J Autom Reasoning (2018) 61:333–365
https://doi.org/10.1007/s10817-018-9455-7

A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality

Jasmin Christian Blanchette[1,2] · **Mathias Fleury**[2,3] · **Peter Lammich**[4] · **Christoph Weidenbach**[2]

Received: 31 March 2017 / Accepted: 24 January 2018 / Published online: 12 March 2018

Abstract We developed a formal framework for conflict-driven clause learning (CDCL) using the Isabelle/HOL proof assistant. Through a chain of refinements, an abstract CDCL calculus is connected first to a more concrete calculus, then to a SAT solver expressed in a functional programming language, and finally to a SAT solver in an imperative language, with total correctness guarantees. The framework offers a convenient way to prove metatheorems and experiment with variants, including the Davis–Putnam–Logemann–Loveland (DPLL) calculus. The imperative program relies on the two-watched-literal data structure and other optimizations found in modern solvers. We used Isabelle's Refinement Framework to automate the most tedious refinement steps. The most noteworthy aspects of our work are the inclusion of rules for forget, restart, and incremental solving and the application of stepwise refinement.

Keywords SAT solvers · CDCL · DPLL · Proof assistants · Isabelle/HOL

✉ Mathias Fleury
mathias.fleury@mpi-inf.mpg.de; s8mafleu@stud.uni-saarland.de

Jasmin Christian Blanchette
j.c.blanchette@vu.nl; jasmin.blanchette@mpi-inf.mpg.de

Peter Lammich
lammich@in.tum.de

Christoph Weidenbach
weidenbach@mpi-inf.mpg.de

1 Section of Theoretical Computer Science, Department of Computer Science, Vrije Universiteit Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

2 Max-Planck-Institut für Informatik, Saarland Informatics Campus E1 4, 66123 Saarbrücken, Germany

3 Saarbrücken Graduate School of Computer Science, Saarland Informatics Campus E1 3, 66123 Saarbrücken, Germany

4 Institut für Informatik, Technische Universität München, Boltzmannstraße 3, Garching, Germany

1 Introduction

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, we started an effort, called IsaFoL (Isabelle Formalization of Logic) [4], that aims at developing libraries and a methodology for formalizing modern research in the field, using the Isabelle/HOL proof assistant [45,46]. Our initial emphasis is on established results about propositional and first-order logic. In particular, we are formalizing large parts of Weidenbach's forthcoming textbook, tentatively called *Automated Reasoning—The Art of Generic Problem Solving*. Our inspiration for formalizing logic is the IsaFoR (Isabelle Formalization of Rewriting) project [55], which focuses on term rewriting.

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.

This article presents our formalization of CDCL (conflict-driven clause learning) based on *Automated Reasoning*, derived as a refinement of Nieuwenhuis, Oliveras, and Tinelli's abstract presentation of CDCL [43]. It is the algorithm implemented in modern propositional satisfiability (SAT) solvers. We start with a family of formalized abstract DPLL (Davis–Putnam–Logemann–Loveland) [17] and CDCL [3,6,40,42] transition systems from Nieuwenhuis et al. (Sect. 3). Some of the calculi include rules for learning and forgetting clauses and for restarting the search. All calculi are proved sound and complete, as well as terminating under a reasonable strategy.

The abstract CDCL calculus is refined into the more concrete calculus presented in *Automated Reasoning* and recently published [57] (Sect. 4). The latter specifies a criterion for learning clauses representing first unique implication points [6, 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. The calculus also supports incremental solving.

The concrete calculus is refined further to obtain a verified, but very naive, functional program extracted using Isabelle's code generator (Sect. 5). The final refinement step derives an imperative SAT solver implementation with efficient data structures, including the well-known two-watched-literal optimization (Sect. 6).

Our work is related to other verifications of SAT solvers, which largely aimed at increasing their trustworthiness (Sect. 7). This goal has lost some of its significance with the emergence of formats for certificates that are easy to generate, even in highly optimized solvers, and that can be processed efficiently by verified checkers [16,33]. In contrast, our focus is on formalizing the metatheory of CDCL, with the following objectives:

- Develop a basic library of formalized results and a methodology aimed at researchers who want to experiment with calculi.
- Study and connect the members of the CDCL family, including newer extensions.
- Check the proofs in *Automated Reasoning* and provide a formal companion to the book.
- Assess the suitability of Isabelle/HOL for formalizing logical calculi.

Compared with the other verified SAT solvers, the most noteworthy features of our framework are the inclusion of rules for forget, restart, and incremental solving and the application of

stepwise refinement [59] to transfer results. The framework is available as part of the IsaFoL repository [20].

Any formalization effort is a case study in the use of a proof assistant. We depended heavily on the following features of Isabelle:

- *Isar* [58] is a textual proof format inspired by the pioneering Mizar system [41]. It makes it possible to write structured, readable proofs—a requisite for any formalization that aims at clarifying an informal proof.
- *Sledgehammer* [7,48] integrates superposition provers and SMT (satisfiability modulo theories) solvers in Isabelle to discharge proof obligations. The SMT solvers, and one of the superposition provers [56], are built around a SAT solver, resulting in a situation where SAT solvers are employed to prove their own metatheory.
- *Locales* [2,25] 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.
- The *Refinement Framework* [30] can be used to express refinements from abstract data structures and algorithms to concrete, optimized implementations. This allows us to reason about simple algebraic objects and yet obtain efficient programs. The *Sepref* tool [31] builds on the Refinement Framework to derive an imperative program, which can be extracted to Standard ML and other programming languages. For example, Isabelle's algebraic lists can be refined to mutable arrays in ML.

An earlier version of this work was presented at IJCAR 2016 [11]. This article extends the conference paper with a description of the refinement to an imperative implementation (Sects. 2.4 and 6) and of the formalization of Weidenbach's DPLL calculus (Sect. 4.1). To make the paper more accessible, we expanded the background material about Sledgehammer (Sect. 2.1) and Isar (Sect. 2.2).

2 Isabelle/HOL

Isabelle [45,46] is a generic proof assistant that supports several object logics. The metalogic is an intuitionistic fragment of higher-order logic (HOL) [15]. 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.

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 Booleans 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 is important operationally but not semantically.

Isabelle adheres to the tradition that started in the 1970s by the LCF system [22]: 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 us 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 [58], Isabelle's input language. Isar proofs are expressed either as a sequence of tactics that manipulate the proof state directly or in a declarative, natural-deduction format inspired by Mizar. Our formalization almost exclusively employs the more readable declarative style.

2.1 Sledgehammer

The Sledgehammer subsystem [7,48] integrates automatic theorem provers in Isabelle/HOL, including CVC4, E, LEO-II, Satallax, SPASS, Vampire, veriT, and Z3. 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 we invoke it dozens of times a day (according to the log files it produces). For example, while formalizing some results that depend on multisets, we found ourselves 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, so we invoked Sledgehammer. Within 30 s, the tool came back with a brief proof text invoking a suitable tactic with a list of ten lemmas from the library, which we could insert into our 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.

2.2 Isar

Without Sledgehammer, proving the above property could easily have taken 5–15 min. 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 count (A ∪ B) a = max (count A a) (count B a) by auto
    moreover have count (A ∩ B) a = min (count A a) (count B a) by auto
    ultimately show count (A ⊎ B) a = count (A ∪ B ⊎ A ∩ B) a by auto
  qed
  ultimately show |A| + |B| = |A ∪ B| + |A ∩ 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 $\cup$).

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. The advantage of Isar proofs over one-line *metis* proofs is that we can follow and understand the steps. However, for lemmas about multisets and other background theories, we are usually content if we can get a proof automatic and carry on with formalizing the more interesting foreground theory.

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 :: τ 'a
  assumes A 'a,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 $A\,'a,\mathsf{c}$ :: *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 ⊆ X t
```

emits the proof obligation $A\,\upsilon,t$, where υ and t :: τ_υ 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 υ and t :: τ_υ.

2.4 Refinement Framework

The Refinement Framework [30] provides definitions, lemmas, and tools that assist in the verification of functional and imperative programs via stepwise refinement [59]. 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*:

datatype *'a nres* = FAIL | RES (*'a set*)

The Isabelle syntax is similar to that of Standard ML and other typed functional programming languages: The type is freely generated by its two constructors, FAIL :: *'a nres* and RES :: *'a set* ⇒ *'a nres*. The set X in RES X specifies the possible values that can be returned. The return statement is defined as a constant RETURN x = RES $\{x\}$ and specifies a single value, whereas 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):

definition sub1_spec :: *nat list* ⇒ *nat list nres* **where**
sub1_spec xs = RETURN (map ($\lambda x.\ x - 1$) xs)

Program refinement uses the same source and target language. The refinement relation $\leq$ is defined by RES $X \leq$ RES $Y \leftrightarrow X \subseteq Y$ and $r \leq$ FAIL for all r. For example, the concrete program RETURN 2 refines ($\leq$) the abstract program RES $\{n \mid n > 0\}$, meaning that all concrete behaviors are possible in the abstract version. The bottom element RES $\{\}$ is an unrefinable program; 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. For example, we can refine the previous specification to a program that uses a 'while' loop:

```
definition sub1_imp :: nat list ⇒ nat list nres where
  sub1_imp xs = do {
    (i, zs) ← WHILET (λ(i, ys). i < |ys|)
      (λ(i, ys). do {
        ASSERT (i < |ys|);
        let zs = list_update ys i ((ys ! i) − 1);
        RETURN (i + 1, zs)
      })
      (0, xs);
    RETURN zs
  }
```

The program relies on the following constructs:

- The 'do' construct is a convenient Haskell-inspired syntax for expressing monadic computations (here, on the nondeterminism monad).
- The WHILE_T combinator takes a condition, a loop body, and a start value. In our example, the loop's state is a pair of the form (i, ys). The $_\text{T}$ 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.
- The $xs\,!\,i$ operation returns the $(i + 1)$st element of xs, and list_update $xs\ i\ y$ replaces the $(i + 1)$st element by y.

To prove the refinement lemma sub1_imp $xs \leq$ sub1_spec xs, we 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 our example, two of the resulting proof obligations correspond to the termination of the 'while' loop and the correctness of the assertion. We can use the measure $\lambda(i,\ ys).\ |ys| - i$ to prove termination.

In a refinement step, we can also change the types. For our small program, if we assume that the natural numbers in the list are all nonzero, we 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:

definition $\mathsf{sub1\_imp}_{\mathsf{int}} :: \mathit{int\ list} \Rightarrow \mathit{int\ list\ nres}$ **where**
$\mathsf{sub1_imp}_{\mathsf{int}}\ xs = \langle$same body as sub1_imp$\rangle$

We 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 us express preconditions and connections between types using higher-order relations called relators:

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

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

The *Imperative HOL* library [14] 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, 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:

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

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 we 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**
sub1_imp$_{\text{int}}$ :: [λ_.True] (array_assn nat_assn)$^{\text{d}}$ → array_assn nat_assn
by *sepref*

The generated array-based program is

```
sub1_imp_code xs =
  do {
    (i, zs) ← heap_WHILE_T (λ(i, ys). do { zs ← Array.len ys;  return (i < zs) })
      (λ(i, ys). do {
        z ← Array.nth ys i − 1;
        zs ← Array.upd ys i z;
        return (i + 1, zs)
      })
      (0, xs);
    return zs
  }
```

The end-to-end refinement theorem, obtained by composing the refinement lemmas, is

$$(\text{sub1\_imp\_code},\ \text{sub1\_spec}) \in [\lambda xs.\ \forall i \in xs.\ i \neq 0]\ (\text{array\_assn int\_of\_nat\_assn})^{\text{d}} \to \text{array\_assn int\_of\_nat\_assn}$$

If we want to execute the program efficiently, we can translate it to Standard ML using Isabelle's code generator [23]. 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.

3 Abstract CDCL

The abstract CDCL calculus by Nieuwenhuis et al. [43] forms the first layer of our 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 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, we 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, we write A, $\neg A$, and $L^\dagger$ for positive, negative, and decision literals, and we 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 [1,57], 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, we write clauses using logical symbols (e.g., $\bot$, L, and $C \vee D$ for $\emptyset$, $\{L\}$, and $C \uplus D$). Given a clause C, we write $\neg C$ for the formula that corresponds to the clause's negation.

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

3.2 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, we 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 $\cdot :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$. The concatenation of two lists is written $M\ @\ M'$. To lighten the notation, we often build lists from elements and other lists by simple juxtaposition, writing MLM' for $M\ @\ L \cdot M'$.

The core of the CDCL calculus is defined as a transition relation DPLL_NOT+BJ, an extension of classical DPLL [17] 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, we abuse notation, implicitly converting $\models$'s first operand from a list to a set and ignoring annotations on literals:

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

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

Backjump $(M' L^{\dagger} M, N) \Longrightarrow_{\text{DPLL\_NOT+BJ}} (L' M, N)$
if N contains a conflicting clause C (i.e., $M' L^{\dagger} M \models \neg C$) and there exists a clause $C' \vee L'$ such that $N \models C' \vee L'$, $M \models \neg C'$, 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 $C' \vee L'$ is a new clause derived from N (but not necessarily in N).

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

```
inductive DPLL_NOT+BJ :: 'st ⇒ 'st ⇒ bool where
  propagate S S' ⟹ DPLL_NOT+BJ S S'
| decide S S' ⟹ DPLL_NOT+BJ S S'
| backjump S S' ⟹ DPLL_NOT+BJ S S'
```

Since there is no call to DPLL_NOT+BJ in the assumptions, we 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) ⟹
```

$S' \sim$ prepend_trail (Decided L) $S \Longrightarrow$
decide_conds S $S' \Longrightarrow$
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.

Theorem 1 *(Termination [20, 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 $S \Longrightarrow_{\text{DPLL\_NOT+BJ}} S'$. 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: We have $[0] \prec \epsilon$ by condition (2), whereas $\epsilon <_{\text{lex}} [0]$. Yet the authors justify well-foundedness by appealing to the well-foundedness of $<_{\text{lex}}$ on bounded lists over finite alphabets. In our proof, we clarify and simplify matters by mapping states S to lists $\big[|M_0|, \ldots, |M_n|\big]$, without appealing to ν. Using the standard lexicographic order, states become *larger* with each transition:

Propagate	$[k_1, \ldots, k_n]$	$<_{\text{lex}}$	$[k_1, \ldots, k_n + 1]$
Decide	$[k_1, \ldots, k_n]$	$<_{\text{lex}}$	$[k_1, \ldots, k_n, 0]$
Backjump	$[k_1, \ldots, k_n]$	$<_{\text{lex}}$	$[k_1, \ldots, k_j + 1]$ 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 $D = \{[k_1, \ldots, k_n] \mid k_1, \ldots, k_n, n \leq V\}$. We take $\prec$ to be the restriction of $>_{\text{lex}}$ 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 $\Longrightarrow$, we 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 2 *(Partial Correctness [20, full_dpll_backjump_final_state_from_init_state]) If* $(\epsilon, N) \Longrightarrow^!_{\text{DPLL\_NOT+BJ}} (M, N)$*, then* N *is satisfiable if and only if* $M \models N$.

We 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.

3.3 Classical DPLL

The locale machinery allows us to derive a classical DPLL calculus from DPLL with backjumping. We 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 $(M' L^\dagger M, N) \Longrightarrow_{\text{DPLL\_NOT}} (-L \cdot M, N)$
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 *(Backtracking [20, 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, we 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, we also verify that these assumptions are consistent. Roughly:

locale DPLL_NOT =
begin
  type_synonym *'v state* = (*'v*, *unit*, *unit*) *ann_literal list* × *'v clause multiset*
  inductive backtrack :: *'v state* ⇒ *'v state* ⇒ *bool* **where** …
end

sublocale DPLL_NOT ⊆ dpll_state fst snd (λL (M, N). ($L \cdot M$, N)) …
sublocale DPLL_NOT ⊆ DPLL_NOT+BJ_ops … (λC L S S'. DPLL.backtrack S S') …
sublocale DPLL_NOT ⊆ DPLL_NOT+BJ …

If a conflict cannot be resolved by backtracking, we would like to have the option of stopping even if some variables are undefined. A state (M, N) is *conclusive* if $M \models N$ 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 4 *(Partial Correctness [20, dpll_conclusive_state_correctness]) If* $(\epsilon, N) \Longrightarrow^*_{\text{DPLL\_NOT}}$ (M, N) *and* (M, N) *is a conclusive state, N is satisfiable if and only if* $M \models N$.

The theorem does not require stopping at the first conclusive state. In an implementation, testing $M \models N$ 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 1. In practice, instead of testing whether $M \models N$, implementations typically apply the rules until every literal is set. When N is satisfiable, this produces a total model.

3.4 The CDCL Calculus

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

$$\textsf{Learn}\ (M, N) \Longrightarrow_{\textsf{CDCL\_NOT}} (M, N \uplus \{C\}) \quad \text{if } N \models C \text{ and each atom of } C \text{ is in } N \text{ or } M$$
$$\textsf{Forget}\ (M, N \uplus \{C\}) \Longrightarrow_{\textsf{CDCL\_NOT}} (M, N) \quad \text{if } N \models C$$

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).

We 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 5 *(Termination [20, wf_cdcl$_{NOT}$_no_learn_and_forget_infinite_chain]) Let* C *be an instance of the* CDCL_NOT *calculus (i.e.,* C ⊆ 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 we restrict the learning so that it is always done immediately before backjumping, we 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:

$$\textsf{Learn+Backjump}\ (M' L^{\dagger} M, N) \Longrightarrow_{\textsf{CDCL\_NOT\_merge}} (L' M, N \uplus \{C' \vee L'\})$$
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 ⊆ CDCL_NOT does not hold. Instead, we have CDCL_NOT_merge ⊆ 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.

3.5 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 initial clauses where it can make progress. Upon a restart, some learned clauses may be removed, and the trail is reset to ϵ. Since our 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 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, we assume that bases states S are pairs (M, N). The calculus C+restartT starts in the state $((\epsilon, N), 0)$ and consists of two rules:

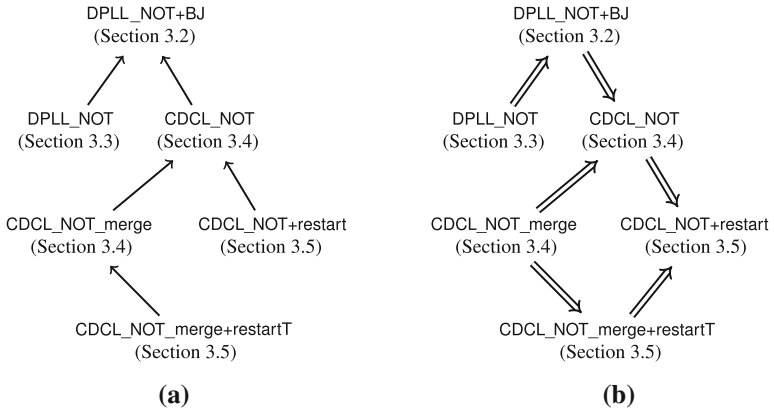

Fig. 1 Connections between the abstract calculi. **a** Syntactic dependencies. **b** Refinements

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

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 us that we count the number of *transitions*; in Sect. 4.5, we 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 $S \Longrightarrow_C^* S' \Longrightarrow_{\textsf{restartT}} S''$, then $\mu_V\ S'' \leq \mu_V\ S$. The measure may depend on V, the number of variables occurring in the problem.

We instantiated the locale parameter C with CDCL_NOT_merge and f with the Luby sequence $(1, 1, 2, 1, 1, 2, 4, \ldots)$ [35], 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 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 1b presents the refinements between the calculi. An arrow $C \Longrightarrow B$ indicates that we proved $C \subseteq B^*$ or some stronger result—either by locale embedding (**sublocale**) or by simulating C's behavior in terms of B.

4 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 [57]. This calculus, CDCL_W, is also formalized in Isabelle and connected to CDCL_NOT.

4.1 The New DPLL Calculus

Independently from the previous section, we 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 $(M, N) \Longrightarrow_{\text{DPLL\_W}} (LM, N)$ if $C \vee L \in N \uplus U$, $M \models \neg C$, and L is undefined in M

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

Backtrack $(M' K^{\dagger} M, N) \Longrightarrow_{\text{DPLL\_W}} (-K \cdot M, N)$
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 (Sect. 3.3).

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

Theorem 6 *(Termination [20, 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:

Propagate	$[k_1, \ldots, k_m, 3, 3, \ldots, 3]$	$>_{\text{lex}}\ [k_1, \ldots, k_m, 1, 3, \ldots, 3]$
Decide	$[k_1, \ldots, k_m, 3, 3, \ldots, 3]$	$>_{\text{lex}}\ [k_1, \ldots, k_m, 2, 3, \ldots, 3]$
Backtrack	$[k_1, \ldots, k_m, 2, l_1, \ldots, l_n]$	$>_{\text{lex}}\ [k_1, \ldots, k_m, 1, 3, \ldots, 3]$

Theorem 7 *(Partial Correctness [20, $dpll_W\_conclusive\_state\_correctness$]) If* $(\epsilon, N) \Longrightarrow^{*}_{\text{DPLL\_W}} (M, N)$ *and* (M, N) *is a conclusive state,* N *is satisfiable if and only if* $M \models N$.

The proof is analogous to the proof of Theorem 2. Some lemmas are shared between both proofs. Moreover, we can link Weidenbach's DPLL calculus with the version we derived from DPLL_NOT+BJ in Sect. 3.3:

Theorem 8 *(DPLL [20, $dpll_W\_dpll_{NOT}$]) For all states* S *that satisfy basic structural invariants,* $S \Longrightarrow_{\text{DPLL\_W}} S'$ *if and only if* $S \Longrightarrow_{\text{DPLL\_NOT}} S'$.

This provides another way to establish Theorems 6 and 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 1).

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

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

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

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

Forget $(M, N, U \uplus \{C\}, \top) \Longrightarrow_{\text{CDCL\_W}} (M, N, U, \top)$ if $M \not\models N$ 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 [6, Chapter 3]:

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

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

Jump $(M' K^{\dagger} M, N, U, D \vee L) \Longrightarrow_{\text{CDCL\_W}} (L^{D \vee L} M, N, U \uplus \{D \vee L\}, \top)$
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 [54].

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:

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

4.3 A Reasonable Strategy

To prove correctness, we assume a *reasonable* strategy: Propagate and Conflict are preferred over Decide; Restart and Forget are not applied. (We will lift the restriction on Restart and Forget in Sect. 4.5.) 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. While this makes sense in an implementation, it is not needed for any of our metatheoretical results.

Theorem 9 *(Partial Correctness [20, full_cdcl$_W$_stgy_final_state_conclusive_from_init_state]) If* $(\epsilon, N, \emptyset, \top) \Longrightarrow^{!}_{\text{CDCL\_W+stgy}} S'$ *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.

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 10 *(No Relearning [20, cdcl$_W$_stgy_distinct_mset_clauses]) If we have* $(\epsilon, N, \emptyset, \top) \Longrightarrow^{*}_{\text{CDCL\_W+stgy}} (M, N, U, D)$, *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, we 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 if induction is needed. The first version of the formal proof was over 700 lines long and is among the most difficult proofs we carried out.

We later refactored the proof. Following the book, each transition in CDCL_W+stgy was normalized by applying Propagate and Conflict exhaustively. For example, we defined Decide+stgy so that $S \Longrightarrow_{\text{Decide+stgy}} U$ if Propagate and Conflict cannot be applied to S and $S \Longrightarrow_{\text{Decide}} T \Longrightarrow^{!}_{\text{Propagate,Conflict}} U$ for some state T. However, normalization is not necessary. It is

simpler to define $S \Longrightarrow_{\text{Decide+stgy}} T$ as $S \Longrightarrow_{\text{Decide}} T$, with the same condition on S as before. This change shortened the proof by about 200 lines. In a subsequent refactoring, we further departed from the book: We 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 we 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 10: We 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.

Using Theorem 10 and assuming that only backjumping has a cost, we get a complexity of $O(3^V)$, where V is the number of different propositional variables. If Conflict is always preferred over Propagate, the learned clause is never redundant in the sense of ordered resolution [57], yielding a complexity bound of $O(2^V)$. We have not formalized this yet.

In *Automated Reasoning*, and in our formalization, Theorem 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, we must show that CDCL_W refines CDCL_NOT.

4.4 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: We 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 conflict-related CDCL_W rules to their high-level counterparts. Our solution is to introduce an intermediate calculus, called CDCL_W_merge, that combines consecutive low-level transitions into a single transition. This calculus refines both CDCL_W and CDCL_NOT_merge and is sufficiently similar to CDCL_W so that we 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:

$$\text{Reduce+Maybe\_Jump } S \Longrightarrow_{\text{CDCL\_W\_ merge}} S'' \quad \text{if } S \Longrightarrow_{\text{Conflict}} S' \Longrightarrow^{!}_{\text{Skip, Resolve, Jump}} S''$$

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 $\bot$, 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 $S \Longrightarrow_{\text{CDCL\_W\_ merge}} S'$, either $S \Longrightarrow_{\text{CDCL\_NOT\_merge}} S'$ 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 Propagate and Reduce+Maybe_Jump over Decide, yielding CDCL_W_merge+stgy. The two strategy-restricted calculi have the same end-to-end behavior:

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

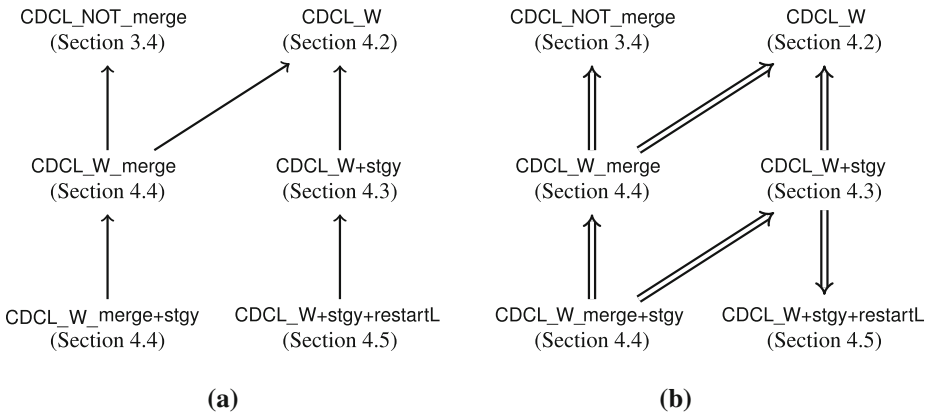

Fig. 2 Connections involving the refined calculi. **a** Syntactic dependencies. **b** Refinements

4.5 A Strategy with Restart and Forget

We could use the same strategy for restarts as in Sect. 3.5, but we prefer to exploit Theorem 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, we 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

$$\textsf{Restart}\ (S, n) \Longrightarrow_{C+\textsf{restartL}} (S''', n+1)$$
$$\text{if } S \Longrightarrow_C^* S' \Longrightarrow_{\textsf{Restart}} S'' \Longrightarrow_{\textsf{Forget}}^* S''' \text{ and } \big|\textsf{learned}\ S'\big| - |\textsf{learned}\ S| \geq f\ n$$
$$\textsf{Finish}\ (S, n) \Longrightarrow_{C+\textsf{restartL}} (S', n+1) \quad \text{if } S \Longrightarrow_C^! S'$$

We formally proved that CDCL_W+stgy+restartL is totally correct. Figure 2 summarizes the situation, following the conventions of Fig. 1.

4.6 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, we designed a calculus CDCL_W+stgy+incr that provides incremental solving on top of CDCL_W+stgy:

$$\textsf{Add\_Nonconflict}_C\ (M, N, U, \top) \Longrightarrow_{\textsf{CDCL\_W+stgy+incr}} S'$$
$$\text{if } M \not\models \neg C \text{ and } (M, N \uplus \{C\}, U, \top) \Longrightarrow^!_{\textsf{CDCL\_W+stgy}} S'$$
$$\textsf{Add\_Conflict}_C\ (M'LM, N, U, \top) \Longrightarrow_{\textsf{CDCL\_W+stgy+incr}} S'$$
$$\text{if } LM \models \neg C,\ -L \in C,\ M' \text{ contains no literal of } C, \text{ and}$$
$$(LM, N \uplus \{C\}, U, C) \Longrightarrow^!_{\textsf{CDCL\_W+stgy}} S'$$

We first run the CDCL_W+stgy calculus on a clause set N, as usual. If N is satisfiable, we 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 11 *(Partial Correctness [20, incremental_conclusive_state]) If state S is conclusive and $S \Longrightarrow_{\text{CDCL\_W+stgy+incr}} S'$, 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 we can reuse the invariants we have already proved about CDCL_W+stgy.

5 A Naive Functional Implementation of CDCL

Sections 3 and 4 presented variants of DPLL and CDCL as parameterized transition systems, formalized using locales and inductive predicates. We now present a deterministic SAT solver that implements CDCL_W+stgy, expressed as a functional program in Isabelle.

When implementing a calculus, we must make many decisions regarding the data structures and the order of rule applications. Our functional SAT solver is very naive and does not feature any optimizations beyond those already present in the CDCL_W+stgy calculus; in Sect. 6, we will refine the calculus further to capture the two-watched-literal optimization and present an imperative implementation relying on mutable data structures.

For our functional implementation, we 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, ⊤) ⇒
    (case find_first_unit_propagation M (N @ U) of
      Some (L, C) ⇒ (Propagated L C · M, N, U, ⊤)
    | None ⇒ S)
  | S ⇒ S)
```

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:

```
function do_all_cdcl_W_stgy :: state ⇒ 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 [27]. For Isabelle to accept the recursive definition of the main loop as a terminating program, we

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, we 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 us 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 [23], we 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.

6 An Imperative Implementation of CDCL

As an impure functional language, Standard ML provides assignment and mutable arrays. We use these features to derive an imperative SAT solver that is much more efficient than the functional implementation. We start by integrating the two-watched-literal optimization into CDCL_W+stgy. Then we refine the calculus to apply rules deterministically, and we generate code that uses arrays to represent clauses and clause sets.

The resulting SAT solver is orders of magnitude faster than the naive functional implementation described in the previous section. However, it is one to two orders of magnitude slower than DPT 2.0 [21], the fastest imperative OCaml solver we know of, because it does not implement restarts or any sophisticated heuristics for learned clause minimization. We expect that many missing heuristics will be straightforward to implement. Due to inefficient memory handling, our solver is not competitive with state-of-the-art solvers.

6.1 The Two-Watched-Literal Scheme

The two-watched-literal (2WL or TWL) scheme [42] is a data structure that makes it possible to efficiently identify candidate clauses for unit propagation and conflict. In each non-unit clause, we distinguish two *watched* literals—the other literals are *unwatched*. Initially, any of a non-unit clause's literals can be chosen to be watched. In the simplest version of the scheme, the solver maintains the following invariant for each non-unit clause:

(α) A watched literal may be false only if all the unwatched literals are false.

As a consequence of this invariant, setting an unwatched literal will never yield a candidate for propagation or conflict, because the two watched literals can then only be true or unset.

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

1. If some of the unwatched literals are not false, we restore the invariant by *updating* the clause: We start watching one of the non-false unwatched literals instead of $-L$.

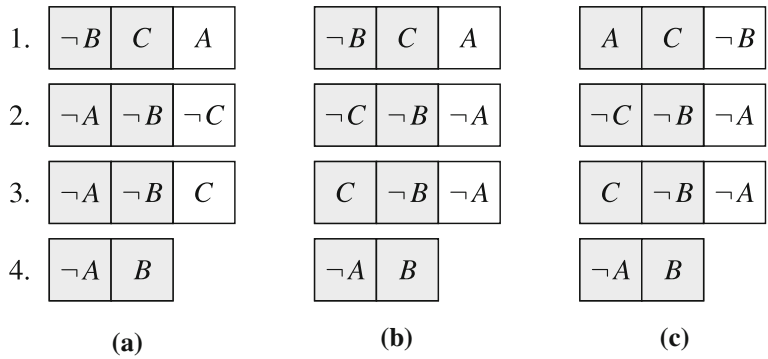

Fig. 3 Evolution of the two-watched-literal data structure on an example

2. Otherwise, we consider the clause's other watched literal:

   2.1. If it is not set, we can propagate it.
   2.2. If it is false, we have found a conflict.
   2.3. If it is true, there is nothing to do.

   In *Automated Reasoning*, a weaker invariant is used, inspired by MiniSat [18]:

   (β) A watched literal may be false only if the other watched literal is true or all the unwatched literals are false.

This invariant is easier to establish than (α): If the other watched literal is true, there is nothing to do, regardless of the truth value of the unwatched literals. The four-step procedure above can easily be adapted, by pulling step 2.3 to the front.

To illustrate how the solver maintains the invariant, whether (α) or (β), we consider the small problem shown in Fig. 3. The clauses are numbered from 1 to 4. Gray cells identify watched literals. Thus, clause 1 is $\neg B \vee C \vee A$, where $\neg B$ and C are watched.

1. We start with an empty trail and an arbitrary choice of watched literals (Fig. 3a).
2. We decide to make A true. The trail becomes $A^{\dagger}$. In clauses 2 and 3, we exchange $\neg A$ with another literal to restore the invariant (Fig. 3b).
3. We propagate B from clause 4. The trail becomes $B\,A^{\dagger}$. In clause 1, we exchange $\neg B$ with A to restore the invariant (Fig. 3c).
4. From clauses 2 and 3, we find out that we can propagate $\neg C$ and C. We choose C. The trail becomes $C\,B\,A^{\dagger}$. Clause 2 is in conflict. The decision made in step 2 was wrong, so we backtrack.

Upon backtracking, there is no need to update the data structure. A key property for the data structure's efficiency is that the invariant is preserved when we remove literals from the trail.

In MiniSat and other implementations, propagation is performed immediately whenever a suitable clause is discovered, and when a conflict is detected, the solver stops updating the data structure and processes the conflict. Using this more efficient strategy, the following scenario is possible for the example of Fig. 3:

1. We start with an empty trail and the same watched literals as before (Fig. 3a).
2. We decide to make A true. The trail becomes $A^{\dagger}$.
3. We propagate B from clause 4. The trail becomes $B\,A^{\dagger}$.
4. We propagate C from clause 3. The trail becomes $C\,B\,A^{\dagger}$. Clause 2 is in conflict. The decision made in step 2 was wrong, so we backtrack.

By making the right arbitrary choices, we could go from propagation to propagation without having to update the clauses. However, neither invariant holds for clauses 1 to 3 after step 3. To capture the new state of affairs, we need a more precise invariant and a richer notion of state that take into account any pending updates. The new invariant is as follows:

> (γ) If there are no pending updates for the clause and no conflict is being processed, invariant (β) holds.

An update is represented by a pair (L, C), where L is a literal that has become false and C is a clause that has L as one of its watched literals. Each time a literal L is added to the trail, all possible updates $(-L,\ C)$ are added to the set of pending updates, which is initially empty. Whenever a conflict is detected, the updates are reset to $\emptyset$. Pending updates can be processed at any time by the calculus.

6.2 The CDCL Calculus with Watched Literals

CDCL with the 2WL data structure is defined as an abstract calculus CDCL_TWL that refines CDCL_W+stgy. Nonunit clauses are represented as TWL_Clause $W\ UW$, where W is the multiset of watched literals (of cardinality 2) and UW the multiset of unwatched literals. Unit clauses are represented as singleton multisets. The state must also keep track of pending updates. States have the form $(M, N, U, D, NP, UP, WS, Q)$, where

- M is the trail;
- N is the initial nonunit clause set in 2WL format;
- U is the learned nonunit clause set in 2WL format;
- D is a conflict clause or $\top$;
- NP is the initial unit clause set;
- UP is the learned unit clause set;
- WS is a multiset of literal–clause pairs (L, C) indicating that clause C must be updated with respect to literal L;
- Q is a set of literals for which further updates are pending.

NP and UP do not influence the calculus; they are ghost components that are useful for connecting a 2WL state to the format expected by CDCL_W:

$$\text{state}_{\text{W\_of}}\ (M, N, U, D, NP, UP, WS, Q) = (M,\ \text{clauses}_{\text{W\_of}}\ N \uplus NP,\ \text{clauses}_{\text{W\_of}}\ U \uplus UP,\ D)$$

The $\text{clauses}_{\text{W\_of}}$ function converts a 2WL clause set to a standard clause set.

The first two rules of CDCL_TWL have direct counterparts in CDCL_W:

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

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

For both rules, the side condition $\forall K \in$ unwatched $C.\ -K \in M$ is necessary because invariant (β) is not required to hold for C while a (L, C) update is pending.

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

Update $(M, N, U, \top, NP, UP, \{(L, C)\} \uplus WS, Q) \Longrightarrow_{\text{CDCL\_TWL}} (M, N', U', \top, NP, UP, WS, Q)$

if $K \in$ unwatched C, $-K \notin M$, and N' and U' are obtained from N and U by replacing $C =$ TWL_Clause $W\ UW$ with TWL_Clause $(W - \{L\} \uplus \{K\})\ (UW - \{K\} \uplus \{L\})$

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

Next_Literal $(M, N, U, \top, NP, UP, \emptyset, \{L\} \uplus Q) \Longrightarrow_{\text{CDCL\_TWL}}$
$(M, N, U, \top, NP, UP, \{(L, C).\ L \in$ watched $C \wedge C \in N \uplus U\}, Q)$

As in CDCL_W+stgy, propagations and conflicts are preferred over decisions. This is achieved by checking that WS and Q are empty when making a decision:

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

The restriction on Decide is enough to ensure that the reasonable strategy is applied in CDCL_TWL. Skip and Resolve are as before, except that they also preserve the 2WL-specific components of the state. The Jump rule is replaced by two rules, because of the distinction between unit and nonunit clauses:

Jump_Nonunit $(M' K^{\dagger} M, N, U, D \vee L, NP, UP, \emptyset, \emptyset) \Longrightarrow_{\text{CDCL\_TWL}}$
$(L^{D \vee L} M, N, U \uplus \{D \vee L\}, \top, NP, UP, \emptyset, \{L\})$
if $D \neq \bot$ and L satisfies the conditions on Jump

Jump_Unit $(M' K^{\dagger} M, N, U, L, NP, UP, \emptyset, \emptyset) \Longrightarrow_{\text{CDCL\_TWL}}$
$(L^{L} M, N, U, \top, NP, UP \uplus \{L\}, \emptyset, \{L\})$ if L satisfies the conditions on Jump

Theorem 12 *(Invariant [20, cdcl_twl_stgy_twl_struct_invs]) If state S satisfies invariant (γ) and $S \Longrightarrow_{\text{CDCL\_TWL}} T$, then T satisfies invariant (γ).*

CDCL_TWL refines CDCL_W+stgy in the following sense:

Theorem 13 *(Refinement [20, full_cdcl_twl_stgy_cdcl$_W$_stgy]) Let S be a state that satisfies invariant (γ). If $S \Longrightarrow^{!}_{\text{CDCL\_TWL}} T$, then* state$_W$_of $S \Longrightarrow^{!}_{\text{CDCL\_W+stgy}}$ state$_W$_of T.

CDCL_TWL refines CDCL_W+stgy's end-to-end behavior and produces final states that are also final states for CDCL_W+stgy. We can apply Theorem 9 to establish partial correctness.

6.3 Derivation of an Executable List-Based Program

The next step is to refine the calculus with watched literals to an executable program. The state is a tuple $(M, NU, n, D, NP, UP, WS, Q)$, where NU is a list (instead of a set) of clauses containing first n initial nonunit clauses followed by the learned nonunit clauses, where clauses are represented as lists of literals starting with the watched ones; M uses indices in NU to represent clause annotations; and WS uses indices in NU to represent clauses. The D, NP, UP, and Q components are as before.

The program's main loop invokes functions that implement specific rules or set of rules. The function for Propagate, Conflict, Update, and Ignore is presented below:

```
definition
  propagate_conflict_update_ignore :: 'v literal ⇒ 'v clause_idx ⇒ 'v state ⇒ 'v state
where
  propagate_conflict_update_ignore L C S = do {
    let (M, NU, n, D, NP, UP, WS, Q) = S;
    let i = (if C ! 0 = L then 0 else 1);
    let L' = (NU ! C) ! (1 − i);
    let pol' = polarity M L';
    if pol' = Some True then
      RETURN (M, NU, n, D, NP, UP, WS, Q)                          (* Ignore *)
    else
      let (pol, j) = find_unwatched M (NU ! C);
      if pol = None
        if pol' = Some False then
          RETURN (M, NU, n, NU ! C, NP, UP, ∅, ∅)                  (* Conflict *)
        else
          RETURN (L'^C M, NU, n, D, NP, UP, WS, {−L'} ⊎ Q)         (* Propagate *)
      else do {
        let K = (NU ! C) ! j;
        let NU' = list_update NU C (list_swap (NU ! C) i j);
        RETURN (M, NU', n, D, NP, UP, WS, Q)                       (* Update *)
      }
  }
```

The values Some True, Some False, and None correspond to positive, negative, and undefined polarity, respectively. As we refine the program, we must provide additional invariants for the data structure—for example, indices in WS are valid and $C :: clause\_idx$ is a valid index. The assertion corresponding to the latter, ASSERT $(C < |NU|)$, is not shown above, but it is needed for code generation.

The main loop is called cdcl_twl_stgy_prog. Although it imposes an order on rule applications, it is not fully deterministic—for example, it does not specify which literal to choose in Decide. The following theorem connects it to the CDCL_TWL calculus:

Theorem 14 *(Refinement [20, cdcl_twl_stgy_prog_spec]) If S is a well-formed state and invariant (γ) holds for all clauses occurring in its NU component, then*

$$\text{cdcl\_twl\_stgy\_prog } S \leq \text{RES}\left\{T \mid \text{state}_{\text{TWL}}\text{\_of } S \Longrightarrow^{!}_{\text{CDCL\_TWL}} \text{state}_{\text{TWL}}\text{\_of } T\right\}$$

where $\text{state}_{\text{TWL}}\text{\_of}$ *translates program states to* CDCL_TWL *states.*

The state returned by the program is final for CDCL_TWL, which means by Theorem 13 that it is also final for CDCL_W+stgy. We conclude that the program is a partially correct implementation of CDCL_W+stgy. In addition, since the specification always specifies a non-FAIL result, the program always terminates normally.

In a further refinement step not presented here, we extend the state with watch lists that map from a literal to the clauses that are watched, instead of recalculating them each time. The watch lists are modeled by a function W such that $W\ L = \{C \in N + U \mid L \in \text{watched } C\}$ and update it in when required.

6.4 Generation of Imperative Code

To be complete in a practical sense, an executable SAT solver must first initialize the 2WL data structure, run the CDCL_TWL calculus, and return "satisfiable" (with a model) or "unsatisfiable," depending on whether a conflict has been found. The initialization step is necessary not only to run the program on actual problems but also to ensure that it is possible to create a 2WL state that satisfies invariant (γ) for any input.

The input is a list of clauses, where each clause is itself a list. We require that the lists are nonempty and contain no duplicates. For each clause C, we perform the following steps:

1. If C is a unit clause L:
   - 1.1 Add L to the state's NP component.
   - 1.2 If $-L$ is in the trail, set the state's D component to L and stop the procedure.
   - 1.3 Otherwise, add L to the state's M and Q components, unless this has already been done.
2. Otherwise, add C to NU. Its first two literals are watched.

The result is a well-formed state that satisfies invariant (γ). If a conflict is found in step 1.2, the program can answer "unsatisfiable" immediately.

Before we can generate imperative code, we must first eliminate the remaining nondeterminism, notably the choice of literal in Decide. We implement the variable-move-to-front heuristic [5]. During initialization, we create a list containing all the literals. This list is used to initialize the doubly linked list needed by the heuristic. We also extract the maximal atom in the list to allocate the list of the polarity-checking optimization (Sect. 6.5) with the correct length.

Second, we must specify the data structures to use the generated code. Lists of clauses are refined to resizable arrays of nonresizable arrays. The dynamic aspect is required for adding learned clauses. Within a clause, only the order of literals needs to change. We had to formalize the data structure ourselves; for technical reasons, the resizable arrays from the Imperative Collection Framework [29,31] cannot contain arrays. We were able to reuse some of the theorems proved on the separation logic level.

We used Sepref to refine the code of the SAT solver, including initialization. We restrict the type of the atoms $'v$ to natural numbers *nat*. In our first version, we also used (unbounded) natural number to literals in the generated code: The literals Pos i and Neg i are encoded by the numbers $2 \cdot i$ and $2 \cdot i + 1$, respectively. However, the extraction of an atom from the literals (the integer division by 2) was inefficient in Standard ML. Therefore, we changed our representation to 32-bits unsigned integers (so only 2^{31} atoms are allowed). The extraction of atoms now becomes bit-shifting.

The end-to-end refinement theorem, relating a semantic satisfiability check on the input problem (model_if_satisfiable that returns None if unsatisfiable) to the Imperative HOL heap code (IsaSAT_code), is stated below, where the clauses_assn relation refines a multiset of multisets of literals to a list of lists of 32-bit unsigned integers, and the option_model_assn relation refines the model that is returned as a list of literals.

Theorem 15 *(End-to-End Correctness [20, IsaSAT_full_correctness]) The following refinement relation holds:*

$$(\text{IsaSAT\_code},\ \text{RETURN} \circ \text{model\_if\_satisfiable})$$
$$\in [\text{no\_duplicate\_no\_false}]\ \text{clauses\_assn}^{k} \to \text{option\_model\_assn}$$

6.5 Fast Polarity Checking

The imperative code described in the previous subsection suffers from a crippling inefficiency: The solver often needs to compute the polarity of a literal, and currently this is achieved by traversing the trail M, which may be very large. In practice, solvers employ a map from atoms to their current polarity.

Using stepwise refinement, we 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 [32], this kind of modularity is invaluable when designing complex data structures.

Since the atoms are natural numbers, we 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 $\text{polarity}_{\text{list\_pair}}$
:: *nat literal* $\Rightarrow$ (*nat, clause_idx*) *ann_literal list* $\times$ *bool option list* $\Rightarrow$ *bool option*
where
$\text{polarity}_{\text{list\_pair}}\ L\ (M, Ls) = \big(\text{case } Ls\ !\ \text{atm\_of}\ L\ \text{of}$
None $\Rightarrow$ None
| Some $b \Rightarrow$ Some (if is_pos L then b else $\neg b$)$\big)$

Given N_1 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 $\text{trail}_{\text{list\_pair}}\text{\_trail\_ref}$
:: $\big(\big((nat,\ clause\_idx)\ ann\_literal\ list \times bool\ option\ list\big)$
$\times$ (*nat, clause_idx*) *ann_literal list*$\big)$ *set*
where
$\text{trail}_{\text{list\_pair}}\text{\_trail\_ref} =$
$\big\{((M', Ls), M).\ M = M' \wedge \forall L \in N_1.\ \text{atm\_of}\ L < |Ls|$
$\wedge\ Ls\ !\ \text{atm\_of}\ L = \text{polarity}\ M\ L\big\}$

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

$$\text{RETURN}\ \big(\text{polarity}_{\text{list\_pair}}\ (M', Ls)\ L\big) \leq \text{RETURN}\ (\text{polarity}\ M\ L) \tag{1}$$

In a subsequent refinement step, we use Sepref to implement the list of polarities by an array, and atoms are mapped to 32-bits unsigned integers (*uint32*), as in Sect. 6.4. Accordingly, we define two auxiliary relations:

- The relation lit_assn :: *nat literal* $\Rightarrow$ *uint32_literal* $\Rightarrow$ *assn* refines a literal with natural number atoms by a literal encoded as a 32-bit unsigned integer.
- The relation $\text{trail}_{\text{list_pair}}$_assn :: (*nat, clause_idx*) *ann_literal list* $\times$ *bool option list* $\Rightarrow$ *uint32_ann_literal list* $\times$ *bool option array* $\Rightarrow$ *assn* refines 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:

$$\begin{aligned}&(\text{polarity\_code},\ \text{RETURN} \circ \text{polarity}_{\text{list\_pair}})\\ &\in \left[\lambda((M, Ls), L).\ \text{atm\_of}\ L < |Ls|\right] \text{trail}_{\text{list\_pair}}\text{\_assn}^{\text{k}} \times \text{lit\_assn}^{\text{k}} \to \text{id\_assn}\end{aligned} \tag{2}$$

The precondition, in square brackets, ensures that we 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 (1) and (2) yields the theorem

$$\begin{aligned}&(\text{polarity\_code},\ \text{RETURN} \circ \text{polarity})\\ &\in [\lambda(M, L).\ L \in N_1]\ \text{trail\_assn}^{\text{k}} \times \text{lit\_assn}^{\text{k}} \to \text{id\_assn}\end{aligned}$$

where trail_assn combines the two refinement relations for trails trail_list_pair_assn and $\text{trail}_{\text{list_pair}}$_trail_ref. The precondition atm_of $L < |Ls|$ is a consequence of $L \in N_1$ and the invariant $\text{trail}_{\text{list_pair}}$_trail_ref. If we 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, we can prove end-to-end correctness as before with respect to the optimized code (cf. Theorem 15).

7 Discussion and Related Work

Our formalization of the DPLL and CDCL calculi consists of about 28 000 lines of Isabelle text. The work was done over a period of 10 months almost entirely by Fleury, who also taught himself Isabelle during that time. It covers nearly all of 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 [19]. The refinement to an imperative program is about 20 000 lines long and took about 6 months to perform.

It is difficult to quantify the cost of formalization as opposed to paper proofs. For a sketchy argument, formalization may take an arbitrarily long time; indeed, Weidenbach's eight-line proof of Theorem 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 [60]. A frequent hurdle to formalization is the lack of suitable libraries. We 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 we did not need any special libraries. We 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.

One of our initial motivations for using locales, besides the ease with which it lets us express relationships between calculi, was that it allows abstracting over the concrete representation of the state. However, we discovered that this is often too restrictive, because some data structures need sophisticated invariants, which we must establish at the abstract level. We found ourselves having to modify the base locale each time we attempted to refine the data structure, an extremely tedious endeavor.

In contrast, the Refinement Framework, with its focus on functions, allows us to exploit local assumptions. Consider the prepend_trail function (Sect. 3.2), 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 (Sect. 6.5) relies on the latter 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, we would have to restrict the specification of prepend_trail to handle only those cases where the literals is 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, we discovered several issues with our program. We had forgotten several assertions (especially array bound checks) and sometimes mixed up the ^{k} and ^{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.

Given the varied level of formality of the proofs in the draft of *Automated Reasoning*, it is unlikely that Fleury will ever catch up with Weidenbach. 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. We also found a major mistake in an extension of CDCL using the branch-and-bound principle: Given a weight function, the calculus aims at finding a model of minimal weight. In the course of formalization, Fleury came up with a counterexample that invalidates the main correctness theorem, whose proof confused partial and total models.

For discharging proof obligations, we relied heavily on Sledgehammer, including its facility for generating detailed Isar proofs [10] and the SMT-based *smt* tactic [13]. We found the SMT solver CVC4 particularly useful, corroborating earlier empirical evaluations [50]. In contrast, the counterexample generators Nitpick and Quickcheck [8] were seldom useful. We often discovered flawed conjectures by observing Sledgehammer fail to solve an easy-looking problem. As one example among many, we 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).

At the calculus level, we followed Nieuwenhuis et al. (Sect. 3) and Weidenbach (Sect. 4), but other accounts exist. In particular, Krstić and Goel [28] 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 10).

Formalizing metatheoretical results about logic in a proof assistant is an enticing, if somewhat self-referential, prospect. Shankar's proof of Gödel's first incompleteness theorem [52], Harrison's formalization of basic first-order model theory [24], and Margetson and Ridge's formalized completeness and cut elimination theorems [36] are some of the landmark results in this area. Recently, SAT solvers have been formalized in proof assistants. Marić [37,38] 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. In addition, he formalized the abstract CDCL calculus by Nieuwenhuis et al. and, together with Janičić [37,39], the more concrete calculus by Krstić and Goel [28]. Marić's methodology is quite different from ours, without the use of refinements, inductive predicates, locales, or even Sledgehammer.

In his Ph.D. thesis, Lescuyer [34] 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 [53]. Most of this work was done by Vaucher during a two-month internship, an impressive achievement. Finally, Oe et al. [47] 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.

8 Conclusion

The advantages of computer-checked metatheory are well known from programming language research, where papers are often accompanied by formalizations and proof assistants are used in the classroom [44,49]. This article, like its predecessors and relatives [9,12,51], reported on some steps we have taken to apply these methods to automated reasoning. Compared with other application areas of proof assistants, the proof obligations are manageable, and little background theory is required.

We presented a formal framework for DPLL and CDCL in Isabelle/HOL, covering the ground between an abstract calculus and a verified imperative SAT solver. Our framework paves the way for further formalization of metatheoretical results. We intend to keep following *Automated Reasoning*, including its generalization of ordered ground resolution with CDCL, culminating with a formalization of the full superposition calculus and extensions. Thereby, we aim at demonstrating that interactive theorem proving is mature enough to be of use to practitioners in automated reasoning, and we hope to help them by developing the necessary libraries and methodology.

The CDCL algorithm, and its implementation in highly efficient SAT solvers, is one of the jewels of computer science. To quote Knuth [26, p. iv], "The story of satisfiability is the tale of a triumph of software engineering blended with rich doses of beautiful mathematics." What fascinates us about CDCL is not only how or how well it works, but also why it works so well. Knuth's remark is accurate, but it is not the whole story.

Acknowledgements Open access funding was provided by the Max Planck Society. Stephan Merz made this work possible in the first place. Dmitriy Traytel remotely cosupervised Fleury's M.Sc. thesis and provided copious advice on using Isabelle. Andrei Popescu gave us his permission to reuse, in a slightly adapted form, the succinct description of locales he cowrote on a different occasion [9]. Simon Cruanes, Anders Schlichtkrull, Mark Summerfield, Dmitriy Traytel, and the reviewers suggested many textual improvements. The work has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (Grant Agreement No. 713999, Matryoshka).

References

1. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, pp. 19–99. Elsevier, Amsterdam (2001)
2. Ballarin, C.: Locales: a module system for mathematical theories. J. Autom. Reason. **52**(2), 123–153 (2014)
3. Bayardo Jr., R.J., Schrag, R.: Using CSP look-back techniques to solve exceptionally hard SAT instances. In: Freuder, E.C. (ed.) CP96. LNCS, vol. 1118, pp. 46–60. Springer, Berlin (1996)
4. Becker, H., Blanchette, J.C., Fleury, M., From, A.H., Jensen, A.B., Lammich, P., Larsen, J.B., Michaelis, J., Nipkow, T., Popescu, A., Schlichtkrull, A., Tourret, S., Traytel, D., Villadsen, J.: IsaFoL: Isabelle Formalization of Logic. https://bitbucket.org/isafol/isafol/. Accessed 13 Feb 2018
5. Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 5584, pp. 237–243. Springer, Berlin (2015)
6. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam (2009)
7. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT solvers. J. Autom. Reason. **51**(1), 109–128 (2013)
8. Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic proof and disproof in Isabelle/HOL. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 12–27. Springer, Berlin (2011)
9. Blanchette, J.C., Popescu, A.: Mechanizing the metatheory of Sledgehammer. In: Fontaine, P., Ringeissen, C., Schmidt, R.A. (eds.) FroCoS 2013. LNCS, vol. 8152, pp. 245–260. Springer, Berlin (2013)
10. Blanchette, J.C., Böhme, S., Fleury, M., Smolka, S.J., Steckermeier, A.: Semi-intelligible Isar proofs from machine-generated proofs. J. Autom. Reason. **56**(2), 155–200 (2016)
11. Blanchette, J.C., Fleury, M., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS, vol. 9706, pp. 25–44. Springer, Berlin (2016)
12. Blanchette, J.C., Popescu, A., Traytel, D.: Soundness and completeness proofs by coinductive methods. J. Autom. Reason. **58**(1), 149–179 (2017)
13. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 179–194. Springer, Berlin (2010)
14. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Berlin (2008)
15. Church, A.: A formulation of the simple theory of types. J. Symb. Log. **5**(2), 56–68 (1940)
16. Cruz-Filipe, L., Heule, M.J.H., Jr., W.A.H., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) CADE-26. LNCS, vol. 10395, pp. 220–236. Springer, Berlin (2017)
17. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
18. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Berlin (2003)
19. 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. Accessed 13 Feb 2018
20. Fleury, M., Blanchette, J.C.: Formalization of Weidenbach's Automated Reasoning—The Art of Generic Problem Solving (2017). https://bitbucket.org/isafol/isafol/src/master/Weidenbach_Book/README.md, Formal proof development. Accessed 13 Feb 2018
21. Goel, A., Grundy, J.: Decision Procedure Toolkit. http://dpt.sourceforge.net/. Accessed 13 Feb 2018
22. Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation, LNCS, vol. 78. Springer, Berlin (1979)
23. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer, Berlin (2010)
24. Harrison, J.: Formalizing basic first order model theory. In: Grundy, J., Newey, M. (eds.) TPHOLs '98. LNCS, vol. 1479, pp. 153–170. Springer, Berlin (1998)
25. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales—a sectioning concept for Isabelle. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs '99. LNCS, vol. 1690, pp. 149–166. Springer, Berlin (1999)
26. Knuth, D.E.: The Art of Computer Programming, vol. 4, Fascicle 6: Satisfiability. Addison-Wesley, Boston (2015)
27. Krauss, A.: Partial recursive functions in higher-order logic. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS, vol. 4130, pp. 589–603. Springer, Berlin (2006)

28. Krstić, S., Goel, A.: Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In: Konev, B., Wolter, F. (eds.) FroCoS 2007. LNCS, vol. 4720, pp. 1–27. Springer, Berlin (2007)
29. Lammich, P.: The Imperative Refinement Framework. Archive of Formal Proofs 2016. http://isa-afp.org/entries/Refine_Imperative_HOL.shtml, Formal proof development. Accessed 13 Feb 2018
30. Lammich, P.: Automatic data refinement. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 84–99. Springer, Berlin (2013)
31. Lammich, P.: Refinement to imperative/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 253–269. Springer, Berlin (2015)
32. Lammich, P.: Refinement based verification of imperative data structures. In: Avigad, J., Chlipala, A. (eds.) CPP 2016, pp. 27–36. ACM, New York (2016)
33. Lammich, P.: Efficient verified (UN)SAT certificate checking. In: de Moura, L. (ed.) CADE-26. LNCS, vol. 10395, pp. 237–254. Springer, Berlin (2017)
34. Lescuyer, S.: Formalizing and implementing a reflexive tactic for automated deduction in Coq. Ph.D. thesis, Université Paris-Sud (2011)
35. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Inf. Process. Lett. **47**(4), 173–180 (1993)
36. Margetson, J., Ridge, T.: Completeness theorem. Archive of Formal Proofs 2004. http://isa-afp.org/entries/Completeness.shtml, Formal proof development. Accessed 13 Feb 2018
37. Marić, F.: Formal verification of modern SAT solvers. Archive of Formal Proofs 2008. http://isa-afp.org/entries/SATSolverVerification.shtml, Formal proof development. Accessed 13 Feb 2018
38. Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theor. Comput. Sci. **411**(50), 4333–4356 (2010)
39. Marić, F., Janičić, P.: Formalization of abstract state transition systems for SAT. Log. Methods Comput. Sci. 7(3) (2011). https://doi.org/10.2168/LMCS-7(3:19)2011
40. Marques-Silva, J.P., Sakallah, K.A.: GRASP–a new search algorithm for satisfiability. In: ICCAD '96, pp. 220–227. IEEE Computer Society Press, Silver Spring (1996)
41. Matuszewski, R., Rudnicki, P.: Mizar: the first 30 years. Mech. Math. Appl. **4**(1), 3–24 (2005)
42. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC 2001, pp. 530–535. ACM, New York (2001)
43. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: from an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53**(6), 937–977 (2006)
44. Nipkow, T.: Teaching semantics with a proof assistant: no more LSD trip proofs. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 24–38. Springer, Berlin (2012)
45. Nipkow, T., Klein, G.: Concrete Semantics: With Isabelle/HOL. Springer, Berlin (2014)
46. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer, Berlin (2002)
47. Oe, D., Stump, A., Oliver, C., Clancy, K.: `versat`: a verified modern SAT solver. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012, LNCS, vol. 7148, pp. 363–378. Springer, Berlin (2012)
48. Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Schulz, S., Ternovska, E. (eds.) IWIL-2010. EPiC, vol. 2, pp. 1–11. EasyChair (2012)
49. Pierce, B.C.: Lambda, the ultimate TA: using a proof assistant to teach programming language foundations. In: Hutton, G., Tolmach, A.P. (eds.) ICFP 2009, pp. 121–122. ACM, New York (2009)
50. Reynolds, A., Tinelli, C., de Moura, L.: Finding conflicting instances of quantified formulas in SMT. In: Claessen, K., Kuncak, V. (eds.) FMCAD 2014, pp. 195–202. IEEE Computer Society Press, Silver Spring (2014)
51. Schlichtkrull, A.: Formalization of the resolution calculus for first-order logic. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 341–357. Springer, Berlin (2016)
52. Shankar, N.: Metamathematics, Machines, and Gödel's Proof, Cambridge Tracts in Theoretical Computer Science, vol. 38. Cambridge University Press, Cambridge (1994)
53. Shankar, N., Vaucher, M.: The mechanical verification of a DPLL-based satisfiability solver. Electr. Notes Theor. Comput. Sci. **269**, 3–17 (2011)
54. Sörensson, N., Biere, A.: Minimizing learned clauses. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 9340, pp. 237–243. Springer, Berlin (2009)
55. Sternagel, C., Thiemann, R.: An Isabelle/HOL formalization of rewriting for certified termination analysis. http://cl-informatik.uibk.ac.at/software/ceta/. Accessed 13 Feb 2018
56. Voronkov, A.: AVATAR: the architecture for first-order theorem provers. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 696–710. Springer, Berlin (2014)

57. Weidenbach, C.: Automated reasoning building blocks. In: Meyer, R., Platzer, A., Wehrheim, H. (eds.) Correct System Design: Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday. LNCS, vol. 9360, pp. 172–188. Springer, Berlin (2015)
58. Wenzel, M.: Isabelle/Isar—a generic framework for human-readable proof documents. In: Matuszewski, R., Zalewska, A. (eds.) From Insight to Proof: Festschrift in Honour of Andrzej Trybulec, Studies in Logic, Grammar, and Rhetoric, vol. 10(23). University of Białystok (2007)
59. Wirth, N.: Program development by stepwise refinement. Commun. ACM **14**(4), 221 (1971)
60. Woodcock, J., Banach, R.: The verification grand challenge. J. Univers. Comput. Sci. **13**(5), 661–668 (2007)