Formalization of Logical Calculi in Isabelle/HOL

Dissertation Draft

A draft to get the degree Doctor of Engineering of the Faculty of Mathematics and Computer Science of Saarland University

by Mathias Fleury

(last update of the draft: 2019-06-19 16:36:15+02:00)

Tag des Kolloqiums:	To be done
Dekan:	
Prüfungsausschuss:	
Vorsitzender	TBA
Berichterstatter	Dr. Jasmin Christian Blanchette
	Prof. Dr. Christoph Weidenbach
	TBA1
	TBA2
Akademischer Mitarbeiter	TBA

Abstract

I develop a formal framework for the conflict-driven clause learning (CDCL) calculus using the Isabelle/HOL proof assistant. The framework offers a convenient way to prove metatheorems and experiment with variants, including the Davis-Putnam-Logemann-Loveland (DPLL) calculus. The key point of my approach is the use of refinement.

I use the formalization to develop three extensions: First, an incremental solving extension of CDCL. Second, I verify an optimizing CDCL (OCDCL): Given a cost function on literals, OCDCL derives an optimal model with minimum cost. Finally, I work on model covering. Thanks to the CDCL framework I can reuse, these extensions are easier to develop.

Through a chain of refinements, I connect the abstract CDCL calculus first to a more concrete calculus, then to a SAT solver expressed in a simple functional programming language, and finally to a SAT solver in an imperative language, with total correctness guarantees. The imperative version relies on the two-watched-literal data structure and other optimizations found in modern solvers. I use the Isabelle Refinement Framework to automate the most tedious refinement steps. After that, I extend this work with further optimizations like blocking literals and the use of machine words as long as possible, before switching to unbounded integers to keep completeness.

Zusammenfassung

Acknowledgments

First and foremost, I want to thank my two supervisors Jasmin Blanchette and Christoph Weidenbach. On the one hand, Jasmin supervises me on a day-to-day basis. He is always ready comment on my (sometimes terrible) draft and he helped me a lot. On the other hand, Christoph made this work possible. He always has multiple interesting ideas, even though I did not have the time to implement most of them.

I am also grateful to all my colleagues at the MPI including Noran Azmy (who briefly shared an office with me), Martin Bromberger (who introduced me to the world of pen-and-paper games), Alberto Fiori (who TODO), Maximilian Jaroschek, Marek Košta (who was always ready to play "Kicker"), Sophie Tourret (who reminded me that emacs is the best text editor and motivated me to extend it in order to be able to use it with Isabelle), Marco Voigt (who understands how the MPI and the MPG actually works and help me understand it), Uwe Waldmann (who I thankfully did never have to meet him in his function as ombudsmen), Daniel Wand (who was always ready to have a chat). Finally, even if Ching Hoo Tang is not part of the MPI anymore, he now works for L4B that has an office on the same floor: He was always ready to exchange opinions on football.

While I could not meet my colleagues from Nancy on a daily basis, they also influenced my work, especially Stephan Merz, Pascal Fontaine, Daniel El Ouraoui, Haniel Barbosa, Hans-Jörg Schurr.

I obviously had discussions with other people. I gratefully thank Dmitriy Traytel (who supervised my master's thesis, that led to this thesis), Andreas Lochbihler (who has many ideas how to do large formalization and did a lot of setup related to code generation), and Peter Lammich (who has developed the Isabelle Refinement Framework and gave me advice on how to use Sepref when I got stuck).

Most of the text in this thesis has appeared previously in various publications. None of this work would have been possible without my coauthors Jasmin Blanchette, Peter Lammich, and Christoph Weidenbach. Many people suggested improvements on these publication: Simon Cruanes, Max Haslbeck, Marijn Heule, Benjamin Kiesl, Peter Lammich, Anders Schlichtkrull, Hans-Jörg Schurr, Mark Summerfield, Dmitriy Traytel, Petar Vukmirović, and every anonymous reviewer, in particular the reviewer who suggested the dual rail encoding for the optimizing CDCL.

Finally, I would like to thank my family for always supporting me, even when I scared them once more by not answering my phone.

Contents

1. Introduction **1**
1.1. Motivation . . . 1
1.2. Plan of the Thesis . . . 4
1.3. Contributions . . . 5

2. Isabelle **7**
2.1. Isabelle/Pure . . . 7
2.2. Isabelle/HOL . . . 8
2.3. Locales . . . 9
2.4. Isar . . . 10
2.5. Sledgehammer . . . 11
2.6. Isabelle/jEdit . . . 11

3. Conflict-Driven Clause Learning **15**
3.1. Abstract CDCL . . . 15
3.1.1. Propositional Logic . . . 16
3.1.2. DPLL with Backjumping . . . 16
3.1.3. Classical DPLL . . . 20
3.1.4. The CDCL Calculus . . . 21
3.1.5. Restarts . . . 22
3.2. A Refined CDCL towards an Implementation . . . 24
3.2.1. The New DPLL Calculus . . . 24
3.2.2. The New CDCL Calculus . . . 25
3.2.3. A Reasonable Strategy . . . 27
3.2.4. Connection with Abstract CDCL . . . 31
3.2.5. A Strategy with Restart and Forget . . . 32
3.2.6. Incremental Solving . . . 33
3.2.7. Backjump and Conflict Minimization . . . 34
3.3. A Naive Functional Implementation of CDCL, IsaSAT-0 . . . 34
3.4. Summary . . . 36

4. CDCL with Branch and Bounds **39**
4.1. Optimizing Conflict-Driven Clause Learning . . . 40

4.2. Formalization of OCDCL . . . 43
4.2.1. OCDCL and CDCL . . . 43
4.2.2. Branch-and-Bound Calculus, CDCL_{BnB} . . . 44
4.2.3. Embedding into CDCL . . . 46
4.2.4. Instantiation with weights, OCDCL_g . . . 47
4.2.5. OCDCL . . . 48
4.3. Optimal Partial Valuations . . . 49
4.4. Formalization of the Partial Encoding . . . 50
4.5. Solving Further Optimization Problems . . . 51
4.5.1. MAX-SAT . . . 51
4.5.2. A Second Instantiation of CDCL_{BnB}: Model Covering . . . 51
4.6. Extending CDCL . . . 53
4.6.1. Restricting CDCL or Adding Shortcuts . . . 53
4.6.2. More General Rules . . . 53
4.7. Summary . . . 54

5. The Two-Watched-Literal Scheme 55
5.1. Code Synthesis with the Isabelle Refinement Framework . . . 56
5.1.1. Isabelle Refinement Framework . . . 56
5.1.2. Sepref and Refinement to Imperative HOL . . . 59
5.1.3. Code Generation of Imperative Programs . . . 60
5.1.4. Sepref and Locales . . . 61
5.2. Watched Literals . . . 62
5.3. Refining the Calculus to an Algorithm . . . 66
5.4. Representing Clauses as Lists . . . 68
5.5. Storing Clauses Watched by a Literal: Watch Lists . . . 69
5.6. Generating Code . . . 71
5.7. Optimizations and Heuristics . . . 73
5.7.1. Variable Move to Front . . . 73
5.7.2. Conflict Clause as a Lookup Table . . . 76
5.7.3. Conflict Clause Minimization . . . 78
5.7.4. State Representation . . . 79
5.7.5. Fast Polarity Checking . . . 81
5.8. Evaluation . . . 83
5.9. Summary . . . 85

6. Optimizing my Verified SAT Solver IsaSAT 87
6.1. Refactoring IsaSAT . . . 88
6.2. Adding Blocking Literals . . . 92
6.3. Improving Memory Management . . . 93

6.4. Implementing Restarts and Forgets 95
6.5. Using Machine Integers 97
6.6. Evaluation 98
6.7. Extracting Efficient Code 100
6.8. Detailed TWL Invariants in Isabelle 102
6.9. Extending IsaSAT 104
6.10. Summary 105

7. Discussion and Conclusion **107**
7.1. Discussion and Related Work 107
7.2. Summary 116
7.3. Future Work 116

A. More Details on IsaSAT and Other SAT Solvers **119**
A.1. Clauses 119
A.1.1. IsaSAT 119
A.1.2. Glucose 120
A.1.3. CaDiCaL 120
A.2. Watch Lists and Propagations 121
A.3. Decision Heuristics 122
A.4. Clause Simplification 122
A.5. Forget 123
A.6. Clause Minimization 123

List of Figures

2.1. Isabelle/jEdit screenshot: Isabelle could not verify the proof highlighted in red . . . 12

3.1. Connections between the abstract calculi . . . 23
3.2. Proof tree for the clauses $A \vee B$, $\neg A \vee C$, and $B \vee \neg C$, where red edges indicate conflicts with one of the three clauses . . . 29
3.3. Connections involving the refined calculi . . . 33

4.1. OCDCL transitions and corresponding CDCL transitions for $N = \{P \vee Q\}$ with cost P = cost $Q = 1$ and cost $\neg P$ = cost $\neg Q$ = 0 where the horizontal lines separate two successive CDCL run 45

5.1. Evolution of the 2WL data structure on a simple example . . . 63
5.2. Example of the VMTF heuristic before and after bumping. . . . 75
5.3. Conversion from the lookup table to a clause, assuming $C \neq$ None . . . 77
5.4. Comparison of performance on the problems classified easy or medium from the SAT Competition 2009 . . . 83
5.5. Summary of the layers used to generate code . . . 85

6.1. Comparison of the code of Ignore rule in Algo before and after refactoring . . . 89
6.2. Different ways of writing the proof that $\mathsf{PCUI}_{\mathsf{list}}$ from Figure 6.1a refines $\mathsf{PCUI}_{\mathsf{algo}}$ . . . 89
6.3. Refinement of the rule Ignore with blocking literals from Algo to WList . . . 92
6.4. Example of arena module with two clauses $A \vee B \vee C$ (initial clause, 'init') and $\neg A \vee \neg B \vee C \vee D$ (learned clause, 'learn') . . 93
6.5. Skipping deleted clauses during iteration over the watch list . . 95
6.6. Performance of some SAT solvers (N/A if no simplification is performed by default) . . . 99
6.7. Benchmarks of variants of IsaSAT-30 before fixing the forget heuristic . . . 99

7.1. Length of various parts of the formalization (in thousands lines of code, not accounting for empty lines) 107

1. Introduction

This thesis describes my formalization of the conflict-driven-clause-learning procedure, how I use the developed framework to capture two variants without redefining or reproving most invariants, and how I extend it to a fully verified executable solver called IsaSAT.

1.1. Motivation

Researchers in automated reasoning spend a substantial portion of their work time developing logical calculi and proving metatheorems about them. These proofs are typically carried out with pen and paper, which is error-prone and can be tedious. Today's proof assistants are easier to use than their predecessors and can help reduce the amount of tedious work, so it makes sense to use them for this kind of research.

In this spirit, a few colleagues and I have started an effort, called IsaFoL (Isabelle Formalization of Logic) [8], that aims at developing libraries and a methodology for formalizing modern research in the field, using the Isabelle/HOL proof assistant [93,94]. The initial emphasis of the effort is on established results about propositional and first-order logic. The inspiration for formalizing logic is the IsaFoR (Isabelle Formalization of Rewriting) project [114], which focuses on term rewriting. My contributions to the overall project are results on propositional logic, the development of libraries (e.g., for partial models), and SAT solving.

The objective of formalization work is not to eliminate paper proofs, but to complement them with rich formal companions. Formalizations help catch mistakes whether superficial or deep, in specifications and theorems; they make it easy to experiment with changes or variants of concepts; and they help clarify concepts left vague on paper.

SAT solvers are widely used to find counter-examples or prove correctness in various fields. Given a propositional problem in conjunctive normal form, they answer SAT if there is a *model* (i.e., an assignment of every atom to either true or false such that every clause has a true literal) and UNSAT if there is no model of the clauses of the propositional problem. They have for example been used to solve long-standing open problems such as the *Boolean Pythagorean Triples Problem* [51] and *Schur Number Five* [53]. The approach used to solve both problems is to encode them into a satisfiability (SAT) problem, before dividing these very large problems into several smaller and easier problems. Then a SAT solver is run on each smaller problem. For each problem, the answer can either be satisfiable (the witness is a model) or unsatisfiable (the witness is a proof of false). While the witness for satisfiable is easy to check, the generated proof is harder to certify. This is done with another tool to rule out that a bug has occurred in the SAT solver while solving the problem. This is a common approach to increase the trustworthiness of SAT solvers: returning independently verifiable *proofs* that certify the correctness of their answers. To increase the trust even further, proof checkers have also been verified in proof assistants [33,69] to prove the correctness. However, the production of proofs does not provide total correctness guarantees: Although a correct proof guarantees that a solver has produced a correct result, it is not guaranteed that the solver will be able to produce a proof in the first place.

The CDCL (conflict-driven clause learning) procedure is the core calculus implemented in most SAT solvers. The input problem is a finite set of clauses, where a clause is a (finite) multiset of positive or negative Boolean variables. The aim is to assign truth values to the variables such that all clauses are true. CDCL gradually builds a partial assignment until either a model of the clauses is found or the constraints are so restrictive that no model can exist. The partial assignment can be extended by propagating information or setting (*deciding*) a new value. Decisions are arbitrary choices and the opposite choice has to be explored later. If the partial model cannot be extended to a model anymore because there is a *conflict* between the problem and the model, the latter has to be adjusted: The conflict is analyzed to derive information from it, especially how to repair the current model and how to avoid ending up in the same dead end.

Marco wants an example here

In this thesis, I present my formalization of CDCL based on Weidenbach's forthcoming textbook, tentatively called *Automated Reasoning—The Art of Generic Problem Solving*. I derive his presentation CDCL as a refinement of Nieuwenhuis, Oliveras, and Tinelli's abstract presentation of CDCL [92]. I start with a family of formalized abstract DPLL (Davis–Putnam–Logemann–Loveland) [35] and CDCL [7,16,87,90] transition systems. These calculi are presented as non-deterministic systems and all aspects that do not need to be specified are kept unspecified, like how decisions are made. All calculi are proved sound and complete, as well as terminating under a reasonable strategy. My first extension is the development of an incremental version of CDCL. If CDCL has found a model, it is now possible to add new constraints to find for example a different model (e.g., in order to enumerate all models).

After developing a verified SAT solver, I build upon the formalization in two different directions. First, I use the framework to capture two extensions of CDCL. The aim of the IsaFoL project is not only to develop tools, but also to be able to develop variants and extensions of already defined calculi. In this spirit, I add rules to the CDCL calculus to find a total model with minimal cost. Given a cost function on literals, the aim is to find an optimal total model. In the context of a product configuration system, one possible application is finding the cheapest product that fulfills some criteria (e.g., what is the cheapest car that has a GPS and no seat heating). Furthermore, I add another set of rules to CDCL to find a set of models, such that every option is true at least once. In the context of a produce configuration system, this answers the question whether every option can be taken or if the constraints are so restrictive that some options are impossible to take. Both variants are formalized within a common framework, CDCL with branch-and-bound: The normal CDCL searches for a better model (either one with lower cost or one new covering model). In turn, this better model is added and is used to exclude several new models ensuring that a subsequent CDCL run does not find the model again. After that, CDCL can be called again to either find a new model or to conclude that there is no further model anymore. The calculus can also use the additional information to cut branches by finding a conflict because some part of the search space has already been excluded. Conflicts are analyzed by the normal CDCL rules to adjust the model. While formalizing the optimizing CDCL, I have discovered that the calculus presented in *Automated Reasoning* was flowed, because optimality for partial models was claimed, but does not hold. To solve this issue, I formalize an encoding to reduce the search for optimal partial models to optimal *total* models.

Furthermore, I extend the formalization towards executable code and efficiency. State-of-the-art SAT solvers are extremely efficient on practical problems. This is both due to the extreme optimization of their source code and due to theoretical ideas to efficiently identify propagations and conflicts. The *two-watched-literal* scheme [90] and *blocking literals* [30] are examples of a theoretical idea. The key point is to reduce the tracking of the status of clauses (Can one propagate information? Is it a conflict?) to the tracking of only these two literals per clause. I develop a new calculus that includes two ideas. It is still presented as a non-deterministic transition system and inherits correctness and termination from CDCL. Several versions of two watched literals have been developed and I verify the version used in modern SAT solvers.

Finally, I refine this non-deterministic transition system to deterministic code written with imperative aspects, like arrays. The resulting SAT solver is called IsaSAT. Heuristics replace the non-deterministic aspects by deterministic code, such as which literal to decide instead of the specification that any unset literal works. Moreover, complicated data structures are used. The resulting code can be exported from the Isabelle/HOL proof assistant to functional languages like Haskell, OCaml, Scala, or Standard ML, with imperative features like arrays. It is interesting how efficient or inefficient the generated code it. Since proof checkers and SAT solvers share similar techniques and data structures, they face similar efficiency challenges, and some of the techniques presented here to optimize the verified SAT solver are applicable to checkers too.

I am not the first to verify an executable SAT solver: There are two other formalization that also included the two-watched literals, namely by Marić [85] and by Oe et al. [97]. First, Marić [85] has not implemented blocking literals and verified a different version of the two-watched-literals scheme. The difference is the interleaving of propagations and update of clauses to make the key invariant true: I propagate *during* the updates, and not update some clauses, then propagate, then update other clauses, and so on. Moreover, he has not developed efficient data structure nor heuristics. The other solver is `versat` by Oe et al. [97]. It has some efficient data structure (including two watched literals but excluding blocking literals). Only the answer UNSAT is verified, whereas the answer SAT requires additional runtime checks. Termination is not proven.

1.2. Plan of the Thesis

The thesis is organized as follows.

- In Chapter 2, I introduce Isabelle/HOL, the interactive theorem prover I use for the formalization.

- Chapter 3 presents the formalization of CDCL: I formalize two variants, one based on Nieuwenhuis et al.'s account [92] and the other based on Weidenbach's presentation [119]. The latter is more concrete and describes some aspects that are left unspecified in the former presentation. I connect these two presentations, making it possible to inherit the proof of termination. The termination result is stronger than Weidenbach's version. The text is based on an article [20].

- Chapter 4 shows one extension of CDCL, namely a CDCL with branch-and-bound. I instantiate this abstract calculus with two calculi: One that derives an optimal total model and another that finds a set of covering models. I also formalize an encoding to reduce the search from total to the search of optimal partial models.

- In Chapter 5, I refine Weidenbach's variant of CDCL with the two-watched-literal scheme. The resulting calculus is still presented as an abstract non-deterministic transition system. It inherits correctness and termination from the original. After that, using the Isabelle Refinement Framework [65], I refine the calculus to executable code. The resulting SAT solver is called IsaSAT. The text is based on a paper [42].

- In Chapter 6, I extend the SAT solvers with other features. Most SAT solvers use watched literals combined with another optimization, *blocking literals*, a technique to reduce the number of memory accesses (and therefore, the number of cache misses). Moreover, the calculus from Chapter 5 does not include the rules Restart and Forget. Restarts try to avoid focusing on a hard part of the search space. Forgets limit the number of clauses because too many of them slow down the solver. The text is based on a paper [40].

- In Chapter 7, I discuss my approach and compare it to other verification attempts. The main difference is the refinement approach: I verify an abstract CDCL that encapsulates the main idea, and then I refine it either to add new rules (for OCDCL), to add watched literals, or generate code. I also give some ideas on how to extend this formalization to verify other calculi.

Chapter 4 is independent of the Chapters 5 and 6.

1.3. Contributions

My main contributions are the following:

1. I developed a rich framework that captures the most important features of CDCL. The calculus is modeled as a transition system that includes rules for forget, restart, and incremental solving and the application of stepwise refinement to transfer results.

2. I used the CDCL framework I developed to verify variants of CDCL, based on a common presentation, CDCL with branch-and-bound. This is, to the best of my knowledge, the first time a CDCL formalization has

been reused for such purpose. Reusing a framework makes it possible to reduce the cost of the development of new variants.

3. I have developed the only verified SAT solver with efficient data structures, restart, forget, blocking literals, that is complete, and whose response (satisfiable or unsatisfiable) is correct without runtime checks. IsaSAT is also the only solver to feature the nowadays standard version of the two-watched-literal scheme with blocking literals.

Beyond the contributions described here, I have worked on some topics that do not fit in this thesis. First, I have extended Isabelle's multiset library: I use multisets to represent clauses and, therefore, I heavily rely on them. Beyond extending Isabelle's library of lemmas, I have created a simplification procedure to simplify results like $A + B + C = E + A + F$ into $B + C = E + F$ by canceling the common multiset A. This makes the handling of multisets easier. This work is presented in a paper with Blanchette and Traytel [21].

Isabelle features a tactic called `smt`. It asks an automatic prover to prove a proof obligation. If a proof is found, then Isabelle replays the produced proof through its kernel. This means that the external prover is not *trusted*. Earlier, only the SMT solver Z3 [91] was supported [23]. I have added support for the SMT solver veriT [24]. This work can also be useful for developers of automated theorem provers, because a failure in proof checking can indicate unsoundness in the prover that might be unnoticed otherwise. A prototype is presented in an article with Barbosa, Blanchette, and Fontaine [4]. Since that, more work has been done with Schurr to make the reconstruction faster [5].

2. Isabelle

Isabelle [93, 94] is a generic proof assistant that supports several object logics. The metalogic is an intuitionistic fragment of higher-order logic (HOL) [31]. This metasystem is called Isabelle/Pure (Section 2.1). The most used object logic is the instantiation with classical higher-order logic (Section 2.2). In the formalization, I heavily rely on the following features of Isabelle:

- *Locales* [3, 58] parameterize theories over operations and assumptions, encouraging a modular style. They are useful to express hierarchies of concepts and to reduce the number of parameters and assumptions that must be threaded through a formal development (Section 2.3).

- *Isar* [121] is a textual proof format inspired by the pioneering Mizar system [89]. It makes it possible to write structured, readable proofs (Section 2.4).

- *Sledgehammer* [19, 99] integrates superposition provers and SMT (satisfiability modulo theories) solvers in Isabelle to discharge proof obligations. The SMT solvers, and one of the superposition provers [117], are built around a SAT solver, resulting in a situation where SAT solvers are employed to prove their own metatheory (Section 2.5).

- Isabelle/jEdit is the official graphical interface to interact with Isabelle (Section 2.6)

2.1. Isabelle/Pure

Isabelle's metalogic is an intuitionistic fragment of higher-order logic (HOL) [31]. This system is called Isabelle/Pure. The types are built from type variables $'a, 'b, \ldots$ and n-ary type constructors, normally written in postfix notation (e.g., $'a\ list$). The infix type constructor $'a \Rightarrow 'b$ is interpreted as the (total) function space from $'a$ to $'b$. Function applications are written in a curried style without parentheses (e.g., $\mathsf{f}\ x\ y$). Anonymous functions $x \mapsto t_x$ are written $\lambda x.\ t_x$. The notation $t :: \tau$ indicates that term t has type τ. Propositions are terms of type *prop*, a type with at least two values. Symbols belonging to the signature (e.g., f) are uniformly called *constants*, even if they are functions or predicates. No syntactic distinction is enforced between terms and formulas. The metalogical operators are universal quantification $\bigwedge :: ('a \Rightarrow prop) \Rightarrow prop$, implication $\Longrightarrow :: prop \Rightarrow prop \Rightarrow prop$, and equality $\equiv :: 'a \Rightarrow 'a \Rightarrow prop$. The notation $\bigwedge x.\ p_x$ abbreviates $\bigwedge (\lambda x.\ p_x)$ and similarly for other binder notations. Several logics have deen developed inside Isabelle/Pure, notably Isabelle/ZF [95] based on the Zaermelo-Frankel axioms and Isabelle/HOL [94] based on simple higher-order logic (HOL).

2.2. Isabelle/HOL

Isabelle/HOL is the instantiation of Isabelle with HOL, an object logic for classical HOL extended with rank-1 (top-level) polymorphism and Haskell-style type classes. It axiomatizes a type *bool* of Boolean as well as its own set of logical symbols ($\forall$, $\exists$, False, True, $\neg$, $\wedge$, $\vee$, $\longrightarrow$, $\longleftrightarrow$, $=$). The object logic is embedded in the metalogic via a constant $\mathsf{Trueprop} :: bool \Rightarrow prop$, which is normally not printed. In practice, the distinction between the two logical levels HOL and Pure is important operationally but not semantically; for example, resolution in Isabelle can derive $\mathsf{q}\ 0$ from $\mathsf{p}\ 0$ and $\mathsf{p}\ x \Longrightarrow \mathsf{q}\ x$, but it would fail if the last formula were $\mathsf{p}\ x \longrightarrow \mathsf{q}\ x$. In this thesis, I will not preserve the distinction between the metalogic and the object logic and will identify *prop* with *bool*, $\bigwedge$ with $\forall$, $\Longrightarrow$ with $\longrightarrow$, and $\equiv$ with $=$.

Isabelle adheres to the tradition that started in the 1970s by the LCF system [46]: All inferences are derived by a small trusted kernel; types and functions are defined rather than axiomatized to guard against inconsistencies. High-level specification mechanisms let the user define important classes of types and functions, notably inductive datatypes, inductive predicates, and recursive functions. Internally, the system synthesizes appropriate low-level definitions and derives the user specifications via primitive inferences.

Isabelle developments are organized as collections of theory files that build on one another. Each file consists of definitions, lemmas, and proofs expressed in Isar [121], Isabelle's input language. Isar proofs are expressed either as a sequence of tactics (**apply** scripts) that manipulate the proof state directly or in a declarative, natural-deduction format inspired by Mizar. My formalization almost exclusively employs the more readable declarative style.

From now on, I will use the name Isabelle to mean Isabelle/HOL.

2.3. Locales

Isabelle locales are a convenient mechanism for structuring large proofs. A locale fixes types, constants, and assumptions within a specified scope. A schematic example follows:

> **locale** X =
>   **fixes** c :: $\tau_{'a}$
>   **assumes** $A_{'a,\mathsf{c}}$
> **begin**
>   ⟨*body*⟩
> **end**

The definition of locale X implicitly fixes a type $'a$, explicitly fixes a constant c whose type $\tau_{'a}$ may depend on $'a$, and states an assumption $A_{'a,\mathsf{c}} :: \mathit{prop}$ over $'a$ and c. Definitions made within the locale may depend on $'a$ and c, and lemmas proved within the locale may additionally depend on $A_{'a,\mathsf{c}}$. A single locale can introduce several types, constants, and assumptions. Seen from the outside, the lemmas proved in X are polymorphic in type variable $'a$, universally quantified over c, and conditional on $A_{'a,\mathsf{c}}$.

Locales support inheritance, union, and embedding. To embed Y into X, or make Y a *sublocale* of X, we must recast an instance of Y into an instance of X, by providing, in the context of Y, definitions of the types and constants of X together with proofs of X's assumptions. The command

> **sublocale** Y $\subseteq$ X t

emits the proof obligation $A_{v,t}$, where v and $t :: \tau_v$ may depend on types and constants available in Y. After the proof, all the lemmas proved in X become available in Y, with $'a$ and c :: $\tau_{'a}$ instantiated with v and $t :: \tau_v$.

Sometimes an embedding is the wrong tool, especially if there are several instantiations, because theorems names can used only once, making it impossible to refine the same locale several times with different parameters. In this case, the command **interpretation** can be used:

> **locale** Y
> **begin**
>   **interpretation** a: X t
>   **sorry**
> **end**

The same proof obligations $A_{v,t}$ are emitted, but the lemmas proved in X become available with the prefix a (e.g., *a.thm* for the theorem *thm* of X).

2.4. Isar

The original and most low-level proof style is the **apply** script: It is a backward style and each tactic creates subgoals. It is ideal for proof exploration and simple proofs. This proof style is, however, hard to maintain. A more readable style states explicit statements of properties in Isar [121]. The styles can be combined: each intermediate step can be recursively justified by apply scripts or Isar. For robustness, I use Isar where practicable. For example, while formalizing some results that depend on multisets, I found myself needing the basic property

lemma $|A| + |B| = |A \cup B| + |A \cap B|$

where A and B are finite multisets, $\cup$ denotes union defined such that for each element x, the multiplicity of x in $A \cup B$ is the maximum of the multiplicities of x in A and B, $\cap$ denotes intersection, and $|\ |$ denotes cardinality. This lemma was not available in Isabelle's underdeveloped multiset library.

A manual proof, expressed in Isar's declarative style, might look like this:

proof –
  have $|A| + |B| = |A + B|$ **by** *auto*
  also have $A \uplus B = (A \cup B) \uplus (A \cap B)$ **unfolding** *multiset_eq_iff*
  proof *clarify*
    fix a
    have $\mathsf{count}\ (A \uplus B)\ a = \mathsf{count}\ A\ a + \mathsf{count}\ B\ a$ **by** *simp*
    moreover have $\mathsf{count}\ (A \cup B \uplus A \cap B)\ a = \mathsf{count}\ (A \cup B)\ a$
        $+\, \mathsf{count}\ (A \cap B)\ a$
      by *simp*
    moreover have $\mathsf{count}\ (A \cup B)\ a = \mathsf{max}\ (\mathsf{count}\ A\ a)\ (\mathsf{count}\ B\ a)$
      by *auto*
    moreover have $\mathsf{count}\ (A \cap B)\ a = \mathsf{min}\ (\mathsf{count}\ A\ a)\ (\mathsf{count}\ B\ a)$
      by *auto*
    ultimately show $\mathsf{count}\ (A \uplus B)\ a = \mathsf{count}\ (A \cup B \uplus A \cap B)\ a$
      by *auto*
  qed
  ultimately show $|A| + |B| = |A \cup B| + |A \cap B|$ **by** *simp*
qed

The count function returns the multiplicity of an element in a multiset. The $\uplus$ operator denotes the disjoint union operation—for each element, it computes the sum of the multiplicities in the operands (as opposed to the maximum for the union operator $\cup$).

In Isar proofs, intermediate properties are introduced using **have** and proved using a tactic such as *simp* and *auto*. Proof blocks (**proof** … **end**) can be nested.

2.5. Sledgehammer

The Sledgehammer subsystem [19,99] integrates automatic theorem provers in Isabelle/HOL, including CVC4 [6], E [108], LEO-II [9], Satallax [26], SPASS [120], Vampire [104], veriT [24], and Z3 [91]. Upon invocation, it heuristically selects relevant lemmas from the thousands available in loaded libraries, translates them along with the current proof obligation to SMT-LIB or TPTP, and invokes the automatic provers. In case of success, the machine-generated proof is translated to an Isar proof that can be inserted into the formal development, so that the external provers do not need to be trusted.

Sledgehammer is part of most Isabelle users' workflow, and I invoke it dozens of times a day (according to the log files it produces). Sledgehammer is able to prove the property presented above: Within 30 s, the tool came back with a brief proof text, which I could insert into my formalization:

> **by** (*metis* (*no_types*) *Multiset.diff_right_commute add.assoc add_left_cancel monoid_add_class.add.right_neutral multiset_inter_commute multiset_inter_def size_union sup_commute sup_empty sup_multiset_def*)

The generated proof refers to 10 library lemmas by name and applies the *metis* search tactic. However, the proof found be Sledgehammer is not very readable and does not give any information to the ready why the theorem holds. The advantage of Isar proofs over one-line *metis* proofs is that I can follow and understand the steps. However, for lemmas about multisets and other background theories, I are usually content if I can get a (reasonably maintainable) proof automatically and carry on with formalizing the more interesting foreground theory.

2.6. Isabelle/jEdit

The main graphical and official interface to interact with Isabelle is based on jEdit and called Isabelle/jEdit [122]. Continuous checking of the visible part and all its dependencies is done. A screenshot is shown in Figure 2.1: The main part is the theory with pretty printing of mathematical symbols. At the

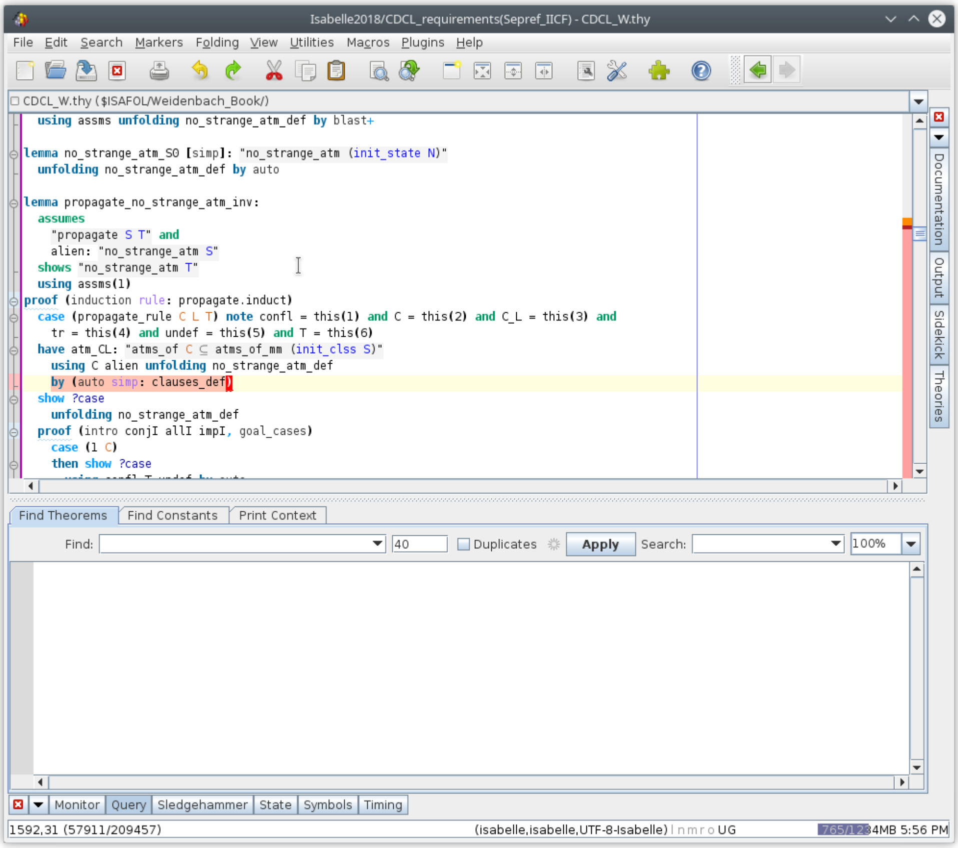


Figure 2.1.: Isabelle/jEdit screenshot: Isabelle could not verify the proof highlighted in red

bottom, there is the current goal, i.e., the proof obligations that remains to prove the current goal holds. When a goal cannot be discharged, the failing tactic is highlight in red.

Isabelle also provides a second way to interact the theory via the use of the language server protocol,[1] developed by Microsoft. Wenzel [123, Section 2.3] has worked on a extending Visual Studio Code to interact with Isabelle, e.g., by pretty-printing the symbols. Fewer features are supported than Isabelle/jEdit (for example, the current goal is printed without colors).

[1] `https://microsoft.github.io/language-server-protocol/`

3. Conflict-Driven Clause Learning

This chapter presents my formalization of CDCL (conflict-driven clause learning), the algorithm implemented in modern propositional satisfiability (SAT) solvers. I start with a family of formalized abstract DPLL (Davis–Putnam–Logemann–Loveland) [35] and CDCL [7, 16, 87, 90] transition systems based on Nieuwenhuis, Oliveras, and Tinelli's abstract presentation of CDCL [92] (Section 3.1). These families of calculi, called DPLL_NOT+BJ and CDCL_NOT, are proved sound, complete, and terminating. Some of the calculi include rules for learning and forgetting clauses and for restarting the search.

After that, I formalize another CDCL calculus based on Weidenbach's account based on *Automated Reasoning* and published earlier [119], called CDCL_W. It is derived as a refinement of CDCL_NOT. The calculus specifies a criterion for learning clauses representing first unique implication points [16, Chapter 3], with the guarantee that learned clauses are not redundant and hence derived at most once. The correctness results (soundness, completeness, termination) are inherited from the abstract calculus. In a minor departure from Weidenbach's presentation, I extend the Jump rule that repairs the model once a conflict is found to be able to express the conflict clause minimization, which is an important technique in implementations that shortens new learned clauses. CDCL_W is closer to an implementation and it is possible to prove complexity results on it: only 2^V clauses can be learned, where V is the number of atoms that appears in the problem. I also extend the calculus to support incremental solving (Section 3.2).

The concrete calculus is refined further to obtain a verified, but very naive, functional program extracted using Isabelle's code generator (Section 3.3). This SAT solver is called IsaSAT-0.

3.1. Abstract CDCL

The abstract CDCL calculus by Nieuwenhuis et al. [92] forms the first layer of my refinement chain. The formalization relies on basic Isabelle libraries for lists and multisets and on custom libraries for propositional logic. Properties such as partial correctness and termination (given a suitable strategy) are inherited by subsequent layers.

3.1.1. Propositional Logic

The DPLL and CDCL calculi distinguish between literals whose truth value has been decided arbitrarily and those that are entailed by the current decisions; for the latter, it is sometimes useful to know which clause entails it. To capture this information, I introduce a type of annotated literals, parameterized by a type $'v$ of propositional variables and a type $'cls$ of clauses:

datatype $'v$ *literal* $=$
  Pos $'v$
| Neg $'v$

datatype $('v, 'cls)$ *ann_literal* $=$
  Decided $('v\ literal)$
| Propagated $('v\ literal)$ $'cls$

The simpler calculi do not use $'cls$; they take $'cls = unit$, a singleton type whose unique value is $()$. Informally, I write A, $\neg A$, and $L^\dagger$ for positive, negative, and decision literals, and I write L^C (with $C :: 'cls$) or simply L (if $'cls = unit$ or if the clause C is irrelevant) for propagated literals. The unary minus operator is used to negate a literal, with $-(\neg A) = A$.

As is customary in the literature [2,119], clauses are represented by multisets, ignoring the order of literals but not repetitions. A *$'v$ clause* is a (finite) multiset over *$'v$ literal*. Clauses are often stored in sets or multisets of clauses. To ease reading, I write clauses using logical symbols (e.g., $\bot$, L, and $C \vee D$ for $\emptyset$, $\{L\}$, and $C \uplus D$). Given a clause C, I write $\neg C$ for the formula that corresponds to the clause's negation. Remark that the negation $\neg C$ of a clause is not a clause anymore, but a multiset of clause, where each clause is a single literal and the negation of one of the literals of C.

Given a set or multiset I of literals, $I \vDash C$ is true if and only if C and I share a literal. This is lifted to sets and multisets of clauses or formulas: $I \vDash N \longleftrightarrow \forall C \in N.\ I \vDash C$. A set or multiset is satisfiable if there exists a consistent set or multiset of literals I such that $I \vDash N$. Finally, $N \vDash N' \longleftrightarrow \forall I.\ I \vDash N \longrightarrow I \vDash N'$. These notations are also extended to formulas.

3.1.2. DPLL with Backjumping

Nieuwenhuis et al. present CDCL as a set of transition rules on states. A state is a pair (M, N), where M is the *trail* and N is the multiset of clauses to satisfy. In a slight abuse of terminology, I will refer to the multiset of clauses as the "clause set." The trail is a list of annotated literals that represents the partial model under construction. The empty list is written ϵ. Somewhat nonstandardly, but in accordance with Isabelle conventions for lists, the trail grows on the left: Adding a literal L to M results in the new trail $L \cdot M$, where $\cdot :: {'a} \Rightarrow {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list}$. The concatenation of two lists is written $M \mathbin{@} M'$. To lighten the notation, I often build lists from elements and other lists by simple juxtaposition, writing MLM' for $M \mathbin{@} L \cdot M'$.

The core of the CDCL calculus is defined as a transition relation called DPLL_NOT+BJ, an extension of classical DPLL [35] with nonchronological backtracking, or *backjumping*. The NOT part of the name refers to Nieuwenhuis, Oliveras, and Tinelli. The calculus consists of three rules, starting from an initial state (ϵ, N). In the following, I abuse notation, implicitly converting $\vDash$'s first operand from a list to a set and ignoring annotations on literals:

Propagate $\quad (M, N) \Longrightarrow_{\mathsf{DPLL\_NOT+BJ}} (LM, N)$
if N contains a clause $C \lor L$ such that $M \vDash \neg C$ and L is undefined in M (i.e., neither $M \vDash L$ nor $M \vDash -L$)

Decide $\quad (M, N) \Longrightarrow_{\mathsf{DPLL\_NOT+BJ}} (L^{\dagger}M, N)$
if the atom of L occurs in N and is undefined in M

Backjump $\quad (M'L^{\dagger}M, N) \Longrightarrow_{\mathsf{DPLL\_NOT+BJ}} (L'M, N)$
if N contains a conflicting clause C (i.e., $M'L^{\dagger}M \vDash \neg C$) and there exists a clause $C' \lor L'$ such that $N \vDash C' \lor L'$, $M \vDash \neg C'$, and L' is undefined in M but occurs in N or in $M'L^{\dagger}$

The Backjump rule is more general than necessary for capturing DPLL, where it suffices to negate the leftmost decision literal. The general rule can also express nonchronological backjumping, if $C' \lor L'$ is a new clause derived from N (but not necessarily in N).

I represented the calculus as an inductive predicate. For the sake of modularity, I formalized the rules individually as their own predicates and combined them to obtain DPLL_NOT+BJ:

inductive DPLL_NOT+BJ :: $'st \Rightarrow 'st \Rightarrow \mathit{bool}$ **where**
propagate $S\ S' \Longrightarrow$ DPLL_NOT+BJ $S\ S'$
| decide $S\ S' \Longrightarrow$ DPLL_NOT+BJ $S\ S'$
| backjump $S\ S' \Longrightarrow$ DPLL_NOT+BJ $S\ S'$

Since there is no call to DPLL_NOT+BJ in the assumptions, I could also have used a plain **definition** here, but the **inductive** command provides convenient introduction and elimination rules. The predicate operates on states of type $'st$. To allow for refinements, this type is kept as a parameter of the calculus, using a locale that abstracts over it and that provides basic operations to manipulate states:

```
locale dpll_state =
  fixes
    trail :: 'st ⇒ ('v, unit) ann_literal list and
    clauses :: 'st ⇒ 'v clause multiset and
    prepend_trail :: ('v, unit) ann_literal ⇒ 'st ⇒ 'st and
    tl_trail :: 'st ⇒ 'st and
    add_clause :: 'v clause ⇒ 'st ⇒ 'st and
    remove_clause :: 'v clause ⇒ 'st ⇒ 'st
  assumes
    state (prepend_trail L S) = (L · trail S, clauses S) and
    state (tl_trail S) = (tl (trail S), clauses S) and
    state (add_cls C S) = (trail S, add_mset C (clauses S)) and
    state (remove_cls C S) = (trail S, remove_all C (clauses S))
```

where state converts an abstract state of type *'st* to a pair (M, N). Inside the locale, states are compared extensionally: $S \sim S'$ is true if the two states have identical trails and clause sets (i.e., if state S = state S'). This comparison ignores any other fields that may be present in concrete instantiations of the abstract state type *'st*.

Each calculus rule is defined in its own locale, based on dpll_state and parameterized by additional side conditions. Complex calculi are built by inheriting and instantiating locales providing the desired rules. For example, the following locale provides the predicate corresponding to the Decide rule, phrased in terms of an abstract DPLL state:

```
locale decide_ops = dpll_state +
  fixes decide_conds :: 'st ⇒ 'st ⇒ bool
begin

inductive decide :: 'st ⇒ 'st ⇒ bool where
  undefined_lit (trail S) L ⟹
  atm_of L ∈ atms_of (clauses S) ⟹
  S' ∼ prepend_trail (Decided L) S ⟹
  decide_conds S S' ⟹
  decide S S'

end
```

Following a common idiom, the DPLL_NOT+BJ calculus is distributed over two locales: The first locale, DPLL_NOT+BJ_ops, defines the DPLL_NOT+BJ calculus; the second locale, DPLL_NOT+BJ, extends it with an assumption expressing a structural invariant over DPLL_NOT+BJ that is instantiated when proving concrete properties later. This cannot be achieved with a single locale, because definitions may not precede assumptions. Moreover, unfolding definitions does not require discharging assumption if the locale does not have assumptions.

Theorem 3.1 (Termination [41, *wf_dpll_bj*] ☑). *The relation DPLL_NOT+BJ is well founded.*

Termination is proved by exhibiting a well-founded relation $\prec$ such that $S' \prec S$ whenever $S \Longrightarrow_{\mathsf{DPLL\_NOT+BJ}} S'$. Let $S = (M, N)$ and $S' = (M', N')$ with the decompositions

$$M = M_n L_n^\dagger \cdots M_1 L_1^\dagger M_0 \qquad\qquad M' = M'_{n'} L'^\dagger_{n'} \cdots M'_1 L'^\dagger_1 M'_0$$

where the trail segments $M_0, \ldots, M_n, M'_0, \ldots, M'_{n'}$ contain no decision literals. Let V be the number of distinct variables occurring in the initial clause set N. Now, let $\nu\, M = V - |M|$, indicating the number of unassigned variables in the trail M. Nieuwenhuis et al. define $\prec$ such that $S' \prec S$ if

(1) there exists an index $i \leq n, n'$ such that

$$[\nu\, M'_0,\ \ldots,\ \nu\, M'_{i-1}] = [\nu\, M_0,\ \ldots,\ \nu\, M_{i-1}]$$

and $\nu\, M'_i < \nu\, M_i$; or

(2) $[\nu\, M_0,\ \ldots,\ \nu\, M_n]$ is a strict prefix of $[\nu\, M'_0,\ \ldots,\ \nu\, M'_{n'}]$.

This order is not to be confused with the lexicographic order: I have $[0] \prec \epsilon$ by condition (2), whereas $\epsilon <_{\mathsf{lex}} [0]$. Yet the authors justify well-foundedness by appealing to the well-foundedness of $<_{\mathsf{lex}}$ on bounded lists over finite alphabets. In my proof, I clarify and simplify matters by mapping states S to lists $[|M_0|, \ldots, |M_n|]$, without appealing to ν. Using the standard lexicographic order, states become *larger* with each transition:

Propagate	$[k_1, \ldots, k_n] <_{\mathsf{lex}} [k_1, \ldots, k_n + 1]$	
Decide	$[k_1, \ldots, k_n] <_{\mathsf{lex}} [k_1, \ldots, k_n, 0]$	
Backjump	$[k_1, \ldots, k_n] <_{\mathsf{lex}} [k_1, \ldots, k_j + 1]$	with $j \leq n$

The lists corresponding to possible states are bounded by the list $[V, \ldots, V]$ consisting of V occurrences of V, thereby delimiting a finite domain $D = \{[k_1, \ldots, k_n] \mid k_1, \ldots, k_n, n \leq V\}$. I take $\prec$ to be the restriction of $>_{\mathsf{lex}}$ to

D. A variant of this approach is to encode lists into a measure $\mu_V\, M = \sum_{i=0}^{n} |M_i|\, V^{n-i}$ and let $S' \prec S \leftrightarrow \mu_V\, M' > \mu_V\, M$, building on the well-foundedness of $>$ over bounded sets of natural numbers.

A *final* state is a state from which no transitions are possible. Given a relation $\Longrightarrow$, I write $\Longrightarrow^!$ for the right-restriction of its reflexive transitive closure to final states (i.e., $S_0 \Longrightarrow^! S$ if and only if $S_0 \Longrightarrow^* S \wedge \forall S'.\ S \not\Longrightarrow S'$).

Theorem 3.2 (Partial Correctness [41, *full_dpll_backjump_final_state_from_init_state*] ☑). *If* $(\epsilon, N) \Longrightarrow^!_{\mathsf{DPLL\_NOT+BJ}} (M, N)$*, then* N *is satisfiable if and only if* $M \models N$.

I first prove structural invariants on arbitrary states (M', N) reachable from (ϵ, N), namely: (1) each variable occurs at most once in M'; (2) if $M' = M_2 L M_1$ where L is propagated, then $M_1, N \models L$. From these invariants, together with the constraint that (M, N) is a final state, it is easy to prove the theorem.

3.1.3. Classical DPLL

The locale machinery allows me to derive a classical DPLL calculus from DPLL with backjumping. I call this calculus DPLL_NOT. It is achieved through a DPLL_NOT locale that restricts the Backjump rule so that it performs only chronological backtracking:

Backtrack $(M' L^\dagger M, N) \Longrightarrow_{\mathsf{DPLL\_NOT}} (-L \cdot M, N)$
if N contains a conflicting clause and M' contains no decision literals

Because of the locale parameters, DPLL_NOT is strictly speaking a family of calculi.

Lemma 3.3 (Backtracking [41, *backtrack_is_backjump*] ☑). *The Backtrack rule is a special case of the Backjump rule.*

The Backjump rule depends on two clauses: a conflict clause C and a clause $C' \vee L'$ that justifies the propagation of L'. The conflict clause is specified by Backtrack. As for $C' \vee L'$, given a trail $M' L^\dagger M$ decomposable as $M_n L^\dagger M_{n-1} L^\dagger_{n-1} \cdots M_1 L^\dagger_1 M_0$ where $M_0, \ldots, M_n$ contain no decision literals, I can take $C' = -L_1 \vee \cdots \vee -L_{n-1}$.

Consequently, the inclusion DPLL_NOT $\subseteq$ DPLL_NOT+BJ holds. In Isabelle, this is expressed as a locale instantiation: DPLL_NOT is made a sublocale of DPLL_NOT+BJ, with a side condition restricting the application of the Backjump rule. The partial correctness and termination theorems are inherited from the base locale. DPLL_NOT instantiates the abstract state type *'st* with a concrete type of pairs. By discharging the locale assumptions emerging with the **sublocale** command, I also verify that these assumptions are consistent. Roughly:

locale DPLL_NOT =
begin
  type_synonym $'v$ *state* = $('v, unit, unit)$ *ann_literal list*
    $\times 'v$ *clause multiset*

  inductive backtrack :: $'v$ *state* $\Rightarrow$ $'v$ *state* $\Rightarrow$ *bool* **where** ...
end

sublocale DPLL_NOT $\subseteq$ dpll_state fst snd $(\lambda L\,(M, N).\,(L \cdot M, N))$
  ...
sublocale DPLL_NOT $\subseteq$ DPLL_NOT+BJ_ops ...
  $(\lambda C\ L\ S\ S'.$ DPLL.backtrack $S\ S')$ ...
sublocale DPLL_NOT $\subseteq$ DPLL_NOT+BJ ...

If a conflict cannot be resolved by backtracking, I would like to have the option of stopping even if some variables are undefined. A state (M, N) is *conclusive* if $M \vDash N$ or if N contains a conflicting clause and M contains no decision literals. For DPLL_NOT, all final states are conclusive, but not all conclusive states are final.

Theorem 3.4 (Partial Correctness [41, *dpll_conclusive_state_correctness*] ☑). *If* $(\epsilon, N) \Longrightarrow^*_{DPLL\_NOT} (M, N)$ *and* (M, N) *is a conclusive state,* N *is satisfiable if and only if* $M \vDash N$*.*

The theorem does not require stopping at the first conclusive state. In an implementation, testing $M \vDash N$ can be expensive, so a solver might fail to notice that a state is conclusive and continue for some time. In the worst case, it will stop in a final state—which is guaranteed to exist by Theorem 3.1. In practice, instead of testing whether $M \vDash N$, implementations typically apply the rules until every literal is set. When N is satisfiable, this produces a total model.

3.1.4. The CDCL Calculus

The abstract CDCL calculus extends DPLL_NOT+BJ with a pair of rules for learning new lemmas and forgetting old ones:

Learn $(M, N) \Longrightarrow_{\mathsf{CDCL\_NOT}} (M, N \uplus \{C\})$ if $N \vDash C$ and each atom of C is in N or M

Forget $(M, N \uplus \{C\}) \Longrightarrow_{\mathsf{CDCL\_NOT}} (M, N)$ if $N \vDash C$

In practice, the Learn rule is normally applied to clauses built exclusively from atoms in M, because the learned clause is false in M. This property eventually guarantees that the learned clause is not redundant (e.g., it is not already contained in N).

I call this calculus CDCL_NOT. In general, CDCL_NOT does not terminate, because it is possible to learn and forget the same clause infinitely often. But for some instantiations of the parameters with suitable restrictions on Learn and Forget, the calculus always terminates.

Theorem 3.5 (Termination [41, *wf_cdcl$_{\mathrm{NOT}}$_no_learn_and_forget_infinite_chain*] ☑). *Let C be an instance of the CDCL_NOT calculus (i.e., $C \subseteq$ CDCL_NOT). If C admits no infinite chains consisting exclusively of Learn and Forget transitions, then C is well founded.*

In many SAT solvers, the only clauses that are ever learned are the ones used for backtracking. If I restrict the learning so that it is always done immediately before backjumping, I can be sure that some progress will be made between a Learn and the next Learn or Forget. This idea is captured by the following combined rule:

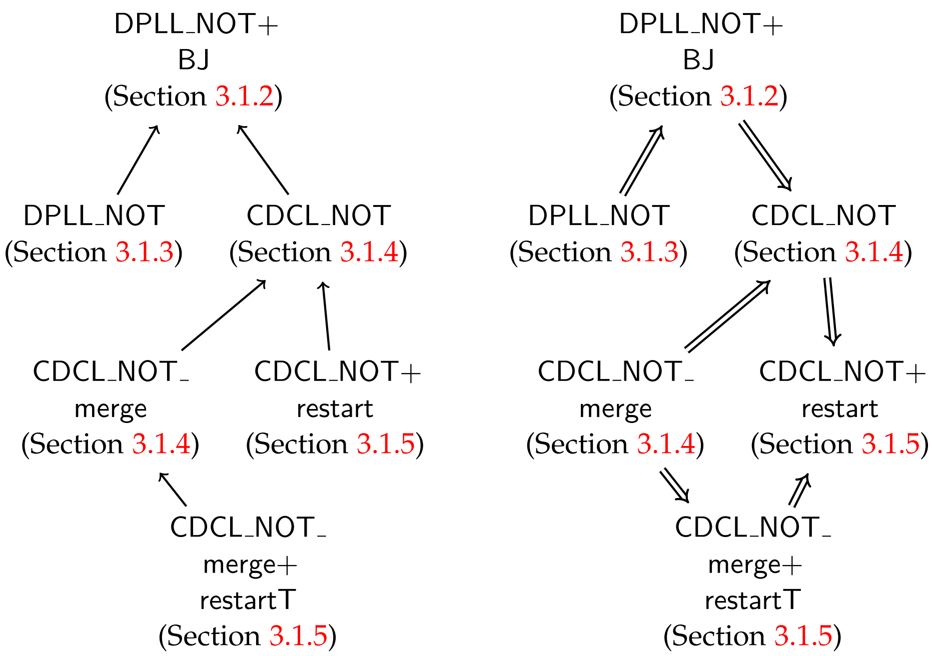
Learn+Backjump $\quad (M'L^{\dagger}M, N) \Longrightarrow_{\mathsf{CDCL\_NOT\_merge}} (L'M, N \uplus \{C' \vee L'\})$
if C', L, L', M, M', N satisfy Backjump's side conditions

The calculus variant that performs this rule instead of Learn and Backjump is called CDCL_NOT_merge. Because a single Learn+Backjump transition corresponds to two transitions in CDCL_NOT, the inclusion CDCL_NOT_merge $\subseteq$ CDCL_NOT does not hold. Instead, I have CDCL_NOT_merge $\subseteq$ CDCL_NOT$^+$. Each step of CDCL_NOT_merge corresponds to a single step in CDCL_NOT or a two-step sequence consisting of Backjump followed by Learn.

3.1.5. Restarts

Modern SAT solvers rely on a dynamic decision literal heuristic. They periodically restart the proof search to apply the effects of a changed heuristic. This helps the calculus focus on a part of the search space where it can make progress. Upon a restart, some learned clauses may be removed, and the trail is reset to ϵ. Since my calculus has a Forget rule, the Restart rule needs only to clear the trail. Adding Restart to CDCL_NOT yields CDCL_NOT+restart. However, this calculus does not terminate, because Restart can be applied infinitely often.

A working strategy is to gradually increase the number of transitions between successive restarts. This is formalized via a locale parameterized by a



(a) Syntactic dependencies **(b)** Refinements

Figure 3.1.: Connections between the abstract calculi

base calculus C and an unbounded function $f :: nat \Rightarrow nat$. Nieuwenhuis et al. require f to be strictly increasing, but unboundedness is sufficient.

The extended calculus C+restartT operates on states of the form (S, n), where S is a state in the base calculus and n counts the number of restarts. To simplify the presentation, I assume that base states S are pairs (M, N). The calculus C+restartT starts in the state $((\epsilon, N), 0)$ and consists of two rules:

Restart $(S, n) \Longrightarrow_{C+\mathsf{restartT}} ((\epsilon, N'), n+1)$
if $S \Longrightarrow_C^m (M', N')$ and $m \geq f\ n$

Finish $(S, n) \Longrightarrow_{C+\mathsf{restartT}} (S', n+1)$ if $S \Longrightarrow_C^! S'$

The symbol $\Longrightarrow_C$ represents the base calculus C's transition relation, and $\Longrightarrow_C^m$ denotes an m-step transition in C. The T in restartT reminds me that I count the number of *transitions*; in Section 3.2.5, I will review an alternative strategy based on the number of conflicts or learned clauses. Termination relies on a measure μ_V associated with C that may not increase from restart to restart: If $S \Longrightarrow_C^* S' \Longrightarrow_{\mathsf{restartT}} S''$, then $\mu_V\ S'' \leq \mu_V\ S$. The measure may depend on V, the number of variables occurring in the problem.

I instantiated the locale parameter C with CDCL_NOT_merge and f with the Luby sequence $(1, 1, 2, 1, 1, 2, 4, \ldots)$ [78], with the restriction that no clause containing duplicate literals is ever learned, thereby bounding the number of learnable clauses and hence the number of transitions taken by C.

Figure 3.1a summarizes the syntactic dependencies between the calculi reviewed in this section. An arrow $C \longrightarrow B$ indicates that C is defined in terms of B. Figure 3.1b presents the refinements between the calculi. An arrow $C \Longrightarrow B$ indicates that I proved $C \subseteq B^*$ or some stronger result—either by locale embedding (**sublocale**) or by simulating C's behavior in terms of B.

3.2. A Refined CDCL towards an Implementation

The CDCL_NOT calculus captures the essence of modern SAT solvers without imposing a policy on when to apply specific rules. In particular, the Backjump rule depends on a clause $C' \vee L'$ to justify the propagation of a literal, but does not specify a procedure for coming up with this clause. For *Automated Reasoning*, Weidenbach developed a calculus that is more specific in this respect, and closer to existing solver implementations, while keeping many aspects unspecified [119]. This calculus, CDCL_W, is also formalized in Isabelle and connected to CDCL_NOT.

3.2.1. The New DPLL Calculus

Independently from the previous section, I formalized DPLL as described in *Automated Reasoning*. The calculus operates on states (M, N), where M is the trail and N is the initial clause set. It consists of three rules:

Propagate $(M, N) \Longrightarrow_{\mathsf{DPLL\_W}} (LM, N)$ if $C \vee L \in N \uplus U$, $M \vDash \neg C$, and L is undefined in M

Decide $(M, N) \Longrightarrow_{\mathsf{DPLL\_W}} (L^\dagger M, N)$ if L is undefined in M and occurs in N

Backtrack $(M' K^\dagger M, N) \Longrightarrow_{\mathsf{DPLL\_W}} (-K \cdot M, N)$
if N contains a conflicting clause and M' contains no decision literals

Backtrack performs chronological backtracking: It undoes the last decision and picks the opposite choice. Conclusive states for DPLL_W are defined as for DPLL_NOT (Section 3.1.3).

The termination and partial correctness proofs given by Weidenbach depart from Nieuwenhuis et al. I also formalized them:

Theorem 3.6 (Termination [41, *wf_dpll*$_W$] ☑). *The relation DPLL_W is well founded.*

Termination is proved by exhibiting a well-founded relation that includes DPLL_W. Let V be the number of distinct variables occurring in the clause set N. The weight $\nu\, L$ of a literal L is 2 if L is a decision literal and 1 otherwise. The measure is

$$\mu\ (L_k \cdots L_1, N) = \big[\nu\, L_1, \ldots, \nu\, L_k, \underbrace{3, \ldots, 3}_{V-k \text{ occurrences}}\big]$$

Lists are compared using the lexicographic order, which is well founded because there are finitely many literals and all lists have the same length. It is easy to check that the measure decreases with each transition:

$$\begin{array}{llll}
\textsf{Propagate} & [k_1,\ldots,k_m,3,3,\ldots,3] & >_{\mathsf{lex}} & [k_1,\ldots,k_m,1,3,\ldots,3] \\
\textsf{Decide} & [k_1,\ldots,k_m,3,3,\ldots,3] & >_{\mathsf{lex}} & [k_1,\ldots,k_m,2,3,\ldots,3] \\
\textsf{Backtrack} & [k_1,\ldots,k_m,2,l_1,\ldots,l_n] & >_{\mathsf{lex}} & [k_1,\ldots,k_m,1,3,\ldots,3]
\end{array}$$

Theorem 3.7 (Partial Correctness [41, *dpll*$_W$*_conclusive_state_correctness*] ☑). *If* $(\epsilon, N) \Longrightarrow^*_{\mathsf{DPLL\_W}} (M, N)$ *and* (M, N) *is a conclusive state,* N *is satisfiable if and only if* $M \vDash N$.

The proof is analogous to the proof of Theorem 3.2. Some lemmas are shared between both proofs. Moreover, I can link Weidenbach's DPLL calculus with the version I derived from DPLL_NOT+BJ in Section 3.1.3:

Theorem 3.8 (DPLL [41, *dpll*$_W$*_dpll*$_{NOT}$] ☑). *If* S *satisfies basic structural invariants, then* $S \Longrightarrow_{\mathsf{DPLL\_W}} S'$ *if and only if* $S \Longrightarrow_{\mathsf{DPLL\_NOT}} S'$.

This provides another way to establish Theorems 3.6 and 3.7. Conversely, the simple measure that appears in the above termination proof can also be used to establish the termination of the more general DPLL_NOT+BJ calculus (Theorem 3.1).

3.2.2. The New CDCL Calculus

The CDCL_W calculus operates on states (M, N, U, D), where M is the trail; N and U are the sets of initial and learned clauses, respectively; and D is a conflict clause, or the distinguished clause $\top$ if no conflict has been detected.

In the trail M, each decision literal L is marked as such ($L^\dagger$—i.e., Decided L), and each propagated literal L is annotated with the clause C that caused its propagation (L^C—i.e., Propagated L C). The level of a literal L in M is the number of decision literals to the right of the atom of L in M, or 0 if the atom is undefined. The level of a clause is the highest level of any of its literals, with 0 for $\bot$, and the level of a state is the maximum level (i.e., the number of decision literals). The calculus assumes that N contains no clauses with duplicate literals and never produces clauses containing duplicates.

The calculus starts in a state $(\epsilon, N, \emptyset, \top)$. The following rules apply as long as no conflict has been detected:

Propagate $(M, N, U, \top) \Longrightarrow_{\mathsf{CDCL\_W}} (L^{C\vee L}M, N, U, \top)$
if $C \vee L \in N \uplus U$, $M \vDash \neg C$, and L is undefined in M

Decide $(M, N, U, \top) \Longrightarrow_{\mathsf{CDCL\_W}} (L^{\dagger}M, N, U, \top)$ if L is undefined in M and occurs in N

Conflict $(M, N, U, \top) \Longrightarrow_{\mathsf{CDCL\_W}} (M, N, U, D)$ if $D \in N \uplus U$ and $M \vDash \neg D$

Restart $(M, N, U, \top) \Longrightarrow_{\mathsf{CDCL\_W}} (\epsilon, N, U, \top)$ if $M \not\vDash N$

Forget $(M, N, U \uplus \{C\}, \top) \Longrightarrow_{\mathsf{CDCL\_W}} (M, N, U, \top)$ if $M \not\vDash N$ and M contains no literal L^C

The Propagate and Decide rules generalize their DPLL_W counterparts. Once a conflict clause has been detected and stored in the state, the following rules cooperate to reduce it and backtrack, exploring a first unique implication point [16, Chapter 3]:

Skip $(L^C M, N, U, D) \Longrightarrow_{\mathsf{CDCL\_W}} (M, N, U, D)$ if $D \notin \{\bot, \top\}$ and $-L$ does not occur in D

Resolve $(L^{C\vee L}M, N, U, D \vee -L) \Longrightarrow_{\mathsf{CDCL\_W}} (M, N, U, C \cup D)$
if D has the same level as the current state

Jump $(M' K^{\dagger} M, N, U, D \vee L) \Longrightarrow_{\mathsf{CDCL\_W}} (L^{D\vee L}M, N, U \uplus \{D \vee L\}, \top)$
if L has the level of the current state, D has a lower level, and K and D have the same level

Exhaustive application of these three rule corresponds to a single step by the combined learning and nonchronological backjumping rule Learn+Backjump from CDCL_NOT_merge. The Learn+Backjump rule is even more general and can be used to express learned clause minimization [113].

In Resolve, $C \cup D$ is the same as $C \vee D$ (i.e., $C \uplus D$), except that it keeps only one copy of the literals that belong to both C and D. When performing propagations and processing conflict clauses, the calculus relies on the invariant that clauses never contain duplicate literals. Several other structural invariants hold on all states reachable from an initial state, including the following: The clause annotating a propagated literal of the trail is a member of $N \uplus U$. Some of the invariants were not mentioned in the textbook (e.g., whenever L^C occurs in the trail, L is a literal of C). Formalization helped develop a better understanding of the data structure and clarify the book.

Like CDCL_NOT, CDCL_W has a notion of conclusive state. A state (M, N, U, D) is *conclusive* if $D = \top$ and $M \models N$ or if $D = \bot$ and N is unsatisfiable. The calculus always terminates, but without a suitable strategy, it can block in an inconclusive state. At the end of the following derivation, neither Skip nor Resolve can process the conflict further:

$$
\begin{aligned}
& (\epsilon, \{A, B\}, \emptyset, \top) \\
\Longrightarrow_{\mathsf{Decide}} \; & (\neg A^\dagger, \{A, B\}, \emptyset, \top) \\
\Longrightarrow_{\mathsf{Decide}} \; & (\neg B^\dagger\, \neg A^\dagger, \{A, B\}, \emptyset, \top) \\
\Longrightarrow_{\mathsf{Conflict}} \; & (\neg B^\dagger\, \neg A^\dagger, \{A, B\}, \emptyset, A)
\end{aligned}
$$

3.2.3. A Reasonable Strategy

To prove correctness, I assume a *reasonable* strategy: Propagate and Conflict are preferred over Decide; Restart and Forget are not applied. (I will lift the restriction on Restart and Forget in Section 3.2.5.) The resulting calculus, CDCL_W+stgy, refines CDCL_W with the assumption that derivations are produced by a reasonable strategy. This assumption is enough to ensure that the calculus can backjump after detecting a nontrivial conflict clause other than $\bot$. The crucial invariant is the existence of a literal with the highest level in any conflict, so that Resolve can be applied. The textbook suggests preferring Confict to Propagate and Propagate to the other rules. But it is not needed for any of my metatheoretical results and not compatible with most implementation.

Correctness. With the reasonable strategy, the calculus terminates in a conclusive state, which allows to conclude on the satisfiability of the original formula:

Theorem 3.9 (Partial Correctness [41, *full_cdcl$_W$_stgy_final_state_conclusive_from_init_state*] ☑). *If $(\epsilon, N, \emptyset, \top) \Longrightarrow^!_{\mathit{CDCL\_W+stgy}} S'$ and N contains no clauses with duplicate literals, S' is a conclusive state.*

Once a conflict clause has been stored in the state, the clause is first reduced by a chain of Skip and Resolve transitions. Then, there are two scenarios: (1) the conflict is solved by a Jump, at which point the calculus may resume propagating and deciding literals; (2) the reduced conflict is $\bot$, meaning that N is unsatisfiable—i.e., for unsatisfiable clause sets, the calculus generates a resolution refutation.

No relearning. The CDCL_W+stgy calculus is designed to have respectable complexity bounds. One of the reasons for this is that the same clause cannot be learned twice:

Theorem 3.10 (No Relearning [41, *cdcl*$_W$*_stgy_distinct_mset_clauses*] ☑). *If* $(\epsilon, N, \emptyset, \top) \Longrightarrow^{*}_{CDCL\_W+stgy} (M, N, U, D)$, *then no Jump transition is possible from the latter state causing the addition of a clause from* $N \uplus U$ *to* U.

The formalization of this theorem posed some challenges. The informal proof in *Automated Reasoning* is as follows (with slightly adapted notations):

> *Proof.* By contradiction. Assume CDCL learns the same clause twice, i.e., it reaches a state $(M, N, U, D \vee L)$ where Jump is applicable and $D \vee L \in N \uplus U$. More precisely, the state has the form $(K_n \cdots K_2 K_1^{\dagger} M_2 K^{\dagger} M_1, N, U, D \vee L)$ where the K_i, $i > 1$ are propagated literals that do not occur complemented in D, as otherwise D cannot be of level i. Furthermore, one of the K_i is the complement of L. But now, because $D \vee L$ is false in $K_n \cdots K_2 K_1^{\dagger} M_2 K^{\dagger} M_1$ and $D \vee L \in N \uplus U$ instead of deciding $K_1^{\dagger}$ the literal L should be propagated by a reasonable strategy. A contradiction. Note that none of the K_i can be annotated with $D \vee L$. □

Many details are missing. To find the contradiction, I must show that there exists a state in the derivation with the trail $M_2 K^{\dagger} M_1$, and such that $D \vee L \in N \uplus U$. The textbook does not explain why such a state is guaranteed to exist. Moreover, inductive reasoning is hidden under the ellipsis notation $(K_n \cdots K_2)$. Such a high-level proof might be suitable for humans, but the details are needed in Isabelle, and Sledgehammer alone cannot fill in such large gaps, especially when induction is needed. The first version of the formal proof was over 700 lines long and is among the most difficult proofs I carried out about CDCL.

I later refactored the proof and the definition of CDCL_W+stgy. Following the book, each transition in CDCL_W+stgy was initially normalized by applying Propagate and Conflict exhaustively. For example, I defined Decide+stgy

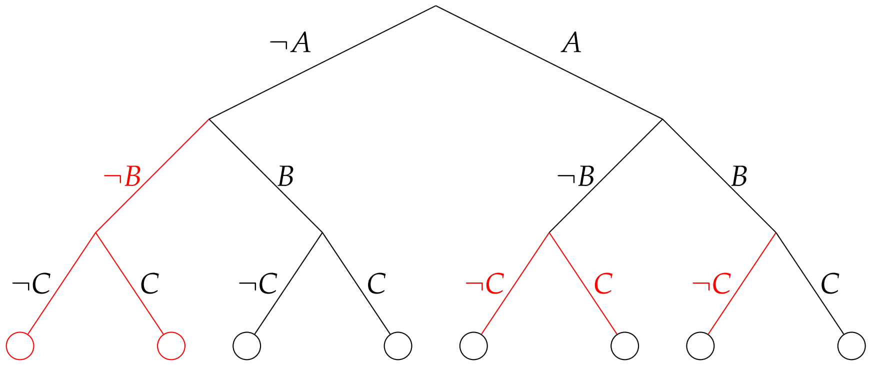


Figure 3.2.: Proof tree for the clauses $A \vee B$, $\neg A \vee C$, and $B \vee \neg C$, where red edges indicate conflicts with one of the three clauses

so that $S \Longrightarrow_{\mathsf{Decide+stgy}} U$ if Propagate and Conflict cannot be applied to S and $S \Longrightarrow_{\mathsf{Decide}} T \Longrightarrow^{!}_{\mathsf{Propagate,Conflict}} U$ for some state T. However, normalization is not necessary. It is simpler to define $S \Longrightarrow_{\mathsf{Decide+stgy}} T$ as $S \Longrightarrow_{\mathsf{Decide}} T$, with the same condition on S as before. This change shortened the proof by about 200 lines. In a subsequent refactoring, I further departed from the book: I proved the invariant that all propagations have been performed before deciding a new literal. The core argument ("the literal L should be propagated by a reasonable strategy") remains the same, but I do not have to reason about past transitions to argue about the existence of an earlier state. The invariant also makes it possible to generalize the statement of Theorem 3.10: I can start from any state that satisfies the invariant, not only from an initial state. The final version of the proof is 250 lines long.

A better bound. Using Theorem 3.10 and assuming that only backjumping has a cost (which is the same as counting the number of learned clauses), I get a complexity of $\mathrm{O}(3^V)$, where V is the number of different propositional variables, but a better complexity bound can be found: $\mathrm{O}(2^V)$. Each time Jump is applied, a new clause is learned and at least a model is excluded. Since there are only 2^V models, the conclusion follows. Another intuitive point of view is to look at all models seen as a tree, where each node contains an atom L, the right branch is L and the left branch is $\neg L$. A decision introduces two branches, while a propagated literal does not introduce a branch. Each leaf represent a total model (Figure 3.2). Since there are only 2^V leafs and each Jump goes to a different branch, the number of learned clauses is bound by 2^V. This point of view is useful to understand the argument, but not useful for a proof, because it implicitly relies on an order of the decisions (the order given by the model tree). However, the argument can be adapted by taking the measure:

$$\mu'\ (L_k \cdots L_1, N) = \left[\nu'\, L_1, \ldots, \nu'\, L_k, \underbrace{0, \ldots, 0}_{V-k \text{ occurrences}}\right]$$

where the weight $\nu'\, L$ of a literal L is 1 if L is a decision literal and 0 otherwise. In Isabelle, I simply consider μ' as the digits of a binary number. It is obviously bound by 2^V. μ' is not a measure for CDCL, since it only decreases for backtrack and propagate, not when decisions are made, and even increases when applying Skip or Resolve. Surprisingly to me, the proof does not depend on the strategy. In the worst case, the calculus will be stuck before reaching a final state. This also explains how crude this bound actually is: It does not depend on propagations and in particular does not require them to be done eagerly.

The proof in Isabelle is a perfect example of how much bookkeeping is sometimes required: In a paper, I would simply say that a CDCL run is an interleaving of the conflict analysis (Conflict, Skip, Resolve, and Backtrack) combined together and of Propagate and Decide combined together. In Isabelle, this interleaving has to be built explicitly from all considered transitions. I did so with a combination of recursive functions and choice operators. This requires several inductions and, even though conceptually important, these induction are only a proof detail and hide the important arguments. Recursive definitions cannot be defined within a proof by the function package [61], and instead, direct calls to the recursor have to be used. This requires to do by hand what is normally done automatically, especially proving simplification rule for the base case. For example, if I consider the sequence of states $(s\,_i)$ such that for all i $s_i \Longrightarrow_{\mathsf{CDCL\_W}} s_{i+1}$, here is the recursive function that enumerates the position of all the Jumps occurring in $(s\,_i)$:

define nth_bj :: *nat* ⇒ *nat* **where**
nth_bj = rec_nat 0
$\big(\lambda\ \_\ j.\ \mathsf{LEAST}\, n.\ (n > j \wedge s_n \Longrightarrow_{\mathsf{Jump}} s_{n+1})\big)$

$\mathsf{LEAST}\, n.\ P\ n$ returns the smallest n such that $P\, n$ is true—it is defined in Isabelle using Hilbert's choice operator. The value O is used to initialize the sequence, instead of the first Jump in the sequence: $\mathsf{LEAST}\, n.\, n \geq 0 \wedge s_n \Longrightarrow_{\mathsf{Jump}} s_{n+1}$. If there are no Jumps in the sequences or if there are no Jumps anymore, then the function returns an unspecified element.

After that, two simplifications rules have to be derived by hand:

$$\begin{aligned}\mathsf{nth\_bj}\,0 &= 0\\ \mathsf{nth\_bj}\,(j+1) &= \mathsf{LEAST}\,n.\,\big(n > \mathsf{nth\_bj}\,j \wedge s_n \Longrightarrow_{\mathsf{Jump}} s_{n+1}\big)\end{aligned}$$

Deriving the theorems is not hard, but not very interesting either. To avoid this kind of reasoning, I usually express whenever possible my invariants as properties on *states* instead of properties on *transitions*. In the case above, this is not avoidable, without defining an alternative calculus that groups the rule as expected.

After that, I define a similar (non-recursive) function that returns when a conflict has been found:

define $\mathsf{nth\_bj}$ :: *nat* $\Rightarrow$ *nat* **where**
$\mathsf{nth\_confl}\,j = \mathsf{LEAST}\,n.\,\big(n > \mathsf{nth\_bj}\,j \wedge j < \mathsf{nth\_bj}\,(j+1) \wedge$
$s_n \Longrightarrow_{\mathsf{Conflict}} s_{n+1}\big)$

One the transitions has been rules, it remains to show that:

$$\begin{aligned}\mu'(\mathsf{trail}\,s_{\mathsf{nth\_confl}\,j}) &> \mu'(\mathsf{trail}\,s_{\mathsf{nth\_bj}\,j})\\ \mu'(\mathsf{trail}\,s_{\mathsf{nth\_bj}\,j}) &> \mu'(\mathsf{trail}\,s_{\mathsf{nth\_confl}\,(1+j)})\end{aligned}$$

This is the core point of the argument. Combined with $\mu'(\mathsf{trail}\,s_{\mathsf{nth\_confl}\,j}) < 2^V$, the final bound can be derived (by yet another induction).

In *Automated Reasoning*, and in my formalization, Theorem 3.10 is also used to establish the termination of CDCL_W+stgy. However, the argument for the termination of CDCL_NOT also applies to CDCL_W irrespective of the strategy, a stronger result. To lift this result, I must show that the calculus CDCL_W refines CDCL_NOT.

3.2.4. Connection with Abstract CDCL

It is interesting to show that CDCL_W refines CDCL_NOT_merge, to establish beyond doubt that CDCL_W is a CDCL calculus and to lift the termination proof and any other general results about CDCL_NOT_merge. The states are easy to connect: I interpret a CDCL_W tuple (M, N, U, C) as a CDCL_NOT pair $(M, N \uplus U)$, ignoring C.

The main difficulty is to relate the low-level conflicts-related CDCL_W rules to their high-level counterparts. My solution is to introduce an intermediate calculus, called CDCL_W_merge, that combines all consecutive low-level transitions Skip, Resolve, and Jump into a single transition. This calculus refines both CDCL_W and CDCL_NOT_merge and is sufficiently similar to CDCL_W so that I can transfer termination and other properties from CDCL_NOT_merge to CDCL_W through it.

Whenever the CDCL_W calculus performs a low-level sequence of transitions of the form Conflict (Skip | Resolve)* Jump?, the CDCL_W_merge calculus performs a single transition of a new rule that subsumes all four low-level rules:

Reduce+Maybe_Jump $\quad S \Longrightarrow_{\mathsf{CDCL\_W\_merge}} S''$
$\quad$ if $S \Longrightarrow_{\mathsf{Conflict}} S' \Longrightarrow^{!}_{\mathsf{Skip,\,Resolve,\,Jump}} S''$

When simulating CDCL_W_merge in terms of CDCL_NOT, two interesting scenarios arise. First, Reduce+Maybe_Jump's behavior may comprise a backjump: The rule can be simulated using CDCL_NOT_merge's Learn+Backjump rule. The second scenario arises when the conflict clause is reduced to the empty clause $\perp$, leading to a conclusive final state. Then, Reduce+Maybe_Jump has no counterpart in CDCL_NOT_merge. The two calculi are related as follows: If $S \Longrightarrow_{\mathsf{CDCL\_W\_merge}} S'$, either $S \Longrightarrow_{\mathsf{CDCL\_NOT\_merge}} S'$ or S is a conclusive state. Since CDCL_NOT_merge is well founded, so is CDCL_W_merge. This implies that CDCL_W without Restart terminates.

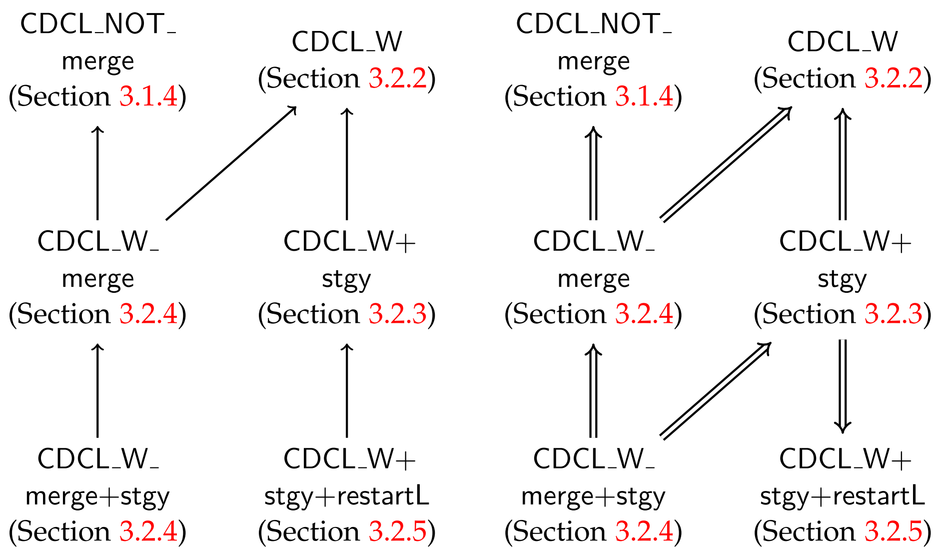
Since CDCL_W_merge is mostly a rephrasing of CDCL_W, it makes sense to restrict it to a *reasonable* strategy that prefers the rules Propagate and Reduce+Maybe_Jump over Decide, yielding CDCL_W_merge+stgy. The two strategy-restricted calculi have the same end-to-end behavior:

$$S \Longrightarrow^{!}_{\mathsf{CDCL\_W\_merge+stgy}} S' \leftrightarrow S \Longrightarrow^{!}_{\mathsf{CDCL\_W+stgy}} S'$$

3.2.5. A Strategy with Restart and Forget

I could use the same strategy for restarts as in Section 3.1.5, but I prefer to exploit Theorem 3.10, which asserts that no relearning is possible. Since only finitely many different duplicate-free clauses can ever be learned, it is sufficient to increase the number of learned clauses between two restarts to ensure termination. This criterion is the norm in modern SAT solvers. The lower bound on the number of learned clauses is given by an unbounded function $f :: nat \Rightarrow nat$. In addition, I allow an arbitrary subset of the learned clauses to be forgotten upon a restart but otherwise forbid Forget. The calculus C+restartL that realizes these ideas is defined by the two rules

Restart $\quad (S, n) \Longrightarrow_{C\mathsf{+restartL}} (S''', n+1)$
$\quad$ if $S \Longrightarrow^{*}_{C} S' \Longrightarrow_{\mathsf{Restart}} S'' \Longrightarrow^{*}_{\mathsf{Forget}} S'''$ and
$\quad |\mathsf{learned}\ S'| - |\mathsf{learned}\ S| \geq f\ n$



(a) Syntactic dependencies   **(b)** Refinements

Figure 3.3.: Connections involving the refined calculi

Finish $\quad (S, n) \Longrightarrow_{C+\mathsf{restartL}} (S', n+1) \quad$ if $S \Longrightarrow^{!}_{C} S'$

I formally proved that CDCL_W+stgy+restartL is totally correct. Figure 3.3 summarizes the situation, following the conventions of Figure 3.1.

3.2.6. Incremental Solving

SMT solvers combine a SAT solver with theory solvers (e.g., for uninterpreted functions and linear arithmetic). The main loop runs the SAT solver on a clause set. If the SAT solver answers “unsatisfiable,” the SMT solver is done; otherwise, the main loop asks the theory solvers to provide further, theory-motivated clauses to exclude the current candidate model and force the SAT solver to search for another one. This design crucially relies on incremental SAT solving: The possibility of adding new clauses to the clause set C of a conclusive satisfiable state and of continuing from there.

As a step towards formalizing SMT (or incremental SAT solving), I designed a three-rule calculus CDCL_W+stgy+incr that provides incremental solving on top of CDCL_W+stgy:

Add_Nonconflict$_C$ $\quad (M, N, U, \top) \Longrightarrow_{\mathsf{CDCL\_W+stgy+incr}} S'$
$\quad$ if $M \not\models \neg C$ and $(M, N \uplus \{C\}, U, \top) \Longrightarrow^{!}_{\mathsf{CDCL\_W+stgy}} S'$

$\mathsf{Add\_Conflict}_C \quad (M'LM, N, U, \top) \Longrightarrow_{\mathsf{CDCL\_W+stgy+incr}} S'$
if $LM \vDash \neg C$, $-L \in C$, M' contains no literal of C, and
$(LM, N \uplus \{C\}, U, C) \Longrightarrow^{!}_{\mathsf{CDCL\_W+stgy}} S'$

I first run the CDCL_W+stgy calculus on a clause set N, as usual. If N is satisfiable, I can add a nonempty, duplicate-free clause C to the set of clauses and apply one of the two above rules. These rules adjust the state and relaunch CDCL_W+stgy.

Theorem 3.11 (Partial Correctness [41, *incremental_conclusive_state*] ☑). *If S is conclusive and $S \Longrightarrow_{\mathsf{CDCL\_W+stgy+incr}} S'$, then S' is conclusive.*

The key is to prove that the structural invariants that hold for CDCL_W+stgy still hold after adding the new clause to the state. Then the proof is easy because I can reuse the invariants I have already proved about CDCL_W+stgy.

3.2.7. Backjump and Conflict Minimization

In order to prepare the refinement to code, I slightly changed and adapted the Jump to be able to express conflict clause minimization. It consists of removing some literals from the conflict clause while ensuring that the clause is still entailed by the other clauses. Typically, literals that have be determined to be false are removed from the conflict clause, because they do not only take space memory but do not change whether the clause can be used to propagate a value or find a conflict. The rule with conflict minimization is:

Jump $(M' \cdot K^{\dagger} M, N, U, D \vee L) \Longrightarrow_{\mathsf{CDCL\_W}} (L^{D' \vee L} M, N, U \uplus \{D' \vee L\}, \top)$
if L has the level of the current state, D has a lower level, $D' \subseteq D$, $N \uplus U \vDash D' \vee L$, and D' has the same level as K

Instead of learning the clause $D \vee L$, the clause $D' \vee L$ is learned, which can be smaller.

One of the invariants on CDCL states that the analyzed clause is entailed by the clauses, i.e. $N \uplus U \vDash N \vee L$. Therefore, the original Jump is a special case of Jump. Minimizing the conflict clause is optional. All results described remains true with the enhanced Jump, including the link to CDCL_NOT.

3.3. A Naive Functional Implementation of CDCL, IsaSAT-0

Sections 3.1 and 3.2 presented variants of DPLL and CDCL as parameterized transition systems, formalized using locales and inductive predicates. I now present a deterministic SAT solver that implements CDCL_W+stgy, expressed as a functional program in Isabelle.

When implementing a calculus, I must make many decisions regarding the data structures and the order of rule applications. My functional SAT solver, called IsaSAT-0, is very naive and does not feature any optimizations beyond those already present in the CDCL_W+stgy calculus; in Chapter 5, I will refine the calculus further to capture the two-watched-literal optimization and present an imperative implementation relying on mutable data structures instead of lists.

For my functional implementation, I choose to represent states by tuples (M, N, U, D), where propositional variables are coded as natural numbers and multisets as lists. Each transition rule in CDCL_W+stgy is implemented by a corresponding function. For example, the function that implements the Propagate rule is given below:

definition do_propagate_step :: *state* ⇒ *state* **where**
  do_propagate_step S =
  (case S of
    $(M, N, U, \top)$ ⇒
    (case find_first_unit_propagation M $(N \mathbin{@} U)$ of
      Some (L, C) ⇒ (Propagated L $C \cdot M, N, U, \top$)
    | None ⇒ S)
  | S ⇒ S)

The data structures are exactly those used to represent CDCL, except that multisets have been replaced by lists. The trail is only represented by a list of propagated or decided literals and testing the polarity of a literals is done by iteration over the list.

The functions corresponding to the different rules are combined into a single function that performs one step. The combinator do_if_not_equal takes a list of functions implementing rules and tries to apply them in turn, until one of them has an effect on the state:

fun do_cdcl_step :: *state* ⇒ *state* **where**
  do_cdcl_step S = do_if_not_equal [do_conflict_step,
    do_propagate_step, do_skip_step, do_resolve_step,
    do_backtrack_step, do_decide_step] S

The main loop applies do_cdcl_step until the transition has no effect, meaning that no further CDCL transition is possible:

function do_all_cdcl$_W$_stgy :: *state* $\Rightarrow$ *state* **where**
do_all_cdcl$_W$_stgy S = (let S' = do_cdcl_step S in
if $S' = S$ then S else do_all_cdcl_stgy S')

The main loop is a recursive program, specified using the **function** command [62]. For Isabelle to accept the recursive definition of the main loop as a terminating program, I must discharge a proof obligation stating that its call graph is well founded. This is a priori unprovable: The solver is not guaranteed to terminate if starting in an arbitrary state.

To work around this, I restrict the input by introducing a subset type that contains a strong enough structural invariant, including the duplicate-freedom of all the lists in the data structure. With the invariant in place, it is easy to show that the call graph is included in the CDCL_W+stgy calculus, allowing me to reuse its termination argument. The partial correctness theorem can then be lifted, meaning that the SAT solver is a decision procedure for propositional logic.

The final step is to extract running code. Using Isabelle's code generator [49], I can translate the program to Haskell, OCaml, Scala, or Standard ML. The resulting program is syntactically analogous to the source program in Isabelle, including its dependencies, and uses the target language's facilities for datatypes and recursive functions with pattern matching. Invariants on subset types are ignored; when invoking the solver from outside Isabelle, the caller is responsible for ensuring that the input satisfies the invariant. The entire program is about 520 lines long in Standard ML. It is not efficient, due to its extensive reliance on lists, but it satisfies the need for a proof of concept.

3.4. Summary

In this chapter, I have presented my formalization of CDCL: Two accounts are presented and formally connected. Both presentations are formalized in Isabelle with abstract transition systems trying to keep some aspects unspecified (like Decide does not specify how the literal is found). I extend Weidenbach's account for CDCL in two directions: First, I make it incremental. Second, I refine it to executable deterministic functional code, that can be exported from Isabelle. This code, IsaSAT-0, features only very naive heuristics: propagations and conflicts are identified by iterating over all clauses. Decisions also iterate over all clauses and stop on the first unset literal.

I use CDCL_W in two ways in the remaining of this thesis. First, I extend it to a CDCL calculus with branch-and-bound (Chapter 4). This stays at the level of a transition system. Second, I extend it towards more efficient execution by adding the two-watched-literal scheme to identify propagation and conflict in a more efficient manner and I add more efficient heuristics and imperative data structures (Chapters 5 and 6).

4. CDCL with Branch and Bounds

In this chapter, I extend the formalization of CDCL presented in the previous chapter. I introduce an optimizing CDCL, called OCDCL (Section 4.1). Given a cost function on literals, it finds an optimal total model. OCDC: is described as an abstract non-deterministic transition system and has the conflict analysis for the first unique implication point built-in. It is well suited for a formalization as its core rules are exactly the rule of the calculus I have formalized earlier. My formalization tries to reuse as much from my previous formalization as possible and especially tries to avoid copy-paste, thanks to the abstractions on states developed earlier. Therefore, I actually develop a framework to express CDCL extensions with branch and bounds, $\mathsf{CDCL}_{\mathrm{BnB}}$. This framework is then instantiated to get OCDCL (Section 4.2). The main idea is to consider the transitions of $\mathsf{CDCL}_{\mathrm{BnB}}$ as special cases of CDCL transitions. This makes it possible to reuse the proofs I have already done previously.

One limitation of OCDCL is that only optimal *total* models can be found. To overcome the limitation of totality of my calculus, I use the dual rail encoding to reduce finding a partial optimal model into a total optimal model (Section 4.3). I also formalized this in Isabelle (Section 4.4). Formalization and verifications in proof assistants are often justified by the idea that they can be reused and extended. This is exactly what we are doing here. In the formalization, we eventually solve the problem of finding optimal models with respect to the two types of valuations using the very same framework. The OCDCL formalization amounts to around 3300 lines of proof. This is small compared to the 2600 lines of shared libraries, plus 6000 lines for the formalization of CDCL [20], and 4500 lines for Nieuwenhuis et al.'s account for CDCL [92]. Thus, thirdly, we show that one of the standard arguments supporting formal verification, the reuse of already proved results, works out in the case of CDCL and OCDCL. The overall formalization took around 1.5 person − month of work and was easy to do. The extension to partial valuations amounts to 1300 lines. We further demonstrate the potential of reuse, by applying our framework to two additional problems: MAX-SAT and covering models. Again the results from OCDCL and the framework, respectively could be reused. The overall effort for the two extensions was

800 and 400 lines, respectively.

Another famous problem, MAX-SAT, can be reduced to OCDCL by adding new literals. Another calculus, a clause covering CDCL is developed in this framework (Section 4.5). It makes it possible to find a set models such that every literal true at least once.

Finally, I give some ideas how to extend the formalization further (Section 4.6), either to add new rules or to restrict applications of current rules.

4.1. Optimizing Conflict-Driven Clause Learning

I assume a total cost function cost on the set of all literals $Lit(\Sigma)$ over Σ into the positive rationals, $\mathrm{cost} : Lit(\Sigma) \to \mathbb{Q}^+$, where our results do not depend specifically on the positive rationals (including 0), e.g., they also hold for the naturals or positive reals. The cost function can be extended to a pair of a literal and a partial valuation $\mathcal{A}$ by $\mathrm{cost}(L, \mathcal{A}) := \mathrm{cost}(L)$ if $\mathcal{A} \models L$ and $\mathrm{cost}(L, \mathcal{A}) := 0$ if L is not defined. The function can be extended to (partial) valuations by $\mathrm{cost}(\mathcal{A}) = \sum_{L \in Lit(\Sigma)} \mathrm{cost}(\mathcal{A}, L)$. We identify partial valuations with consistent sequences $M = [L_1 \ldots L_n]$ of literals. Trails are always consistent. A valuation I is total over clauses N when all atoms of N are defined in I.

The optimizing conflict-driven clause learning calculus (OCDCL) solves the weighted SAT problem on total valuations. Compared with a normal CDCL state, a component O is added resulting in a five tuple $(M; N; U; D; O)$. O either stores the best model so far or $\top$. I extend the cost function to $\top$ by defining $\mathrm{cost}\,(\top) = \infty$ (i.e., $\top$ is the worst possible outcome). OCDCL is a strict extension of the CDCL rules with additional rules to take the cost of models into account. The additional component O is ignored by the original CDCL rules.

The start state for some clause set N is $(\epsilon; N; \emptyset; \top; \top)$. The calculus searches for models in the usual CDCL-style. Once a model is found, it is ruled out by generating a conflict clause resulting from its negation which is then processed by the standard CDCL conflict analysis (rule Improve, defined below). If a partial model M already exceeds the current cost bound, a conflict clause is generated (rule ConflOpt, defined below). The OCDCL calculus always terminates in deriving the empty clause $\bot$. If in this case $O = \top$, then N was unsatisfiable. Otherwise, O contains a cost-optimal total model for N.

The level of a literal is the number of decisions left of its atom in the trail M. I lift the definition to clauses, by defining the level of a clause as the maximum of the levels of its literals or 0 if it is empty.

First, there are three rules involving the last component O that implement a branch-and-bound approach on the models:

Improve $(M;N;U;\top;O) \Longrightarrow_{\text{OCDCL}} (M;N;U;\top;M)$

provided $M \models N$, M is total over N and $\text{cost}(M) < \text{cost}(O)$.

ConflOpt $(M;N;U;\top;O) \Longrightarrow_{\text{OCDCL}} (M;N;U;\neg M;O)$

provided $O \neq \top$ and $\text{cost}(M) \geq \text{cost}(O)$.

Prune $(M;N;U;\top;O) \Longrightarrow_{\text{OCDCL}} (M;N;U;\neg M;O)$

provided for all total trail extensions MM' of M, $\text{cost}(MM') \geq \text{cost}(O)$.

The Prune rule is not necessary for the correctness and completeness. In practice, Prune would be an integral part of any optimizing solver where a lower-bound on the cost of all extensions of M is maintained for efficiency.

The other rules are unchanged imports from the CDCL calculus. They simply ignore the additional component O. The rules Propagate and Decide extend the trail searching for a model. The rule ConflSat detects a conflict. All three rules implement the classical CDCL-style model search until conflict or success.

Propagate $(M;N;U;\top;O) \Longrightarrow_{\text{OCDCL}} (L^{C\vee L}M;N;U;\top;O)$

provided $C \vee L \in N \cup U$, $M \models \neg C$, L is undefined in M.

Decide $(M;N;U;\top;O) \Longrightarrow_{\text{OCDCL}} (L^{\dagger}M;N;U;\top;O)$

provided L is undefined in M, contained in N.

ConflSat $(M;N;U;\top;O) \Longrightarrow_{\text{OCDCL}} (M;N;U;D;O)$

provided $D \in N \cup U$ and $M \models \neg D$.

Once a conflict has been found, it is analyzed to derive a new clause that is then a first unique implication point [16].

Skip $(L^{C\vee L}M;N;U;D;O) \Longrightarrow_{\text{OCDCL}} (M;N;U;D;O)$

provided $D \notin \{\top, \bot\}$ and $\neg L$ does not occur in D.

Resolve $(L^{C\vee L}M;N;U;D \vee -L;O) \Longrightarrow_{\text{OCDCL}} (M;N;U;D \vee C;O)$

provided D is of level k, where k is the number of decisions in M.

Backtrack $(M_2K^{\dagger}M_1;N;U;D \vee L;O) \Longrightarrow_{\text{OCDCL}} (L^{D\vee L}M_1;N;U \cup \{D \vee L\};\top;O)$

provided L is of level k and D and K are of level $i < k$.

The typical CDCL-learned-clause mechanism in the context of searching for (optimal) models does not apply with respect to partial valuations. Consider the clause set $N = \{P \vee Q\}$ and cost function $\text{cost}(P) = 3$, $\text{cost}(\neg P) = \text{cost}(Q) = \text{cost}(\neg Q) = 1$. An optimal-cost model based on total valuations is $[\neg P, Q]$ at overall cost 2, whereas an optimal-cost model based on partial valuations is just $[Q]$ at cost 1. The cost of undefined variables is always considered to be 0. Now the run of an optimizing branch-and-bound CDCL framework may start by deciding $[P^{\dagger}]$ and detect that this is already a model for N. Hence, it learns $\neg P$ and establishes 3 as the best current bound on an optimal-cost model. After backtracking, it can propagate Q with trail $[Q^{P\vee Q}, \neg P^{\neg P}]$ resulting in a model of cost 2 learning the clause $P \vee \neg Q$. The resulting clause set $\{P \vee Q, \neg P, P \vee \neg Q\}$ is unsatisfiable and hence 2 is considered to be the cost-optimal result. The issue is that although this CDCL run already stopped as soon as a partial valuation (trail) is a model for the clause set, it does not compute the optimal result with respect to partial valuations. From the existence of a model with respect to a partial valuation $[P]$ I cannot conclude the clause $\neg P$ to eliminate all further models containing P, because P could be undefined.

Definition 1 (Reasonable OCDCL Strategy)**.** *An OCDCL strategy is* reasonable *if* ConflSat *is preferred over* ConflOpt*, which is preferred over* Improve*, which is preferred over* Propagate*, which is preferred over the remaining rules.*

Lemma 1 (OCDCL Termination, wf_ocdcl$_W$ ☑)**.** *OCDCL with a reasonable strategy terminates in a state $(M; N; U; \bot; O)$.*

Proof. If the derivation started from $(\epsilon, N, \emptyset, \top, top)$, the following function is a measure for OCDCL:

$$\mu((M; N; U; D; O)) = \begin{cases} (3^n - 1 - |U|, 1, n - |M|, \mathrm{cost}(O)) & \text{if } D = \top \\ (3^n - 1 - |U|, 0, |M|, \mathrm{cost}(O)) & \text{otherwise} \end{cases}$$

It is decreasing for the lexicographic order. The hardest part of the proof is the decrease when Backjump: $3^n - 1 - |U \cup \{D \vee L\}| < 3^n - 1 - |U|$ is decreasing since no clause is relearned. The proof is similar to the one for CDCL. □

Theorem 4.1 (Correctness, full_ocdcl$_w$_stgy_no_conflicting_clause_from_init_state ☑)**.** *An OCDCL run with a reasonable strategy starting from state $(\epsilon; N; \emptyset; 0; \top; \epsilon)$ terminates in a state $(M; N; U; 0; \bot; O)$. If $O = \epsilon$ then N is unsatisfiable. If $O \neq \epsilon$ then $O \models N$ and for any other total model M' with $M' \models N$ it holds* $\mathrm{cost}(M') \geq \mathrm{cost}(O)$.

The rule Improve can actually be generalized to situations where M is not total, but all literals with weights have been set.

Improve$^+$ $\quad (M; N; U; \top; O) \implies_{\mathrm{OCDCL}} (M; N; U; \top; MM')$

provided $M \models N$, the model MM' is a total extension, $\mathrm{cost}(M) < \mathrm{cost}(O)$, and for any total extension MM'' of the trail, it holds $\mathrm{cost}(M) = \mathrm{cost}(MM'')$.

Lemma 2 (Improve$^+$, wf_ocdcl$_W$_p ☑ and full_ocdcl$_W$_p_stgy_no_conflicting_clause_from_init_state ☑)**.** *The rule Improve can be replaced by rule Improve$^+$: All previously established OCDCL properties are preserved.*

The rules ConflOpt can produce very long conflict clauses. Even with conflict minimization, they will contain the negation of all decision literals from the trail. It can be advantageous to generate the conflict composed of only the literals with a non-zero weight, i.e., $\neg\{L \in M \mid \text{cost}\, L > 0\}$ instead of $\neg M$. In this case a more general Skip is required, such that the eventual conflict before application of Backtrack contains one literal of highest level. As said, this is not always beneficial, e.g., the rule used in Lingeling [11] switches between the two options by taking the shortest clause.

The rules Restart and Forget can also be added to OCDCL with the same well-known implications from CDCL. For example, completeness is only preserved if Restart is applied after longer and longer intervals.

4.2. Formalization of OCDCL

If I ignore the Improve rule, the remaining OCDCL transitions are very similar to a generalized CDCL (Section 4.2.1)

In the formalization, I abstract over this clause set by using instead a set $\mathcal{T}_N(O)$ and use a predicate is_improving $M\ M'\ O$ to indicate that Improve can be applied. This yields a more abstract branch-and-bound calculus CDCL_{BnB} (Section 4.2.2). CDCL_{BnB} is seen as a special case of CDCL, where the additional clauses $\mathcal{T}_N(O)$ are part of the initial set of clauses. This reduces many proofs to reusing their CDCL counterpart: ConflOpt becomes equivalent to ConflSAT, since it picks a clause of $\mathcal{T}_N(O)$. I do not specify the type of O (Section 4.2.3). This reduces the burden to develop the new variant and makes it possible to reuse many proofs and especially all the invariants about the states: I neither have to redefine nor reprove most of them.

I instantiate CDCL_{BnB} to get a generalized version OCDCL_g: The set of clauses $\mathcal{T}_N(O)$ is instantiated by the set $\{D \mid \{\neg C.\ \text{cost}\, C \geq \text{cost}\, O\} \vDash D\}$ (Section 4.2.4). Finally, I specialize OCDCL_g to get OCDCL from Section 4.1 (Section 4.2.5).

4.2.1. OCDCL and CDCL

An OCDCL run is very similar to a CDCL run where the set of clause is enriched each time Improve is applied (see example in Figure 4.1): The negations of all models of weight larger that the optimal model are included in the set of clauses of CDCL, i.e. $\{\neg C \mid \operatorname{cost} C \geq \operatorname{cost} O\}$.

In a CDCL run corresponding to an OCDCL run, each time Improve is run, the CDCL state changes due to the new clauses. Hence, I have separated the different phases with lines. It is possible to see those Improve as a restart, followed by adding new clauses and repropagating and redeciding the literals on the trail.

This additional set of clauses is too weak to be able to express the rule Prune in the set of clauses. Therefore, I actually use the clauses that are entailed by the clauses of weight larger than the optimal model found so far, i.e. the clauses D such that $\{\neg C \mid \operatorname{cost} C \geq \operatorname{cost} O\} \vDash D$. In this case, N_2 would contain $\bot$. This in particular means that the CDCL run cannot be run with the strategy.

The run above does not depend on the weights of the information of optimal model. It only depends on the additional set of clauses, written $\mathcal{T}_N$, and a predicate is_improving $M \;\; O$ to indicate that Improve can be applied. This is the approach I use in the formalization: I work on the more abstract calculus, called $\mathsf{CDCL_{BnB}}$, and only instantiate it to get OCDCL.

4.2.2. Branch-and-Bound Calculus, $\mathsf{CDCL_{BnB}}$

I use a similar approach to our CDCL formalization with an abstract state and selectors, except that I add an additional component representing information on the optimizing branch-and-bound part of the calculus. I do not yet specify the type of this additional component. I parameterize the calculus by a set of clauses $\mathcal{T}_N$ that contains the conflicting clauses that can be used by the rules ConflOpt and Prune, and a predicate is_improving $M\ M'\ O$ to indicate that Improve can be applied. For weights, the predicate is_improving $M\ M'\ O$ means that the current trail M is a model, M' is the information that will be stored, and O is the currently stored information. $\mathcal{T}$ represents all the clauses that are entailed. I require that:

- the atoms of $\mathcal{T}_N\,(O)$ are included in the atoms of N. I do not introduce new variables.
- the clauses of $\mathcal{T}_N\,(O)$ do not contain duplicate literals. Duplicates are incompatible with the conflict analysis.
- if is_improving $M\ M'\ O$, then $\mathcal{T}_N\,(O) \subseteq \mathcal{T}_N\,(M')$.
- if is_improving $M\ M'\ O$, then $\neg M \in \mathcal{T}_N\,(M')$.

OCDCL Run	Corresponding CDCL Run
$(\epsilon, N, \emptyset, \top, \top)$	$(\epsilon, N \cup N_0, \emptyset, \top, \top)$ where $N_0 = \emptyset$
$\Longrightarrow^{\star}_{\mathsf{Decide}} (P^\dagger Q^\dagger, N, \emptyset, \top, \top)$	$\Longrightarrow^{\star}_{\mathsf{Decide}} (P^\dagger Q^\dagger, N \cup N_0, \emptyset, \top)$
$\Longrightarrow_{\mathsf{Improve}} (P^\dagger Q^\dagger, N, \emptyset, \top, PQ)$	$\Longrightarrow (P^\dagger Q^\dagger, N \cup N_1, \emptyset, \top)$ where $N_1 = N_0 \cup \{\neg P \vee \neg Q\}$
$\Longrightarrow_{\mathsf{ConflOpt}} (P^\dagger Q^\dagger, N, \emptyset, \neg P \vee \neg Q, PQ)$	$\Longrightarrow_{\mathsf{Conflict}} (P^\dagger Q^\dagger, N \cup N_1, \emptyset, \neg P \vee \neg Q)$
$\Longrightarrow_{\mathsf{Backtrack}} (P^\dagger (\neg Q)^{\neg P \vee \neg Q}, N, \{\neg P \vee \neg Q\}, \top, PQ)$	$\Longrightarrow_{\mathsf{Backtrack}} (P^\dagger (\neg Q)^{\neg P \vee \neg Q}, N \cup N_1, \{\neg P \vee \neg Q\}, \top)$
$\Longrightarrow_{\mathsf{Improve}} (P^\dagger (\neg Q)^{\neg P \vee \neg Q}, N, \{\neg P \vee \neg Q\}, \top, \neg PQ)$	$\Longrightarrow (P^\dagger (\neg Q)^{\neg P \vee \neg Q}, N \cup N_1, \{\neg P \vee \neg Q\}, \top)$ where $N_2 = N_1 \cup \{P \vee \neg Q, \neg P \vee Q\}$
$\Longrightarrow_{\mathsf{ConflOpt}} (P^\dagger \neg Q^\dagger, N, \emptyset, P \vee \neg Q, PQ)$	$\Longrightarrow_{\mathsf{Conflict}} (P^\dagger \neg Q^\dagger, N \cup N_2, \emptyset, \neg P \vee \neg Q)$
$\Longrightarrow_{\mathsf{Backtrack}} (\neg P^{\neg P}, N, \{\neg P \vee \neg Q, \neg P\}, \top, P\neg Q)$	$\Longrightarrow^{\star}_{\mathsf{Backtrack}} (\neg P^{\neg P}, N \cup N_2, \{\neg P \vee \neg Q, \neg P\}, \top)$
$\Longrightarrow_{\mathsf{Propagate}} (\neg P^{\neg P} Q^{P \vee Q}, N, \{\neg P \vee \neg Q, \neg P\}, \top, \neg PQ)$	$\Longrightarrow_{\mathsf{Propagate}} (P^\dagger Q^{P \vee Q}, N \cup N_2, \{\neg P \vee \neg Q, \neg P\}, \top)$
$\Longrightarrow^{\star}_{\mathsf{ConflictOpt+Resolve}} (\epsilon, N, \{\neg P \vee \neg Q, \neg P\}, \bot, \neg PQ)$	$\Longrightarrow^{\star}_{\mathsf{Conflict+Resolve}} (\epsilon, N \cup N_2, \{\neg P \vee \neg Q, \neg P\}, \bot)$

Figure 4.1.: OCDCL transitions and corresponding CDCL transitions for $N = \{P \vee Q\}$ with $\operatorname{cost} P = \operatorname{cost} Q = 1$ and $\operatorname{cost} \neg P = \operatorname{cost} \neg Q = 0$ where the horizontal lines separate two successive CDCL run

The rules $\mathsf{ConflOpt}_{\mathsf{BnB}}$, $\mathsf{Improve}_{\mathsf{BnB}}$, and $\mathsf{Backtrack}_{\mathsf{BnB}}$ are defined as follows:

ConflOpt$_{\mathbf{BnB}}$ $(M;N;U;k;\top;O) \Longrightarrow_{\mathrm{OCDCL}} (M;N;U;k;\neg M;O)$

provided $\neg M \in \mathcal{T}_N(O)$

Improve$_{\mathbf{BnB}}$$^+$ $(M;N;U;k;\top;O) \Longrightarrow_{\mathrm{OCDCL}} (M;N;U;k;\neg M;M')$

provided $\mathsf{is\_improving}$ M M' O holds

Backtrack$_{\mathbf{BnB}}$ $(M_2K^{\dagger}M_1;N;U;D \vee L;O) \Longrightarrow_{\mathrm{OCDCL}} (L^{D' \vee L}M_1;N;U \cup \{D' \vee L\};\top;O)$

provided L is of maximum level, $D' \subseteq D$, $N + U + \mathcal{T}_N(O) \vDash D' \vee L$, D' and K are of same level i, and i strictly less than the maximum level

I can simply embed into our CDCL formalization the states with the weights and reuse the previous definitions, properties, and invariants by mapping OCDCL states (M, N, U, D, O) to CDCL states (M, N, U, D). For example, I can reuse the Decide rule and the proofs on it. At this level, anything can be stored in O.

Compared with the rule from Section 4.1, I make it possible to express conflict-clause minimization: Instead of $D \vee L$, a clause $D' \vee L$ is learned such that $D' \subseteq D$ and $N + U + \mathcal{T}_N(O) \vDash D' \vee L$. While all other CDCL rules are reused, the Backtrack rule is not reused for OCDCL: If I had reused Backtrack from CDCL, only the weaker entailment $N + U \vDash D' \vee L$ would be used. The latter version is sufficient to express conflict minimization as implemented in most SAT solvers [113], but the former is stronger and makes it possible for example to remove decision literals without cost from D.

I use the $\mathsf{Improve}^+$ rule instead of the Improve rule, because the latter is a special case of the former. The strategy favors Conflict and Propagate over all other rules. I do not need to favor ConflictOpt over the other rules for correctness, although doing so helps in an implementation.

4.2.3. Embedding into CDCL

In order to reuse the proof I did previously about CDCL, $\mathsf{CDCL}_{\mathsf{BnB}}$ is seen as a special instance of CDCL by mapping the states $(M;N;U;D;O)$ to the CDCL state $(M;N \cup \mathcal{T}_N(O);U;D)$.

In Isabelle, the most direct solution would be to instantiate the CDCL calculus with a selector returning $N + \mathcal{T}_N(O)$ instead of just N. For technical reasons, I cannot do so: This confuses Isabelle, because it leads to duplicated theorems and notations. Instead, I add an explicit conversion from $(M;N;U;D;O)$ to $(M;N + \mathcal{T}_N(O);U;D)$ and consider CDCL on tuples of the latter.

Except for the Improve rule, every OCDCL rule can be mapped to a CDCL rule: The $\mathsf{ConflictOpt}_{\mathsf{BnB}}$ rule corresponds to the Conflict rule (because it can pick a clause from $\mathcal{T}_N\,(O)$) and the extended Backtrack rule is mapped to CDCL's Backtrack. On the other hand, the Improve rule has no counterpart and requires some new proofs, but adding clauses is compatible with the CDCL invariants.

In our formalization, I distinguish the structural from the strategy-specific properties. The strategy-specific properties ensure that the calculus does not get stuck in a state where I cannot conclude on the satisfiability of the clauses. The strategy-specific properties do not necessarily hold: The clause $\bot$ might be in $\mathcal{T}_N\,(O)$ without being picked by the $\mathsf{ConflictOpt}_{\mathsf{BnB}}$ rule. However, I can easily prove that they hold for $\mathsf{CDCL}_{\mathsf{BnB}}$ and I can reuse the proof I have already done for most transitions. To reuse some proofs on CDCL's Backtrack, I generalized some proofs by removing the assumption $N + U \vDash D' \vee L'$ when not required. This is the only generalization I did on CDCL.

Not all transitions of CDCL can be taken by OCDCL: Propagating of clauses in $\mathcal{T}_N\,(O)$ is not possible. The structural properties are sufficient to prove that OCDCL is terminating as long as $\mathsf{Improve}^+$ can be applied only finitely often, because the CDCL calculus is terminating. At this level, $\mathsf{Improve}^+$ is too abstract to prove that it terminates. With the additional assumptions that Improve can always be applied when the trail is a total model satisfying the clauses (if one exists), I show that the final set of clauses is unsatisfiable.

4.2.4. Instantiation with weights, $\mathbf{OCDCL}_g$

Finally, I instantiate $\mathcal{T}_N(O)$ with weights and save the best current found model in O. I assume the existence of a cost function that is monotone with respect to inclusion:

locale cost =
  fixes cost :: $'v$ *literal multiset* $\Rightarrow$ $'c$
  assumes $\forall C.$ consistent_interp $B \wedge$ distinct_mset $B \wedge A \subseteq B \longrightarrow$
    cost $A \leq$ cost B

We assume that cost is function is monotone with respect to inclusion for consistent duplicate-free models. This is natural for trails, which by construction do not contain duplicates. The monotonicity is less restrictive than the condition from Section 4.1, which mandates that the cost is a sum over the literals. I take

$$
\begin{aligned}
\mathcal{T}_N(O) = \{C.\ & \mathrm{atom}(C) \subseteq \mathrm{atom}(N) \\
& \wedge C \text{ is not a tautology nor contains duplicates} \\
& \wedge \{-D.\ \mathrm{cost}\,(D) \geq \mathrm{cost}\,(O)\} \vDash C\}\} \\
\text{is\_improving } M\ M'\ O \leftrightarrow\ & M' \text{ is a total extension of } M,\ M \vDash N, \\
& \text{any total extensions of } M \text{ has the same cost, and} \\
& \mathrm{cost}\, M < \mathrm{cost}\, O
\end{aligned}
$$

and then discharge the assumptions over it.

OCDCL_g inherit from the invariants from $\mathsf{CDCL}_{\mathsf{BnB}}$. For termination, I only have to prove that $\mathsf{Improve}^+$ terminates to reuse the proof I already made on $\mathsf{CDCL}_{\mathsf{BnB}}$. The key property of OCDCL_g is the following:

Isabelle Lemma 4.2 (entails_too_heavy_clauses_too_heavy_clauses ☑). *If I is a total consistent model of N, then either* $\mathrm{cost}\,(I) \geq \mathrm{cost}\,(O)$ *or I is a total model of* $N \cup \mathcal{T}_N\,(O)$.

Proof. Assume $\mathrm{cost}\,(I) < \mathrm{cost}\,(O)$. First, I can show that $I \vDash \{-C \mid \mathrm{cost}\,(C) \geq \mathrm{cost}\,(O)\}$. Let D be a clause of $\{-C \mid \mathrm{cost}\,(C) \geq \mathrm{cost}\,(O)\}$. C is not a subset of I (by monotonicity of cost, $\mathrm{cost}\,(I) \geq \mathrm{cost}\,(C)$). Therefore, there is at least a literal L in C such that $-L$ in I. Hence $I \vDash C$.

By transitivity, since I is total, I is also a model of $\mathcal{T}_N\,(O)$ and therefore of $N \cup \mathcal{T}_N\,(O)$. □

This is the proof that breaks if partial models are allowed. Some additional proofs are required to specify the content of the component O. First, the sequence of literals O is always a total consistent model. This property cannot be inherited from the correctness of CDCL, because it does not express any property about the component O.

4.2.5. OCDCL

Finally, I can refine the calculus to precisely the rules expressed in Section 4.1. I define two calculi: one with only the rule $\mathsf{Improve}$, and the other with both $\mathsf{Improve}^+$ and Prune. In both cases, the rule $\mathsf{ConflictOpt}$ is only applied when $\mathrm{cost}\,(M) > \mathrm{cost}\,(O)$ and is therefore a special case of $\mathsf{ConflictOpt}_{\mathsf{BnB}}$. The Prune rule is also seen as a special case of $\mathsf{ConflictOpt}_{\mathsf{BnB}}$. Therefore, every transition is also a transition of OCDCL_g. Moreover, since final states of both calculi are the same, a completed run of OCDCL is also a completed run of OCDCL_g. Therefore, the correctness theorem can be inherited.

Overall, the full formalization was easy to do, once I got the idea how to see OCDCL as a special case of CDCL. Formalizing a changing target is different than an already fixed version calculus: I had to change our formalization several times to take into account additional rules: The Prune rule requires to use $\{D \mid \{-C.\ \text{cost}(C) \geq \text{cost}(O)\} \vDash D\}$, while the set of clauses $\{-C.\ \text{cost}(C) \geq \text{cost}(O)\}$ is sufficient for Improve^{+}.

4.3. Optimal Partial Valuations

To reduce the search from optimal partial valuations to optimal total valuations, I use the dual rail encoding [27, 98]. For every proposition variable P, I create two variables P^1 and P^0 indicating that P is defined positively or negatively. I also add the clause $\neg P^1 \vee \neg P^0$ to ensure that P is not defined positively and negatively at the same time. The resulting set is called $\text{penc}(N)$.

More precisely, the encoding penc is defined on literals by $\text{penc}(P) := (P^1)$, $\text{penc}(\neg P) := (P^0)$, and lifted to clauses and clause sets by $\text{penc}(L_1 \vee \dots \vee L_n) := \text{penc}(L_1) \vee \dots \vee \text{penc}(L_n)$, and, $\text{penc}(C_1 \wedge \dots \wedge C_m) := \text{penc}(C_1) \wedge \dots \wedge \text{penc}(C_m)$. I call Σ' the set of all newly introduced atoms.

The important property of this encoding is that $\neg P^1$ does not entail P^0: If P is not positive, it does not have to be negative either.

Given the encoding $\text{penc}(N)$ of N the cost function is extended to a valuation $\mathcal{A}'$ on $\Sigma \cup \Sigma'$ by $\text{cost}'(\mathcal{A}') = \text{cost}\left(\{L \mid L^1 \in \mathcal{A}'\} \cup \{-L \mid L^0 \in \mathcal{A}'\}\right)$.

Let $\text{pdec}(\mathcal{A}) : P \mapsto \begin{cases} 1 & \text{if } \mathcal{A}(P^1) = 1 \\ 0 & \text{if } \mathcal{A}(P^0) = 1 \\ \text{unset} & \text{otherwise} \end{cases}$ a function that transforms a total model of $\text{penc}(N)$ into a model of N and $\text{pdec}^{-}(\mathcal{A})$ does the opposite transformation, with $\text{pdec}^{-}(\mathcal{A})(P^1) = 1$ if $\mathcal{A}(P) = 1$, $\text{pdec}^{-}(\mathcal{A})(P^1) = 0$ if $\mathcal{A}(P) = 0$, $\text{pdec}^{-}(\mathcal{A})(P^0) = 1$ if $\mathcal{A}(P) = 0$, $\text{pdec}^{-}(\mathcal{A})(P^0) = 0$ if $\mathcal{A}(P) = 1$, unset otherwise.

Lemma 3 (Partial and Total Valuations Coincide Modulo penc, penc_ent_postp ☑ and penc_ent_upostp ☑)**.** *Let N be a clause set.*

1. *If $\mathcal{A} \models N$ for a partial model $\mathcal{A}$ then* $\text{pdec}^{-}(\mathcal{A}) \models \text{penc}(N)$*;*
2. *If $\mathcal{A}' \models \text{penc}(N)$ for a total model $\mathcal{A}'$, then* $\text{pdec}(\mathcal{A}') \models N$*.*

Lemma 4 (penc Preserves Cost Optimal Models, full_encoding_OCDCL_correctness ☑)**.** *Let N be a clause set and* cost *a cost function over literals from N. If*

$\mathcal{A}'$ is a cost-optimal total model for $\text{penc}(N)$ *over* cost'*, resulting in* $\text{cost}'(\mathcal{A}') = m$*, then the partial model* $\text{pdec}\,(\mathcal{A}')$ *is cost-optimal for* N *and* $\text{cost}(\text{pdec}\,(\mathcal{A}')) = m$*.*

Proof. Assume there is a partial model $\mathcal{A}$ for N with $\text{cost}(\mathcal{A}) = k$. The model $\text{pdec}^{-}\,\mathcal{A}$ is another model of N. As $\mathcal{A}'$ is cost optimal, $\text{cost}'(\text{pdec}^{-}\,(\mathcal{A})) \geq \text{cost}'(\mathcal{A}')$. Moreover, $\text{cost}'(\text{pdec}\,(\mathcal{A}')) = \text{cost}(\mathcal{A}')$ and $\text{cost}'(\text{pdec}\,(\mathcal{A})) = \text{cost}(\mathcal{A})$. Ultimately, $\mathcal{A}$ is not better than $\mathcal{A}'$ and $\mathcal{A}'$ has cost m. □

$\text{penc}(N)$ contains $|N| + |\Sigma|$ clauses. Recall that for n propositional variables there are 2^n total valuations and 3^n partial valuations.

Non-Machine-Checked Lemma 4.3 (OCDCL on the Encoding)**.** *Consider a reasonable CDCL run on* $\text{penc}(N)$*. If rule* Decide *is restricted to deciding either P^1 or P^0 for any propositional variable, and* ConflOpt *only considers the decision literals out of M as a conflict clause, then OCDCL performs at most 3^n Backtrack steps.*

Proof. Using the strategy on P^1 or P^0 there are exactly three combinations that can occur on a trail: a decision P^1 and $\neg P^0$ by propagation, or the other way round, or $\neg P^1$ and $\neg P^0$. In summary, for each propositional variable, a run considers at most 3 cases, overall 3^n cases for n different variables in N. □

4.4. Formalization of the Partial Encoding

In Isabelle, total valuations are defined by Herbrand interpretations, i.e., a set of all true atoms (all others being implicitly false) [107], but I use partial models for CDCL, similar to a trail by adding a predicate to indicate whether a model is total. I distinguish between literals that can have a weight $\Delta\Sigma$ from the others ($\Sigma \setminus \Delta\Sigma$) that can be left unchanged by the encoding.

The proofs are very similar to the proofs described in Section 4.3. I instantiate the OCDCL calculus with the cost' function:

interpretation OCDCL **where** $\text{cost} = \text{cost}'$

I have to prove the proof obligation that cost' is monotone.

Finally, I can prove the correctness Theorem 4. The formalization is 800 lines long for the encoding, and 500 additional lines to restrict Decide.

I have not yet formalized the complexity bound of 3^n of Lemma 4.3. So far, I have only verified the correctness of the variant of ConflOpt. It can be seen as a special case of conflict analysis and backtrack thanks to conflict minimization: $(M_1K^\dagger M_2, N, U, \neg(M_1K^\dagger M_2)) \Longrightarrow^\star_{\mathsf{Resolve}} (M_1K^\dagger, N, U, \neg(M_1K^\dagger)) \Longrightarrow^\star_{\mathsf{Backtrack}} (M_1\neg K^{D'}, N, U \cup \{D'\}, \top)$, where D' is the negation of the decisions of $M_1K^\dagger$ and M_2 does not contain any decision. If there are no decision in the trail, I set the conflict to $\bot$.

4.5. Solving Further Optimization Problems

In this section I show how OCDCL can be used to solve MAX-SAT (Section 4.5.1) and can be extended to solve minimal model coverage. Both extensions are verified using Isabelle.

4.5.1. MAX-SAT

The maximum satisfiability problem (MAX-SAT) is a well-known optimization problem [76]. It consists of two clause sets N_H (hard constraints, mandatory to satisfy) and N_S (soft constraints, optional to satisfy). The set N_S comes with a cost function for clauses that are not satisfied. The aim is to find a total model with minimal cost.

Theorem 4.4 (partial_max_sat_is_weight_sat ☑). *Let* $(N_H, N_S, \mathrm{cost})$ *be a MAX-SAT problem and let active* $: N_S \to \Sigma'$ *be an injective and surjective mapping for a set* Σ' *of fresh propositional variables that assigns to each soft constraint an activation variable.*

Let I *be the solution to the OPT-SAT problem* $N = N_H \cup \{\mathit{active}(C) \vee C \mid C \in N_S\}$ *with the cost function* $\mathrm{cost}'(L) = \mathrm{cost}(C)$ *if active*$(C) = L$ *for some* $C \in N_S$ *and* $\mathrm{cost}'(L) = 0$ *otherwise.*

If there is no model I *of* N*, the NAX-SAT problem has no solution. Otherwise,* I *without the additional atoms from* Σ' *is an optimal solution to the MAX-SAT problem.*

Proof.

- N_H is satisfiable iff MAX-SAT has a solution. Therefore, if there is no model I of N, then N_H is unsatisfiable.

- Let $I' = \{L \mid L \in I \wedge \mathrm{atom}(L) \notin \Sigma'\}$. Let J be any other model of $(N_H \cup N_S)$ and J' its total extension to Σ': $J' = J \cup \{\mathrm{active}(C) \mid C \in N_S \wedge J \not\models C\} \cup \{\neg\mathrm{active}(C) \mid C \in N_S \wedge J \models C\}$ to N.

  J' satisfies N_H and is a total consistent model of N. Hence, $\mathrm{cost}'(J') \geq \mathrm{cost}(I)$, because I is the optimal model of N. By definition, $\mathrm{cost}'(I) = \mathrm{cost}(I')$ and $\mathrm{cost}'(J) = \mathrm{cost}(J')$. Therefore, I is an optimal MAX-SAT model. □

4.5.2. A Second Instantiation of $\mathbf{CDCL_{BnB}}$: Model Covering

My second example demonstrates that our framework can be applied beyond OCDCL. I consider the calculation of covering models. Again this is motivated by a product configuration scenario where a propositional variable encodes the containment of a certain component in a product. For product testing, finding a bucket of products is typically required such that every component occurs at least once in the bucket. Translated into propositional logic: given a set N of clauses I search for a set of models $\mathcal{M}$ such that for each propositional variable P occurring in N, $M \vDash P$ for at least one $M \in \mathcal{M}$ or there is no model of N such that P holds.

In order to solve the model covering problem, I define a domination relation: A model is dominated if there is another model that contains more true propositional variables. More formally, if I and J are total models for N, then I is *dominated* by J if $\{P \mid I \vDash P\} \subseteq \{Q \mid J \vDash Q\}$. If a total model is dominated by a model already contained in $\mathcal{M}$, then it is not required in a minimal solution. The model covering can be computed by creating another CDCL extension, where the set $\mathcal{M}$ is explicitly added as a component to a state and used for a branch-and-bound optimization approach, similar to OCDCL. The extension to CDCL are the two additional rules:

ConflCM $\quad (M; N; U; \top; \mathcal{M}) \implies_{\mathsf{CDCL}_{\mathsf{CM}}} (M; N; U; \neg M; \mathcal{M})$

provided for all total extensions MM' with $MM' \models N$, there is an $I \in \mathcal{M}$ which dominates MM'.

Add $\quad (M; N; U; k; \top; \mathcal{M}) \implies_{\mathsf{CDCL}_{\mathsf{CM}}} (M; N; U; k; \top; \mathcal{M} \cup \{M\})$

provided $M \models N$, all literals from N are defined in M and M is not dominated by a model in $\mathcal{M}$.

The $\mathsf{CDCL}_{\mathsf{CM}}$ calculus does not necessarily compute a minimal set of covering models. Minimization is a classical NP-complete problem [59] and can then be done in a second step. This calculus is another instance of $\mathsf{CDCL}_{\mathsf{BnB}}$. In the formalization, we instantiate $\mathsf{CDCL}_{\mathsf{BnB}}$ with:

$$
\begin{aligned}
\mathcal{T}_N(\mathcal{M}) = \{C.\ & \mathrm{atom}(C) \subseteq \mathrm{atom}(N) \\
& \wedge C \text{ is not a tautology nor contains duplicates} \\
& \wedge \{-D.\ \mathsf{is\_dominating}\ \mathcal{M}\ D,\ \mathsf{total}\} \cup N \vDash C\} \\
\mathsf{is\_improving}\ M\ M'\ \mathcal{M} \leftrightarrow\ & M = M' \text{ and } M \vDash N \\
& \text{and } M \text{ is not dominated by } \mathcal{M} \\
& \text{and } M \text{ is consistent, total, duplicate free}
\end{aligned}
$$

Compared with OCDCL, $\mathcal{T}_N(\emptyset)$ is never empty, because it contains at least the clause set N.

Theorem 4.5 ($\mathsf{CDCL}_{\mathsf{CM}}$ Correctness, cdclcm_correctness ☑). *If the clauses in N do not contain duplicated literals, then a $\mathsf{CDCL}_{\mathsf{CM}}$ starting from $(\epsilon, N, \emptyset, \top, \emptyset)$ and* *in a state $(\epsilon, N, U, \bot, \mathcal{M})$, and for every variable P in N, there is a model M of N, $M \in \mathcal{M}$, where $M \models P$, or there is no model satisfying both P and N.*

The proof involves a lemma similar to Lemma 4.2: Every model is dominated by a model in $\mathcal{M}$ or is still a model of $N \cup \mathcal{T}_N(\mathcal{M})$.

4.6. Extending CDCL

I briefly describe how to extend the formalization. The simplest way consists in restricting the rules, like required for decision (Section 4.2). It is impossible to me explain how to add rules to CDCL for every possible set of rules, but I will hive some ideas how to do so (Section sec:more-general-rules).

4.6.1. Restricting CDCL or Adding Shortcuts

Restricting rules or adding rules that combine several other rules tends to be reasonably easy. The idea is to show that the behavior is one of the behaviors that were previously possible (if $S \Longrightarrow^{+}_{\mathsf{CDCL\_W'}} T$, then $S \Longrightarrow^{+}_{\mathsf{CDCL\_W}} T$) and the final states are the same (by contraposition, if $S \Longrightarrow_{\mathsf{CDCL\_W}} T$, then $\exists T.\, S \Longrightarrow^{*}_{\mathsf{CDCL\_W'}} T$). For example, a restriction of Decide is done in Section 4.2, and the Jump rule without the conflict minimization, which is a special case of Jump with it (this was used in the naive implementation of IsaSAT of Section 3.3). In this case, the following stronger version holds: if $S \Longrightarrow_{\mathsf{CDCL\_W'}} T$, then $S \Longrightarrow_{\mathsf{CDCL\_W}} T$. The termination proof can be inherited.

Remark that one of the invariants of CDCL is that all atoms that have to appear in the set of initial clauses. It is not possible to add a literal only to the learned clauses. In some cases, to prove that the required invariants remain true for the new rules, it might be necessary to extract the proofs that are currently inlined. The proofs corresponds to the file with name `CDCL_W*.thy` in the repository [41].

4.6.2. More General Rules

There is no way for me to give a precise path on how to add more general rules, but here are some ideas: In some cases, it might be possible to simulate a new rule with restart, adding clauses or doing some transformation, then followed by applying the usual CDCL rules. This could be useful, for example, in an SMT solver, if the theory solver provides a clause that should be used to justify a propagation: With a restart (trail: ϵ), followed by adding the clause and the reuse of the trail, it is possible to get back exactly to the point where the propagation takes place. In this case, the termination proof must be adapted.

4.7. Summary

I have presented here a framework for CDCL with branch and bounds, called $\mathsf{CDCL}_{\mathsf{BnB}}$. I have instantiated it to find optimal models (CDCL) and to find a set of covering models. The formalization fits nicely into the framework I have previously developed and the abstraction I have used in Isabelle to simplify reuse and study variants and extensions.

I have also used the dual rail encoding to reduce the search of optimal models with respect to partial valuations to the search of optimal models with respect to total valuations.

5. The Two-Watched-Literal Scheme

A crucial optimization in modern SAT solvers is the two-watched-literal [90] data structure. It allows for efficient unit propagation and conflict detection—the core CDCL operations. It is much more efficient than my functional implementation (Section 3.3) that iterates over all clauses to find them. I introduce an abstract transition system, called TWL, that captures the essence of a SAT solver with this optimization as a nondeterministic transition system (Section 5.2). Weidenbach's book draft only presents the main invariant, without a precise description of the optimization. I enrich the invariant based on MiniSat's [37] source code and prove that it is maintained by all transitions.

I refine the TWL calculus in several correctness-preserving steps. The stepwise refinement methodology enables me to inherit invariants, correctness, and termination from previous refinement steps. The first refinement step implements the rules of the calculus in a more algorithmic fashion, using the nondeterministic programming language provided by the Isabelle Refinement Framework [66] (Section 5.3). The next step refines the data structure: Multisets are replaced by lists, and clauses justifying propagations are represented by indices into a list of clauses (Section 5.4). A key ingredient for an efficient implementation of watched literals is a data structure called *watch lists*. These index the clauses by their two watched literals—literals that can influence their clauses' truth value in the solver's current state. Watch lists are introduced in a separate refinement step (Section 5.5).

Next, I use the Sepref tool [65] to synthesize imperative code for a functional program, together with a refinement proof. Sepref replaces the abstract functional data structures by concrete imperative implementations, while leaving the algorithmic structure of the program unchanged. Isabelle's code generator can then be used to extract a self-contained SAT solver in imperative Standard ML (Section 5.6). Finally, to obtain reasonably efficient code, I need to implement further optimizations and heuristics (Section 5.7). In particular, the literal selection heuristic is crucial. I use variable move to front [14] with phase saving [101].

To measure the gap between my solver, *IsaSAT-17*, and the state of the art, I compare IsaSAT's performance with four other solvers: the leading solver

Glucose [1]; the well-known MiniSat [37]; the OCaml-based DPT;[1] and the most efficient verified solver I know of, `versat` [97] (Section 5.8). Although my solver is competitive with `versat`, the results are sobering.

5.1. Code Synthesis with the Isabelle Refinement Framework

The Isabelle Refinement Framework approach is at the core of my approach: I start from a transition system that includes the two-watched-literal scheme. From there, I refine it by changing data structure and defining heuristics, before synthesizing imperative code.

5.1.1. Isabelle Refinement Framework

The Isabelle Refinement Framework [66] provides definitions, lemmas, and tools that assist in the verification of functional and imperative programs via stepwise refinement [125]. The framework defines a programming language that is built on top of a nondeterminism monad. A program is a function that returns an object of type $'a\ nres$:

$$\textbf{datatype}\ 'a\ nres\ =\ \mathsf{FAIL}\ |\ \mathsf{RES}\ ('a\ set)$$

The set X in $\mathsf{RES}\ X$ specifies the possible values that can be returned. The return statement is defined as a constant $\mathsf{RETURN}\ x = \mathsf{RES}\ \{x\}$ and specifies a single value, whereas $\mathsf{RES}\ \{n \mid n > 0\}$ indicates that an unspecified positive number is returned. The simplest program is a semantic specification of the possible outputs, encapsulated in a RES constructor. The following example is a nonexecutable specification of the function that subtracts 1 from every element of the list xs (with $0 - 1$ defined as 0 on natural numbers):

$$\begin{array}{l}\textbf{definition}\ \mathsf{sub1\_spec} :: nat\ list \Rightarrow nat\ list\ nres\ \textbf{where}\\ \quad \mathsf{sub1\_spec}\ xs = \mathsf{RETURN}\ (\mathsf{map}\ (\lambda x.\ x - 1)\ xs)\end{array}$$

Program refinement uses the same source and target language. The refinement relation $\leq$ is defined by $\mathsf{RES}\ X \leq \mathsf{RES}\ Y \longleftrightarrow X \subseteq Y$ and $r \leq \mathsf{FAIL}$ for all r. For example, the concrete program $\mathsf{RETURN}\ 2$ refines ($\leq$) the abstract program $\mathsf{RES}\ \{n \mid n > 0\}$, meaning that all concrete behaviors are possible in the abstract version. The bottom element $\mathsf{RES}\ \{\}$ is an unrefinable program;

[1] http://dpt.sourceforge.net/

the top element FAIL represents a run-time failure (e.g., a failed assertion) or divergence.

Refinement can be used to change the program's data structures and algorithms, towards a more deterministic and usually more efficient program for which executable code can be generated. I can refine the previous specification to a program that uses a 'while' loop:

$$
\begin{array}{l}
\textbf{definition}\ \mathsf{sub1\_imp} :: \mathit{nat\ list} \Rightarrow \mathit{nat\ list\ nres}\ \ \textbf{where} \\
\quad \mathsf{sub1\_imp}\ \mathit{xs} = \mathrm{do}\ \{ \\
\qquad (i, \mathit{zs}) \leftarrow \mathsf{WHILE_T}\ (\lambda(i, \mathit{ys}).\ i < |\mathit{ys}|) \\
\qquad\quad (\lambda(i, \mathit{ys}).\ \mathrm{do}\ \{ \\
\qquad\qquad \mathsf{ASSERT}\ (i < |\mathit{ys}|); \\
\qquad\qquad \mathrm{let}\ \mathit{zs} = \mathsf{list\_update}\ \mathit{ys}\ i\ ((\mathit{ys} \,!\, i) - 1); \\
\qquad\qquad \mathsf{RETURN}\ (i + 1, \mathit{zs}) \\
\qquad\quad \}) \\
\qquad\quad (0, \mathit{xs}); \\
\qquad \mathsf{RETURN}\ \mathit{zs} \\
\quad \}
\end{array}
$$

The program relies on the following constructs. The 'do' is a Haskell-inspired syntax for expressing monadic computations (here, on the nondeterminism monad). The $\mathsf{WHILE_T}$ combinator takes a condition, a loop body, and a start value. In my example, the loop's state is a pair of the form (i, ys). The $_\mathsf{T}$ subscript in the combinator's name indicates that the loop must not diverge. Totality is necessary for code generation. The ASSERT statement takes an assertion that must always be true when the statement is executed. Finally, the $\mathit{xs} \,!\, i$ operation returns the $(i + 1)$st element of xs, and $\mathsf{list\_update}\ \mathit{xs}\ i\ y$ replaces the $(i + 1)$st element by y.

To prove the refinement lemma $\mathsf{sub1\_imp}\ \mathit{xs} \leq \mathsf{sub1\_spec}\ \mathit{xs}$, I can use the *refine_vcg* proof method provided by the Refinement Framework. This method heuristically aligns the statements of the two programs and generates proof obligations, which are passed to the user. If the abstract program has the form RES X or RETURN x, as is the case here, *refine_vcg* applies Hoare-logic-style rules to generate the verification conditions. For my example, two of the resulting proof obligations correspond to the termination of the 'while' loop and the correctness of the assertion. I can use the measure $\lambda(i, \mathit{ys}).\ |\mathit{ys}| - i$ to prove termination.

The goals generated by *refine_vcg* are often easy to discharge with standard Isabelle tactics, but they may also point to missing lemmas or invariants. The primary technical challenge during proof development is to handle cases where the verification condition generator fails to properly align the programs and generates nonsensical, and usually unprovable, proof obligations. In some cases, the tool generates error messages, but these are often cryptic. Another hurdle is that refinement proof goals can be very large, and the Isabelle/jEdit graphical interface is painfully slow at displaying them. I suspect that this is mostly due to type annotations and other metainformation available as tooltips.

In a refinement step, I can also change the types. For my small program, if I assume that the natural numbers in the list are all nonzero, I can replace them by integers and use the subtraction operation on integers (for which $0 - 1 = -1 \neq 0$). The program remains syntactically identical except for the type annotation:

$$\textbf{definition}\ \mathsf{sub1\_imp}_{\mathsf{int}} :: \mathit{int\ list} \Rightarrow \mathit{int\ list\ nres}\ \textbf{where}$$
$$\mathsf{sub1\_imp}_{\mathsf{int}}\ \mathit{xs} = \langle \text{same body as } \mathsf{sub1\_imp} \rangle$$

I want to establish the following relation: If all elements in $xs :: \mathit{nat\ list}$ are nonzero and the elements of $ys :: \mathit{int\ list}$ are positionwise numerically equal to those of xs, then any list of integers returned by $\mathsf{sub1\_imp}_{\mathsf{int}}\ ys$ is positionwise numerically equal to some list returned by $\mathsf{sub1\_imp}\ xs$. The framework lets me express preconditions and connections between types using higher-order relations called relators:

$$(\mathsf{sub1\_imp}_{\mathsf{int}}, \mathsf{sub1\_imp})$$
$$\in\ [\lambda xs.\ \forall i \in xs.\ i \neq 0]\ \langle \mathsf{int\_of\_nat\_rel} \rangle \mathsf{list\_rel} \to \langle \langle \mathsf{int\_of\_nat\_rel} \rangle \mathsf{list\_rel} \rangle \mathsf{nres\_rel}$$

The relation $\mathsf{int\_of\_nat\_rel} :: (\mathit{int} \times \mathit{nat})\ \mathit{set}$ relates natural numbers with their integer counterparts (e.g., $(5 :: \mathit{int}, 5 :: \mathit{nat}) \in \mathsf{int\_of\_nat\_rel}$). The syntax of relators mimics that of types; for example, if R is the relation for $'a$, then $\langle R \rangle \mathsf{list\_rel}$ is the relation for $'a\ \mathit{list}$, and $\langle R \rangle \mathsf{nres\_rel}$ is the relation for $'a\ \mathit{nres}$. The ternary relator $[p]\, R \to S$, for functions $'a \Rightarrow 'b$, lifts the relations R and S for $'a$ and $'b$ under precondition p.

The theorem can also be written:

$$(\forall i \in xs.\ i \neq 0) \wedge (xs', xs) \in \langle \mathsf{int\_of\_nat\_rel} \rangle \mathsf{list\_rel} \Longrightarrow$$
$$\mathsf{sub1\_imp}_{\mathsf{int}}\ xs' \leq \Downarrow_{\langle \mathsf{int\_of\_nat\_rel} \rangle \mathsf{list\_rel}} (\mathsf{sub1\_imp}\ xs)$$

The Refinement Framework uses both versions: The first version is used when synthesizing code, while the later version is used for every other purpose. Therefore, I have some theorems that transform the first version to the later.

5.1.2. Sepref and Refinement to Imperative HOL

The *Imperative HOL* library [28] defines a heap monad that can express imperative programs with side effects. On top of Imperative HOL, a separation logic, with assertion type *assn*, can be used to express relations $'a \Rightarrow 'b \Rightarrow \mathit{assn}$ between plain values, of type $'a$, and data structures on the heap, of type $'b$. For example, $\mathsf{array\_assn}\ R :: 'a\ \mathit{list} \Rightarrow 'b\ \mathit{array} \Rightarrow \mathit{assn}$ relates lists of $'a$ elements with mutable arrays of $'b$ elements, where $R :: 'a \Rightarrow 'b \Rightarrow \mathit{assn}$ is used to relate the elements. The relation between the ! operator on lists and its heap-based counterpart Array.nth can be expressed as follows:

$$\begin{aligned}&\big((\lambda(xs,\ i).\ \mathsf{Array.nth}\ xs\ i),\ (\lambda(xs,\ i).\ \mathsf{RETURN}\ (xs\,!\,i))\big)\\ \in\ &[\lambda(xs,\ i).\ i < |xs|]\ (\mathsf{array\_assn}\ R)^{\mathrm{k}} \times \mathsf{nat\_assn}^{\mathrm{k}} \rightarrow R\end{aligned}$$

The arguments' relations are annotated with $^{\mathrm{k}}$ ("keep") or $^{\mathrm{d}}$ ("destroy") superscripts that indicate whether the previous value can still be accessed after the operation has been performed. Reading an array leaves it unchanged, whereas updating it destroys the old array.

The *Sepref* tool automates the transition from the nondeterminism monad to the heap monad. It keeps track of the values that are destroyed and ensures that they are not used later in the program. Given a suitable source program, it can automatically generate the target program and prove the corresponding refinement lemma automatically. The main difficulty is that some low-level operations have side conditions, which I must explicitly discharge by adding assertions at the right points in the source program to guide Sepref.

The following command generates a heap program called sub1_imp_code from the source program $\mathsf{sub1\_imp}_{\mathsf{int}}$:

sepref_definition sub1_imp_code **is**
  $\mathsf{sub1\_imp}_{\mathsf{int}} :: [\lambda\_.\mathsf{True}]\ (\mathsf{array\_assn}\ \mathsf{nat\_assn})^{\mathrm{d}} \rightarrow$
    $\mathsf{array\_assn}\ \mathsf{nat\_assn}$
  by *sepref*

The generated array-based program is

$$\begin{aligned}&\mathsf{sub1\_imp\_code}\ xs\ =\ \mathrm{do}\ \{\\ &\quad (i, zs) \leftarrow \mathsf{heap\_WHILE}_{\mathsf{T}}\ (\lambda(i,\ ys).\ \mathrm{do}\ \{\ zs \leftarrow \mathsf{Array.len}\ ys;\\ &\qquad\quad \mathsf{return}\ (i < zs)\ \})\\ &\qquad (\lambda(i,\ ys).\ \mathrm{do}\ \{\\ &\qquad\quad z \leftarrow \mathsf{Array.nth}\ ys\ i - 1;\\ &\qquad\quad zs \leftarrow \mathsf{Array.upd}\ ys\ i\ z;\\ &\qquad\quad \mathsf{return}\ (i + 1,\ zs)\ \ \})\end{aligned}$$

$(0, xs)$;
return zs
}

The Refinement Framework provides a way to compose all the specifications. The end-to-end refinement theorem, obtained by composing the refinement lemmas, is

$$\begin{aligned}&(\mathsf{sub1\_imp\_code},\ \mathsf{sub1\_spec})\\ \in\ &[\lambda xs.\ \forall i \in xs.\ i \neq 0]\ (\mathsf{array\_assn}\ \mathsf{int\_of\_nat\_assn})^{\mathrm{d}} \to\\ &\quad \mathsf{array\_assn}\ \mathsf{int\_of\_nat\_assn}\end{aligned}$$

5.1.3. Code Generation of Imperative Programs

If I want to execute the program efficiently, I can translate it to Standard ML using Isabelle's code generator [49]. The following imperative code, including its dependencies, is generated (in slightly altered form):

```
fun sub1_imp_code xs = (fn () =>
  let
    val (i, zs) =
      heap_WHILET
        (fn (i, ys) => fn () => i < len heap_int ys)
        (fn (i, ys) => fn () =>
           let val z = nth heap_int ys i - 1 in
             (i + 1, upd heap_int i z ys) end)
        (0, xs) ();
  in zs end);
```

The ML idiom `(fn () => ...) ()` is inserted to delay the evaluation of the body, so that the side effects occur in the intended order.

Code generation in Isabelle is built around a mapping from Imperative HOL operations to concrete code in the target language. This mapping is composed of *code equations* translating code and the correctness of the mapping cannot be verified in Isabelle. For example, accessing the n-th element of an Imperative HOL array is mapped to accessing the n-th elements of the target language (e.g., `nth` in the code above which maps around `Array.sub`). These equations are the *trusted code base*. They cannot be proved correct without a full semantics of the target language and a compiler in Isabelle from Isabelle's language to this language.

5.1.4. Sepref and Locales

Code synthesis in locale is more complicated than normal code synthesis, because otherwise, the assumptions of the locale would also be assumptions on the definition, making code generation from Isabelle impossible, because the code generator cannot discharge that the assumptions of the definition holds. The standard idiom is the following:

locale X
begin
  sepref_register sub1_imp
  sepref_thm sub1_imp_refined **is**
    PR_CONST $\mathsf{sub1\_imp}_{\mathsf{int}}$
    :: $[\lambda\_.\mathsf{True}]$ $(\mathsf{array\_assn}\ \mathsf{nat\_assn})^{\mathsf{d}} \rightarrow \mathsf{array\_assn}\ \mathsf{nat\_assn}$
    by *sepref*
  concrete_definition (**in** $-$) sub1_imp_code
    uses X.sub1_imp_refined.refine_raw
    is $(?f, \_) \in \_$
  prepare_code_thms (**in** $-$) sub1_imp_code_def
end

The command **sepref_thm** is similar to **sepref_definition** but does not introduce a new definition: It simply synthesizes the code and prove the refinement relation (here under the name sub1_imp_refined). **concrete_definition** is the command in charge of creating the definition with the key difference that this is done *outside the locale*: The Isabelle command (**in** $-$) locally exits the locale. After that, **prepare_code_thms** takes care of some additional code setup to be able to generate code. The later is not necessary when the synthesis does nat happen inside a locale.

There are two additional complications: First, the constant PR_CONST (defined to be simply the identity) is used for technical reasons to protect functions to synthesize, if the functions depends on parameters of the locale.[2] More precisely, assume that the locale X depends on some parameter N. This is emulated by Isabelle by having the constant X.sub1_imp taking N as first parameter. Sepref normalizes a call to sub1_imp (inside the locale) to the call (outside of the locale) PR_CONST $(\mathsf{X.sub1\_imp}\, N)$. PR_CONST makes sure

[2]If there are not parameters, PR_CONST is harmful and makes it impossible to synthesize code calling this function. Whether it is necessary is sometimes so complicated to find out, that I do not always know and wait to see if the code generation fails.

that $(\mathsf{X.sub1\_imp}\,N)$ are considered together and N is not considered as an argument. Second, exiting the locale can become complicated and requires additional (sometimes very fragile) proofs.

5.2. Watched Literals

The two-watched-literal (2WL or TWL) scheme [90] is a data structure that makes it possible to efficiently identify candidate clauses for unit propagation and conflict. In each nonunit clause (i.e., a clause with at least two literals), I distinguish two *watched* literals; the remaining literals are *unwatched*. Initially, any of a nonunit clause's literals can be chosen to be watched. The solver maintains the following *2WL invariant* for each clause:

> Unless a conflict has been found, a watched literal may be false only if the other watched literal is true and all the unwatched literals are false.

This is the invariant given by Weidenbach. It is inspired by MiniSat's code. A consequence of this invariant is that setting an unwatched literal will never yield a candidate for propagation or conflict. This can dramatically reduce the number of candidate clauses to consider.

For each literal L, the clauses that contain a watched L are chained together in a list, called a *watch list*. When a literal L becomes true, the solver needs to iterate only through the watch list for $-L$ to find candidates for propagation or conflict. For each candidate clause, there are four possibilities:

1. If the other watched literal is true, do nothing.
2. If one of the unwatched literals L' is not false, restore the invariant by *updating* the clause so that it watches L' instead of $-L$.
3. Otherwise, consider the other watched literal L' in the clause:
   - 3.1. If it is not set, propagate L'.
   - 3.2. Otherwise, L' is false, and I have found a conflict.

Propagation is performed eagerly. When a conflict is detected, the solver stops updating the data structure and processes the conflict.

To illustrate how the solver maintains the 2WL invariant, I consider the small problem shown in Figure 5.1. The clauses are numbered from 1 to 4. Gray cells identify the watched literals. Thus, clause 1 is $\neg B \vee C \vee A$, where $\neg B$ and C are watched. The following scenario is possible:

1.	$\neg B$	C	A
2.	$\neg C$	$\neg B$	$\neg A$
3.	$\neg A$	$\neg B$	C
4.	$\neg A$	B	

(a)

$\neg B$	C	A
$\neg C$	$\neg B$	$\neg A$
C	$\neg B$	$\neg A$
$\neg A$	B	

(b)

Figure 5.1.: Evolution of the 2WL data structure on a simple example

1. I start with an empty trail and the clauses shown in Figure 5.1a. I decide to make A true. The trail becomes $A^\dagger$. I need to consider every clause where $\neg A$ is watched, i.e., clauses 3 and 4, in any order.

2. I first consider clause 4 for $\neg A$. I propagate B from it. The trail becomes $BA^\dagger$. I still need to consider clause 3 for $\neg A$ and the clauses for $\neg B$.

3. I consider clause 3 for $\neg A$. Since C is unwatched and not false, I swap C and $\neg A$, resulting in the clauses shown in Figure 5.1b. I must still consider clauses 1, 2, and 3 for $\neg B$.

4. I consider clause 3 for $\neg B$: I propagate C. The trail becomes $CBA^\dagger$. I still need to update the clauses 1 and 2 for $\neg B$ and the clauses for $\neg C$.

5. I consider clause 2. All its literals are false—a conflict. Thanks to the invariant's precondition ("unless a conflict has been found"), I do not need to update clause 1 or the clauses for $\neg C$.

Compatibility with the Jump rule is important for efficiency: When removing literals from the trail, the invariant is preserved without requiring any update.

To capture the 2WL data structure formally, I need a notion of state that takes into account pending updates. These can concern a specific clause or all the clauses associated with a literal. As in the example above, I first process the clause-specific updates; then I move to the next literal and start updating its associated clauses.

States have the form $(M, N, U, D, NP, UP, WS, Q)$, of type $'v\ state_{\mathsf{TWL}}$. The pending updates are stored in the last two components: the *work stack* WS is a multiset $\{(L, C_1), \ldots, (L, C_n)\}$, where L is a false literal and the clauses C_i watch L and may require an update. The other literals to update are stored in the *queue* Q. For example, at the end of step 4 above, WS is $\{(\neg B, \neg B \vee C \vee A), (\neg B, \neg C \vee \neg B \vee \neg A)\}$ and Q is $\{\neg C\}$.

Moreover, I store the unit clauses separately from the nonunit clauses. The unit clauses are put in the NP and UP components as singleton multisets. The nonunit clauses are put in N and U. Each nonunit clause is represented by a value $\mathsf{Clause}_{\mathsf{TWL}}\ W\ UW$, where W is the multiset of watched literals, of cardinality 2, and UW the multiset of unwatched literals.

The $\mathsf{state}_{\mathsf{W}}\mathsf{\_of}$ function converts a TWL state to a CDCL_W state:

$$\begin{array}{l}\textbf{definition}\ \mathsf{state}_{\mathsf{W}}\mathsf{\_of} :: {}'v\ \mathit{state}_{\mathsf{TWL}} \Rightarrow {}'v\ \mathit{state}_{\mathsf{W}}\ \textbf{where}\\ \quad \mathsf{state}_{\mathsf{W}}\mathsf{\_of}\ (M, N, U, D, NP, UP, WS, Q) =\\ \quad (M,\ \mathsf{image\ clause}_{\mathsf{W}}\mathsf{\_of}\ N \uplus NP,\\ \quad\ \ \mathsf{image\ clause}_{\mathsf{W}}\mathsf{\_of}\ U \uplus UP,\ D)\end{array}$$

where $\mathsf{clause}_{\mathsf{W}}\mathsf{\_of}\ (\mathsf{Clause}_{\mathsf{TWL}} W\ UW) = W \uplus UW$ and $\mathsf{image}\ f\ N$ applies the function f to each element of multiset N.

The first two TWL rules have direct counterparts in CDCL_W:

Propagate $(M, N, U, \top, NP, UP, \{(L, C)\} \uplus WS,\ Q) \Longrightarrow_{\mathsf{TWL}}$
$(L'^{C} M, N, U, \top, NP, UP, WS, \{-L'\} \uplus Q)$
if $\mathsf{watched}\ C = \{L, L'\}$, L' is not set in M, and
$\forall K \in \mathsf{unwatched}\ C.\ -K \in M$

Conflict $(M, N, U, \top, NP, UP, \{(L, C)\} \uplus WS, Q) \Longrightarrow_{\mathsf{TWL}}$
$(M, N, U, C, NP, UP, \emptyset, \emptyset)$
if $\mathsf{watched}\ C = \{L, L'\}$, $-L' \in M$, and
$\forall K \in \mathsf{unwatched}\ C.\ -K \in M$

For both rules, C cannot be a unit clause. The condition stating that $\forall K \in \mathsf{unwatched}\ C.\ -K \in M$ is necessary because the 2WL invariant trivially holds for C as long as an update on C is pending.

The next rules manipulate the state's 2WL-specific components, without affecting its semantics as seen through the function $\mathsf{state}_{\mathsf{W}}\mathsf{\_of}$:

Update $(M, N, U, \top, NP, UP, \{(L, C)\} \uplus WS, Q) \Longrightarrow_{\mathsf{TWL}}$
$(M, N', U', \top, NP, UP, WS, Q)$
if $K \in \mathsf{unwatched}\ C$, $-K \notin M$, and N' and U' are obtained from N and U by replacing the clause $C = \mathsf{Clause}_{\mathsf{TWL}}\ W\ UW$ with $\mathsf{Clause}_{\mathsf{TWL}}\ (W - \{L\} \uplus \{K\})(UW - \{K\} \uplus \{L\})$

Ignore $(M, N, U, \top, NP, UP, \{(L, C)\} \uplus WS, Q) \Longrightarrow_{\mathsf{TWL}}$
$(M, N, U, \top, NP, UP, WS, Q)$
if $\mathsf{watched}\ C = \{L, L'\}$ and $L' \in M$

Next_Literal $(M, N, U, \top, NP, UP, \emptyset, \{L\} \uplus Q) \Longrightarrow_{\mathsf{TWL}}$
$(M, N, U, \top, NP, UP,$
$\{(L, C) \mid L \in \mathsf{watched}\ C \wedge C \in N \uplus U\}, Q)$

As in W+stgy, I postpone decisions. This is achieved by requiring that WS and Q are empty in the Decide rule. Skip and Resolve are as before, except that they also preserve the 2WL-specific components of the state. Due to the distinction between unit and nonunit clauses, I need two rules for nonchronological backjumping:

Decide $(M, N, U, \top, NP, UP, \emptyset, \emptyset) \Longrightarrow_{\mathsf{TWL}}$
$(L^{\dagger}M, N, U, \top, NP, UP, \emptyset, \{-L\})$
if L is not defined in M and appears in N

Jump_Nonunit $(M' \cdot K^{\dagger}M, N, U, D \vee L, NP, UP, \emptyset, \emptyset) \Longrightarrow_{\mathsf{TWL}}$
$(L^{D \vee L}M, N, U \uplus \{\mathsf{Clause}_{\mathsf{TWL}}\ \{L, L'\}\ (D' - \{L'\})\}, \top,$
$NP, UP, \emptyset, \{L\})$
if the conditions on Jump are satisfied by D, D', and L, $L' \in D$, and L' has the highest level among D''s literals

Jump_Unit $(M' \cdot K^{\dagger}M, N, U, D \vee L, NP, UP, \emptyset, \emptyset) \Longrightarrow_{\mathsf{TWL}}$
$(L^{L}M, N, U, \top, NP, UP \uplus \{L\}, \emptyset, \{L\})$
if the conditions on Jump are satisfied by D, $D' = \emptyset$, and L

In Jump_Nonunit, I need to choose a literal L' of D' with the highest level among D''s literals, or the next-highest level in $D' \vee L$ (since L has a higher level than L'). Jump_Nonunit is documented in MiniSat's code ("find the first literal assigned at the next-highest level"). Remarkably, this important property is mentioned neither in Weidenbach's book draft nor in the description of MiniSat [37].

Theorem 5.1 (Invariant [41, *cdcl_twl_stgy_twl_struct_invs*]). *If the state S satisfies the 2WL invariant and $S \Longrightarrow_{TWL} T$, then T satisfies the 2WL invariant.*

Theorem 5.2 (Refinement [41, *full_cdcl_twl_stgy_cdcl$_W$_stgy*]). *Let S be a state that satisfies the 2WL invariant. If $S \Longrightarrow^{!}_{TWL} T$, then*

$$\mathsf{state}_{\mathsf{W}}\mathsf{\_of}\ S \Longrightarrow^{!}_{CDCL\_W} \mathsf{state}_{\mathsf{W}}\mathsf{\_of}\ T.$$

TWL refines W+stgy's end-to-end behavior and produces final states that are also final states for CDCL_W. I can apply Theorem 3.9 to establish partial correctness. Termination of TWL is a direct consequence of the termination of CDCL_W.

5.3. Refining the Calculus to an Algorithm

I want to obtain an executable SAT solver from TWL. I do this by refining the calculus in multiple consecutive steps until I reach an implementation.

The Isabelle Refinement Framework [65–67] provides a tool chain for program development via stepwise refinement. It is based on the *nondeterminism monad* over the datatype $'a\ nres = \mathsf{FAIL} \mid \mathsf{RES}\ ('a\ set)$. If the program has an execution that diverges or raises an error, its result is FAIL; otherwise, the result is RES X, where X is the set of possible return values. The function RETURN x, which abbreviates RES $\{x\}$, returns the value x; bind $m\ f$ nondeterministically chooses a return value from m and applies f to it. Based on these constructs and Isabelle's standard 'if–then–else' and 'case' expressions, the Refinement Framework defines higher-level constructs such as 'while' and 'for each' loops. The Haskell-style 'do' monadic notation is also supported: do $\{a \leftarrow m;\ f\,a\}$ is syntactic sugar for bind $m\ f$.

The first step in the refinement chain is to implement the calculus as a program in the nondeterminism monad. The program operates on states of type $'v\ state_{\mathsf{TWL}}$, as in the TWL calculus, but it reduces some of the calculus's nondeterminism. The program consists of a few functions that implement mutually disjoint sets of rules. I focus on the function that applies Propagate, Conflict, Update, or Ignore, assuming that its first argument, the pair $LC = (L, C)$, has already been removed from the WS component of S:

definition $\mathsf{PCUI}_{\mathsf{algo}}$ ::
  $'v\ lit \times 'v\ clause \Rightarrow 'v\ state_{\mathsf{TWL}} \Rightarrow 'v\ state_{\mathsf{TWL}}$
where
  $\mathsf{PCUI}_{\mathsf{algo}}\ LC\ S = \mathrm{do}\ \{$
    let $(M, N, U, D, NP, UP, WS, Q) = S$;
    let $(L, C) = LC$;
    $L' \leftarrow \mathsf{RES}\ \{L' \mid L' \in \mathsf{watched}\ C - \{L\}\}$;
    if $L' \in M$ then (∗ Ignore ∗)
      RETURN S
    else
      if $\forall L \in \mathsf{unwatched}\ C.\ -L \in M$ then
        if $-L' \in M$ then (∗ Conflict ∗)
          RETURN $(M, N, U, C, NP, UP, \emptyset, \emptyset)$
        else (∗ Propagate ∗)
          RETURN $(L'^{C} M, N, U, D, NP, UP, WS,$
            $\{-L'\} \uplus Q)$
      else do { (∗ Update ∗)

$$
\begin{array}{l}
\qquad\qquad K \leftarrow \mathsf{RES}\,\{K \mid K \in \mathsf{unwatched}\,C \wedge -K \notin M\};\\
\qquad\qquad (N', U') \leftarrow \mathsf{RES}\,\{(N', U') \mid\\
\qquad\qquad\qquad \mathsf{update\_clss}\,(N, U)\,C\,L\,K\,(N', U')\};\\
\qquad\qquad \mathsf{RETURN}\,(M, N', U', D, NP, UP, WS, Q)\\
\qquad \}\\
\}
\end{array}
$$

The predicate $\mathsf{update\_clss}\,(N, U)\,C\,L\,K\,(N', U')$ updates the clause C by exchanging the watched literal L and the unwatched literal K in C. The clause is updated in either N or U, yielding N' and U'. Since propagations are performed eagerly, WS never refers to unit clauses.

The $\mathsf{PCUI}_{\mathsf{algo}}$ algorithm still contains abstract, nondeterministic parts. For example, in the Update part, I leave the choice of the new watched literal K underspecified.

To allow me to specify the connection between two programs, the Refinement Framework defines a partial order $\leq$ on $'a\ nres$, with FAIL as the top element: $\mathsf{RES}\,X \leq \mathsf{RES}\,Y$ if and only if $X \subseteq Y$, and $r \leq \mathsf{FAIL}$ for all r. The bottom element $\mathsf{RES}\,\{\}$ is an unrefinable program. I also use this partial order to state program correctness: The statement $P\,x \Longrightarrow f\,x \leq \mathsf{RES}\,\{y \mid Q\,y\}$ expresses the total correctness of program f with precondition P and postcondition Q. For $\mathsf{PCUI}_{\mathsf{algo}}$, I have the following refinement theorem:

Lemma 5.3 (Refinement [41, *unit_propagation_inner_loop_body_add*]). *If the 2WL invariant holds for all clauses occurring in the N and U components of S, then*

$$\mathsf{PCUI}_{\mathsf{algo}}\,(L, C)\,S \leq \mathsf{RES}\ \{T \mid \mathsf{add\_to\_WS}\,(L, C)\,S \Longrightarrow_{\mathsf{PCUI}} T\}$$

The PCUI subscript on the transition arrow refers to the fragment of TWL consisting of the four rules Propagate, Conflict, Update, and Ignore, whereas $\mathsf{add\_to\_WS}\,(L, C)\,S$ returns the state obtained by adding (L, C) to S's WS component. For the entire SAT solver, I have the following theorem:

Theorem 5.4 (Refinement [41, *cdcl_twl_stgy_prog_spec*]). *If the 2WL invariant holds for all clauses occurring in the N and U components of S, then*

$$\mathsf{TWL}_{\mathsf{algo}}\,S \leq \mathsf{RES}\ \{T \mid S \Longrightarrow^{!}_{TWL} T\}$$

The state returned by the program is a final state for TWL. From Theorem 5.2, I deduce that it is also a final state for W+stgy. Hence, the program $\mathsf{TWL}_{\mathsf{algo}}$ is a SAT solver by Theorem 3.9.

5.4. Representing Clauses as Lists

The nondeterministic program $\mathsf{TWL}_{\mathsf{algo}}$ presented in Section 5.3 relies on the same state type as the TWL calculus. This changes with the next refinement step: I now store the initial and learned clauses together in a list, and I use indices to refer to the clauses. States are now tuples $(M, NU, u, D, NP, UP, WS', Q)$:

- NU is the list of all nonunit clauses. It simultaneously refines N and U. The initial clauses occupy indices 1 to $u - 1$, and the learned clauses start at index u. The list's first element is left unused to keep index 0 as a null clause reference.

- M is the trail, where the annotations are replaced by numeric indices. For nonunit clauses, L^i is used instead of L^C if $NU \,!\, i = C$, where the ! operator denotes 0-based list access. When annotating literals with unit clauses (which are not present in NU), I use the special index 0—i.e., I put L^0 on the trail to mean L^L.

- In WS', I implement a pair (L, C) by the index of clause C. The literal L, which is the same for all pairs in WS, is stored locally in the refined unit propagation algorithm.

Abusing notation, I will use the letter C to refer to clause indices and will not distinguish between a clause and its index.

In addition to the modifications to the state, I also transform the representation of clauses, from a pair of multisets holding the watched and unwatched literals to a list of literals such that its first two elements are watched. Given a nonunit clause (index) C, its watched literals are available as $(NU \,!\, C) \,!\, 0$ and $(NU \,!\, C) \,!\, 1$. Furthermore, I set the stage for future refinements by replacing the test $L \in M$ by a call to a function, polarity, that returns Some True if $L \in M$, Some False if $-L \in M$, and None otherwise.

The refined version of the $\mathsf{PCUI}_{\mathsf{algo}}$ algorithm follows

definition $\mathsf{PCUI}_{\mathsf{list}}$ ::
  $'v\ lit \Rightarrow {}'v\ clause\_idx \Rightarrow {}'v\ state_{\mathsf{list}} \Rightarrow {}'v\ state_{\mathsf{list}}$
where
  $\mathsf{PCUI}_{\mathsf{list}}\ L\ C\ S = \mathrm{do}\ \{$
    let $(M, NU, u, D, NP, UP, WS, Q) = S;$
    let $i = \mathrm{if}\ (NU \,!\, C) \,!\, 0 = L\ \mathrm{then}\ 0\ \mathrm{else}\ 1;$
    let $L' = (NU \,!\, C) \,!\, (1 - i);$

```
let pol′ = polarity M L′;
if pol′ = Some True then                                  (* Ignore *)
  RETURN (M, NU, u, D, NP, UP, WS, Q)
else
  case find_unwatched M (NU ! C) of
    None ⇒
      if pol′ = Some False then                           (* Conflict *)
        RETURN (M, NU, u, NU ! C, NP, UP, ∅, ∅)
      else                                                (* Propagate *)
        RETURN (L′^C M, NU, u, D, NP, UP, WS,
            {−L′} ⊎ Q)
  | Some j ⇒ do {                                         (* Update *)
      let NU′ = list_update NU C
          (list_swap (NU ! C) i j);
      RETURN (M, NU′, u, D, NP, UP, WS, Q)
    }
}
```

5.5. Storing Clauses Watched by a Literal: Watch Lists

In the Next_Literal rule of the TWL calculus, the set of clauses that watch a given literal is calculated. A refinement step eliminates this gratuitous inefficiency: Instead of iterating over all clauses, I maintain a map from literals to the clauses that contain them as watched literals. States now have the form $(M, NU, u, D, NP, UP, Q, W)$, where $W :: {'v\ lit} \Rightarrow clause\_idx\ list$ maps each literal to its watch list.

The abstract state stores all the clauses that watch the current literal L and still require processing in its WS component. In the concrete algorithm, I use a local variable w to traverse the watch list. After processing a clause, there are two cases. If the clause still watches L (rules Propagate, Conflict, and Ignore), I increment w to move to the next clause. Otherwise, the clause no longer watches L (rule Update). I exchange the element at index w with the watch list's last element and shorten the list by one (function delete_idx_and_swap). Since the traversal order is irrelevant, this is an efficient way to delete an element in constant time based on arrays. This technique is implemented in many solvers.

The refined PCUI algorithm is presented below, where the syntax $f(x := y)$ denotes the function that maps x to y and otherwise coincides with f:

definition $\mathsf{PCUI}_{\mathsf{wlist}}$ ::

$'v\ lit \Rightarrow nat \Rightarrow {}'v\ state_{\mathsf{wlist}} \Rightarrow nat \times {}'v\ state_{\mathsf{wlist}}$
where

$\mathsf{PCUI}_{\mathsf{wlist}}\ L\ w\ S = \mathrm{do}\ \{$
  let $(M, NU, u, D, NP, UP, Q, W) = S$;
  let $C = W\ L\ !\ w$;
  let $i =$ if $C\ !\ 0 = L$ then 0 else 1;
  let $L' = (NU\ !\ C)\ !\ (1 - i)$;
  let $pol' = \mathsf{polarity}\ M\ L'$;
  if $pol' = \mathsf{Some\ True}$ then (∗ Ignore ∗)
    RETURN $(w + 1, (M, NU, u, D, NP, UP, Q, W))$
  else
    case $\mathsf{find\_unwatched}\ M\ (NU\ !\ C)$ of
      None ⇒
      if $pol' = \mathsf{Some\ False}$ then (∗ Conflict ∗)
        RETURN $(w + 1, (M, NU, u, NU\ !\ C, NP, UP,$
          $\emptyset, W))$
      else (∗ Propagate ∗)
        RETURN $(w + 1, (L'^{C} M, NU, u, D, NP, UP,$
          $\{-L'\} \uplus Q, W))$
    | Some j ⇒ do { (∗ Update ∗)
        let $K = (NU\ !\ C)\ !\ j$;
        let $NU' = \mathsf{list\_update}\ NU\ C$
          $(\mathsf{list\_swap}\ (NU\ !\ C)\ i\ j)$;
        let $W' =$
          $W(L := \mathsf{delete\_idx\_and\_swap}\ (W\ L)\ w)$
            $(K := W\ K \cdot C)$;
        RETURN $(w, (M, NU', u, D, NP, UP, Q, W'))$
      }
}

When performing a chain of refinements, I often want to reuse information from earlier refinement steps. Assume that I have previously shown the refinement relation

$$g\,y \le \Downarrow_{\{(t,s) \in R \mid I_1\,t \wedge I_2\,t\,s\}} f\,x, \tag{5.1}$$

where R relates concrete and abstract states and I_1 and I_2 are invariants. Now suppose I want to refine g by the function h with relation S and invariant J. The invariant J typically consists of a genuinely new part J_{new} and a part inherited from higher abstraction levels. I first prove the new part:

$$h\,z \le \Downarrow_{\{(u,t) \in S \mid J_{\mathrm{new}}\,u\,t\}} g\,y \tag{5.2}$$

Then I can combine it with equation (5.1), using the invariant I_1 that does not depend on a state s, yielding

$$h\,z \leq \Downarrow_{\{(u,t)\in S \mid J_{\text{new}}\,u\,t\,\wedge\,I_1\,t\}}\,g\,y \tag{5.3}$$

Finally, I can prove the desired refinement relation $h\,z \leq \Downarrow_{\{(u,t)\in S \mid J\,u\,t\}}\,g\,y$, by showing the inclusion

$$\{(u,t) \in R \mid J_{\text{new}}\,u\,t \wedge I_1\,t\} \subseteq \{(u,t) \in R \mid J\,u\,t\} \tag{5.4}$$

Because I frequently needed to combine large invariants to derive refinement lemmas such as (5.3), I developed a specialized tactic in the Eisbach language [88]. It takes as input the relations (5.1) and (5.2). It separates I_1 and I_2, based on their syntactic dependencies, and derives the relation (5.3). Another Eisbach tactic takes (5.3) and the desired refinement goal as arguments and leaves (5.4) as the goal. Eisbach is very useful for such tedious but straightforward manipulations, especially for goals containing large formulas.

5.6. Generating Code

For technical reasons, I need an intermediate refinement step between the introduction of watch lists (Section 5.5) and the change of data structures. This step amounts to adding assertions in the watch list algorithms stating that all literals belong to a fixed, finite domain. Given the set of all literals $\mathcal{L}_{\text{in}}$ that appear in the clause set N, I need to consider only the literals that appear in $\mathcal{L}_{\text{in}}$ or whose negation appear in $\mathcal{L}_{\text{in}}$. I call this set $\mathcal{L}_{\text{all}}$. The intermediate refinement step involves stating and discharging assertions of the form $L \in \mathcal{L}_{\text{all}}$. This layer is called $\mathsf{TWL}_{\mathsf{wlist}}{+}\mathcal{L}_{\text{all}}$. This sets the stage for many subsequent optimizations, by allowing me to allocate arrays that are large enough to represent mappings from atoms or literals. Arrays are used for watch lists (which map literals to clauses), polarity caching (which map atoms to their polarities, corresponding to the polarity function), and other optimizations.

Some of the data structures I need are already available in the Imperative Collections Framework [68], while others must be developed specifically for this project. Since the code in Imperative HOL is deterministic, I must commit to a strategy for applying the calculus rules. The precise heuristics and other optimizations are described in Section 5.7.

The solver state is enriched with information necessary for optimizations and heuristics, and its components are implemented by efficient data structures. For example, literals are refined to 32-bit unsigned integers, representing a positive literal $\mathsf{Pos}\,i$ by $2 \cdot i$ and a negative literal $\mathsf{Neg}\,i$ by $2 \cdot i + 1$. All required operations, such as atom extraction and negation, can be efficiently implemented on this representation. The use of 32-bit numbers restricts my implementation to at most 2^{31} atoms (which seems to be a common restriction for SAT solvers).

The encoding of literals as unsigned integers can be used to represent a map from literals by an array, indexed by the literal representation. In this way, I implement the *W* function that maps literals to its watch lists by an array of arrays. The outer array's size is determined by the actual number of atoms in the problem, while I use a dynamic resizing strategy for the inner arrays that hold the watch lists. Using the same literal encoding, clauses are represented by arrays of 32-bit integers. In contrast, the indices used as annotations in the trail and in the *WS* component are unbounded integers.

Internally, the refinement of the state is done in two steps: The first step handles the addition of the data for optimizations and heuristics, and the second step uses Sepref to refine the functional representations of the state's components to efficient mutable data structures.

To obtain a complete SAT solver, I must provide code to initialize the data structure with the 2WL invariant using the list of the atoms in the problems. Initialization works as follows: I first go through the clauses and extract all the atoms by taking the literal where it occurs first. Then for each clause, either it contains at least two literals, in which case the first two are watched, or it is a unit clause, in which case the literal is propagated (or a conflict is marked) and the clause is added to *NP*. If there is a conflict, there is no need to analyze it—the clauses are unsatisfiable.

Once I have refined TWL into an imperative program and combined it with a function initializing the data structure from a list of clauses, I define the complete imperative SAT solver as a function $\mathsf{IsaSAT}_{\mathsf{code}}$ in Imperative HOL. The abstract specification of the solver is given by

$$\begin{aligned}&\mathsf{model\_if\_satisfiable} =\\ &\quad \mathsf{RES}\;\{M \mid \text{if } \mathsf{satisfiable}\,CS \text{ then } M \neq \mathsf{None} \wedge \mathsf{the}\,M \vDash CS\\ &\qquad\qquad\quad \text{else } M = \mathsf{None}\}\end{aligned}$$

where $\mathsf{the}\,(\mathsf{Some}\;x) = x$. This abstract program returns None if the input clauses are unsatisfiable; otherwise, it returns Some M, where M is a model of the clauses. By combining the refinement theorems for all refinement steps, I obtain end-to-end correctness for the entire solver.

Theorem 5.5 (End-to-End Correctness [41, *IsaSAT_code_full_correctness*])**.** *The im-*

perative SAT solver returns a model if its input is satisfiable:

$$(\mathsf{IsaSAT}_{\mathsf{code}}, \mathsf{model\_if\_satisfiable})$$
$$\in [\mathsf{no\_duplicate\_no\_false}]\ \mathsf{clauses\_assn}^{\mathsf{k}} \rightarrow$$
$$\mathsf{option\_assn}\ (\mathsf{list\_assn}\ \mathsf{lit\_assn})$$

The clauses_assn relation refines a multiset of multisets of literals to a list of lists of 32-bit literals, and option_assn (list_assn lit_assn) refines an optional list of literals to an optional list of 32-bit literals.

Finally, I invoke Isabelle's code generator [49] to extract Standard ML code from the Imperative HOL program. The result is a self-contained program consisting of about 2700 lines of code. It is extended with a simple unverified parser for SAT problems in conjunctive normal form. To give a flavor of the program, I show its main loop below (slightly reformatted for readability):

```
fun IsaSAT_code initial_state () =
  let val (_, final_state) =
    heap_WHILET
      (fn (done, _) => fn () => not done)
      (fn (_, T) =>
         analyze_or_decide_code
           (PCUI_and_Next_Literal T ()) ())
      (false, initial_state) ()
  in final_state end
```

5.7. Optimizations and Heuristics

My imperative SAT solver relies on a few optimizations that deserve to be explained in more detail: an efficient decision heuristic, a representation of conflicts as a lookup table, conflict clause minimization, and the elimination of redundant components from the state.

5.7.1. Variable Move to Front

The variable-move-to-front (VMTF) heuristic [14], based on the move-to-front algorithm [106], selects which atom to decide next. It offers similar performance to the better-known variable-state-independent-decaying-sum (VSIDS) scheme [90]. VMTF's main advantage, from a formalization point of view, is that it does not require floating-point arithmetic.

VMTF works on a list of atoms As, which must contain all atoms from $\mathcal{L}_{\text{in}}$ in some order. Two operations access or modify this list: When a decision is needed, VMTF traverses As to *find* the first unset atom with respect to the trail. When an atom is heuristically determined to be important to the problem, it is moved to the front of As so that it is found next—an operation called *rescoring*.

To speed up these operations, I implement some of the optimizations described by Biere and Fröhlich [14]:

- To efficiently remove atoms from As, I represent it by a doubly linked list. Moreover, I store it in an array ns indexed by the atoms, enabling fast accesses to the associated nodes. Each entry in ns has the form $(st, prev, next)$, where st is the timestamp indicating when the atom was rescored, and $prev$ and $next$ are the linked-list "pointers," or rather indices in As (with None representing a null pointer).

- I extend the data structure with a *next_search* component that stores an atom. If $As = As_0 \cdot \ldots \cdot As_{|As|-1}$, with $next\_search = As_j$, all atoms $As_0, \ldots, As_{j-1}$ are set in the trail. When searching for an undefined atom, I can start at index j.

- Timestamps enable me to efficiently unset a literal (e.g., when jumping). Since atoms are sorted in reverse timestamp order in As, I need to update *next_search* only if the unset atom has a higher timestamp than the current *next_search* atom.

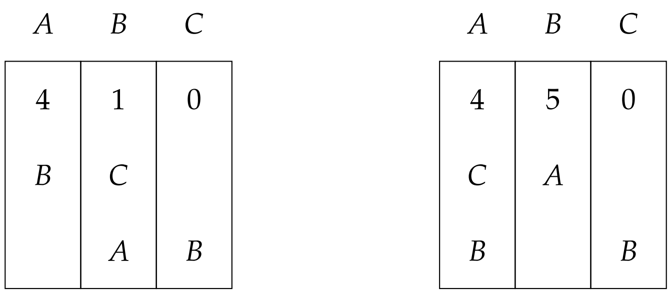
- I batch the rescoring of atoms. Atoms are not removed from As, until the end of the next Jump when rescoring takes place: I sort the atoms to rescore by their timestamps and prepend them to As.

The tuple $vmtf = ((ns, st, fst, next\_search), to\_rescore)$ captures the VMTF data structure. The ns component corresponds the doubly linked list described above; st is the maximum timestamp; fst gives the first atom in As; and *to_rescore* is the batch of atoms that are awaiting rescoring.

In Isabelle, I define the inductive predicate vmtf $As\ st\ ns$ that checks whether ns stores a doubly linked list corresponding to As and the timestamps are bounded by st. It is defined by the following introduction rules:

Empty list vmtf $\epsilon\ st\ ns$, where ϵ denotes the empty list;

Singleton list vmtf $i\ st\ ns$
if $i < |ns|$ and $ns\,!\,i = (st, \text{None}, \text{None})$;



(a) Doubly linked list for the VMTF heuristics for $As = [A, B, C]$, $st = 4$, and $ns = [(4, \mathsf{Some}\ A, \mathsf{None}); (1, \mathsf{Some}\ C, \mathsf{Some}\ A); (0, \ \mathsf{None}, \mathsf{Some}\ A)]$

(b) Doubly linked list for the VMTF heuristics after bumping B for $As = [B, A, C]$ and $st = 5$

Figure 5.2.: Example of the VMTF heuristic before and after bumping.

List of length 2 or more $\mathsf{vmtf}\ (ijAs)\ (st + 1)\ ns$
if $\mathsf{vmtf}\ (jAs)\ st\ ns'$, $i \neq j$, $i < |ns|$, and ns is ns' where $ns\,!\,i = (st + 1, \mathsf{None}, \mathsf{Some}\ j)$ and the *prev* component of $ns'\,!\,j$ has been updated to $\mathsf{Some}\ i$.

An example is shown in Figure 5.2. In the function that finds the next unset literal, I iterate over the doubly linked list stored in *ns*:

$$
\begin{array}{l}
\mathsf{find\_next\_undef}\ ((ns, st, fst, next\_search), \_)\ M = \mathrm{do}\ \{\\
\quad \mathsf{WHILE_T}\ (\lambda next\_search.\ next\_search \neq \mathsf{None}\\
\qquad\quad \wedge\ \mathsf{defined\_atm}\ M\ (\mathsf{the}\ next\_search))\\
\qquad (\lambda next\_search.\\
\qquad\quad \mathsf{RETURN}\ (\mathsf{get\_next}\ (A\,!\,\mathsf{the}\ next\_search)))\\
\qquad next\_search\\
\}
\end{array}
$$

The $\mathsf{defined\_atm}$ predicate tests whether an atom is set in the trail. The $\mathsf{get\_next}\ i$ function returns the *next* component of the node associated with atom i—i.e., the atom following atom i in As (or None if i is the last element in As).

To prove this program correct, I must show the termination of the while loop, which amounts to the well-foundedness of the relation

$$
\begin{array}{l}
\{(\mathsf{get\_next}\ (\mathsf{ns}\,!\,\mathsf{the}\ next\_search), next\_search)\ |\\
\ next\_search \neq \mathsf{None}\}
\end{array}
$$

This, in turn, amounts to showing that the chain of get_next calls contains no loops. I achieve this by showing that the chain is a traversal of the list *As*, which is finite. On the example from Figure 5.2, first *A* would be test, then *B*, and finally *C*.

When implementing a heuristic such as VMTF, I must prove that it does not fail (e.g., because of an out-of-bound array access) and that it returns a correct result. I do not need to prove that my implementation is actually a "VMTF" as defined by Biere and Fröhlich [14]. For example, there are no formal guarantees that the sorting function I use to rescore the batched atoms by their timestamps is correct; it is sufficient to show that sorting introduces no new atoms.

VMTF gives only the next atom to decide (if one exists). I also need to choose the literal's polarity. I use the *phase saving* heuristic [101]. It is a mapping φ from an atom to a polarity, implemented as an array of Booleans. Initially, all atoms are mapped to a negative polarity. Then for each conflict, the mapping is updated: Every atom involved in the conflict will be mapped to the polarity it has in the trail.

5.7.2. Conflict Clause as a Lookup Table

In the TWL calculus and the refinements shown so far, the conflict clause is either $\top$ (None) or an actual clause (Some C). Four operations access or modify the conflict clause:

- The Conflict rule replaces $\top$ by a conflict clause.
- The Resolve rule merges the conflict clause with another clause, removing duplicates.
- The choice between the Resolve and Skip rules depends on whether the trail's head appears in the conflict clause.
- The choice between Resolve and Jump requires an iteration through the clause to evaluate the maximum level of the clause minus one literal.

Initially, I tried representing the conflict as an optional resizable array that is converted to a nonresizable array when the clause is learned (by rule Jump_Nonunit). However, this led to many memory allocations and to inefficient code for resolution (rule Resolve).

Inspired by MiniSat, I moved to an encoding of the conflict clause as a lookup table. I use an array *ps* such that the entry at position *i* indicates

Intermediate code	Refinement relation	Imperative HOL code
	(b', n', ps'_C) refines the clause C as a lookup table; L' and K' refine the literals L and K	
let $n = \mathsf{size}\ (\mathsf{the}\ C)$;		$n' \leftarrow \mathsf{size\_conflict\_code}\ (b', n', ps'_C)$
	The 32-bit unsigned integer n' is equal to the natural number n	
let $D = \mathsf{replicate}\, n\, K$;		$D' \leftarrow \mathsf{Array.new}\ n'\ K'$
	The array D' has the same length and same content as the list D	
let $D = D[1 := L]$;		$D' \leftarrow \mathsf{Array.upd}\ 1\ D'\ L'$
	The array D' refines the updated list D; both contain K at position 1	
let $C' = \mathsf{Some}\ (\mathsf{the}\ C - \{K, L\})$;		$(b', n', ps'_{C'}) \leftarrow \mathsf{remove\_from}\ K'\ L'\ (b', n', ps'_C)$
	$(b', n' - 2, ps'_C)$ refines C'	
$\mathsf{RES}\ \{(E, \mathsf{None}) \mid E\,!\,0 = K2 \land E\,!\,1 = L \land \lvert E \rvert \geq 2 \land C' =\}\ \mathsf{mset}\ (\mathsf{drop}\ 2\ E)\}$		$(E', (b', n', ps'_\top)) \leftarrow (\lambda\_\ \_.\ \mathsf{conflict\_with\_cls})\ K'\ L'\ D'\ (b', n', ps'_{C'})$
	The array E' refines the clause C, and $(b', n', ps'_\top)$ refines $\top$	

Figure 5.3.: Conversion from the lookup table to a clause, assuming $C \neq \mathsf{None}$

the polarity of atom i in the conflict clause—i.e, whether the literal i occurs positively, negatively, or not at all in the clause. More precisely, a conflict clause is represented by a triple (b, n, ps), where b indicates whether the conflict is $\top$ and n stores the size of the conflict clause. The n component is useful to quickly test whether the conflict clause is empty, or whether it has size one.

There are two main differences between the lookup table and the original version. First, duplicate literals and tautologies cannot be represented. I know from my invariants that this is not an issue. Second, the clause can only contain atoms that are smaller than the length of the array.

To give a sense of what this involves, I describe the refinement of a small program fragment from the abstract level, where a conflict is an optional multiset, to the concrete level, where a conflict is a lookup table. At the end of Jump_Nonunit, I need to convert the conflict clause C to a list that I can add to my list of clauses such that two given literals $L, L' \in C$ are watched (i.e., are at positions 0 and 1). This conversion is specified abstractly as

$$\begin{aligned}\mathsf{RES}\,\{(D, \mathsf{None}) \mid D\,!\,0 = L \land D\,!\,1 = L' \land \mathsf{mset}\,D = C \\ \land |D| \geq 2\}\end{aligned}$$

The condition $|D| \geq 2$ ensures that the accesses to positions 0 and 1 are well-defined. In the refined code, I convert the lookup table to an array (D in the specification) and empty the lookup table (instead of reallocating a new one later; this is the None in the specification).

The refinement is done in two steps. I first refine the specification to an intermediate function that describes the implementation on the level of the abstract data structures (leftmost column of Figure 5.3). In a second step, the abstract data structures and operations are refined to concrete data structures and operations (rightmost column of Figure 5.3). The middle column gives the refinement relation that connects the notions of states used on either side, before and after every statement. Each statement from the intermediate code is mapped to a concrete function, such that the refinement relation of the result is also the refinement relation of the arguments of the next statement. Since intermediate and concrete functions must have the same number of arguments, some arguments are ignored on the concrete side (indicated by the unbound argument _ in the λ-abstractions).

5.7.3. Conflict Clause Minimization

I follow a minimization scheme due to Sörensson and Biere [113]. If the conflict is $E \lor K$, where E contains the literal L that is always kept in rule Jump, and I can show that $N \uplus U \vDash E \vee -K$, then by resolution I have $N \uplus U \vDash E$ and the conflict can be reduced to E. More precisely, minimization is a recursive procedure that considers each literal K of the conflict distinct from L in turn:

1. If K appears in E, then $E \vee K$ can be reduced to E.
2. If $-K$ is set at level 0 in the trail, then $-K$ is entailed by $N \uplus U$ and $E \vee K$ can be reduced to E.
3. If $(-K)^{-K \vee C}$ appears in the trail and for each literal K' of C, I have that $E \vee K'$ be recursively reduced to E, then $E \vee K$ can be reduced to E.
4. Otherwise (e.g., if K was decided), the literal K is kept.

The minimization procedures terminates because the literals K' have been set earlier than K. To optimize the procedure, I cache the clause's minimization status: "can be minimized", "cannot be minimized", or "not determined yet." This turns out to be the trickiest part of the proof. After exploring many dead ends, I found that I can define "can be minimized" as $N \uplus U \vDash E^{\succ_M K} \vee -K$, where $E^{\succ_M K}$ denotes the subclause of E consisting only of literals that appear to the right of K in the trail M.

Minimization is specified abstractly in terms of multisets and refined to an efficient implementation using the lookup-table representation. To simplify the code, when propagating a literal I ensure it appears at the first position in the clause, as in MiniSat. Similarly to VMTF, I prove correctness but no notion of optimality: I especially do not prove that all literals of level 0 are removed.

5.7.4. State Representation

The states I are considering before generating code in Imperative HOL are eight-tuples $(M, NU, u, D, NP, UP, Q, W)$. However, two components are redundant and can be eliminated: Unit clauses are added to NP and UP but never accessed afterwards.

Initially, I wrote code as I have shown in Section 5.5: All function bodies started with let $(M, NU, u, D, NP, UP, WS, Q) = S$. This made it convenient to refer to the components individually, or to refine them. I could also add information to the components during refinement. For example, since the VMTF heuristic depends on the trail, its *vmtf* tuple could only be added to the refined trail component. However, this approach works only if the additional information depends on a single component. Moreover, it offers no means of eliminating redundant components such as NP and UP.

After gathering some experience with the Refinement Framework, I decided to move to a different scheme, following which all state manipulation is mediated by accessor functions. I can then refine each of these functions individually. For example, when refining $(M, NU, u, D, NP, UP, WS, Q)$ to the intermediate representation $(M, NU, u, D, WS, Q, vmtf, \varphi)$ with heuristics (where *vmtf* is the VMTF data structure and φ is the mapping used for phase saving), the get_queue function that selects the eighth tuple component is mapped to a function that selects the sixth tuple component.

There is, however, a difficulty with this scheme. In an imperative implementation, a getter that returns a component of a state that is stored on the heap must either copy the component or return a pointer into the state. The first option can be very inefficient, and the alternative is not supported by the Sepref tool, which does not permit pointer aliases. My solution is to provide ad hoc getters to extract the relevant information from the state, without exposing parts of the state simultaneously to the whole state (which would require aliasing). Similarly, I provide setter functions to update components of the state.

For example, after reducing a conflict (rules Resolve and Skip), I must distinguish between either jumping (rules Jump_Unit and Jump_Nonunit) or stopping the solver by testing whether the conflict was reduced to $\bot$:

$$\mathsf{the}\ (\mathsf{get\_conflict}_{\mathsf{wlist}}\ S) = \emptyset$$

(The result is unspecified if the conflict is $\top$, i.e., None.)

Since all I need is the emptiness check and not the conflict clause itself, I can define a specialized getter:

$$\mathsf{conflict\_is\_empty}_{\mathsf{wlist}}\ S \longleftrightarrow \mathsf{the}\ (\mathsf{get\_conflict}_{\mathsf{wlist}}\ S) = \emptyset$$

Then I refine it to the intermediate state with heuristics:

$$\begin{aligned}&\mathsf{conflict\_is\_empty}_{\mathsf{heuristic}}\ (M, NU, u, D, WS, Q, vmtf, ')\\ &\longleftrightarrow \mathsf{conflict\_is\_empty}\ D\end{aligned}$$

with the following auxiliary function that operates only on the D component:

$$\mathsf{conflict\_is\_empty}\ D \longleftrightarrow \mathsf{the}\ D = \emptyset$$

Next, I refine the auxiliary function to use the lookup-table representation:

$$\mathsf{conflict\_is\_empty}_{\mathsf{lookup}}\ (b, n, ps) \longleftrightarrow n = 0$$

Finally, this function is given to Sepref, which generates Imperative HOL code. This, in turn, makes it possible to synthesize $\mathsf{conflict\_is\_empty}_{\mathsf{heuristic}}$.

The representation of states changes between refinement layers. It can also change within a layer, to store temporary information. Consider the number of literals of maximum level in the conflict clause. When it reaches 1, the Resolve rule no longer applies. Keeping this number around, in a locally enriched state tuple, can be much more efficient than iterating over the conflict clause to evaluate the maximum level. With my initial concrete notion of state as an eight-tuple, adding this information would have required a new layer of refinement, since the level depends simultaneously on two state components (the trail and the conflict clause).

5.7.5. Fast Polarity Checking

SAT solvers test very often the polarity of literals. Therefore testing it by iterating over the trail is too inefficient. In practice, solvers employ a map from atoms to their current polarity. Since the atoms are natural numbers, I enrich the trail data structure with a list of polarities (of type *bool option*), such that the $(i+1)$st element gives the polarity of atom i. The new polarity function is defined as follows:

definition $\mathsf{polarity}_{\mathsf{list\_pair}}$
$:: \mathit{nat\ literal} \Rightarrow (\mathit{nat},\ \mathit{clause\_idx})\ \mathit{ann\_literal\ list} \times \mathit{bool\ option\ list} \Rightarrow \mathit{bool\ option}$
where
$\mathsf{polarity}_{\mathsf{list\_pair}}\ L\ (M, Ls) = \big(\mathsf{case}\ Ls\ !\ \mathsf{atm\_of}\ L\ \mathsf{of}$
$\quad \mathsf{None} \Rightarrow \mathsf{None}$
$\mid \mathsf{Some}\ b \Rightarrow \mathsf{Some}\ (\mathsf{if}\ \mathsf{is\_pos}\ L\ \mathsf{then}\ b\ \mathsf{else}\ \neg\, b)\big)$

Given $\mathcal{L}_{\mathrm{all}}$ the set of all valid literals (i.e., the positive and negative version of all atoms that appear in the problem), the refinement relation between the trail with the list of polarities and the simple trail is defined as follows:

definition $\mathsf{trail}_{\mathsf{list\_pair}}\mathsf{\_trail\_ref}$
$:: \big(\big((\mathit{nat},\ \mathit{clause\_idx})\ \mathit{ann\_literal\ list} \times \mathit{bool\ option\ list}\big) \times (\mathit{nat},\ \mathit{clause\_idx})\ \mathit{ann\_literal\ list}\big)\ \mathit{set}$
where
$\mathsf{trail}_{\mathsf{list\_pair}}\mathsf{\_trail\_ref} =$
$\big\{((M', Ls), M).\ M = M' \wedge \forall L \in \mathcal{L}_{\mathrm{all}}.\ \mathsf{atm\_of}\ L < |Ls| \wedge Ls\ !\ \mathsf{atm\_of}\ L = \mathsf{polarity}\ M\ L\big\}$

This invariant ensures that the list Ls is long enough and contains the polarities. I can link the new polarity function to the simpler one. If $((M', Ls), M) \in$ $\mathsf{trail}_{\mathsf{list\_pair}}\mathsf{\_trail\_ref}$, then

$$\mathsf{RETURN}\ \left(\mathsf{polarity}_{\mathsf{list\_pair}}\ (M', Ls)\ L\right) \leq \mathsf{RETURN}\ \left(\mathsf{polarity}\ M\ L\right) \tag{5.5}$$

In a subsequent refinement step, I use Sepref to implement the list of polarities by an array, and atoms are mapped to 32-bits unsigned integers (*uint32*), as in Section 5.6. Accordingly, I define two auxiliary relations:

- The relation $\mathsf{lit\_assn} :: \mathit{nat\ literal} \Rightarrow \mathit{uint32\_literal} \Rightarrow \mathit{assn}$ refines a literal with natural number atoms by a literal encoded as a 32-bit unsigned integer.

- $\mathsf{trail}_{\mathsf{list\_pair}}\mathsf{\_assn} :: (\mathit{nat}, \mathit{clause\_idx})\ \mathit{ann\_literal\ list} \times \mathit{bool\ option\ list} \Rightarrow$ $\mathit{uint32\_ann\_literal\ list} \times \mathit{bool\ option\ array} \Rightarrow \mathit{assn}$ is a relation refining the trail data structure to use an array of polarities (instead of a list) and annotated literals of type *uint32_ann_literal*, using the 32-bit representation of literals. The clause indices of type *clause_idx* remain unbounded unsigned integers.

Sepref generates the imperative program polarity_code and derives the following refinement theorem:

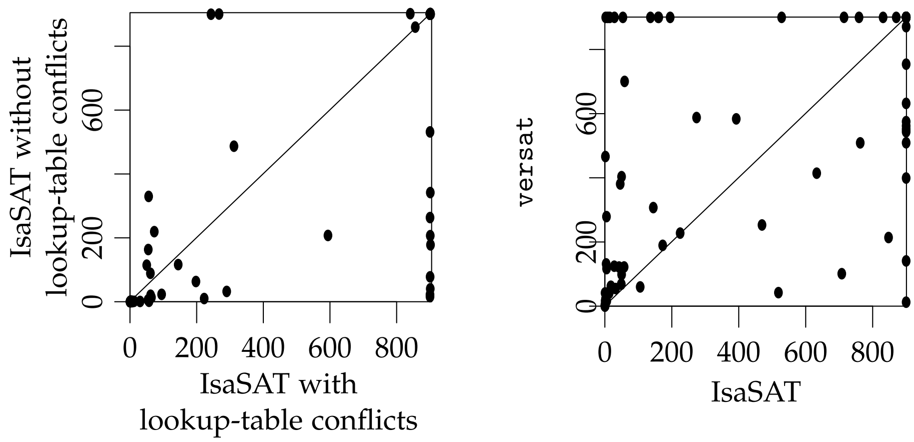
$$\begin{aligned} &(\mathsf{polarity\_code}, \mathsf{RETURN} \circ \mathsf{polarity}_{\mathsf{list\_pair}}) \\ \in &\left[\lambda((M, Ls), L).\ \mathsf{atm\_of}\ L < |Ls|\right] \mathsf{trail}_{\mathsf{list\_pair}}\mathsf{\_assn}^{\mathsf{k}} \times \mathsf{lit\_assn}^{\mathsf{k}} \rightarrow \mathsf{id\_assn} \end{aligned} \tag{5.6}$$

The precondition, in square brackets, ensures that I can only take the polarity of a literal that is within bounds. The term after the arrow is the refinement for the result, which is trivial here because the data structure for polarities remains *bool option*.

Composing the refinement steps (5.5) and (5.6) yields the theorem

$$\begin{aligned} (\mathsf{polarity\_code},\ \mathsf{RETURN} \circ \mathsf{polarity}) \in [\lambda(M, L).\ L \in \mathcal{L}_{\mathrm{all}}] \\ \mathsf{trail\_assn}^{\mathsf{k}} \times \mathsf{lit\_assn}^{\mathsf{k}} \rightarrow \mathsf{id\_assn} \end{aligned}$$

where trail_assn combines both refinement relations for trails trail_list_pair_assn and $\mathsf{trail}_{\mathsf{list\_pair}}\mathsf{\_trail\_ref}$. The precondition $\mathsf{atm\_of}\ L < |Ls|$ is a consequence of $L \in \mathcal{L}_{\mathrm{all}}$ and the invariant $\mathsf{trail}_{\mathsf{list\_pair}}\mathsf{\_trail\_ref}$. If I invoke Sepref now and discharge polarity_code's preconditions, all occurrences of the unoptimized polarity function are be replaced by polarity_code. After adapting the initialization to allocate the array for Ls of the correct size, I can prove end-to-end correctness as before with respect to the optimized code.



(a) IsaSAT with and without the lookup-table conflict representation (Section 5.7.2), without caching the number of literals of highest level in the conflict

(b) IsaSAT and `versat`

Figure 5.4.: Comparison of performance on the problems classified easy or medium from the SAT Competition 2009

Using stepwise refinement, I integrate this optimization into the imperative data structure used for the trail. This refinement step is isolated from the rest of the development, which only relies on its final result: a more efficient implementation of the trail and its operations. As Lammich observed elsewhere [68], this kind of modularity is invaluable when designing complex data structures.

5.8. Evaluation

I compare the performance of my solver, IsaSAT, with Glucose 4.1 [1], MiniSat 2.2 [37], DPT 2.0, and `versat` [97].

`versat`, by Oe et al. [97], is specified and verified using the Guru proof assistant [115], which can generate C code. `versat` consists of 15 000 lines of C code. Optimized data structures are used, including for watched literals and conflict analysis (but not for conflict minimization), and a variant of VSIDS heuristic is in charge of decisions. However, termination is not guaranteed, and model soundness is proved trivially by means of a run-time check of the models; if this check fails, the solver's outcome is "unknown."

I ran all five solvers on the 150 problems classified easy or medium from the SAT Competition 2009, with a time limit of 900 s. Glucose solves 147 problems, spending 51 s on average per problem it solves. MiniSat solves 143 problems in 98 s. DPT solves 70 problems in 206 s on average. `versat` solves 53 problems in 235 s on average.

To evaluate the lookup-table-conflict representation, I ran IsaSAT without caching of the number of literals of maximum level. IsaSAT without the lookup table solves 43 problems in 126 s on average, while the version with a lookup table solves only 36 problems in 127 s on average (including four problems that the version without lookup table could not solve). IsaSAT with every optimization solves 56 problems in 183 s on average. As an indication of how far I have come, the functional solver implementing the CDCL_W calculus [22, Section 5] and my first imperative unoptimized version with watched literals do not solve *any* of the problems. The solvers were run on a Xeon E5-2680 with 256 GB of memory, with Intel Turbo Boost deactivated. Globally, the experiments show that Glucose and MiniSat are much faster than the other solvers and that DPT solves substantially more instances than IsaSAT and `versat`, which are roughly comparable.

A more precise comparison of performance of my solver with and without the lookup table is shown in Figure 5.4a. A point at coordinates (x, y) indicates that the version with the lookup table took x seconds, whereas the version without the table took y seconds. Points located above the main diagonal correspond to problems for which the table pays off. Figure 5.4b compares `versat` and the optimized IsaSAT: It shows that either solver solves some problems on which the other solver times out. This is to be expected given that the two solvers implement different decision heuristics.

There are several reasons explaining why my solver is much slower than the state of the art. First, it lacks restarts and forgetting. This limit will be lifted in Chapter 6. Glucose and MiniSat also use preprocessing techniques to simplify the initial set of clauses. Other SAT solvers, such as Lingeling [11], also use inprocessing techniques to simplify initial and learned clauses after restarts.

Another difference is that Isabelle/HOL can only generate code in impure functional languages, whereas most unverified SAT solvers are developed in C or C++. Although I proved that all array accesses are within bounds, functional languages nonetheless check array bounds at run-time. Moreover, other features, such as the arbitrary-precision arithmetic (which I use for clause indices), tend to be less efficient than their C++ counterparts.

To reduce these effects, I implemented literals by 32-bit unsigned integers (which required some extra work to prove absence of overflows). This increased the speed of my solver by a factor between two and four. In a slight extension of the trusted base of the code generation, I convert literals directly

Refinement Level	Name	
1	CDCL_W	is correct and terminates.
2	TWL	adds watched literals.
3	Algo	enters the nondeterminic transition monad.
4	$\mathsf{TWL}_{\mathsf{list}}$	uses lists instead of multisets.
5	$\mathsf{TWL}_{\mathsf{wlist}}$	adds watch lists to the state.
6	$\mathsf{TWL}_{\mathsf{wlist}}+\mathcal{L}_{\mathrm{all}}$	restricts literals to be in the input problems.
7	Heur	adds all heuristics.
8	IsaSAT	is synthesized by Sepref.

Figure 5.5.: Summary of the layers used to generate code

to machine-size integers (32- or 64-bit), instead of taking the detour through unbounded integers. This simple change improved performance by another factor of two.

5.9. Summary

In this chapter, I have presented a refinement from CDCL_W to TWL, a calculus that include the two-watched-literal scheme. Starting from the later calculus, I refine it to an executable version. This refinement is done gradually and each step changes the data structures or specialize the behavior. The different layers are summarized in Figure 5.5. For example, the watch lists, although critical for performance, are added only in a later step. Finally, the complete SAT solver, IsaSAT, is obtained. Once combined with an unverified parser, I can compare it to state-of-the-art SAT solver. It is slower than state-of-the-art SAT solvers, but faster than `versat`. In the next chapter, I will optimize it further by adding Restart and Forget and other optimization.

6. Optimizing my Verified SAT Solver IsaSAT

In this chapter, I optimize IsaSAT further. Because some idioms made the proofs hard to maintain and slow to process, I first refactored the Isabelle formalization (Section 6.1). The first optimization is the use of *blocking literals* [30] to improve Boolean constraint propagation (Section 6.2). The idea is to cache a literal for each clause—if the literal is true in the current partial model of the solver, the clause can be ignored (saving a likely cache miss by not accessing the clause).

To avoid focusing on hard parts of the search space, the search of a SAT solver is heuristically restarted and the search direction changed, because the decision heuristic VMTF is dynamic and, therefore, leads to different decisions. Keeping too many clauses slows down unit propagation. Hence clauses that are deemed useless are also forgotten. However, the standard heuristics rely on the presence of meta-information in clauses that can be efficiently accessed. To make this possible, I redesigned the clause representation, which also allowed me to implement the *position saving* [44] heuristic (Section 6.3). Extending the SAT solver with *restart* and *forget* required the extension of the calculus with watched literals: Both behaviors were already present in the abstract calculus CDCL_W but were not implemented in the next refinement step. Heuristics are critical and easy to verify, but hard to implement in a way that improves performance (Section 6.4).

Using machine integers instead of unbounded integers is another useful optimization. The new IsaSAT thus uses machine integers until the numbers do not fit in them anymore, in which case unbounded integers are used to maintain completeness: theoretically, IsaSAT could have to learn more than 2^{64} clauses before reaching the conclusion SAT or UNSAT, which would overflow clause counters and indices to clauses in the watch list. The code is duplicated in the generated code but specified and refined only once (Section 6.5): Sepref is able to synthesize both versions.

I analyze the importance of the different features and compare IsaSAT with state-of-the-art solvers (Section 6.6). Even though the new features improve the performance IsaSAT significantly, much more work is required to match the best unverified solvers.

Finally, IsaSAT is a very large development and I will give some ideas where to start, if someone wants to extend it (Section 6.9).

6.1. Refactoring IsaSAT

The optimizations require changes in the proofs and in the code. My first step is a refactoring to simplify maintenance and writing of proofs.

Proof Style. The original and most low-level proof style is the apply script: It is a forward style and each tactic creates subgoals. It is ideal for proof exploration and simple proofs. It is, however, hard to maintain. A more readable style states explicit statements of properties in Isar [121]. The styles can be combined: each intermediate step can be recursively justified by apply scripts or Isar. For robustness, I use Isar where possible.

The tactics aligning goals are inherently apply style, but I prefer Isar. I will show the difference on the example of the refinement of $\mathsf{PCUI}_{\mathsf{algo}}$ (Figure 6.1a) by $\mathsf{PCUI}_{\mathsf{list}}$ (Figure 6.1b). Assume the arguments of the function are related by the relation $((LC, S), (LC', S')) \in R_{\mathrm{state}}$. The first two goals stemming from aligning $\mathsf{PCUI}_{\mathsf{algo}}$ with $\mathsf{PCUI}_{\mathsf{list}}$ are

$$\forall L'\ L\ C\ C'.\ ((LC, S), (LC', S')) \in R_{\mathrm{state}} \land LC = (L, C) \land LC' = (L', C') \rightarrow \\ (LC, LC') \in R_{\mathrm{watched}} \tag{6.1}$$

$$\forall L'\ L\ C\ C'.\ ((LC, S), (LC', S')) \in R_{\mathrm{state}} \land LC = (L, C) \land LC' = (L', C') \\ \land (LC, LC') \in R_{\mathrm{watched}} \rightarrow \\ \mathsf{RES}\ (\mathsf{watched}\ C - \{L\}) \le\Downarrow R_{\mathrm{other\ watched}}(\mathsf{RES}\ (\mathsf{watched}\ C' - \{L'\})) \tag{6.2}$$

where equation (6.1) relates the two lets, equation (6.2) the two RES, and the relations R_{watched} and $R_{\mathrm{other\ watched}}$ are two schematic variables that have to be instantiated during the proof (e.g., by the identity). Although I strive to use sensible variable names, they are lost when aligning the programs, making the goals harder to understand.

A slightly modified version of Haftmann's `explore` tool [48] transforms the goals into Isar statements. The workflow to use it is the following. First, use Sepref's tactic to align two programs. Then, `explore` prints the structured statements. Finally, those statements can be inserted in the theory, before the goal. Figure 6.2a shows the output: equations (6.1) and (6.2) corresponds to the two `have` statements, where **have** $R\,x$ **if** $P\,x$ **and** $Q\,x$ **for** x stands for

definition $\mathsf{PCUI}_{\mathsf{algo}}$ **where**
  $\mathsf{PCUI}_{\mathsf{algo}}\ LC\ S = \text{do}\ \{$
    $\text{let}\ (L, C) = LC;$
    $L' \leftarrow \mathsf{RES}\ (\mathsf{watched}\ C - \{L\});$
    $\text{if}\ L' \in \mathsf{trail}_{\mathsf{list}}\ S\ \text{then}$
      $\mathsf{RETURN}\ S$
    $\text{else} \ldots$
  $\}$

(a) Ignore rule after refactoring

definition $\mathsf{PCUI}_{\mathsf{wlist}}$ **where**
  $\mathsf{PCUI}_{\mathsf{list}}\ LC\ S = \text{do}\ \{$
    $\text{let}\ (L, C) = LC;$
    $L' \leftarrow \mathsf{RES}\ (\mathsf{watched}\ C - \{L\});$
    $\text{if}\ L' \in \mathsf{trail}_{\mathsf{list}}\ S\ \text{then}$
      $\mathsf{RETURN}\ S$
    $\text{else} \ldots$
  $\}$

(b) Ignore rule after refactoring

Figure 6.1.: Comparison of the code of Ignore rule in Algo before and after refactoring

have $(LC, LC') \in R_{\text{watched}}$
  if $LC = (L, C)$ **and** $LC' = (L', C')$
    and $((LC, S), (LC', S')) \in R_{\text{state}}$
  for $L'\ L\ C\ C'$
  sorry
have $\mathsf{RES}\ (\mathsf{watched}\ C - \{L\})$
    $\leq \Downarrow R_{\text{other watched}}$
      $(\mathsf{RES}\ (\mathsf{watched}\ C' - \{L'\})$
  if $(LC, LC') \in R_{\text{watched}}$ **and**
    $LC = (L, C)$ **and** $LC' = (L', C')$
    and $((LC, S), (LS', S')) \in R_{\text{state}}$
  for $L'\ L\ C\ C'\ C\ C'$
  sorry

(a) Proof as generated by `explore`: no sharing of variables and assumptions across goals

have $(LC, LC') \in R_{\text{watched}}$
  if $LC = (L, C)$ **and** $LC' = (L', C')$
    and $((LC, S), (LC', S')) \in R_{\text{state}}$
  for $L'\ L\ C\ C'$
  sorry
have $\mathsf{RES}\ (\mathsf{watched}\ C - \{L\})$
    $\leq \Downarrow R_{\text{other watched}}$
      $(\mathsf{RES}\ (\mathsf{watched}\ C' - \{L'\})$
  if $(LC, LC') \in R_{\text{watched}}$ **and**
    $LC = (L, C)$ **and** $LC' = (L', C')$
    and $((LC, S), (LS', S')) \in R_{\text{state}}$
  for $L'\ L\ C\ C'\ C\ C'$
  sorry

(b) Proof with contexts as generated `explore_context`, with sharing of variables and assumptions across goals

Figure 6.2.: Different ways of writing the proof that $\mathsf{PCUI}_{\mathsf{list}}$ from Figure 6.1a refines $\mathsf{PCUI}_{\mathsf{algo}}$

the unstructured goal $\forall x.\,(P\,x \wedge Q\,x \longrightarrow R\,x)$. Each goal can be named and used to solve one proof obligations arising from the alignment of the two programs.

`explore` does not change the goals and hence, variables and assumptions are not shared between proof steps, leading to duplication across goals. I later expanded the `explore` to preprocess the goals before printing them: It uses **context**s (Figure 6.2b) that introduces blocks sharing variables and assumptions. These proofs are now faster to check and write and minor changes are easier to do. There is no formal link between the statements and the goal obligations: If the goal obligations changes, the Isar statements have to be updated by hand. After big changes in the refined functions, it can be easier to regenerate the new statements, re-add them to the theory, and reprove them than to adapt the old one. Thankfully, this only happens a few times, usually when significantly changing the function anyway, which also significantly changes the proof.

Heuristics and Data Structures. At first, the implementation of heuristics and optimized data structures was carried out in three steps:

1. use specification and abstract data structure in Heur (e.g., the conflict clause is an optional multiset);

2. map the operations on abstract to concrete functions (e.g., the function converting a clause to a conflict clause is refined to a specific function converting a clause to a lookup table);

3. discharge the preconditions from step 2 with Sepref (e.g., no duplicate literal).

In principle, if step 2 is changed, Sepref can synthesize a new version of the code without other changes, making it easy to generate several versions to compare heuristics and data structures. However, in practice, this never happens because optimizing code further always requires stronger invariants, requiring to change the proofs for step 3. Moreover, Sepref's failures to discharge preconditions are tedious to debug. To address this, I switched to a different approach:

1′. introduce the heuristics and data structures in Heur (e.g., the conflict is a lookup table);

2′. add assertions for preconditions on code generation to Heur.

The theorems used to prove steps 2 are now used during the refinement to Heur. Sepref is also faster since the proofs of 2′ are now mostly trivial: Either the invariant is asserted or the synthesis should fail. In one extreme case, Sepref took 24 minutes before failing with the old approach. After identifying the error, the solution was to add another theorem, recall Sepref, and wait. Thanks to this simpler approach and the entire-state based refinement, Sepref now takes only 16 s to synthesize the code (or fail).

Layer Conception. As described in Section 5.6, IsaSAT initially relied on a locale parametrized by $\mathcal{L}_{\mathtt{all}}$ (layer $\mathsf{TWL}_{\mathsf{wlist}}{+}\mathcal{L}_{\mathsf{all}}$), the set of all literals that appear in the initial set of clauses or their negation. This is also the set of all literals that can appear during the execution of the program. This set is very useful to express conditions on the size of lists for heuristics (Section 5.7.5). However, using locales has some drawbacks: First, all functions are defined in the same namespace. This is not issue as long as Isabelle is not entering and exiting locales too often, because switching between locales is slow. However, this is exactly what happens each time code is synthesized (as described in Section 5.1.4). A more dramatic issue is that synthesizing the code for the whole SAT solver is complicated: $\mathcal{L}_{\mathtt{all}}$ is both a parameter of *the functions* and of the *refinement relation*. Therefore, a relation refining f by g would have the form

$$(\mathsf{g}, \mathsf{f}) \in [\lambda \mathcal{L}'_{\mathtt{all}}.\ \mathcal{L}'_{\mathtt{all}} = \mathcal{L}_{\mathtt{all}}]\ R\,\mathcal{L}_{\mathtt{all}} \rightarrow S\,\mathcal{L}_{\mathtt{all}}$$

where $R\,\mathcal{L}_{\mathtt{all}}$ is the relation refining the arguments, $S\,\mathcal{L}_{\mathtt{all}}$ the relation refining the image. The precondition $\mathcal{L}'_{\mathtt{all}} = \mathcal{L}_{\mathtt{all}}$ ensures that $\mathcal{L}_{\mathtt{all}}$ is the only possible argument. However, such relations are not supported by Sepref: Free variables like $\mathcal{L}_{\mathtt{all}}$ are not instantiated. At first, I found a workaround: Internally, Sepref uses an intermediate form, called heap-nres (or hnr), which is then used for code synethesis. I could transform the theorems into hnr-form, then using reflexivity to force the variables to be equal, and finally using this version of the theorem for synthesis. However, after some other changes, the setup broke and I decided to replace all the occurrences of $\mathcal{L}_{\mathtt{all}}$ by the set of literals in the problem itself, since they are equal and every function already contained the assertion that the two sets were equal. This is a bit more cumbersome, because $\mathcal{L}_{\mathtt{all}}$ now has to be passed as argument to every relation (e.g. $\mathsf{trail}_{\mathsf{list\_pair}}\mathsf{\_trail\_ref}$ of Section 5.7.5). However, it also simplified the code generation of the whole SAT solver and the formalization became a bit faster to check.

definition $\mathsf{PCUI}_{\mathsf{algo}}$ **where**
$\mathsf{PCUI}_{\mathsf{algo}}\, LC\, S = \text{do}\ \{$
  let $(L, C) = LC$;
  $L' \leftarrow \mathsf{RES}\ \{L' \mid L' \in C\}$;
  if $L' \in \mathsf{trail}\ S$ then
    $\mathsf{RETURN}\ S$
  else do $\{$
    $L'' \leftarrow \mathsf{RES}\ (\mathsf{watched}\ C - \{L\})$;
    if $L'' \in \mathsf{trail}\ S$ then
      $\mathsf{RETURN}\ S$
    else ...
  $\}$
$\}$

(a) Ignore part of the $\mathsf{PCUI}_{\mathsf{algo}}$ in Algo with blocking literals

definition $\mathsf{PCUI}_{\mathsf{wlist}}$ **where**
$\mathsf{PCUI}_{\mathsf{wlist}}\, L\, i\, S = \text{do}\ \{$
  let $(L', C) = \mathsf{watch\_list\_at}\ S\ L\ i$;
  let $L' = L'$;
  if $L' \in \mathsf{trail}\ S$ then
    $\mathsf{RETURN}\ S$
  else do $\{$
    $L'' \leftarrow \mathsf{RES}\ (\mathsf{watched}\ C - \{L\})$;
    if $L'' \in \mathsf{trail}\ S$ then
      $\mathsf{RETURN}\ S$
    else ...
  $\}$
$\}$

(b) Ignore in WList with watch lists and blocking literals

Figure 6.3.: Refinement of the rule Ignore with blocking literals from Algo to WList

6.2. Adding Blocking Literals

Blocking literals [30] are an extension of the two-watched-literal scheme and are composed of two parts: a relaxed invariant and the caching of a literal. Most SAT solvers implement both aspects. Blocking literals reduce the number of memory accesses (and, therefore, of cache misses).

Invariant. IsaSAT-17's version of the two-watched-literal scheme is inspired by MiniSAT 1.13. The key invariant is the following [42]:

> A watched literal can be false only if *the other watched literal* is true or all the unwatched literals are false.

I now relax the condition by replacing "the other watched literal" by "any other literal". This weaker version means that there are fewer changes to do to the watched literals: If there is a true literal, no change is required. Accordingly, the side conditions of the Ignore rule of TWL can be relaxed from $L' \in \mathsf{watched}\ C$ to $L' \in C$. Adapting the proof of correctness was relatively easy. The proofs are easy to fix (after adding some key lemmas) thanks to Sledgehammer [18], a tool that uses automatic theorem provers to find proofs.

The generalized Ignore rule is refined to the non-determinism monad (Figure 6.3a). Since the calculus has only been generalized, no change in the refinement would have been necessary. In the code, the rule can be applied in three different ways: Either L', the other watched literal L'', or another literal from the clause is true (the last case is not shown in Figure 6.3). Any literal (even the false watched literal L) can be chosen for L'.

Caching of a literal. Most SAT solvers contain an second part: When visiting a clause, it is often sufficient to visit a single literal [106]. Therefore, to avoid a likely cache miss, a literal per clause, called *blocking literal*, is cached in the watch lists. If it is true, no additional work is required; otherwise, the clause is visited: If a true literal is found, this literal is elected as new blocking literal, requiring no update of the watch lists.

In the refinement step WList, the choice is fixed to the cached literal from the watch list (Figure 6.3b). The identity "let $L' = L'$;" helps the tactics of the Refinement Framework to recognize L' as the choice for RES $\{L' \mid L' \in C\}$, i.e. yielding the goal obligation $L' \in$ RES $\{L' \mid L' \in C\}$.

IsaSAT's invariant on the blocking literal forces the blocking literal to be *different* from the associated watch literal (corresponding to the condition $L \neq L'$ in Figure 6.3). This is not necessary for correctness but offers better performance (since L is always false) and enables special handling of binary clauses: No memory access is necessary to know the content of the clause. IsaSAT's watched lists contain an additional Boolean indicating whether the clause is binary.

6.3. Improving Memory Management

The representation of clauses and their metadata used for heuristics is crucial for the performance of SAT solvers. Most solvers use two ideas: First, they keep the metadata and clauses together. For example, MiniSAT puts the metadata before the clause. The second idea is that memory allocation puts clauses one after the other in memory to improve locality.

init	3	A	B	C	learn	4	$\neg A$	$\neg B$	C	D

Figure 6.4.: Example of arena module with two clauses $A \vee B \vee C$ (initial clause, 'init') and $\neg A \vee \neg B \vee C \vee D$ (learned clause, 'learn')

However, none of these two tricks can be directly obtained by refinement and Isabelle offers no control over the memory allocator. Therefore, I implemented both optimizations at once, similarly to the implementation in CaDiCaL [12]. The implementation uses a large array, the *arena*, to allocate each clause one after the other, with the metadata before the clauses (Figure 6.4): The lengths (here 3 and 4) precede the clause. Whereas the specifications allow the representation to contain holes between clauses, the concrete implementation avoids it.

In IsaSAT-17, the clauses were a list of clauses, each one being a list of literals (both lists being refined to arrays). This representation could not be refined to an arena. Moreover, it was not compatible with removing clauses without shifting the positions. For example, if the first clause was removed from the list $[A \vee B \vee C;\ \neg A \vee \neg B \vee C \vee D]$, then the position of the second clause changed. This was a problem as the indices are used in the trail. Therefore, I first changed the representation from a list of lists to a mapping from natural numbers to clauses. Then, every element of the domain was mapped to a clause in the arena with the same index (for example, in Figure 6.4, the clause 2 is $A \vee B \vee C$; 7 is $\neg A \vee \neg B \vee C \vee D$; there are no other clauses).

Introducing arenas requires some subtle changes to the existing code base. First, the arena contains natural numbers (clause length) and literals (clause content). Therefore, I use a datatype (as a tagged union) that contains either a literal or a natural number. Both types are refined to the same type, a 32-bits word and the datatype is removed when synthesizing code. An invariant on the whole arena describes its content. Moreover, because literals are refined to 32-bit machine words, the length has to fit in 32 bits. However, as the input problems can contain at most 2^{16} different atoms and duplicate-free tautologies, the maximum length of a clause is 2^{32}. To make it possible to represent all clauses including those of size 2^{32}, the arena actually keeps the number of unwatched literals (i.e., the length minus 2), unlike Figure 6.4.

While introducing the arena, I also optimized parts of the formalization. I replaced loops on a clause starting at position C in the arena (i.e., iterations on $C + i$ for i in $[0, \text{length}\, C]$) by loops on the arena fragment (i.e., iteration on i for i in $[C, C + \text{length}\, C]$). This makes it impossible to compare IsaSAT-30 with and without the memory module without changes in the formalization. The impact of the arena was small (improvement of 2%, and a few more problems could be solved), but arenas make it possible to add metadata for heuristics.

Position Saving. I implemented a heuristic called *position saving* [44], which requires an additional metadata. It considers a clause as a circular buffer:

When looking for a new literal, the search starts from the last searched position instead of starting from the first non-watched literal of the clause. The position is saved as a metadata of the clause. Similarly to CaDiCaL [12], the heuristic is only used for long clauses (length larger than four). Otherwise, the position field is not allocated in the arena (i.e., the size of the metadata depends on the clause size). Incorporating the heuristic was easy thanks to non-determinism. For example, to apply the Ignore rule, finding a true literal is sufficient, *how* it is found is not specified. This makes it easy to verify a different search algorithm.

Although there exist some benchmarks showing that this technique improve the performance of solvers [13], only CaDiCaL and Lingeling [12] implement it and I did not know if it would improve IsaSAT: The generated code is hardly readable and hard to change in order to test such techniques. However, it was easy to add and it improves performance on most problems (see Section 6.6).

6.4. Implementing Restarts and Forgets

CDCL-based SAT solvers have a tendency to get stuck in a fruitless area of the search space and to clutter their memory with too many learned clauses. Most modern SAT solvers offer two countermeasures. Restarts try to avoid focusing on a hard part of the search space. Forgets limit the number of clauses because too many of them slow down the solver.

Completeness is not guaranteed anymore if restart and forget are applied too often. To keep completeness, I delay them more and more. TWL does not propagate clauses of length 1, because they do not fit in the two-watched-literal scheme. These clauses are propagated during the initialization are

$\mathit{to\_skip} \ \leftarrow \mathsf{RES}\ \{n.\ \mathsf{True}\};$
$\mathsf{WHILE}(\lambda(\mathit{to\_skip}, i, S).\ \langle\text{there is a clause to update or } \mathit{to\_skip} > 0\rangle))$
$\quad(\lambda(\mathit{to\_skip}, i, S).\,\mathsf{do}\ \{$
$\quad\quad\mathit{skip\_element} \leftarrow \mathsf{RES}\ \{b \mid b \rightarrow \mathit{to\_skip} > 0\}$
$\quad\quad\mathsf{if}\ \mathit{skip\_element}\ \mathsf{then}\ \mathsf{RETURN}(\mathit{to\_skip} - 1, i, S) \qquad\qquad (*\ \text{do nothing}\ *)$
$\quad\quad\mathsf{else\ do}\{$
$\quad\quad\quad LC \leftarrow \langle\text{some literal and clause to update}\rangle;$
$\quad\quad\quad\mathsf{PCUI}_{\mathsf{algo}}\ LC\ S\ \}$
$\quad\})$

Figure 6.5.: Skipping deleted clauses during iteration over the watch list

cannot be removed from the trail. However, such clauses will always be repropagated by CDCL_W. Therefore, a TWL restart corresponds to a CDCL_W restart and some propagations. If decisions are also kept, then IsaSAT can reuse parts of the trail [102]. This technique avoids redoing some work after a restart. The trail could even be entirely reused if the decision heuristics would do the same decisions.

When forgetting several clauses at once, called one *reduction step*, IsaSAT uses the LBD [1] (least block distance) to sort the clauses by importance, and then keeps only linearly many (linear in the number restarts). All other learned clauses are deleted. I have not yet implemented garbage collection for the arena, so deleted clauses currently remain in memory forever.

After clauses have been marked as deleted, the watch lists are not garbage collected. Instead, before accessing a clause, IsaSAT tests if the clause has been deleted or not. However, this is an implementation-specific detail I do not want to mirror in Algo. To address this, I changed Algo in a less intrusive way. Before Algo was iterating over *WS*. After the change, a finite number of no-ops is added to the while loop (Figure 6.5). When aligning the two programs, an iteration over a deleted clause is mapped to a no-op. More precisely, there are two tests: whether the blocking literal is true and whether the clause is marked as deleted. If the blocking literal is true, the state does not change (whether the clause is deleted or not). Otherwise, the clause has to be accessed. If the clause is deleted, it is removed from the watch list.

IsaSAT uses the EMA-14 heuristic [15], which is based on two exponential moving averages of scores, implemented using fixed-point numbers: a "slow" average measuring the long-term tendency of the scores and a "fast" one for the local tendency. If the fast average is worse than the slow one, the heuristic is triggered. Then, depending on the number of clauses, either restart or reduce is triggered. The heuristic follows the unpublished implementation of CaDiCaL [12], with fixed-point calculations. This is easier to implement than Glucose's queue for scores. Due to programming errors, it took several iterations to get EMA-14 right: The first version never restarted while the second did as soon as possible. Both versions even the second were complete because the number of conflicts between successve restart slowly increased: It was initially 50, then 51, after that 52, ... Once I understood where the programming error was, I fixed it and IsaSAT performed much better.

Later, I found out that I forget some bitshifting during the calculation of the moving average. Fixing it had two effects. First, it led to noticeably more restarts (roughly ten times more restarts), bringing the number of restarts closer to the values printed by other SAT solver. Second, it decreased performance. It is not clear if this is indicates another error in the implementation of the heuristic (it cannot be a correctness bug), or if this is only an issue due to the choices of the constants.

The reuse of parts of the trail was not hard to implement, but it does not seem to happen very often. The VMTF decision heuristic probably changes too fast. Therefore, the decisions on the trail are very likely to be different after a restart. I tested that on a few examples and barely anything of the trail is reused. Most of the time, zero or one level is reused. However, the maintenance and performance cost is low. Therefore, I kept it in the formalization.

6.5. Using Machine Integers

When I started to work on IsaSAT, it was natural to use unbounded integers to index clauses in the arena (refined from Isabelle's natural numbers). First, they are the only way to write lists accesses in Isabelle (further refined to array accesses). Second, they are also required for completeness to index the clauses and there was also no code-generation setup for array accesses with machine words. Finally, the Standard ML compiler I use, MLton [118], efficiently implements numbers first as machine words and then as unbounded GMP integers with one bit indicating whether something is a pointer or a machine integer For these reasons, using machine words seemed unnecessary. However, profiling showed that subtractions and additions took among them around 10% of the time.

I decided to switch to machine words. Instead of failing upon overflow or restarting the search from scratch with unbounded integers, IsaSAT switches in the middle of the search:

while $\neg$ *done* $\wedge$ $\neg$ *overflow* do
  $\langle$invoke the 64-bit version of the solver's body$\rangle$;
if $\neg$ *done* then
  $\langle$convert the state from 64-bit to unbounded integers$\rangle$;
  while $\neg$ *done* do
    $\langle$invoke the unbounded version of the solver's body$\rangle$

The switch is done pessimistically. When the length of the arena is longer than $2^{64} - 2^{16} - 5$ (maximum size of a non-tautological clause without duplicate literals is 2^{16} and 5 is the maximal number of header fields), the solver switches to unbounded integers, regardless of the size of the next clause. This bound is large enough to make a switch unlikely in practice. In Isabelle, the two versions of the solver's body are just two instances of the same function where Sepref has refined Isabelle's natural numbers differently during the synthesis. To synthesize machine words, Sepref must prove that numbers cannot overflow. For example, if i is refined to the 64-bit machine word w, then the machine-word addition $w + 1$ refines $i + 1$ if the addition does not overflow, i.e., $i + 1 < 2^{64}$. The code for data structures like resizable arrays (used for watch lists) has not been changed and, therefore, still uses unbounded integers. However, some code was changed to limit manipulation on the length of resizable arrays.

IsaSAT uses 64-bit machine words instead of 32-bit machine words. They are used in the trail but mostly in the watch lists. Using 32-bits words would be more cache friendlier for the trail. However, this would not make any difference for watch lists. Each element in a watch list contains a clause index, a 32-bit literal, and a Boolean. Due to padding, there is not size difference for 32 and 64-bit words. Moreover, the SAT Competition contains problems that require more memory than fits in 32 bits: After hitting the limit, IsaSAT would switch to the slower unbounded version of the solver, whereas no switch is necessary for 64-bit indices.

6.6. Evaluation

I evaluated IsaSAT-30 on preprocessed problems from the SAT Competitions 2009 to 2017 and from the SAT Race 2015 using a timeout of 1800 s. The hardware was an Intel Xeon E5620, 2.40 GHz, 4 cores, 8 threads. Each instance was limited to 10 GB of RAM. The problems were preprocessed by CryptoMiniSat [112]. The motivation behind this is that preprocessing can significantly simplify the problem. Detailed results can be found on the companion web page[1].

State-of-the-art solvers solve more problems than IsaSAT with the default options (Figure 6.6). Since the instances have already been preprocessed, the difference comes from a combination of simplifications (pre- and inprocessing), better heuristics, and a better implementation. To assess the difference, I have also benchmarked the solvers without simplification (third column of Figure 6.6). For Glucose and Minisat the difference is small, unlike for CaDiCaL and CryptoMiniSat, who are doing much more inprocessing (and the heuristics are optimized towards it). Heule's MicroSAT [52] aims at being very short (240 lines of code including comments). Compared with IsaSAT, it has neither position saving nor blocking literals but is highly optimized and its heuristics work well together. The version without the four presented optimizations differs from IsaSAT-17 by a faster conflict analysis, a different

[1] https://people.mpi-inf.mpg.de/~mfleury/paper/results-NFM/results.html

SAT solver	Default options Solved	Default options Average time (s)	No simplification Solved	No simplification Average time (s)
CryptoMiniSat	1774	349	1637	349
Glucose	1703	320	1696	303
CaDiCaL	1677	361	1602	346
MiniSAT	1388	326	1373	317
MicroSAT	1018	310	N/A	
IsaSAT-30 fixed heuristic	801	359	N/A	
zChaff	573	345	N/A	
IsaSAT-30 without the four optimizations	433	301	N/A	
IsaSAT-17	393	220	N/A	
`versat`	368	224	N/A	

Figure 6.6.: Performance of some SAT solvers (N/A if no simplification is performed by default)

Reduction	Restarts	Position saving	Machine words	Solved	Average time (s)	Average memory (GB)
				520	294	2.1
			✓	551	291	2.3
		✓		526	281	2.1
		✓	✓	547	289	2.3
	✓			666	292	2.2
	✓		✓	713	312	2.5
	✓	✓		712	294	2.4
	✓	✓	✓	**753**	306	2.7
✓				433	213	1.6
✓			✓	448	207	1.7
✓		✓		446	212	1.6
✓		✓	✓	456	204	1.7
✓	✓			677	336	2.8
✓	✓		✓	738	339	3.1
✓	✓	✓		705	324	2.9
✓	✓	✓	✓	749	338	3.2

Figure 6.7.: Benchmarks of variants of IsaSAT-30 before fixing the forget heuristic

decision heuristic, blocking literals, and various minor optimizations. IsaSAT performs better than the only other verified SAT solver with efficient data structures I know of, `versat` [97]. I also include the older solver zChaff 04 [80], that was state-of-the-art around 2004.

I compared the impact of reduction, restart, position saving, and machine words (Figure 6.7). Since Standard ML is garbage-collected, the peak memory usage depends on the system's available memory. The results show that restarts and machine words have a significant impact on the number of solved problems. The results are less clear for the other features. Position saving mostly has a positive impact. The negative influence of reduction hints at a bad heuristic: I later tuned the heuristic by keeping clauses involved in the conflict analysis and the results improved from 749 to 801 problems. The fact that garbage collection of the arena is not implemented could also have an impact, as memory is wasted.

6.7. Extracting Efficient Code

When refining the code, it is generally not clear which invariants will be needed later. However, I noticed that improvements on data structures also require stronger properties. Therefore, proving them early can help further refinement but also makes the proofs more complicated. Another issue is that the generated code is not readable, which makes it extremely hard to change in order to test if a data structure or a heuristic improves speed.

Profiling is crucial to obtain good performance. First, it shows if there are some obvious gains. However, profiling Standard ML code is not easy. MLton has a profiler which only gives the total amount of time spent in the function (not including the function calls in its body) and not the time per path in the call graph. So performance bugs in functions that do not dominate run time are impossible to identify. One striking example was the insertion sort used to sort the clauses during reduction. It was the comparison function that was dominating the run time, not the sort itself, which I changed to quicksort.

Continuous testing also turned out to be important. It can catch performance regression before any change in the search behavior is done, allowing me to debug them. One extreme example was the special handling of binary clauses: A Boolean was added to every element of the watch list, changing the type from `word64 * word32` to `word64 * (word32 * bool)`. This change in the critical spot of any SAT solver caused a performance loss of around 20% due to 3.5 times as many cache misses. Since the search behavior had not changed, I took a single problem and tried to understand where the regression came from. First, `word64 * (word32 * bool)` is less efficient than `word64 * word32 * bool` as it requires a pointer for `word32 * bool`. This can be alleviated by using a single constructor datatype (the code generator generates the later version and the single constructor is optimized away). However, there is a second issue: The tuple uses three 64-bit words, whereas only two would be used in the equivalent C structure. I added code equations to merge the `word32 * bool` into a single `word64` (with 31 unused bits), solving the regression. Developers of non-verified SAT solvers face similar issues[2] but they are more tools for C and C++.

Code generation in Isabelle is built on a mapping from Imperative HOL operation to concrete code in the target language. This mapping is composed of *code equation* translating code (like array access) and the correctness of the mapping cannot be verified without semantics of the target language. While working on the SAT solver, I added several code equations to the trusted code base. The additional code equations are either trying to avoid the conversions to unbounded integers (`IntInf`) and back (as would happen by default when accessing arrays) or related to printing (statistics during the execution). Whether or not the equations are safe is not always obvious. For example, the code equations to access arrays *without* converting the numbers to unbounded integers and back[3] are safe as long as the array bounds are checked.

However, IsaSAT is compiled with an option that deactivates array-access bound checks. When accessing elements outside of an array, the behavior is undefined. As long as I am using Sepref and clauses of the input do not contain duplicate literals, validity of the memory accesses was proved. Without the custom code equations and with bound checks, only 536 problems (average time: 283 s) are solved, instead of 749.

Equivalent C code would be more efficient. First, as already mentioned, there are differences in the memory guarantees. Standard ML does not provide information on the alignment. A second issue are spurious reallocations. A simple example is the function `fun (propa, s) => (propa + 1, s)`. This simple function (counting the number of propagations) is responsible for 1.7% of all allocations although I would expect no extra allocation. A third issue is that the generated code is written in a functional style with many unit arguments `fun () => ...` to ensure that side effects are done in the right order. Not every compiler supports optimizing these additional constructs away.

---

[2] e.g., https://www.msoos.org/2016/03/memory-layout-of-clauses-in-minisat/

[3] although the Standard ML specification encourages compilers to optimize such code

All the optimizations have an impact on the length of the formalization. The whole formalization is around 31 000 lines of proof for refinement from TWL to the last layer Heur, 35 000 lines (Heur and code generation), and 9000 lines for libraries. The generated Standard ML code including all auxiliary functions is 8100 lines long.

6.8. Detailed TWL Invariants in Isabelle

In this section, I describe in detail the invariants to prove correctness of the watched literals invariants. This can serve as a base for testing or adding assertion when implemented an SMT solver or SAT solver where clauses are added.

I first define two predicates, one indicating whether a literal is a blocking and whether a clause has a true literal with respect to a given literal.

definition is_blit :: $('a, 'b)$ *ann_literals* $\Rightarrow$ *'a clause* $\Rightarrow$ *'a literal* $\Rightarrow$ *bool* **where**
is_blit $M\,D\,L = (L \in D \land L \in M)$
definition has_blit :: $('a, 'b)$ *ann_literals* $\Rightarrow$ *'a clause* $\Rightarrow$ *'a literal* $\Rightarrow$ *bool* **where**
has_blit $M\,D\,L' = (\exists L.\,$ is_blit $M\,D\,L \land$ get_level $M\,L \leq$ get_level $M\,L')$

The restriction get_level $M\,L \leq$ get_level $M\,L'$ is not important in a SAT solver, because L' is typically a literal whose negation has been propagated after the last decision. Therefore, L' is always of maximum level, making the inequality automatically true. While this is automatically true in a SAT solver, this is the core argument of the compatibility of the two-watched-literal scheme with Backjump and Restart.

In practice, the invariants for a clause C on watched literals do not hold most of the time. They only hold if C does need any update:

definition twl_is_an_exception **where**
twl_is_an_exception $C\;Q\;WS =$
$(\exists L.\; L \in Q \land L \in$ watched $C) \lor (\exists L.\; (L, C) \in WS)$

When a false literal is watched and the clause does not have a blocking literal, then this literal is of maximum level and is of level higher than all other literals. This invariant is natural in SAT solver, since conflicts are always detected eagerly before any further decision is made.

fun twl_lazy_update :: $('a, 'b)$ *ann_literals* $\Rightarrow$ *'a twl_cls* $\Rightarrow$ *bool* **where**
twl_lazy_update M (TWL_Clause $W\;UW) =$

$$\forall L.\,(L \in W \land -L \in M \land \neg\mathsf{has\_blit}\,M\,(W+UW)\,L) \longrightarrow$$
$$(\forall K \in UW.\,\mathsf{get\_level}\,M\,L \geq \mathsf{get\_level}\,M\,K \land -K \in M)$$

fun watched_literals_false_of_max_level :: $('a,'b)\,\mathit{ann\_literals} \Rightarrow {'a}\,\mathit{twl\_cls} \Rightarrow$ *bool* **where**
$$\mathsf{watched\_literals\_false\_of\_max\_level}\,M\,(\mathsf{TWL\_Clause}\,W\,UW) =$$
$$\forall L.\,L \in W \land -L \in M \land \neg\mathsf{has\_blit}\,M\,(W+UW)\,L \longrightarrow$$
$$\mathsf{get\_level}\,M\,L = \mathsf{count\_decided}\,M$$

The previous can be combined to an invariant that is always true when executing TWL:

fun twl_st_inv :: $'v\,\mathit{twl\_st} \Rightarrow \mathit{bool}$ **where**
$$\mathsf{twl\_st\_inv}(M,N,U,D,NE,UE,WS,Q) =$$
$$(\forall C \in N+U.\,\mathsf{struct\_wf\_twl\_cls}\,C) \land$$
$$(\forall C \in N+U.\,D = \mathsf{None} \land \neg\mathsf{twl\_is\_an\_exception}\,C\,Q\,WS \longrightarrow$$
$$\mathsf{twl\_lazy\_update}\,M\,C) \land$$
$$(\forall C \in N+U.\,D = \mathsf{None} \longrightarrow \mathsf{watched\_literals\_false\_of\_max\_level}\,M\,C)$$

An important property is the compatibility with backtrack and restarts. It is given by the following invariant:

fun past_invs :: $'v\,\mathit{twl\_st} \Rightarrow \mathit{bool}$ **where**
$$\mathsf{past\_invs}(M,N,U,D,NE,UE,WS,Q) =$$
$$\forall M1\,M2\,K.\,M = M2@\mathsf{Decided}\,K\#M1 \longrightarrow ($$
$$(\forall C \in N+U.\,\mathsf{twl\_lazy\_update}\,M1\,C \land$$
$$\mathsf{watched\_literals\_false\_of\_max\_level}\,M1\,C \land$$
$$\mathsf{twl\_exception\_inv}\,(M1,N,U,\mathsf{None},NE,UE,\emptyset,\emptyset)\,C) \land$$
$$\mathsf{confl\_cands\_enqueued}\,(M1,N,U,\mathsf{None},NE,UE,\emptyset,\emptyset) \land$$
$$\mathsf{propa\_cands\_enqueued}\,(M1,N,U,\mathsf{None},NE,UE,\emptyset,\emptyset) \land$$
$$\mathsf{clauses\_to\_update\_inv}\,(M1,N,U,\mathsf{None},NE,UE,\emptyset,\emptyset))$$

It states that all invariants still holds after Backtrack and Restart.

Failing to fulfill the invariants can lead to unforeseen issues. One example is an issue that is in the SAT solver from the solver SPASS-SATT [4] [25] (technically not an SMT solver, because only one theory at a time is supported): The theory solver can give a new clause to the SAT solver to justify a propagation. However, this is not done eagerly and, therefore, it can happen that a propagation is not done at the right level. For example, if the trail is $B^\dagger A^\dagger$,

[4] https://www.mpi-inf.mpg.de/departments/automation-of-logic/software/spass-workbench/spass-satt/

the theory solver can ask the SAT solver to learn and propagate the clause $\neg A \vee C$, yielding the trail $C^{\neg A \vee C} B^{\dagger} A^{\dagger}$. If the SAT solver backtracks or restarts to level 1, the trail becomes $A^{\dagger}$, but the clause $\neg A \vee C$ is not repropagated. While this should not lead to bugs in SPASS-SATT, it can lead to efficiency losses. The lost clause might be repropagated after the next restart (if few enough literals of the trail are reused) or it will found during the next round inprocessing (because it is associated with restart at level 0). In a SAT solver, an important invariant is that there is at least on literal of highest level in the conflict, which is not the case here anymore whenever such a clause becomes a conflicting clause. However, the core of most SMT solvers can also use the Skip rule over decisions, which solves the issue.

The invariants above are not the only invariants that have to be fulfilled. For example, IsaSAT relies on the fact that if $L^{C \vee L}$ is in the trail, then L has to be at first position in the clause $C \vee L$ (unless the clause $C \vee L$ is binary).

6.9. Extending IsaSAT

The simplest case is the extension of IsaSAT with different heuristics. In this case, only the Heur layer needs to be changed. This corresponds to the file with name `IsaSAT_*.thy` in the repository. In some cases, especially if the propagation loop is changed, then also refinement of the two-watched-literal calculus might have to change (files `Watched_Literals*.thy`).

Currently no features are provided to add or remove literals during the run (which is also an issue for inprocessing). The best point to change the problem during a run is during a restart: change and simplify the problem there, adapt the data structures, and restart the run of the solver. Remark that especially adding literals should be done with care, as keeping the same number of literals is currently the essence of the termination proof. Removing the restriction that literals fit in 32-bit words is not hard, but should be done with care to avoid harming performance more than necessary, even though I expect that switching to 64-bit literals to be relatively harmless, except for the merge trick used in the watch list (Section 6.7).

I started the extension of CDCL to enumerate models: It is a system that is similar to a very naive SMT solver, where the theory only supports partial models and does not create any propagation (and does not add any variable). I refined this version to the two-watched-literal scheme (without restarts and forget) but I did not refine it past WList, although the main function calls IsaSAT. Hence, heuristics used in IsaSAT can be reused (files `Model_Enumeration.thy` and `Watched_Literals_*_Enumeration.thy`). It can also be used as a starting point to refine OCDCL from Chapter 4 to verified executable code, because the proof obligations will be similar. Termination comes from the fact that either a model is found, or no model exists anymore (no variable is added, only the negation of the decisions of the trail is added to the clauses).

6.10. Summary

In this chapter, I have optimized my verified SAT solver further by adding adding two features that were already included in CDCL_W, but not yet refined to TWL, forget and restart. I have also extended the TWL calculus to include blocking literals in a very abstract way, before extending the $\mathsf{TWL}_{\mathsf{wlist}}$ layer to efficiently use blocking literals. The inclusion of forget required some changes in the memory representation. Finally, the use of machine words instead of unbounded integers in IsaSAT as long as possible improves the performance of the overall solver.

7. Discussion and Conclusion

In the final chapter of this thesis, I discuss in more details the other formalizations and SAT solvers, and compare them to my own work. The most important difference is the *refinement* approach I used all along my formalization to inherit, reuse properties, and extend the formalization (Section 7.1). After a brief summary of this thesis (Section 7.2), I give some ideas of future work (Section 7.3)

7.1. Discussion and Related Work

Discussion on the CDCL formalization My formalization of the DPLL and CDCL calculi consists of about 17 000 lines of Isabelle text (Figure 7.1). The work was done over a period of 10 months, and I taught myself Isabelle during that time. It covers nearly all the metatheoretical material of Sections 2.6 to 2.11 of *Automated Reasoning* and Section 2 of Nieuwenhuis et al., including normal form transformations and ground unordered resolution, which I partly formalized during my Master's thesis [39]. The formalization of CDCL_W and the functional implementation were already formalized in my Master's thesis [39]. However, I did not formalize CDCL_NOT, Nieuwenhuis et al. account of CDCL (and therefore neither the link between CDCL_NOT and CDCL_W), I did not use locales, the calculus did not include conflict min-

Formalization part	Length (kloc)
Libraries for CDCL	3
CDCL	17
CDCL Extensions	5
Libraries for refinement and code generation	6
From TWL to $\mathsf{TWL}_{\mathsf{wlist}}+\mathcal{L}_{\mathrm{all}}$	26
Heur and code generation	35

Figure 7.1.: Length of various parts of the formalization (in thousands lines of code, not accounting for empty lines)

imization, and I did not use the refinement approach. All of this is part of my PhD thesis.

It is difficult to quantify the cost of formalization as opposed to paper proofs. For a sketchy paper proof, formalization may take an arbitrarily long time; indeed, Weidenbach's eight-line proof of Theorem 3.10 initially took 700 lines of Isabelle. In contrast, given a very detailed paper proof, one can sometimes obtain a formalization in less time than it took to write the paper proof [126].

A frequent hurdle to formalization is the lack of suitable libraries. I spent considerable time adding definitions, lemmas, and automation hints to Isabelle's multiset library, and the refinement to resizable arrays of arrays required an elaborate setup, but otherwise I did not need any special libraries. I also found that organizing the proof at a high level, especially locale engineering, is more challenging, and perhaps even more time-consuming, than discharging proof obligations. Sometimes having alternate definitions of invariants makes them easier to use for Isabelle's built-in tactics or less likely to cause loops (especially, during when the simplifier runs).

Given the varied level of formality of the proofs in the draft of *Automated Reasoning*, it is unlikely that I will ever formalize the whole textbook. But the insights arising from formalization have already enriched the textbook in many ways. For the calculi described in this paper, the main issues were that fundamental invariants were omitted and some proofs may have been too sketchy to be accessible to the book's intended audience.

For discharging proof obligations, I relied heavily on Sledgehammer, including its facility for generating detailed Isar proofs [18] and the SMT-based *smt* tactic [23]. I found the SMT solver CVC4 particularly useful, corroborating earlier empirical evaluations [103]; although they were much less useful during the Heur step of refinement. In contrast, the counterexample generators Nitpick and Quickcheck [17] were seldom useful. We often discovered flawed conjectures by observing Sledgehammer fail to solve an easy-looking problem. As one example among many, I lost perhaps one hour working from the hypothesis that converting a set to a multiset and back is the identity. Because Isabelle's multisets are finite, the property does not hold for infinite sets A; yet Nitpick and Quickcheck fail to find a counterexample, because they try only finite values for A (and Quickcheck cannot cope with underspecification anyway).

Other CDCL formalizations. At the calculus level, I followed Nieuwenhuis et al. (Section 3.1) and Weidenbach (Section 3.2), but other accounts exist. In particular, Krstić and Goel [63] present a calculus that lies between CDCL_NOT and CDCL_W on a scale from abstract to concrete. Unlike Nieuwenhuis et al., they have a concrete Backjumping rule. On the other hand, whereas Weidenbach only allows to resolve the conflict (Resolution) with the clause that was used to propagate a literal, Krstić and Goel allow any clause that could have cause the propagation (rule Explain). Another difference is that their Learn and Backtrack rules must explicitly check that no clause is learned twice (cf. Theorem 3.10). The authors mention that the check is not required in MiniSAT-like implementations, but no proof of this statement is provided.

Formalizing metatheoretical results about logic in a proof assistant is an enticing, even though somewhat self-referential, prospect. Shankar's proof of Gödel's first incompleteness theorem [110], Harrison's formalization of basic first-order model theory [50], and Margetson and Ridge's formalized completeness and cut elimination theorems [82] are some landmark results in this area.

In his Ph.D. thesis, Lescuyer [74] presents the formalization of the CDCL calculus and the core of an SMT solver in Coq. He also developed a reflexive DPLL-based SAT solver for Coq, which can be used as a tactic in the proof assistant. Another formalization of a CDCL-based SAT solver, including termination but excluding two watched literals, is by Shankar and Vaucher in PVS [111]. Most of this work was done by Vaucher during a two-month internship, an impressive achievement.

CDCL extensions. I am not aware of any attempt to formalize extensions of CDCL in a proof assistant, but there are several presentations of optimizing SAT. Larossa et al. have developed a similar approach to mine [71]. They define cost optimality with respect to models (without specifying if total or partial models are meant), but their Improve rule only considers total models. Our calculus is slightly more general due to the inclusion of the rule $\mathsf{Improve}^+$. Moreover, the first unique implication point is built into our calculus. The Pruning rule can be simulated by their Learn rule: $\neg M \lor c \geq \mathrm{cost}(O)$ is entailed by the clauses.

A related problem to finding the minimum partial model is called minimum-weight propositional satisfiability by Sebastiani et al. [109]. It assumes that negative literals do not cost anything: In my case, the opposite of L is $\neg L$ or L undefined. In this case, $\neg L$ and L undefined have the same weight. This makes his version compatible with the learning mechanism of CDCL.

Liberatore developed a variant of DPLL to solve this problem [77]. Each time a variable is decided, it is first set to true, then set to false. If the current model is larger than a given bound, then the search stops exploring the current branch. When a new better model is found, the search is restarted with the new lower bound. A version lifted to CDCL was implemented in zChaff [45] to solve MAX-SAT. Although Liberatore's method can return partial models, it is only an Herbrand model: It is entirely given by the set of all true atoms. Therefore, the method actually builds total models.

I have presented here a variant of CDCL to find optimal models and used the dual rail encoding to reduce the search of optimal models with respect to partial valuations to the search of optimal models with respect to total valuations. Both have been formalized using the proof assistant Isabelle/HOL. This formalization fits nicely into the framework we have previously developed and the abstraction we have used in Isabelle to simplify reuse and studying variants and extensions.

I started the encoding for cost-minimal models with respect to partial valuations by introducing three extra variables for each variable based on Zimmer's upcoming Bachelor thesis. Compared with the dual rail encoding, the third extra variable explicitly modeled whether a variable is defined or undefined. I performed the content of Section 4.3 and Section 4.4 with this encoding and only afterwards were pointed by a reviewer to the dual rail encoding. It took us half a day to redo the overall formalization. For me, this is another example that the reuse of formalizations can work. This is further demonstrated by the application of the OCDCL results to MAX-SAT and the reuse of the formalization framework to verify the model covering calculus CDCLcm, Section 4.5. Minimization of the model covering set computed by CDCLcm can also be solved by an afterwards application of a CDCL calculus with branch-and-bound [81], and would probably fit in my framework.

Discussion on Refinement. I found formalizing the two watched literals challenging. In the literature, only variants of the invariant from Section 5.2 are presented. However, there are several other key properties that are necessary to prove that no work is needed when backjumping. For example, the invariant states that "a watched literal may be false only if the other watched literal is true," but this is not the whole story. It would be more precise to state that "a watched literal may be false only if the other watched literal is true *and this false literal's level is greater than or equal to the true literal's level*." This version of the invariant explains why no update is required after Jump: Either both watched literals are now unset in the trail, or only the true literal remains.

One difficulty I faced when adding optimizations is that the "edit, compile, run” cycle is much longer when code is developed through the Isabelle Refinement Framework instead of directly in a programming language such as C++. For example, the change to the conflict-clause representation took two weeks to prove and implement, before I found out that the overall solver gets slower. I have yet to find a good methodology for carrying out quick experiments.

The distinguishing feature of my work is the systematic application of refinement to connect abstract calculi with generated code. The Refinement Framework allows me to generate imperative code while keeping programs underspecified for as long as possible. It makes it straightforward to change the implementation or to derive multiple implementations from the same abstract specification. Its support for assertions makes it possible to reuse properties proved on an abstract level to reason about more concrete levels.

One of my initial motivations for using locales, besides the ease with which it lets me express relationships between calculi, was that it allows abstracting over the concrete representation of the state. My first idea was instantiating the CDCL_W locale with the more complicated TWL data structures and convert these structures in the selectors to the data structures from CDCL_W. However, I discovered that this is often too restrictive, because some data structures need sophisticated invariants, which I must establish at the abstract level. I found myself having to modify the base locale each time I attempted to refine the data structure, an extremely tedious endeavor.

In contrast, the Refinement Framework, with its focus on functions, allows me to exploit local assumptions. Consider the prepend_trail function (Section 3.1.2), which adds a literal to the trail. Whenever the function is called, the literal is not already set and appears in the clauses. The polarity-checking optimization (Section 5.7.5) relies on this property to avoid checking bounds when updating the atom-to-polarity map. With the Refinement Framework, there are enough assumptions in the context to establish the property. With a locale, I would have to restrict the specification of prepend_trail to handle only those cases where the literals are in the set of clauses, leading to changes in the locale definition itself and to all its uses, well beyond the polarity-checking code.

While refining to the heap monad, I discovered several issues with my program. I had forgotten several assertions (especially array bound checks) and sometimes mixed up the $^{\mathrm{k}}$ and $^{\mathrm{d}}$ annotations, resulting in large, hard-to-interpret proof obligations. Sepref is a very useful tool, but it provides few safeguards or hints when something goes wrong. Moreover, the Isabelle/jEdit user interface can be unbearably slow at displaying large proof obligations.

The Refinement Framework’s lack of support for pointer aliasing impacted our solver in two main ways. First, I had to use array indices instead of pointers to clauses. This moved the dependency between the array and the clause from the code level to the abstract specification level. Second, array access $NU ! C$ must take a copy of the array at position C. I avoided this issue by consistently using two-dimensional indexing, $(NU ! C) ! i$, which yields an unsigned 32-bit integer representing a literal.

The longest part was the refinement from the abstract algorithm to the executable version. To improve performance, I studied the generated code and looked for bottlenecks. This was tedious: The code is hardly readable, with generated variable names (only function names are kept). But at least, at every step I knew that the code was correct.

Formalizing heuristics turns out to be surprisingly hard: There is no guarantee that they behave correctly and it is extremely hard to compare the behavior to other SAT solvers. For example, during restarts, the beginning of the trail can be reused (Section 6.4). When testing it, I have remarked that most of the time only one or two are levels are reused. I do not know if this is normal (it could be a side effect of the VMTF decision heuristic that often shuffles the order) or it indicates that this a performance problem in the interplay of the heuristics.

Other Formalized SAT solvers. SAT solvers have been formalized in proof assistants, with the aim of obtaining to executable code. Marić [83,85] verified a CDCL-based SAT solver in Isabelle/HOL, including two watched literals, as a purely functional program. The solver is monolithic, which complicates extensions. Marić's methodology is quite different from mine, without the use of refinements, inductive predicates, locales, or even Sledgehammer. More precisely, he developed.

1. *A CDCL calculus*: He formalized the abstract CDCL calculus by Nieuwenhuis et al. and, together with Janičić [83,86], the more concrete calculus by Krstić and Goel [63].

2. *The two-watched-literal scheme*: It is a different scheme than the one I verified. Most notably, propagations are not done immediately. In my notation, that corresponds to only propagating when taking a literal out of *WS*, instead of directly propagating it when adding it to *WS*.

3. *A code extraction of an executable SAT solver [85]*: A SAT solver was derived from the refined calculus. This solver does not have any efficient data structure and is implemented in a purely functional style. This solver

contains only very naive heuristics: For example, the decision heuristic selects a random undefined literal.

4. *A connection by hand to the C++ solver Argo [84]*: Through a chain of refinement partly on paper and partly in Isabelle, Marić connected ArgoSAT to his functional code. This solver contains features that are not included in the code generated from Isabelle, such as conflict minimization, restart, and forget.

Oe et al. [97] verified an imperative and fairly efficient CDCL-based SAT solver, expressed using the Guru language for verified programming. Optimized data structures are used, including for two watched literals and conflict analysis. However, termination is not guaranteed, and model soundness is achieved through a run-time check and not proved. The two-watched-literal scheme used in `versat` is different from IsaSAT: Instead of updating one watched literal at a time, both can be updated at the same time.[1] They use the invariant described in Section 6.2, but do not have blocking literals. Unlike Marić's version and similarly to IsaSAT, literals are immediately propagated. The code of `versat` uses bounded integers: The code actually relies on C integers (`int`) to be exactly 32-bit words; otherwise, the behavior is undefined. Therefore, the solver is not complete and actually crashes when trying to solve some larger problems of the SAT competition. Technically, they have verified a checker inside a SAT solver: only the resolutions are certified. Proving this requires some additional proofs on the SAT solver (no undefined behavior, no crashes). To take the example of watch lists, this requires to prove that the pointer in the watch lists are valid pointers that points to clauses entailed by or present in the problem, but there was no reason to prove that every clause appears twice in the watch lists.

There are several formalizations of DPLL, including Berger et al. [10], whose Haskell solver outperforms `versat` on large pigeon-hole problems. (CDCL is not faster than DPLL on such problems, because the learned clauses are useless at pruning the search space.) Like `versat`, the resolution steps are certified, but termination and correctness of the returned model is proved. A partial verification is included in Roe's Ph.D. thesis [105]. He developed tools for Coq to automatically prove the correctness of structural invariants. He applied it on the verification of the two-watched-literal scheme in a DPLL solver. Only some properties have been verified (8 out of 83) and he only focuses on the structural invariants, not on heuristics. Although his code is obtained by parsing a C program, the data structures are non-standard in

[1] I am not aware of any state-of-the-art SAT solver doing so.

SAT solvers. The solver operates on structures called `clause`. Each `clause` is a C structure that contains an array of type `bool [NUM_ATMS]` where true at the ith position indicates that the atom i is watched and false that it is unwatched or not present in the clause. Therefore, finding the watched literals requires iterating over all atoms in the input problem. More generally, the data structures are not optimized for efficiency, since each clause contains several arrays of type `bool [NUM_ATMS]` (for positive literals, negative literals, watched atoms, next watched clauses, previous watched clauses), like a hashmap to indicate that an atom is present. They make iterating over the clause inefficient and the clause needs a lot of memory.

Other Verification Approaches. Given that I had formalized CDCL in Isabelle/HOL, it was natural to use the Isabelle Refinement Framework and Sepref. For Coq, the Fiat tool is available [36]. Like Sepref, it applies automatic data refinement to obtain efficient implementations. However, it is limited to purely functional implementations and does not support recursive programs. Nor does it support assertions, which are an important mechanism to move facts down the refinement chain instead of reproving them at each level.

Gries and Volpano [47] describe a data refinement approach that, like Sepref, automatically transforms abstract to concrete data structures, by replacing abstract with concrete operations. It refines imperative to imperative programs, whereas Sepref connects functional to imperative programs. To my knowledge, their approach does not use a theorem prover, i.e., the correctness of the transformations must be trusted.

Unlike the top-down approach used here, the verification of the seL4 microkernel [60] relies on abstracting the program to verify. An abstract specification in Isabelle is refined to an Haskell program. Then, a C program is abstracted and connected to the Haskell program. Unbounded integers are not supported in C and therefore achieving completeness of a SAT solver would not be possible: If there are more than 2^{64} clauses, integers larger than 64 bits are required. Whether such a computer exists is a different question. The goal obligations arising from such abstraction would be similar to the one I have already discharged. For example, I could start with a C version of CaDiCaL and import it to Isabelle with autocorres [96] and connect it to $\mathsf{TWL}_{\mathsf{wlist}}+\mathcal{L}_{\mathsf{all}}$. CaDiCaL uses the VMTF heuristic that, even if different variables are considered important, has similar invariants to IsaSAT.

Other techniques to abstract programs exist, like Chargueraud's characteristic formulas [29]. Another option is Why3 [38] or a similar verification condition generator like Dafny [73]. Some meta-arguments in Why3 (for example, incrementing a 64-bit machine integer initialized with 0 will not overflow in a reasonable amount of time; therefore, machine integers are safe [32]) would simplify the generation of efficient code. In any case, refinement helps to verify a large program.

Isabelle's code generator does not formally connect the generated code to the original function. On the one hand, Hupel's verified compiler [56, 57] from Isabelle to the semantics of the verified Standard ML compiler CakeML could bridge the gap. However, code export from Imperative HOL is not yet supported. On the other hand, HOL4 in conjunction with CakeML makes it possible to bridge this gap and also to reason about input and output like parsing the input file and printing the answer [64]. There is, however, no way to eliminate the array-access checks, because these conditions cannot be embedded in the program as done by assertion and no precondition can express that no array check is required in a function. For example, consider the two following programs:

definition $\mathsf{f1} :: \mathit{unit} \Rightarrow \mathit{nat}$ **where**
$\mathsf{f1}\,() = 42$

definition $\mathsf{f2} :: \mathit{unit} \Rightarrow \mathit{nat}$ **where**
$\mathsf{f2}\,() = (\mathbf{let}\ \_ = [\,]\ !\ 12\ \mathbf{in}\ 42)$

Since functions are total in HOL, both programs are well defined: $[\,]\ !\ 12$ is defined and returns an arbitrary (undefined) value. Moreover, the programs are equal in HOL, because they both return 42. The execution $\mathsf{f1}\,()$ is safe, but not the execution of $\mathsf{f2}$ in general, because $\mathsf{f2}$ accesses the empty array $[\,]$ outside of the bounds. Therefore, without array checks, the execution of $\mathsf{f2}\,()$ is undefined, but HOL cannot distinguish both definitions. Hence, no precondition on programs can ensure that a function is safe. One possible solution are assertions as done in the nondeterministic exception monad. The code in the CakeML semantics does not have assertions, making it impossible to prove that all arrays bounds have been checked.

Besides the array checks, CakeML uses boxed machine words unlike MLton: This means that every access to a machine word (including in arrays) requires following a pointer (and possibly a cache miss), which likely leads to a significant slowdown in the overall solver.

Certification. Instead of verifying a SAT solver, another way to obtain trustworthy results is to have the solver produce a certificate, which can be processed by a checker. While certificates for satisfiable formulas are simply a valuation of the variables and can easily be generated and checked, certificates for unsatisfiable formulas are more complicated. The de facto standard format is DRAT (deletion resolution asymmetric tautology) [55], which can be easily generated by solvers. The standard DRAT certificate checker [124] is, however, an unverified C program. Recent research [33,34,54,70] shows that it is now possible to have efficient verified checkers. Like a SAT solver, checkers uses many arrays and accesses then often. Therefore, they would likely benefit from machine words to improve performance.

7.2. Summary

In this thesis, I have presented my formalization of the conflict-driven-clause-learning procedure, based on two different accounts by Weidenbach and Nieuwenhuis et al. The most important feature in the formalization is the use of a nondeterministic transition system to represent the calculi and the approach by refinement. This makes it possible to reuse and extend the formalization by inheriting properties.

I have used the CDCL formalization as a framework to expand it further by making it incremental and developing an abstract CDCL with branch-and-bound, which is instantiated to find a total model with minimum weight and a covering set of models. The most important feature of the $\mathsf{CDCL}_{\mathrm{BnB}}$ framework is the reuse of the invariants and definitions from CDCL, reducing the hurdle to verify other variants of CDCL.

After that, I refined CDCL to include two important features for efficiency, the two-watched-literal scheme and blocking literals. I still present this system as a nondeterministic transition system.

Finally, after several further steps of refinement, I refined the transition system to executable code with efficient data structures. This solver is obtained by refining code and synthesizing code with imperative data structures from code with only functional data structures. Once combined with a trusted parser, it is efficient enough to solve problems from the SAT Competition and it performs better than any other verifies SAT solver.

7.3. Future Work

Lammich is currently working on generating LLVM [72] code which could give more control on the generated code (e.g., the tuples representation is more efficient). It could also enable to write structures instead of tuples that must be decomposed and recomposed in each function, and give access to more benchmarking tools. Generating LLVM changes the trusted part of the code: Instead of trusting the translation from Isabelle to Standard ML, the Isabelle version of the LLVM semantics must be trusted. As the code generator of Isabelle is taken out of the equation, there would no more `(fn () => ...) ()`, that must be optimized away by the compiler.[2]

There are several techniques missing IsaSAT compared with state-of-the-art SAT solvers, currently garbage collection of the arena module, but mostly pre- and inprecessing. Besides preprocessing that can tremendously simplify problems and is now standard in most SAT solvers, a technique called vivification [75,79] is now implemented in many SAT solvers that took part in the SAT Competition 2018. The technique is not new [100], but it is now included in several solvers that do not extensively focus on inprocessing.

I would like to extend my calculus to be able to represent CDCL($\mathcal{T}$), the calculus behind SMT solvers. The theory of linear arithmetic has already been implemented by Thiemann [116]. The authors of the CeTA checker would also be interested in using a verified SAT solver. They do not use Imperative HOL and there is no way to extract a result from code in Imperative HOL. Therefore, some work is required to adapt IsaSAT. One possible solution is to use Sepref to generate functional code in Imperative HOL and generate from it a version in HOL (i.e., outside of the nondeterminism monad).

[2]The most striking example of a missed optimization was the use of the constant 2^{32} in the propagation loop. MLton was not able to precalculate the value, which was instead re-evaluated every time. In this specific case, the slowdown was massive enough to be obvious and the fix very simple: replace 2^{32} by its value during code generation.

A. More Details on IsaSAT and Other SAT Solvers

This chapter describes some features of other SAT solvers and compares them to the implementation in IsaSAT. This section is independent and not necessary to understand the rest of the thesis.

A.1. Clauses

The clause representation is the central point of the performance of SAT solvers. The representations of IsaSAT and other SAT solvers are similar, even though only IsaSAT offers no abstraction over the clause representation: How the clauses are represented in memory is usually only handled by the allocator and does not impact the remaining code, while in IsaSAT, the arena module impacts every clause access, due to a lack of pointers (which in particular make sit impossible to write a generic memory manager).

A.1.1. IsaSAT

The arena module is presented in Section 6.3. With all heuristics, IsaSAT has the following headers before a clause:

- the saved position (Section 6.3), which is optional and kept only for long clauses.
- the status (initial, learned, or deleted) and whether the clause has been used. This field is extremely wasteful as only 3 bits are used (out of 32 bits).
- the activity of the clause and whether the clause has been used during the conflict analysis (rule Resolve).
- the LBD score of the clause.
- the size of the clause.

A.1.2. Glucose

Glucose uses a similar memory presentation to IsaSAT, with a flat memory representation. However, it is handled using an allocator defined in the file `XAlloc`, where the relevant part is the following:

```
template<class T>
class RegionAllocator
{
    T*          memory;
    uint32_t    sz;
    uint32_t    cap;
    uint32_t    wasted_;

}
```

The pointer `memory` is the allocated region and clauses are added to it.

A.1.3. CaDiCaL

IsaSAT's memory representation is inspired by CaDiCaL's. However, C++ makes the implementation easier to achieve. Clauses are defined in a class:

```
class Clause {
public:

  int _pos;

  int alignment;

  bool extended:1;

  /* several other Boolean flags */

  signed int glue : LD_MAX_GLUE;

  int size;

  union {
    int literals[2];
    Clause * copy;
  };
```

```
}
```

In the code of CaDiCaL, `literals[2]` is only a trick (although technically undefined behavior in C++) and the clause can (obviously) contain more than two literals. After that, the memory allocator takes care to allocate the clause in a flat representation and not to allocate the field `_pos` when the clause is short (similarly to the representation in IsaSAT). Compared with the representation in IsaSAT, the headers are represented in a more compact fashion (both because not all bits of the LBD score are stored and because there are fewer wasted bits) Especially, the compiler takes care of putting the headers in the least number of bytes.

Another difference is the fact that the pointer to the clause is represented either as a clause or a pointer if the clause was reallocated. This makes it easier to reallocate the clause and update the reasons in the trail. The union makes garbage collection more efficient. In IsaSAT, I found no way to represent that: If I assume that clause indices are represented on by 64-bit numbers, I could do it by encoding it into two 32-bit numbers and include it in the arena. However, I do not see any way to represent this behavior for unbounded integers. So far, I avoided introducing behaviors that differ between the bounded and the unbounded version to be able to compare the performance of both versions.

A.2. Watch Lists and Propagations

Glucose use two different watch lists: One only for binary clauses and one for non-binary clauses. This avoids testing if a clause is binary during the propagation loop, but requires two loops. I thought about doing that too in IsaSAT: At the CDCL_W level, I would still have a single list for watched list, which would get split into two at Heur. However, I have never managed to prove that I can split a single loop into two different loops during a refinement step. I talked to Peter Lammich, but he did not see any simple solution to it either.

In CaDiCaL, an element of a watch list contains an additional redundant information whether the clause is redundant or not:

```
struct Watch {
  Clause * clause;
  signed int blit;
  bool redundant;
  bool binary;
```

```
}
```

Even if 29 bits are still wasted (integers `signed int` are 32 bits, Boolean `bool` only one bit, and pointers `Clause*` 64 bits), whether the clause is redundant can be used during inprocessing: If an initial clause is subsumed by a binary clause, then the latter can be promoted to an initial clause and the former can be removed. All this information can be accessed without cach miss. In IsaSAT the redundant information is not accessed (instead 30 bits are wasted).

A.3. Decision Heuristics

There are several decision heuristics, by order of invention: Glucose (and Minisat) uses VSIDS, CaDiCaL uses VMTF (like IsaSAT), while MapleSAT variants have LRB. The later seems especially efficient for satisfiable instances.

The main reason for not implementing VSIDS or LRB in IsaSAT is that VMTF is much simpler but does not seem to perform worse than VSIDS [14], and especially does not require the use of floating-point arithmetic. The behaviour of floating-point numbers has been formalized in Isabelle by Yu [127], but it has not been used in for refinement and there is currently no setup for code synthesis.

A.4. Clause Simplification

There are several algorithms used to simplify the clause set. Here is a limited list.

Removal of true/false literals If L is true, then $L \vee C$ can be removed from the clause set and $-L$ can be removed from $-L \vee D$.

Variable elimination This technique tries to eliminate variables. This is usually done if the number of clauses is not increased. A special case is *pure literal deletion*: If a literal appears only positively, it can be removed and the clauses it appears in (this corresponds to setting to true).

Variable addition This is the opposite of the previous technique: clauses are simplified by adding new literals.

Subsumption-resolution If $L \vee C$ and $-L \vee D$ is included in the clause set and $C \subseteq D$, then the clause $-L \vee D$ is redundant and can be removed.

Subsumption If $C \subseteq D$, then the clause D is redundant and can be removed.

MiniSAT and Glucose mostly uses techniques as preprocessing: The input problems are simplified initially, then not at all, except for removal of true/false literals (and very recently vivification for Glucose).

CryptoMiniSat, Lingeling, and to a lesser extent CaDiCaL perform several of these techniques as inprocessing. It is difficult to schedule them in a way that does not harm performance, but can be extremely helpful for example on benchmarks for cryptography.

IsaSAT does not perform any simplification. There are two main problems. First, our CDCL calculus relies on the fact than the initial clauses have not been modified. This could be changed, but the termination proof requires that atoms can only be deleted. The second problem is that my invariants are not compatible with removing atoms. For example, if a literal appears a single time in a clause that is removed, then this literal must also be removed from the VMTF heuristic.

A.5. Forget

In IsaSAT, clauses are sorted by LBD and by activity and half of the clauses are deleted, except for initial clauses, binary clauses, clauses of LBD less than three[1], or clauses that have been used for rule Resolve. This is a move-to-front like scheme: used clauses are always kept at least once. The activity is the number of times a clause is used in a conflict. Using the activity for ties seems important on the benchmarks.

A.6. Clause Minimization

The efficiency of the conflict minimization algorithm is important, because this procedure is called extremely often.

Compared with IsaSAT, Glucose includes conflict minimization with binary clauses: by iterating over the watch lists, some literals are removed (this is only tried if the LBD score of the conflict clause is low enough). If the conflict clause is $A \vee B \vee C$, A is the literal of highest level, and the clause $A \vee \neg B$ is a clause of the problem, then the conflict clause can be shortened to $A \vee C$ by resolution. This is a special case of inprocessing, which is very efficient since all necessary information is in the watch lists.

[1]if the LBD calculation is correct, this supersedes the previous point, but I don't prove any correction of the LBD calculation

CaDiCaL does not minimize the conflict clause with binary clauses (this is only done during inprocessing), but the algorithm for clause minimization differs slightly:

- it is based on van Gelder's poison idea [43] to limit the number of recursive attempts. This, however, requires more complicated data structures.
- it contains an idea by Knuth to abort earlier.
- the implementation is recursive and not iterative (and contains a depth check to limit the recursion depth).

Bibliography

[1] Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: C. Boutilier (ed.) IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009, pp. 399–404. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2009). URL http://ijcai.org/Proceedings/09/Papers/074.pdf

[2] Bachmair, L., Ganzinger, H., McAllester, D., Lynch, C.: Resolution theorem proving. In: A. Robinson, A. Voronkov (eds.) Handbook of Automated Reasoning, vol. I, pp. 19–99. Elsevier (2001). doi:10.1016/b978-044450813-3/50004-7

[3] Ballarin, C.: Locales: A module system for mathematical theories. J. Autom. Reasoning **52**(2), 123–153 (2013). doi:10.1007/s10817-013-9284-7

[4] Barbosa, H., Blanchette, J.C., Fleury, M., Fontaine, P.: Scalable fine-grained proofs for formula processing. J. Autom. Reasoning (2019). doi:10.1007/s10817-018-09502-y

[5] Barbosa, H., Blanchette, J.C., Fleury, M., Fontaine, P., Schurr, H.J.: Better SMT proofs for easier reconstruction. In: T.C. Hales, C. Kaliszyk, R. Kumar, S. Schulz, J. Urban (eds.) 4th Conference on Artificial Intelligence and Theorem Proving (AITP 2019) (2019)

[6] Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: G. Gopalakrishnan, S. Qadeer (eds.) Computer Aided Verification—23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, *Lecture Notes in Computer Science*, vol. 6806, pp. 171–177. Springer (2011). doi:10.1007/978-3-642-22110-1_14

[7] Bayardo Jr., R.J., Schrag, R.: Using CSP look-back techniques to solve exceptionally hard SAT instances. In: E.C. Freuder (ed.) Proceedings of the Second International Conference on Principles and Practice of

Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996, *Lecture Notes in Computer Science*, vol. 1118, pp. 46–60. Springer (1996). doi:10.1007/3-540-61551-2_65

[8] Becker, H., Bentkamp, A., Blanchette, J.C., Fleury, M., From, A.H., Jensen, A.B., Lammich, P., Larsen, J.B., Michaelis, J., Nipkow, T., Peltier, N., Popescu, A., Robillard, S., Schlichtkrull, A., Tourret, S., Traytel, D., Villadsen, J., Petar, V.: IsaFoL: Isabelle Formalization of Logic. `https://bitbucket.org/isafol/isafol/`

[9] Benzmüller, C., Sultana, N., Paulson, L.C., Theiß, F.: The higher-order prover LEO-II. J. Autom. Reasoning **55**(4), 389–404 (2015). doi:10.1007/s10817-015-9348-y

[10] Berger, U., Lawrence, A., Nordvall Forsberg, F., Seisenberger, M.: Extracting verified decision procedures: DPLL and resolution. Log. Meth. Comput. Sci. **11**(1) (2015). doi:10.2168/lmcs-11(1:6)2015

[11] Biere, A.: Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016. In: T. Balyo, M.J.H. Heule, M. Järvisalo (eds.) Proc. of SAT Competition 2016–Solver and Benchmark Descriptions, *Department of Computer Science Series of Publications B*, vol. B-2016-1, pp. 44–45. University of Helsinki (2016)

[12] Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT Competition 2017. In: T. Balyo, M.J.H. Heule, M. Järvisalo (eds.) SAT Competition 2017: Solver and Benchmark Descriptions, *Department of Computer Science Series of Publications B*, vol. B-2017-1, pp. 14–15. University of Helsinki (2017)

[13] Biere, A.: Deep bound hardware model checking instances, quadratic propagations benchmarks and reencoded factorization problems. In: T. Balyo, M.J.H. Heule, M. Järvisalo (eds.) SAT Competition 2017: Solver and Benchmark Descriptions, *Department of Computer Science Series of Publications B*, vol. B-2017-1, pp. 37–38. University of Helsinki (2017)

[14] Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: SAT, *Lecture Notes in Computer Science*, vol. 9340, pp. 405–422. Springer (2015). URL `978-3-319-24318-4_29`

[15] Biere, A., Fröhlich, A.: Evaluating CDCL restart schemes. In: M.J.H. Heule, S. Weaver (eds.) Theory and Applications of Satisfiability Testing – SAT 2015—18th International Conference, Austin, TX, USA, September

24-27, 2015, Proceedings, *Lecture Notes in Computer Science*, vol. 9340, pp. 405–422. EasyChair (2015). doi:10.29007/89dw

[16] Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, *Frontiers in Artificial Intelligence and Applications*, vol. 185. IOS Press (2009)

[17] Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic proof and disproof in Isabelle/HOL. In: C. Tinelli, V. Sofronie-Stokkermans (eds.) Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings, *Lecture Notes in Computer Science*, vol. 6989, pp. 12–27. Springer (2011). doi:10.1007/978-3-642-24364-6_2

[18] Blanchette, J.C., Böhme, S., Fleury, M., Smolka, S.J., Steckermeier, A.: Semi-intelligible isar proofs from machine-generated proofs. J. Autom. Reasoning **56**(2), 155–200 (2015). doi:10.1007/s10817-015-9335-3

[19] Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending sledgehammer with SMT solvers. J. Autom. Reasoning **51**(1), 109–128 (2013). doi:10.1007/s10817-013-9278-5

[20] Blanchette, J.C., Fleury, M., Lammich, P., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. J. Autom. Reasoning **61**(1-4), 333–365 (2018). doi:10.1007/s10817-018-9455-7

[21] Blanchette, J.C., Fleury, M., Traytel, D.: Nested multisets, hereditary multisets, and syntactic ordinals in Isabelle/HOL. In: D. Miller (ed.) 2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK, *LIPIcs*, vol. 84, pp. 11:1–11:18. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik (2017). doi:10.4230/LIPIcs.FSCD.2017.11

[22] Blanchette, J.C., Fleury, M., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. In: N. Olivetti, A. Tiwari (eds.) Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, *Lecture Notes in Computer Science*, vol. 9706, pp. 25–44. International Joint Conferences on Artificial Intelligence Organization (2017). doi:10.24963/ijcai.2017/667

[23] Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: M. Kaufmann, L.C. Paulson (eds.) Interactive Theorem Proving, First

International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings, *Lecture Notes in Computer Science*, vol. 6172, pp. 179–194. Springer (2010). doi:10.1007/978-3-642-14052-5_14

[24] Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: `veriT`: An open, trustable and efficient SMT-solver. In: R.A. Schmidt (ed.) Automated Deduction—CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings, *Lecture Notes in Computer Science*, vol. 5663, pp. 151–156. Springer (2009). doi:10.1007/978-3-642-02959-2_12

[25] Bromberger, M., Fleury, M., Schwarz, S., Weidenbach, C.: SPASS-SATT a CDCL(LA) solver. In: P. Fontaine (ed.) Accepted at Automated Deduction—CADE 27—27th International Conference on Automated Deduction, Natal, Brazil (2019)

[26] Brown, C.E.: Satallax: An automatic higher-order prover. In: B. Gramlich, D. Miller, U. Sattler (eds.) Automated Reasoning—6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings, *Lecture Notes in Computer Science*, vol. 7364, pp. 111–117. Springer (2012). doi:10.1007/978-3-642-31365-3_11

[27] Bryant, R.E., Beatty, D., Brace, K., Cho, K., Sheffler, T.: COSMOS: A compiled simulator for MOS circuits. In: A. O'Neill, D. Thomas (eds.) 24th ACM/IEEE conference proceedings on Design automation conference—DAC '87, pp. 9–16. ACM Press (1987). doi:10.1145/37888.37890

[28] Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: O.A. Mohamed, C.A. Muñoz, S. Tahar (eds.) Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings, *Lecture Notes in Computer Science*, vol. 5170, pp. 134–149. Springer (2008). doi:10.1007/978-3-540-71067-7_14

[29] Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Proceeding of the 16th ACM SIGPLAN international conference on Functional programming—ICFP '11, pp. 418–430. ACM Press (2011). doi:10.1145/2034773.2034828

[30] Chu, G., Harwood, A., Stuckey, P.J.: Cache conscious data structures for Boolean satisfiability solvers. JSAT **6**(1-3), 99–120 (2009). URL `https://satassociation.org/jsat/index.php/jsat/article/view/71`

[31] Church, A.: A formulation of the simple theory of types. J. symb. log. **5**(02), 56–68 (1940). doi:10.2307/2266170

[32] Clochard, M., Filliâtre, J., Paskevich, A.: How to avoid proving the absence of integer overflows. In: Verified Software: Theories, Tools, and Experiments—7th International Conference, VSTTE 2015, San Francisco, CA, USA, July 18-19, 2015. Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 9593, pp. 94–109. Springer (2015). doi:10.1007/978-3-319-29613-5_6

[33] Cruz-Filipe, L., Heule, M.J.H., Hunt Jr., W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: L.M. de Moura (ed.) Automated Deduction—CADE 26—26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings, *Lecture Notes in Computer Science*, vol. 10395, pp. 220–236. Springer (2017). doi:10.1007/978-3-319-63046-5_14

[34] Cruz-Filipe, L., Marques-Silva, J.P., Schneider-Kamp, P.: Efficient certified resolution proof checking. In: A. Legay, T. Margaria (eds.) Tools and Algorithms for the Construction and Analysis of Systems—23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I, *Lecture Notes in Computer Science*, vol. 10205, pp. 118–135. Springer (2017). doi:10.1007/978-3-662-54577-5_7

[35] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962). doi:10.1145/368273.368557

[36] Delaware, B., Pit-Claudel, C., Gross, J., Chlipala, A.: Fiat: Deductive synthesis of abstract data types in a proof assistant. In: S.K. Rajamani, D. Walker (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '15, pp. 689–700. ACM Press (2015). doi:10.1145/2676726.2677006

[37] Eén, N., Sörensson, N.: An extensible SAT-solver. In: E. Giunchiglia, A. Tacchella (eds.) Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers, *Lecture Notes in Computer Science*, vol. 2919, pp. 502–518. Springer (2003). doi:10.1007/978-3-540-24605-3_37

[38] Filliâtre, J., Paskevich, A.: Why3—where programs meet provers. In: Programming Languages and Systems—22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, *Lecture Notes in Computer Science*, vol. 7792, pp. 125–128. Springer (2013). doi:10.1007/978-3-642-37036-6_8

[39] Fleury, M.: Formalisation of ground inference systems in a proof assistant. M.Sc. thesis, École normale supérieure de Rennes (2015). https://www.mpi-inf.mpg.de/fileadmin/inf/rg1/Documents/fleury_master_thesis.pdf

[40] Fleury, M.: Optimizing a verified SAT solver. In: J.M. Badger, K.Y. Rozier (eds.) NASA Formal Methods, pp. 148–165. Springer International Publishing, Cham (2019). doi:978-3-030-20652-9_10

[41] Fleury, M., Blanchette, J.C.: Formalization of Weidenbach's *Automated Reasoning—The Art of Generic Problem Solving* (2018). https://bitbucket.org/isafol/isafol/src/master/Weidenbach_Book/README.md, Formal proof development

[42] Fleury, M., Blanchette, J.C., Lammich, P.: 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, pp. 158–171. ACM Press (2018). doi:10.1145/3167080

[43] Gelder, A.V.: Improved conflict-clause minimization leads to improved propositional proof traces. In: O. Kullmann (ed.) Theory and Applications of Satisfiability Testing—SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30–July 3, 2009. Proceedings, *Lecture Notes in Computer Science*, vol. 5584, pp. 141–146. Springer (2009). doi:10.1007/978-3-642-02777-2_15

[44] Gent, I.P.: Optimal implementation of watched literals and more general techniques. jair **48**, 231–252 (2013). doi:10.1613/jair.4016

[45] Giunchiglia, E., Maratea, M.: Solving optimization problems with DLL. In: G. Brewka, S. Coradeschi, A. Perini, P. Traverso (eds.) ECAI 2006, 17th European Conference on Artificial Intelligence, August 29–September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings, *Frontiers in Artificial Intelligence and Applications*, vol. 141, pp. 377–381. IOS Press (2006)

[46] Gordon, M.J., Milner, A.J., Wadsworth, C.P.: Edinburgh LCF, *Lecture Notes in Computer Science*, vol. 78. Springer Berlin Heidelberg (1979). doi:10.1007/3-540-09724-4

[47] Gries, D., Volpano, D.M.: The Transform—A new language construct. Structured Programming **11**(1), 1–10 (1990)

[48] Haftmann, F.: Draft toy for proof exploration (2013). URL www.mail-archive.com/isabelle-dev@mailbroy.informatik.tu-muenchen.de/msg04443.html

[49] Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: M. Blume, N. Kobayashi, G. Vidal (eds.) Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings, *Lecture Notes in Computer Science*, vol. 6009, pp. 103–117. Springer (2010). doi:10.1007/978-3-642-12251-4_9

[50] Harrison, J.: Formalizing basic first order model theory. In: J. Grundy, M.C. Newey (eds.) Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'98, Canberra, Australia, September 27—October 1, 1998, Proceedings, *Lecture Notes in Computer Science*, vol. 1479, pp. 153–170. Springer (1998). doi:10.1007/BFb0055135

[51] Heule, M.J., Kullmann, O., Marek, V.W.: Solving very hard problems: Cube-and-conquer, a hybrid SAT solving method. In: N. Creignou, D.L. Berre (eds.) Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, *Lecture Notes in Computer Science*, vol. 9710, pp. 228–245. International Joint Conferences on Artificial Intelligence Organization (2017). doi:10.24963/ijcai.2017/683

[52] Heule, M.J.H.: microsat (2014). URL https://github.com/marijnheule/microsat. Open source development; last accessed 2019-03-11

[53] Heule, M.J.H.: Schur number five. In: S.A. McIlraith, K.Q. Weinberger (eds.) Proceedings of AAAI 18, pp. 6598–6606. AAAI Press (2018). URL https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16952

[54] Heule, M.J.H., Hunt, W.A., Wetzler, N.: Trimming while checking clausal proofs. In: M. Ayala-Rincón, C.A. Muñoz (eds.) 2013 Formal Methods in Computer-Aided Design (FMCAD), *Lecture*

Notes in Computer Science, vol. 10499, pp. 269–284. IEEE (2013). doi:10.1109/fmcad.2013.6679408

[55] Heule, M.J.H., Hunt, W.A., Wetzler, N.: Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. Softw. Test. Verif. Reliab. **24**(8), 593–607 (2014). doi:10.1002/stvr.1549

[56] Hupel, L.: Verified code generation from Isabelle/HOL. Ph.D. thesis, TUM (2019). URL `https://lars.hupel.info/pub/phd-thesis_hupel.pdf`. Submitted

[57] Hupel, L., Nipkow, T.: A verified compiler from Isabelle/HOL to CakeML. In: A. Ahmed (ed.) Programming Languages and Systems—27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, *Lecture Notes in Computer Science*, vol. 10801, pp. 999–1026. Springer (2018). doi:10.1007/978-3-319-89884-1_35

[58] Kammüller, F., Wenzel, M., Paulson, L.C.: Locales—A sectioning concept for Isabelle. In: Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, L. Théry (eds.) Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings, *Lecture Notes in Computer Science*, vol. 1690, pp. 149–166. Springer (1999). doi:10.1007/3-540-48256-3_11

[59] Karp, R.M.: Reducibility among combinatorial problems. In: R.E. Miller, J.W. Thatcher (eds.) Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA, The IBM Research Symposia Series, pp. 85–103. Plenum Press, New York (1972). URL `http://www.cs.berkeley.edu/\%7Eluca/cs172/karp.pdf`

[60] Klein, G., Norrish, M., Sewell, T., Tuch, H., Winwood, S., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R.: seL4: Formal verification of an operating-system kernel. Commun. ACM **53**(6), 107 (2010). doi:10.1145/1743546.1743574

[61] Krauss, A.: Partial recursive functions in higher-order logic. In: U. Furbach, N. Shankar (eds.) Automated Reasoning, Third International Joint

Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings, *Lecture Notes in Computer Science*, vol. 4130, pp. 589–603. Springer (2006). doi:10.1007/11814771_48

[62] Krauss, A.: Partial recursive functions in higher-order logic. In: U. Furbach, N. Shankar (eds.) Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings, *Lecture Notes in Computer Science*, vol. 4130, pp. 589–603. Springer (2006). doi:10.1007/11814771_48

[63] Krstić, S., Goel, A.: Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In: B. Konev, F. Wolter (eds.) Frontiers of Combining Systems, 6th International Symposium, FroCoS 2007, Liverpool, UK, September 10-12, 2007, Proceedings, *Lecture Notes in Computer Science*, vol. 4720, pp. 1–27. Springer (2007). doi:10.1007/978-3-540-74621-8_1

[64] Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ml. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages—POPL '14, *Lecture Notes in Computer Science*, vol. 10900, pp. 646–662. ACM Press (2014). doi:10.1145/2535838.2535841

[65] Lammich, P.: Refinement to Imperative/HOL. In: C. Urban, X. Zhang (eds.) Interactive Theorem Proving—6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings, *Lecture Notes in Computer Science*, vol. 9236, pp. 253–269. Springer (2015). doi:10.1007/978-3-319-22102-1_17

[66] Lammich, P.: Refinement based verification of imperative data structures. In: S. Blazy, C. Paulin-Mohring, D. Pichardie (eds.) Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs—CPP 2016, *Lecture Notes in Computer Science*, vol. 7998, pp. 84–99. ACM Press (2016). doi:10.1145/2854065.2854067

[67] Lammich, P.: Refinement based verification of imperative data structures. In: Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs—CPP 2016, *Lecture Notes in Computer Science*, vol. 7406, pp. 166–182. ACM Press (2016). doi:10.1145/2854065.2854067

[68] Lammich, P.: Refinement based verification of imperative data structures. In: J. Avigad, A. Chlipala (eds.) Proceedings of the 5th ACM

SIGPLAN Conference on Certified Programs and Proofs—CPP 2016, pp. 27–36. ACM Press (2016). doi:10.1145/2854065.2854067

[69] Lammich, P.: Efficient verified (UN)SAT certificate checking. In: L. de Moura (ed.) Automated Deduction—CADE 26—26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings, *Lecture Notes in Computer Science*, vol. 10395, pp. 237–254. Springer (2017). doi:10.1007/978-3-319-63046-5_15

[70] Lammich, P.: The GRAT tool chain—efficient (UN)SAT certificate checking with formal correctness guarantees. In: S. Gaspers, T. Walsh (eds.) Theory and Applications of Satisfiability Testing – SAT 2017—20th International Conference, Melbourne, VIC, Australia, August 28–September 1, 2017, Proceedings, *Lecture Notes in Computer Science*, vol. 10491, pp. 457–463. Springer (2017). doi:10.1007/978-3-319-66263-3_29

[71] Larrosa, J., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: A framework for certified boolean branch-and-bound optimization. J. Autom. Reasoning **46**(1), 81–102 (2010). doi:10.1007/s10817-010-9176-z

[72] Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004., pp. 75–88. IEEE (2004). doi:10.1109/cgo.2004.1281665

[73] Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Logic for Programming, Artificial Intelligence, and Reasoning—16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 6355, pp. 348–370. Springer (2010). doi:10.1007/978-3-642-17511-4_20

[74] Lescuyer, S.: Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. Theses, Université Paris Sud—Paris XI (2011). URL `https://tel.archives-ouvertes.fr/tel-00713668`

[75] Li, C., Xiao, F., Luo, M., Manyà, F., Lü, Z., Li, Y.: Clause vivification by unit propagation in CDCL SAT solvers. CoRR **abs/1807.11061** (2018). URL `http://arxiv.org/abs/1807.11061`

[76] Li, C.M., Manyà, F.: MaxSAT, hard and soft constraints. In: Handbook of Satisfiability, *Frontiers in Artificial Intelligence and Applications*, vol. 185, pp. 613–631. IOS Press (2009). doi:10.3233/978-1-58603-929-5-613

[77] Liberatore, P.: Algorithms and experiments on finding minimal models. Tech. Rep. 09-99, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza" (1999). URL http://www.dis.uniroma1.it/%7eliberato/papers/libe-99.ps.gz

[78] Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Information Processing Letters **47**(4), 173–180 (1993). doi:10.1016/0020-0190(93)90029-9

[79] Luo, M., Li, C.M., Xiao, F., Manyà, F., Lü, Z.: An effective learnt clause minimization approach for CDCL SAT solvers. In: C. Sierra (ed.) Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, pp. 703–711. International Joint Conferences on Artificial Intelligence Organization (2017). doi:10.24963/ijcai.2017/98

[80] Mahajan, Y.S., Fu, Z., Malik, S.: zChaff2004: An efficient SAT solver. In: H.H. Hoos, D.G. Mitchell (eds.) Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 3542, pp. 360–375. Springer (2004). doi:10.1007/11527695_27

[81] Manquinho, V., Marques-Silva, J.P.: Satisfiability-based algorithms for pseudo-boolean optimization using gomory cuts and search restarts. In: 17th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'05), pp. 150–155. IEEE (2005). doi:10.1109/ictai.2005.113

[82] Margetson, J., Ridge, T.: Completeness theorem. Archive of Formal Proofs (2004). http://isa-afp.org/entries/Completeness.shtml, Formal proof development

[83] Marić, F.: Formal verification of modern SAT solvers. Archive of Formal Proofs (2008). http://isa-afp.org/entries/SATSolverVerification.shtml, Formal proof development

[84] Marić, F.: Formalization and implementation of modern SAT solvers. J. Autom. Reasoning **43**(1), 81–119 (2009). doi:10.1007/s10817-009-9127-8

[85] Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theoretical Computer Science **411**(50), 4333–4356 (2010). doi:10.1016/j.tcs.2010.09.014

[86] Marić, F., Janičić, P.: Formalization of abstract state transition systems for SAT. Log.Meth.Comput.Sci. **7**(3) (2011). doi:10.2168/lmcs-7(3:19)2011

[87] Marques Silva, J.P., Sakallah, K.: GRASP—a new search algorithm for satisfiability. In: Proceedings of International Conference on Computer Aided Design, pp. 220–227. IEEE Comput. Soc. Press (1996). doi:10.1109/iccad.1996.569607

[88] Matichuk, D., Murray, T., Wenzel, M.: Eisbach: A proof method language for Isabelle. J. Autom. Reasoning **56**(3), 261–282 (2016). doi:10.1007/s10817-015-9360-2

[89] Matuszewski, R., Rudnicki, P.: Mizar: The first 30 years. Mechanized Mathematics and Its Applications **4**(1), 3–24 (2005). doi:10.1.1.151.6028

[90] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient sat solver. In: Proceedings of the 38th conference on Design automation—DAC '01, pp. 530–535. ACM Press (2001). doi:10.1145/378239.379017

[91] de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: C.R. Ramakrishnan, J. Rehof (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, *Lecture Notes in Computer Science*, vol. 4963, pp. 337–340. Springer (2008). doi:10.1007/978-3-540-78800-3_24

[92] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). JACM **53**(6), 937–977 (2006). doi:10.1145/1217856.1217859

[93] Nipkow, T., Klein, G.: Concrete Semantics. Springer International Publishing (2014). doi:10.1007/978-3-319-10542-0

[94] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, *Lecture Notes in Computer Science*, vol. 2283. Springer Berlin Heidelberg (2002). doi:10.1007/3-540-45949-9

[95] Noel, P.A.J.: Experimenting with Isabelle in ZF set theory. J. Autom. Reasoning **10**(1), 15–58 (1993). doi:10.1007/bf00881863

[96] Noschinski, L., Rizkallah, C., Mehlhorn, K.: Verification of certifying computations through AutoCorres and Simpl. In: J.M. Badger, K.Y. Rozier (eds.) NASA Formal Methods—6th International Symposium, NFM 2014, Houston, TX, USA, April 29–May 1, 2014. Proceedings, *Lecture Notes in Computer Science*, vol. 8430, pp. 46–61. Springer (2014). doi:10.1007/978-3-319-06200-6_4

[97] Oe, D., Stump, A., Oliver, C., Clancy, K.: versat: A verified modern SAT solver. In: Lecture Notes in Computer Science, *Lecture Notes in Computer Science*, vol. 7148, pp. 363–378. Springer Berlin Heidelberg (2012). doi:10.1007/978-3-642-27940-9_24

[98] Palopoli, L., Pirri, F., Pizzuti, C.: Algorithms for selective enumeration of prime implicants. Artif. Intell. **111**(1-2), 41–72 (1999). doi:10.1016/s0004-3702(99)00035-1

[99] Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a Practical link between automatic and interactive theorem provers. In: Proceedings of the 2nd Workshop on Practical Aspects of Automated Reasoning, PAAR-2010, Edinburgh, Scotland, UK, July 14, 2010, *EPiC Series in Computing*, vol. 9, pp. 1–10. EasyChair (2010). URL `http://www.easychair.org/publications/paper/52675`

[100] Piette, C., Hamadi, Y., Sais, L.: Vivifying propositional clausal formulae. In: M. Ghallab, C.D. Spyropoulos, N. Fakotakis, N.M. Avouris (eds.) ECAI 2008—18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings, *Frontiers in Artificial Intelligence and Applications*, vol. 178, pp. 525–529. IOS Press (2008). doi:10.3233/978-1-58603-891-5-525

[101] Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: J.P. Marques-Silva, K.A. Sakallah (eds.) Theory and Applications of Satisfiability Testing – SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings, *Lecture Notes in Computer Science*, vol. 4501, pp. 294–299. Springer (2007). doi:10.1007/978-3-540-72788-0_28

[102] Ramos, A., van der Tak, P., Heule, M.J.H.: Between restarts and backjumps. In: K.A. Sakallah, L. Simon (eds.) Theory and Applications of Satisfiability Testing – SAT 2011—14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings, *Lecture*

Notes in Computer Science, vol. 6695, pp. 216–229. Springer (2011). doi:10.1007/978-3-642-21581-0_18

[103] Reynolds, A., Tinelli, C., de Moura, L.: Finding conflicting instances of quantified formulas in SMT. In: 2014 Formal Methods in Computer-Aided Design (FMCAD), pp. 195–202. IEEE (2014). doi:10.1109/fmcad.2014.6987613

[104] Riazanov, A., Voronkov, A.: The design and implementation of VAMPIRE. AI Commun. **15**(2-3), 91–110 (2002). URL `http://content.iospress.com/articles/ai-communications/aic259`

[105] Roe, K., Smith, S.F.: Using the coq theorem prover to verify complex data structure invariants. Ph.D. thesis, Johns Hopkins University (2018). URL `https://www.cs.jhu.edu/~roe/root.pdf`

[106] Ryan, L.: Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University (2004). URL `https://www.cs.sfu.ca/~mitchell/papers/ryan-thesis.ps`

[107] Schlichtkrull, A.: Formalization of the resolution calculus for first-order logic. In: J.C. Blanchette, S. Merz (eds.) Interactive Theorem Proving—7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings, *Lecture Notes in Computer Science*, vol. 9807, pp. 341–357. Springer (2016). doi:10.1007/978-3-319-43144-4_21

[108] Schulz, S.: E—a brainiac theorem prover. AI Commun. **15**(2-3), 111–126 (2002). URL `http://content.iospress.com/articles/ai-communications/aic260`

[109] Sebastiani, R., Giorgini, P., Mylopoulos, J.: Simple and minimum-cost satisfiability for goal models. In: A. Persson, J. Stirna (eds.) Advanced Information Systems Engineering, 16th International Conference, CAiSE 2004, Riga, Latvia, June 7-11, 2004, Proceedings, *Lecture Notes in Computer Science*, vol. 3084, pp. 20–35. Springer (2004). doi:10.1007/978-3-540-25975-6_4

[110] Shankar, N.: Metamathematics, Machines, and Gödels's Proof, *Cambridge Tracts in Theoretical Computer Science*, vol. 38. Cambridge University Press (1994). doi:10.1017/cbo9780511569883

[111] Shankar, N., Vaucher, M.: The mechanical verification of a DPLL-based satisfiability solver. Electron. Notes Theor. Comput. Sci. **269**, 3–17 (2011). doi:10.1016/j.entcs.2011.03.002

[112] Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: O. Kullmann (ed.) Lecture Notes in Computer Science, *Lecture Notes in Computer Science*, vol. 5584, pp. 244–257. Springer Berlin Heidelberg (2009). doi:10.1007/978-3-642-02777-2_24

[113] Sörensson, N., Biere, A.: Minimizing learned clauses. In: O. Kullmann (ed.) Theory and Applications of Satisfiability Testing – SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30—July 3, 2009. Proceedings, *Lecture Notes in Computer Science*, vol. 5584, pp. 237–243. Springer (2009). doi:10.1007/978-3-642-02777-2_23

[114] Sternagel, C., Thiemann, R.: An Isabelle/HOL formalization of rewriting for certified termination analysis. `http://cl-informatik.uibk.ac.at/software/ceta/`

[115] Stump, A., Deters, M., Petcher, A., Schiller, T., Simpson, T.: Verified programming in guru. In: T. Altenkirch, T.D. Millstein (eds.) Proceedings of the 3rd workshop on Programming languages meets program verification—PLPV '09, pp. 49–58. ACM Press (2008). doi:10.1145/1481848.1481856

[116] Thiemann, R.: Extending a verified simplex algorithm. In: G. Barthe, K. Korovin, S. Schulz, M. Suda, G. Sutcliffe, M. Veanes (eds.) LPAR-22 Workshop and Short Paper Proceedings, *Kalpa Publications in Computing*, vol. 9, pp. 37–48. EasyChair (2018). doi:10.29007/5vlq

[117] Voronkov, A.: AVATAR: The architecture for first-order theorem provers. In: A. Biere, R. Bloem (eds.) Computer Aided Verification—26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, *Lecture Notes in Computer Science*, vol. 8559, pp. 696–710. Springer (2014). doi:10.1007/978-3-319-08867-9_46

[118] Weeks, S.: Whole-program compilation in MLton. In: Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006, p. 1. ACM Press (2006). doi:10.1145/1159876.1159877

[119] Weidenbach, C.: Automated reasoning building blocks. In: Correct System Design—Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings, *Lecture Notes in Computer Science*, vol. 9360, pp. 172–188. Springer (2015). doi:10.1007/978-3-319-23506-6_12

[120] Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: R.A. Schmidt (ed.) Automated Deduction—CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings, *Lecture Notes in Computer Science*, vol. 5663, pp. 140–145. Springer (2009). doi:10.1007/978-3-642-02959-2_10

[121] Wenzel, M.: Isabelle/Isar—A generic framework for human-readable proof documents. In: R. Matuszewski, A. Zalewska (eds.) From Insight to Proof: Festschrift in Honour of Andrzej Trybulec, *Studies in Logic, Grammar, and Rhetoric*, vol. 10(23). University of Białystok (2007)

[122] Wenzel, M.: System description: Isabelle/jEdit in 2014. In: C. Benzmüller, B.W. Paleo (eds.) Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, UITP 2014, Vienna, Austria, 17th July 2014., *EPTCS*, vol. 167, pp. 84–94 (2014). doi:10.4204/EPTCS.167.10

[123] Wenzel, M.: Further scaling of Isabelle technology. In: T. Nipkow, L.C. Paulson, M. Wenzel (eds.) Isabelle Workshop 2018 (2018)

[124] Wetzler, N., Heule, M.J.H., Hunt Jr., W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: C. Sinz, U. Egly (eds.) Theory and Applications of Satisfiability Testing – SAT 2014—17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings, *Lecture Notes in Computer Science*, vol. 8561, pp. 422–429. Springer (2014). doi:10.1007/978-3-319-09284-3_31

[125] Wirth, N.: Program development by stepwise refinement. Commun. ACM **14**(4), 221–227 (1971). doi:10.1145/362575.362577

[126] Woodcock, J., Banach, R.: The verification grand challenge. J. Univers. Comput. Sci. **13**(5), 661–668 (2007). doi:10.3217/jucs-013-05-0661

[127] Yu, L.: A formal model of IEEE floating point arithmetic. Archive of Formal Proofs (2013). `http://isa-afp.org/entries/IEEE_Floating_Point.html`, Formal proof development

Index

blocking literal, 92

CDCL_NOT, 21
CDCL_W, 25
CDCL_W+stgy+incr, 33
CDCL_W+stgy, 27
CDCL_NOT_merge, 22
CDCL_W_merge, 31
conflict minimization, 34

DPLL_NOT, 20
DPLL_NOT+BJ, 17
DPLL_W, 24

`explore`, 88

Imperative HOL, 59
Isabelle Refinement Framework, 56

literal, 16

$\mathsf{PCUI}_{\mathsf{algo}}$, 66
$\mathsf{PCUI}_{\mathsf{list}}$, 68
position saving, 94

reasonable strategy, 27

Sepref, 59

trail reuse, 97
TWL, 64

variable move to front, 73
VMTF, 73

watch lists, 69