fleury et al. notes in [21] that they spent 2 weeks on implementing and proving a change in the conflict-clause representation, only to have it be slower. We do not have to prove code to test its efficacy, which should yield faster iteration time. Another gain here is that we can prove our correctness guarantees incrementally: first safety, then soundness, then completeness, whereas Isabelle has to be given a specification which is correct and which terminates.

This is, to the best of our knowledge, a capability for verified CDCL SAT solvers which has not existed before, and ties in with the fact that we have targeted an imperative implementation directly. The full advantages and eventual disadvantages of this approach remain to be seen, though we are optimistic, both in terms of further improvements to proof and to performance.

Before we revisit the thesis statement, we will briefly mention TrueSAT, which we compared to Robinson in the previous chapter. TrueSAT is verified in Dafny, and the structure of the proof is similar to the structure of the proof of Robinson. As we saw, Robinson was around 3 to 5 times faster than TrueSAT, and both Robinson and TrueSAT were over an order of magnitude slower than CreuSAT. This is the case, despite TrueSAT having both count-based identification of unit clauses, as well as a slightly better decision heuristic. We thus view Robinson as a more performant realization of the goal of achieving a verified SAT solver through deductive verification, and CreuSAT as a further improvement upon Robinson.

Finally, we look at the thesis statement which we set out to explore in the start of the thesis:

- To investigate whether it is possible to create a formally verified SAT solver with comparative performance to those based on proof assistants, while targeting an imperative implementation directly.

Considering that we are currently faster than all but one of the verified SAT solvers which are based on proof assistant, we believe that it is indeed possible to create a formally verified SAT solver with comparative performance to those based on proof assistants, while targeting an imperative implementation directly.

7.4 Future work

We will in this section look at work that could be done in extension of Robinson and CreuSAT.

7.4.1 Extending Robinson

With the exception of solving pigeon hole problems efficiently, the need for fast DPLL solvers is about non-existent. It is therefore not very likely

that we will ever improve Robinson substantially, but it may be that others may want to. We identify the following areas of improvement, ranked in increasing order of difficulty:

- Implementing and proving better decision heuristics.
- Implementing and proving pure literal elimination.
- Implementing and proving count based unit propagation.
- Making the solver iterative by implementing and proving a trail mechanism.
- Implementing and proving two watched literals.

Robinson should be much easier to understand and to do changes in than CreuSAT, and should be a decent starting point for those wishing to try out proving SAT solvers with CREUSOT.

7.4.2 Improving CreuSAT

Though we are very happy with what we have managed to achieve, CreuSAT is a piece of software, and, as it usually is with software, there are possibilities for improvements. In this section we discuss various improvements to CreuSAT which we would like to pursue in the future.

7.4.2.1 Proving completeness

Though we believe sacrificing completeness for performance has been the correct choice, a solver which is proven complete is obviously better than one which is not. Proving completeness for CreuSAT would require proving the semantics of the 2WL scheme. This took Mathias Fleury 15 months to do, after already having the abstract CDCL calculus, and experience with Isabelle [29]. We do not believe that proving this in CREUSOT will be easy, but we conjecture that it will take less than 15 months to do. A key realization here is that completeness can be ensured by proving the correctness of 2WL for the initial formula, and then simply maintain that the original formula is always watched. It is very likely that we will go for such a scheme, as it means that further watches, as well as the addition and deletion of learned clauses can be treated entirely separately.

7.4.2.2 Improving performance and adding features

Though CreuSAT has very solid performance when considering it being verified, it has still some more to go to become competitive with the state-of-the-art. An aspect which became apparent once we started benchmarking, is that we should have benchmarked much earlier, and tuned our solver to do better at the benchmarks. We believe there is a decent amount of gain to be had simply in making the code more efficient, improving the layout of the data structures, in addition to adding further optimizations.

The performance optimizations which we consider to be of the highest yield at the moment:

- Building up more possibilities in the logic to model actions as resolution. This desire is based on the insight that most actions which are done by a CDCL SAT solver can be modelled as resolution. In addition to being useful in the clause learning algorithm, a better resolution framework would enable self-subsuming resolution. Many actions, such as simplification after learning a unit clause, or learned clause minimization, should be possible to efficiently model as doing self-subsuming resolution. We will likely build the logical capacity of modelling actions as resolution together with concrete implementations of features such as learned clause minimization and simplification.
- Moving to an arena-based memory management scheme, with clauses and clause header represented together, as well as special handling of clauses of various length. This is not something which has been looked at much, but it should be possible by proving the correctness of the representation and defining the correct abstraction barriers for the rest of the specification to use.
- Implementing the target phases technique of CaDiCaL [11]. We did not realize the significance of this technique until quite recently, but it should yield quite a substantial improvement on SAT instances without a substantial amount of work in implementing and proving. Target phases is essentially an optimization on phase saving, with the addition of adding "phases" to the search processes. As anything related to decisions can be proven by simply proving safety, this should be an optimization with very high yield.

There are also various other techniques, many of which offer small improvements, which we would like to look at implementing and proving. To identify which of these to prioritize, we will have to experiment with an unverified solver.

7.4.2.3 Reducing the time needed to prove the solver

We have demonstrated CREUSOT's ability to generate obligations which require little resources to prove. We have however not expended much resources into making these obligations prove even faster. We believe that there is a significant possibility for improvements, both for making proof discovery go faster, and for making the proof with a given proof tree go faster. We believe some of this can be gained through changes in CreuSAT, some of this can be gained through changes in CREUSOT, and some through tuning the heuristics of WHY3. This work is largely out of scope for this thesis, but is something which we would like to explore in the future.

Bibliography

[1] Cezar-Constantin Andrici and Ştefan Ciobâcă. 'Who Verifies the Verifiers? A Computer-Checked Implementation of the DPLL Algorithm in Dafny'. In: *arXiv:2007.10842 [cs]* (19th July 2020). arXiv: 2007.10842. URL: http://arxiv.org/abs/2007.10842 (visited on 21/11/2021).

[2] Gilles Audemard and Laurent Simon. 'On the Glucose SAT solver'. In: *International Journal on Artificial Intelligence Tools* 27 (Feb. 2018), p. 1840001. DOI: 10.1142/S0218213018400018.

[3] Gilles Audemard and Laurent Simon. 'Predicting learnt clauses quality in modern SAT solvers'. In: *Twenty-first international joint conference on artificial intelligence*. 2009.

[4] Gilles Audemard and Laurent Simon. 'Refining restarts strategies for SAT and UNSAT'. In: *International Conference on Principles and Practice of Constraint Programming*. Springer, 2012, pp. 118–126.

[5] John G. Barnes. *Programming in ADA*. Addison-Wesley Longman Publishing Co., Inc., 1984. 340 pp.

[6] Clark Barrett et al. 'CVC4'. In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 171–177. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_14.

[7] Patrick Baudin et al. 'ACSL: ANSI C Specification Language'. In: *CEA-LIST, Saclay, France, Tech. Rep. v1* 2 (2008).

[8] Armin Biere. 'Splatz, lingeling, plingeling, treengeling, yalsat entering the sat competition 2016'. In: *Proc. of SAT Competition* (2016), pp. 44–45.

[9] Armin Biere and Mathias Fleury. 'Chasing target phases'. In: *Proceedings of Pragmatics of (SAT)* (2020).

[10] Armin Biere and Andreas Fröhlich. 'Evaluating CDCL variable scoring schemes'. In: *International conference on theory and applications of satisfiability testing*. Springer, 2015, pp. 405–422.

[11] Armin Biere et al. 'CaDiCaL, kissat, paracooba, plingeling and treengeling entering the SAT competition 2020'. In: *Proc. of SAT Competition* B-2020-1 (2020), p. 4.

[12] Yves Bertot Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2004. 472 pp. ISBN: 978-3-662-07964-5.

[13] Sylvain Conchon et al. 'Alt-Ergo 2.2'. In: SMT Workshop: International Workshop on Satisfiability Modulo Theories. 12th July 2018. URL: https://hal.inria.fr/hal-01960203 (visited on 26/10/2021).

[14] Stephen A. Cook. 'The complexity of theorem-proving procedures'. In: *Proceedings of the Third annual ACM symposium on Theory of computing*. 1971, pp. 151–158.

[15] Martin Davis, George Logemann and Donald Loveland. 'A Machine Program for Theorem-Proving'. In: *Commun. ACM* 5.7 (July 1962). Place: New York, NY, USA Publisher: Association for Computing Machinery, pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557. URL: https://doi.org/10.1145/368273.368557.

[16] Martin Davis and Hilary Putnam. 'A Computing Procedure for Quantification Theory'. In: *J. ACM* 7.3 (July 1960). Place: New York, NY, USA Publisher: Association for Computing Machinery, pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034. URL: https://doi.org/10.1145/321033.321034.

[17] Xavier Denis, Jacques-Henri Jourdan and Claude Marché. *The CREUSOT Environment for the Deductive Verification of Rust Programs*. Research Report RR-9448. Inria Saclay - Île de France, 2021. URL: https://hal.inria.fr/hal-03526634.

[18] Niklas Eén and Niklas Sörensson. 'An Extensible SAT-solver'. In: *Theory and Applications of Satisfiability Testing*. Ed. by Enrico Giunchiglia and Armando Tacchella. Vol. 2919. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 502–518. ISBN: 978-3-540-24605-3. DOI: 10.1007/978-3-540-24605-3_37.

[19] Jean-Christophe Filliâtre and Andrei Paskevich. 'Why3 — Where Programs Meet Provers'. In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Red. by David Hutchison et al. Vol. 7792. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128. ISBN: 978-3-642-37035-9 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_8. URL: http://link.springer.com/10.1007/978-3-642-37036-6_8 (visited on 12/05/2022).

[20] Mathias Fleury. 'Optimizing a Verified SAT Solver'. In: *NASA Formal Methods*. Ed. by Julia M. Badger and Kristin Yvonne Rozier. Vol. 11460. Lecture Notes in Computer Science. Springer International Publishing, 2019, pp. 148–165. ISBN: 978-3-030-20652-9. DOI: 10.1007/978-3-030-20652-9_10.

[21] Mathias Fleury, Jasmin Christian Blanchette and Peter Lammich. 'A verified SAT solver with watched literals using imperative HOL'. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. New York, NY, USA: Association for Computing Machinery, 8th Jan. 2018, pp. 158–171. ISBN: 978-1-4503-5586-5. DOI: 10.1145/3167080. URL: https://doi.org/10.1145/3167080 (visited on 21/11/2021).

[22] Ian P. Gent. 'Optimal implementation of watched literals and more general techniques'. In: *Journal of Artificial Intelligence Research* 48 (2013), pp. 231–252.

[23] John Hatcliff et al. 'Behavioral interface specification languages'. In: *ACM Computing Surveys* 44.3 (June 2012). Number: 3, pp. 1–58. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/2187671.2187678. URL: https://dl.acm.org/doi/10.1145/2187671.2187678 (visited on 02/05/2022).

[24] Marijn J.H. Heule, Matti Järvisalo and Martin Suda. 'SAT competition 2018'. In: *Journal on Satisfiability, Boolean Modeling and Computation* 11.1 (2019). Publisher: IOS Press, pp. 133–154.

[25] Son Ho and Jonathan Protzenko. *Aeneas: Rust Verification by Functional Translation*. Version 1.0. May 2022. DOI: 10.5281/zenodo.6597014. URL: https://doi.org/10.5281/zenodo.6597014.

[26] Charles AR Hoare. 'Proof of a program: FIND'. In: *Communications of the ACM* 14.1 (1971). Publisher: ACM New York, NY, USA, pp. 39–45.

[27] *isafol / isafol — Bitbucket*. URL: https://bitbucket.org/isafol/isafol/src/master/ (visited on 13/06/2022).

[28] Ralf Jung et al. 'RustBelt: securing the foundations of the Rust programming language'. In: *Proceedings of the ACM on Programming Languages* 2 (POPL 27th Dec. 2017), 66:1–66:34. DOI: 10.1145/3158154. URL: https://doi.org/10.1145/3158154 (visited on 21/11/2021).

[29] Peter Lammich, Mathias Fleury and Jasmin C. Blanchette. 'A Verified SAT Solver with Watched Literals Using Imperative HOL'. Matryoshka 2018. Amsterdam, The Netherlands, 26th June 2018. URL: https://matryoshka-project.github.io/matryoshka2018/slides/Matryoshka-2018-Fleury-A-Verified-SAT-Solver.pdf (visited on 07/06/2022).

[30] Gary T. Leavens, Albert L. Baker and Clyde Ruby. 'Preliminary design of JML: A behavioral interface specification language for Java'. In: *ACM SIGSOFT Software Engineering Notes* 31.3 (2006). Publisher: ACM New York, NY, USA, pp. 1–38.

[31] K. Rustan M. Leino. 'Dafny: An automatic program verifier for functional correctness'. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2010, pp. 348–370.

[32] Stéphane Lescuyer and Sylvain Conchon. 'A reflexive formalization of a SAT solver in Coq'. In: *21st International Conference on Theorem Proving in Higher Order Logics*. 2008, pp. 64–75.

[33] Michael Luby, Alistair Sinclair and David Zuckerman. 'Optimal speedup of Las Vegas algorithms'. In: *Information Processing Letters* 47.4 (1993). Publisher: Elsevier, pp. 173–180.

[34] Filip Marić. 'Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL'. In: *Theoretical Computer Science* 411.50 (12th Nov. 2010), pp. 4333–4356. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2010.09.014. URL: https://www.sciencedirect.com/science/article/pii/S0304397510004937 (visited on 21/11/2021).

[35] Filip Marić. 'Formalization and Implementation of Modern SAT Solvers'. In: *Journal of Automated Reasoning* 43.1 (June 2009), pp. 81–119. ISSN: 0168-7433, 1573-0670. DOI: 10.1007/s10817-009-9127-8. URL: http://link.springer.com/10.1007/s10817-009-9127-8 (visited on 21/11/2021).

[36] Filip Marić and Predrag Janičić. 'Formal Correctness Proof for DPLL Procedure'. In: *Informatica* 21.1 (1st Jan. 2010). Publisher: Institute of Mathematics and Informatics, pp. 57–78. ISSN: 0868-4952. URL: https://content.iospress.com/articles/informatica/info21-1-05 (visited on 21/11/2021).

[37] J. P. Marques-Silva and K. A. Sakallah. 'GRASP: a search algorithm for propositional satisfiability'. In: *IEEE Transactions on Computers* 48.5 (1999), pp. 506–521. DOI: 10.1109/12.769433.

[38] Yusuke Matsushita, Takeshi Tsukada and Naoki Kobayashi. 'RustHorn: CHC-Based Verification for Rust Programs'. In: *Programming Languages and Systems*. Ed. by Peter Müller. Lecture Notes in Computer Science. Springer International Publishing, 2020, pp. 484–514. ISBN: 978-3-030-44914-8. DOI: 10.1007/978-3-030-44914-8_18.

[39] Yusuke Matsushita et al. 'RustHornBelt: A Semantic Foundation for Functional Verification of Rust Programs with Unsafe Code'. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. event-place: San Diego, CA, USA. New York, NY, USA: Association for Computing Machinery, 2022, pp. 841–856. ISBN: 978-1-4503-9265-5. DOI: 10.1145/3519939.3523704. URL: https://doi.org/10.1145/3519939.3523704.

[40] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992. 594 pp. ISBN: 0-13-247925-7.

[41] F. Lockwood Morris and Cliff B. Jones. 'An early program proof by Alan Turing'. In: *IEEE Annals of the History of Computing* 6.2 (1984). Publisher: IEEE Computer Society, pp. 139–143.

[42] Matthew W. Moskewicz et al. 'Chaff: engineering an efficient SAT solver'. In: *Proceedings of the 38th annual Design Automation Conference*. DAC '01. New York, NY, USA: Association for Computing Machinery, 22nd June 2001, pp. 530–535. ISBN: 978-1-58113-297-7. DOI: 10.1145/378239.379017. URL: https://doi.org/10.1145/378239.379017 (visited on 20/11/2021).

[43] Leonardo de Moura and Nikolaj Bjørner. 'Z3: An Efficient SMT Solver'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[44] Peter Müller, Malte Schwerhoff and Alexander J. Summers. 'Viper: A verification infrastructure for permission-based reasoning'. In: *International conference on verification, model checking, and abstract interpretation*. Springer, 2016, pp. 41–62.

[45] Tobias Nipkow, Markus Wenzel and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[46] U. Norell. 'Towards a Practical Programming Languages Based on Dependent Type Theory'. PhD thesis. Department of Computer Science and Engineering, Chalmers, 2007.

[47] Peter O'Hearn. 'Separation logic'. In: *Communications of the ACM* 62.2 (2019). Publisher: ACM New York, NY, USA, pp. 86–95.

[48] Duckki Oe et al. 'versat: A verified modern SAT solver'. In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2012, pp. 363–378.

[49] Chanseok Oh. 'Between SAT and UNSAT: the fundamental difference in CDCL SAT'. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2015, pp. 307–323.

[50] Sam Owre, John M. Rushby and Natarajan Shankar. 'PVS: A prototype verification system'. In: *International Conference on Automated Deduction*. Springer, 1992, pp. 748–752.

[51] Lawrence Ryan. 'Efficient algorithms for clause-learning SAT solvers'. Master's thesis. Simon Fraser University, 2004.

[52] Natarajan Shankar and Marc Vaucher. 'The Mechanical Verification of a DPLL-Based Satisfiability Solver'. In: *Electronic Notes in Theoretical Computer Science*. Proceedings of the Fifth Logical and Semantic Frameworks, with Applications Workshop (LSFA 2010) 269 (22nd Apr. 2011), pp. 3–17. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2011.03.002. URL: https://www.sciencedirect.com/science/article/pii/S1571066111000594 (visited on 21/11/2021).

[53] Jan Smans, Bart Jacobs and Frank Piessens. 'Implicit dynamic frames'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 34.1 (2012). Publisher: ACM New York, NY, USA, pp. 1–58.

[54] Alexander J. Summers. 'Prusti: deductive verification for Rust (keynote)'. In: *Proceedings of the 22nd ACM SIGPLAN International Workshop on Formal Techniques for Java-Like Programs*. FTfJP 2020. New York, NY, USA: Association for Computing Machinery, 23rd July 2020, p. 1. ISBN: 978-1-4503-8186-4. DOI: 10.1145/3427761.3432348.

URL: https://doi.org/10.1145/3427761.3432348 (visited on 21/11/2021).

[55] Grigori S. Tseitin. 'On the complexity of derivation in propositional calculus'. In: *Automation of reasoning*. Springer, 1983, pp. 466–483.

[56] Hantao Zhang. 'SATO: An efficient propositional prover'. In: *International Conference on Automated Deduction*. Springer, 1997, pp. 272–275.

[57] Lintao Zhang and S. Malik. 'Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications'. In: *2003 Design, Automation and Test in Europe Conference and Exhibition*. 2003, pp. 880–885. DOI: 10.1109/DATE.2003.1253717.

[58] Lintao Zhang and Sharad Malik. 'Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications'. In: *2003 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 2003, pp. 880–885.