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

Jasmin Christian Blanchette[1,2], Mathias Fleury[2], and Christoph Weidenbach[2]

[1] Inria Nancy – Grand Est & LORIA, Villers-lès-Nancy, France
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany

Abstract. We developed a formal framework for CDCL (conflict-driven clause learning) in Isabelle/HOL. Through a chain of refinements, an abstract CDCL calculus is connected to a SAT solver expressed in a functional programming language, with total correctness guarantees. The framework offers a convenient way to prove metatheorems and experiment with variants. Compared with earlier SAT solver verifications, the main novelties are the inclusion of rules for forget, restart, and incremental solving and the application of refinement.

1 Introduction

Researchers in automated reasoning spend a significant portion of their work time specifying 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. As proof assistants are becoming easier to use, it makes sense to employ them.

In this spirit, we started an effort, called IsaFoL (Isabelle Formalization of Logic), that aims at developing libraries and methodology for formalizing modern research in the field, using the Isabelle/HOL proof assistant [8]. 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 project, which focuses on term rewriting [40].

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 paper presents our formalization of CDCL from *Automated Reasoning* on propositional satisfiability (SAT), developed via a refinement of Nieuwenhuis, Oliveras, and Tinelli's account of CDCL [30]. CDCL is the algorithm implemented in modern SAT solvers. We start with a family of abstract DPLL [12] and CDCL [2, 3, 27, 29] transition systems (Section 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 [42] (Section 4). The latter specifies a criterion for learning clauses representing first unique implication points (1UIPs) [3], with the guarantee that learned clauses are

not redundant and hence derived at most once. The calculus also supports incremental solving. This concrete calculus is refined further, as a certified functional program extracted using Isabelle's code generator (Section 5).

Any formalization effort is a case study in the use of a proof assistant. Beyond the code generator, we depended heavily on the following features of Isabelle:

- *Isar* [43] is a textual proof format inspired by the pioneering Mizar system [28]. It makes it possible to write structured, readable proofs—a requisite for any formalization that aims at clarifying an informal proof.
- *Locales* [1,19] parameterize theories over operations and assumptions, encouraging a modular style of development. They are useful to express hierarchies of related concepts and to reduce the number of parameters and assumptions that must be threaded through a formal development.
- *Sledgehammer* integrates superposition provers and SMT (satisfiability modulo theories) solvers in Isabelle to discharge proof obligations. The SMT solvers, and one of the superposition provers [41], are built around a SAT solver, resulting in a situation where SAT solvers are employed to prove their own metatheory.

Our work is related to other verifications of SAT solvers, typically with the aim of increasing their trustworthiness (Section 6). 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 [18]. In contrast, our focus is on formalizing the metatheory of CDCL, to study and connect the various members of the family. The main novelties of our framework are the inclusion of rules for forget, restart, and incremental solving and the application of refinement to transfer results. The framework is available online as part of the IsaFoL repository [14].

2 Isabelle

Isabelle [32,33] is a generic proof assistant that supports many object logics. The metalogic is an intuitionistic fragment of higher-order logic (HOL) [11]. 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 (e.g., $\mathsf{f}\ x\ y$). Anonymous functions $x \mapsto y_x$ are written $\lambda x.\ y_x$. The judgment $t :: \tau$ indicates that term t has type τ. Propositions are simply terms of type *prop*. Symbols belonging to the signature are uniformly called *constants*, even if they are functions or predicates. The metalogical operators include universal quantification $\bigwedge :: ('a \Rightarrow prop) \Rightarrow prop$ and implication $\Longrightarrow :: prop \Rightarrow prop \Rightarrow prop$. The notation $\bigwedge x.\ p_x$ is syntactic sugar for $\bigwedge (\lambda x.\ p_x)$ 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, $\neg$, $\wedge$, $\vee$, $\longrightarrow$, $\leftrightarrow$, $=$). The object logic is embedded in the metalogic via a constant $\mathsf{Trueprop} :: bool \Rightarrow prop$, which is normally not printed. The distinction between the two logical levels is important operationally but not semantically.

Isabelle adheres to the tradition initiated in the 1970s by the LCF system [15]: 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 predicates and recursive functions. Internally, the system synthesizes appropriate low-level definitions.

Isabelle developments are organized as collections of theory files, or modules, that build on one another. Each file consists of definitions, lemmas, and proofs expressed in Isar, Isabelle's input language. Proofs are specified either as a sequence of tactics that manipulate the proof state directly or in a declarative, natural deduction format. Our formalization almost exclusively employs the more readable declarative style.

The Sledgehammer tool [5, 35] 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.

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

$$\texttt{locale X} = \texttt{fixes c} :: \tau_{'a} \texttt{ assumes } A_{'a,\texttt{c}}$$

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 $A_{'a,\texttt{c}} :: \mathit{prop}$ over $'a$ and c. Definitions made within the locale may depend on $'a$ and c, and lemmas proved within the locale may additionally depend on $A_{'a,\texttt{c}}$. 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 $A_{'a,\texttt{c}}$.

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

3 Abstract CDCL

The abstract CDCL (conflict-driven clause learning) calculus by Nieuwenhuis et al. [30] forms the first layer of our refinement chain. Our formalization relies on basic Isabelle libraries for lists and multisets and on custom libraries for propositional logic. Properties such as partial correctness and termination are inherited by subsequent layers.

3.1 Propositional Logic

We represent raw and annotated literals by freely generated datatypes parameterized by the types $'v$ (propositional variable), $'lvl$ (decision level), and $'cls$ (clause):

```
datatype 'v literal =
  Pos 'v
| Neg 'v

datatype ('v, 'lvl, 'cls) ann_literal =
  Decided ('v literal) 'lvl
| Propagated ('v literal) 'cls
```

The syntax is similar to that of Standard ML and other typed functional programming languages. For example, *literal* has two constructors, Pos and Neg, of type $'v \Rightarrow 'v$ *literal*. Informally, we write A, $\neg A$, and $L^\dagger$ for positive, negative, and decided literals, and $-L$ for the negation of a literal, with $-(\neg A) = A$. The simpler calculi do not use $'lvl$ or $'cls$; they take $'lvl = 'cls = unit$, a singleton type whose unique value is denoted by $()$.

A $'v$ *clause* is a (finite) multiset over $'v$ *literal*. Clauses themselves are often stored in 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 set I of literals, $I \models C$ is true if and only if C and I share a literal. This is lifted to (multi)sets of clauses: $I \models N \leftrightarrow \forall C \in N.\ I \models C$. A set is satisfiable if there exists a (consistent) set of literals I such that $I \models N$. Finally, $N \models N' \leftrightarrow \forall I.\ I \models N \longrightarrow I \models N'$.

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 set of clauses to satisfy. The trail is a list of annotated literals that represents the partial model under construction. 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 the list constructor $\cdot$ has type $'a \Rightarrow 'a\ list \Rightarrow 'a\ list$. The concatenation of two lists is written $M \mathbin{@} M'$. To lighten the notation, we often build lists from elements and other lists by simple juxtaposition, writing MLM' for $M \mathbin{@} L \cdot M'$.

The core of the CDCL calculus is defined as a transition relation DPLL+BJ, an extension of classical DPLL (Davis–Putnam–Logemann–Loveland) [12] with nonchronological backtracking, or *backjumping*. We write $S \Longrightarrow_{\mathsf{DPLL+BJ}} S'$ for DPLL+BJ S S'. The DPLL+BJ calculus consists of three rules, starting from an initial state (ϵ, N):

Propagate $\quad (M, N) \Longrightarrow_{\mathsf{DPLL+BJ}} (LM, N)$
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 $M \models -L$)

Backjump $\quad (M' L^\dagger M, N) \Longrightarrow_{\mathsf{DPLL+BJ}} (L' M, N)$
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 $M' L^\dagger$

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

The Backjump rule is more general than necessary for capturing DPLL, where it suffices to swap the leftmost decision literal. In this form, the rule can also represent CDCL backjumping, if $C' \vee L'$ is a new clause derived from N.

A natural representation of such rules in Isabelle is as an inductive predicate. Isabelle's `inductive` command lets us specify the transition rules as introduction rules. From this specification, it produces elimination rules to perform a case analysis on a hypothesis of the form DPLL+BJ S S'. In the interest of modularity, we formalized the rules individually as their own predicates and combined them to obtain DPLL+BJ:

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

The predicate operates on states (M, N) 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, unit) ann_literal list and
    clauses :: 'st ⇒ 'v clause multiset and
    prepend_trail :: ('v, unit, unit) ann_literal ⇒ 'st ⇒ 'st and ... and
    remove_clause :: 'v clause ⇒ 'st ⇒ 'st
  assumes
    ⋀S L. trail (prepend_trail L S) = L · trail S and ... and
    ⋀S C. clauses (remove_cls C S) = remove_mset C (clauses S)
```

The predicates corresponding to the individual calculus rules are phrased in terms of such an abstract state. For example:

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

States are compared extensionally: $S \sim S'$ is true if the two states have identical trails and clause sets, ignoring other fields. This flexibility is necessary to allow refinements with more sophisticated data structures.

In addition, each rule is defined in its own locale, parameterized by additional side conditions. Complex calculi are built by inheriting and instantiating locales providing the desired rules. Following a common idiom, the DPLL+BJ calculus is distributed over two locales: The first locale, DPLL+BJ_ops, defines the DPLL+BJ calculus; the second locale, DPLL+BJ, extends it with an assumption expressing a structural invariant over DPLL+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 [14, wf_dpll_bj]). *The relation* DPLL+BJ *is well founded.*

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

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

where $M_0, \ldots, M_n, M'_0, \ldots, M'_{n'}$ 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 $i \leq n, n'$ for which $[\nu\, M'_0, \ldots, \nu\, M'_{i-1}] = [\nu\, M_0, \ldots, \nu\, M_{i-1}]$ 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 $\epsilon <_{\mathsf{lex}} [0]$. Yet the authors justify well-foundedness by appealing to the well-foundedness of $<_{\mathsf{lex}}$ on bounded lists over finite alphabets. In our proof, we clarify and simplify matters by mapping states to lists $\big[|M_0|, \ldots, |M_n|\big]$, without appealing to ν. Using the standard lexicographic ordering, states become *larger* with each transition:

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

The lists corresponding to possible states are $\prec$-bounded by the list 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 $>_{\mathsf{lex}}$ to D. A variant of this approach is to encode lists into a measure $\mu_V\, M = \sum_{i=0}^{n} |M_i|\, V^{n-i}$ and let $S' \prec S \leftrightarrow \mu_V\, M' > \mu_V\, M$, building on the well-foundedness of $>$ over bounded sets of integers.

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.

Theorem 2 (Partial Correctness [14, full_dpll_backjump_final_state_from_init_state]). *If* $(\epsilon, N) \Longrightarrow^{*!}_{\mathsf{DPLL+BJ}} (M, N)$*, then* N *is satisfiable if and only if* $M \vDash N$.

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 $M_1, N \vDash L$. From these invariants, together with the constraint that (M, N) is a final state, it is easy to prove the conclusion.

3.3 Classical DPLL

The locale machinery allows us to derive a classical DPLL [12] calculus from DPLL with backjumping. This is achieved through a DPLL locale that restricts the Backjump rule so that it performs only chronological backtracking:

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

Lemma 3 (Backtracking [14, backtrack_is_backjump]). *Backtracking is a special case of backjumping.*

The Backjump rule depends on 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 decided literals, we can take $C' = -L_1 \vee \cdots \vee -L_{n-1}$.

Consequently, the inclusion DPLL $\subseteq$ DPLL+BJ holds. In Isabelle, this is expressed as a locale instantiation: DPLL is made a sublocale of DPLL+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 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 =
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 ⊆ dpll_state fst snd (λL (M, N). (L · M, N)) ...
sublocale DPLL ⊆ DPLL+BJ_ops ... (λC L S S'. DPLL.backtrack S S') ...
sublocale DPLL ⊆ DPLL+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 decided literals. For DPLL, all final states are conclusive, but not all conclusive states are final.

Theorem 4 (Partial Correctness [14, dpll_conclusive_state_correctness]). *If* $(\epsilon, N) \Longrightarrow^*_{\mathsf{DPLL}} (M, N)$ *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 continue for a while. In the worst case, it will stop in a final state—which exists by Theorem 1.

3.4 The CDCL Calculus

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

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

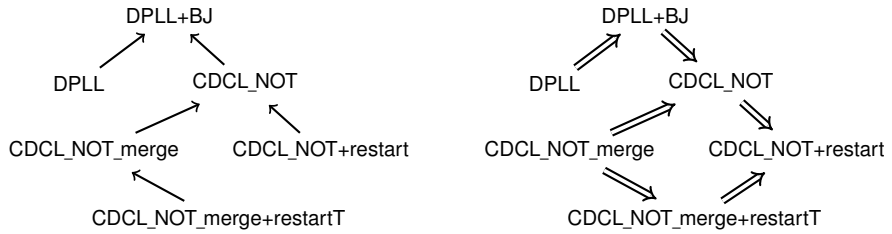
Forget $\quad (M, N \uplus \{C\}) \Longrightarrow_{\mathsf{CDCL\_NOT}} (M, N) \quad$ if $N \models C$

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 after Nieuwenhuis, Oliveras, and Tinelli. Because of the locale parameters, it is strictly speaking a family of calculi. 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. In particular, DPLL+BJ always terminates.

Theorem 5 (Termination [14, wf_cdcl$_{\mathsf{NOT}}$_no_learn_and_forget_infinite_chain]). *Let C be an instance of the* CDCL_NOT *calculus (i.e.,* $C \subseteq$ CDCL_NOT*). If C admits no infinite chains consisting exclusively of* Learn *and* Forget *transitions, then C is well founded.*

In many SAT solvers, the only clauses that are ever learned are the ones used for backtracking. If 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:



(a) Syntactic dependencies (b) Refinements

Fig. 1. Connections between the abstract calculi

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

The calculus variant that performs this rule instead of Learn or Backjump is called CDCL_NOT_merge. Because a single Learn+Backjump transition corresponds to two transitions in CDCL_NOT, the inclusion CDCL_NOT_merge $\subseteq$ CDCL_NOT does not hold. Instead, we have CDCL_NOT_merge $\subseteq$ CDCL_NOT$^+$, which is proved by simulation.

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, our 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 $f :: \mathbb{N} \Rightarrow \mathbb{N}$. Nieuwenhuis et al. require f to be strictly increasing, but unboundedness is sufficient.

The extended calculus C+restartT is defined by the two rules

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

The T in restartT reminds us that we count the number of *transitions*; in Section 4.4, 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 $S \Longrightarrow_C^* S' \Longrightarrow_{\mathsf{restartT}} S''$, then $\mu_V\, S'' \leq \mu_V\, S$. The measure may depend on V, the number of variables occurring in the problem. We instantiated the locale parameter C with CDCL_NOT_merge and f with the Luby sequence $(1,1,2,1,1,2,4,\dots)$ [23], 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 1(a) 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 1(b) 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 implementations, while keeping many aspects unspecified [42]. This calculus, CDCL_W, is also formalized in Isabelle and connected to CDCL_NOT.

4.1 The New CDCL Calculus

The CDCL_W calculus operates on states (M, N, U, k, D), where M is the trail; N and U are the sets of initial and learned clauses, respectively; k is the decision level (i.e., the number of decision literals in M); D is a conflict clause, or the distinguished clause $\top$ if no conflict has been detected. In M, each decision literal is annotated with a level (Decided $L\,k$ or L^k), and each propagated literal is annotated with the clause that caused its propagation (Propagated $L\,C$ or L^C). The level of a propagated literal L is the level of the closest decision literal that follows it in the trail, or 0 if no such literal exists. The level of a clause is the highest level of any of its literals (0 for $\bot$). The calculus assumes that N contains no duplicate literals and never produces clauses containing duplicates.

The calculus starts in a state $(\epsilon, N, \emptyset, 0, \top)$. The following rules apply as long as no conflict has been detected:

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

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

Decide $\quad (M, N, U, k, \top) \Longrightarrow_{\mathsf{CDCL\_W}} (L^{k+1} M, N, U, k+1, \top)$
if L is undefined in M and occurs in N

Restart $\quad (M, N, U, k, \top) \Longrightarrow_{\mathsf{CDCL\_W}} (\epsilon, N, U, 0, \top)$ if $M \not\models N$

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

Once a conflict clause is detected and stored in the state, the following rules collaborate to reduce it and backtrack, exploring a first unique implication point [3]:

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

Resolve $(L^{C \vee L} M, N, U, k, D \vee -L) \Longrightarrow_{\mathsf{CDCL\_W}} (M, N, U, k, C \cup D)$ if D is of level k

Backtrack $(M' K^{i+1} M, N, U, k, D \vee L) \Longrightarrow_{\mathsf{CDCL\_W}} (L^{D \vee L} M, N, U \uplus \{D \vee L\}, i, \top)$
if L is of level k and D is of level i

(In Resolve, $C \cup D$ is the same as $C \vee D$, except that it avoids duplicating literals present in both C and D.) In combination, these three rules can be simulated by the combined learning and nonchronological backjump rule Learn+Backjump from CDCL_NOT_merge.

Several structural invariants hold on all states reachable from an initial state, including the following: The trail is consistent; the k decided literals in the trail are annotated with levels k to 1; and the clause annotating a propagated literal of the trail is contained in $N \uplus U$. 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, k, D) is *conclusive* if $D = \top$ and $M \models N$ or if $D = \bot$ and N is unsatisfiable. The calculus always terminates but, without suitable strategy, it can stop in an inconclusive state. Consider this derivation: $(\epsilon, \{A, B\}, \emptyset, 0, \top) \Longrightarrow_{\mathsf{Decide}} (\neg A^1, \{A, B\}, \emptyset, 1, \top) \Longrightarrow_{\mathsf{Decide}} (\neg B^2\ \neg A^1, \{A, B\}, \emptyset, 2, \top) \Longrightarrow_{\mathsf{Conflict}} (\neg B^2\ \neg A^1, \{A, B\}, \emptyset, 2, A)$. The conflict cannot be processed by Skip or Resolve. The calculus is blocked.

4.2 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 Section 4.4.) 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 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.

Theorem 6 (Partial Correctness [14, full_cdcl$_{\mathsf{W}}$_stgy_final_state_conclusive_from_init_state]). *If $(\epsilon, N, \emptyset, 0, \top) \Longrightarrow^{*!}_{\mathsf{CDCL\_W+stgy}} S'$ 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 Backtrack, 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 7 (Relearning [14, cdcl$_{\mathsf{W}}$_stgy_distinct_mset_clauses]). *Let $(\epsilon, N, \emptyset, 0, \top) \Longrightarrow^{*}_{\mathsf{CDCL\_W+stgy}} (M, N, U, k, D)$. No* Backtrack *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 $(M, N, U, k, D \vee L)$ where Backtrack is applicable and $D \vee L \in N \uplus U$. More precisely, the state has the form $(K_n \cdots K_2 K_1^k M_2 K^{i+1} M_1, N, U, k, D \vee L)$ where the K_i, $i > 1$ are propagated literals that do not occur complemented in D, as for 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 $K_n \cdots K_2 K_1^k M_2$ $K^{i+1} M_1$ and $D \vee L \in N \uplus U$ instead of deciding K_1^k 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 $M_2 K^{i+1} M_1$, 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 full formal proof is over 700 lines long and was among the most difficult proofs we carried out.

Using this theorem and assuming that only backjumping has a cost, we get a complexity of $\mathrm{O}(3^V)$, where V is the number of different propositional variables. If Conflict is always preferred over Propagate, the learned clause in never redundant in the sense of ordered resolution [42], yielding a complexity bound of $\mathrm{O}(2^V)$. Formalizing this is planned for future work.

In *Automated Reasoning*, and in our formalization, Theorem 7 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.

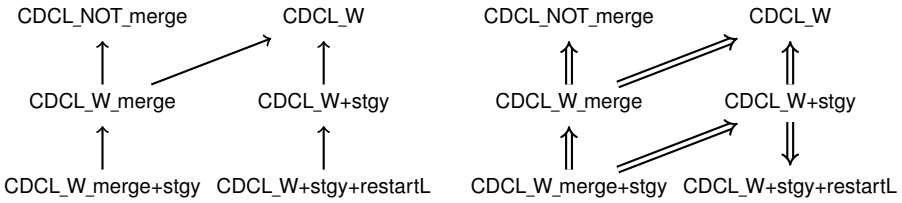
4.3 Connection with Abstract CDCL

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, k, C) as a CDCL_NOT pair (M, N).

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 $\mathsf{Conflict}\,(\mathsf{Skip} \mid \mathsf{Resolve})^*\,\mathsf{Backtrack}^?$, the CDCL_W_merge calculus performs a single transition of a new rule that subsumes all four low-level rules:

Reduce+Maybe_Backtrack $\quad S \Longrightarrow_{\mathsf{CDCL\_W\_merge}} S''$
if $S \Longrightarrow_{\mathsf{Conflict}} S' \Longrightarrow^{*!}_{\mathsf{Skip} \mid \mathsf{Resolve} \mid \mathsf{Backtrack}} S''$



(a) Syntactic dependencies (b) Refinements

Fig. 2. Connections involving the refined calculi

When simulating CDCL_W_merge in terms of CDCL_NOT, two interesting scenarios arise. In the first case, Reduce+Maybe_Backtrack's behavior comprises a backtrack. The rule can then 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_Backtrack has no counterpart in CDCL_NOT_merge. More formally, 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 CDCL_W_merge to a *reasonable* strategy that prefers Propagate and Reduce+Maybe_Backtrack over Decide, yielding CDCL_W_merge+stgy. The two strategy-restricted calculi have the same end-to-end behavior:

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

4.4 A Strategy with Restart and Forget

We could use the same strategy for restarts as in Section 3.5, but we prefer to exploit Theorem 7, 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 obtain termination. This criterion is the norm in existing implementations. The lower bound on the number of learned clauses is given by an unbounded function $f :: \mathbb{N} \Rightarrow \mathbb{N}$. 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

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

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

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

4.5 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 set of clauses. 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:

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

Add_Conflict$_C$ $\quad (M'LM, N, U, k, \top) \Longrightarrow_{\text{CDCL\_W+stgy+incr}} S'$
if $LM \vDash \neg C$, $-L \in C$, M' contains no literal of C, L is of level i in LM, and
$(LM, N \uplus \{C\}, U, i, C) \Longrightarrow^{*!}_{\text{CDCL\_W+stgy}} S'$

We first run the CDCL_W+stgy calculus on a set of clauses 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 8 (Partial Correctness [14, incremental_conclusive_state]). *If S is a conclusive state and $S \Longrightarrow_{\text{CDCL\_W+stgy+incr}} S'$, then S' is a conclusive state.*

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. The proof is easy because we can reuse the invariants we have already proved about CDCL_W+stgy.

5 An Implementation of CDCL

The previous sections presented variants of DPLL and CDCL as parameterized transition systems, formalized using locales and inductive predicates. The final link in our refinement chain is 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. We choose to represent states by tuples (M, N, U, k, D), where propositional variables are coded as natural numbers and multisets as lists.[1] 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:

[1] We have started formalizing the two-watched-literal optimization [29] but have yet to connect it with our SAT solver implementation. The `README.md` file in our repository is frequently updated to mention the latest developments [14].

```
definition do_propagate_step :: 'v solver_state ⇒ 'v solver_state where
  do_propagate_step S =
  (case S of
    (M, N, U, k, ⊤) ⇒
    (case find_first_unit_propagation M (N @ U) of
      Some (L, C) ⇒ (Propagated L C · M, N, U, k, ⊤)
    | None ⇒ S)
  | S ⇒ S)
```

The main loop invokes the functions for the rules, looking for conflicts before propagating literals. It is a recursive program, specified using the `function` command [21]. 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 CDCL_W+stgy, 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 [16], we can translate the program into Haskell, OCaml, Scala, or Standard ML code. 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 700 lines long in OCaml. It is not efficient, due to its extensive reliance on lists, but it satisfies the need for a proof of concept.

6 Discussion and Related Work

Our formalization consists of about 28 000 lines of Isabelle text. It 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 [13].

It is difficult to quantify the cost of formalization as opposed to paper proofs. For a sketchy paper proof, formalization may take an arbitrarily long time; indeed, Weidenbach's nine-line proof of Theorem 7 took 700 lines of Isabelle. In contrast, given a very detailed paper proof, one can obtain a formalization in less time than it took to write the paper proof [44]. A common hurdle to formalization is often the lack of suitable libraries. For CDCL, we spent considerable time adding definitions, lemmas, and automation hints to Isabelle's multiset library but otherwise did not need any special libraries. We also found that organizing the proof at a high level—especially locale engineering—is more challenging than discharging 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. The most damning mistake was in the proof of the resolution calculus without reductions, where the completeness theorem was stated with "$N \Longrightarrow^* \{\bot\}$" instead of "$N \Longrightarrow^* N'$ and $\bot \in N'$." For CDCL, the main issues were that key invariants were omitted and some proofs were too sketchy to be accessible to the intended audience of the book.

For discharging proof obligations, we relied extensively on Sledgehammer, including its facility for generating detailed Isar proofs [4] and the SMT-based *smt* tactic [10]. We found the SMT solver CVC4 particularly useful, corroborating earlier empirical evaluations [37]. In contrast, the counterexample generators Nitpick and Quickcheck [6] were seldom of any use. We often discovered flawed conjectures by seeing 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: $\mathsf{set\_mset}\ (\mathsf{mset\_set}\ A) = A$. Because Isabelle 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 sets as values for A.

Formalizing logic in a proof assistant is an enticing, if somewhat self-referential, prospect. Shankar's proof of Gödel's first incompleteness theorem [38], Harrison's formalization of basic first-order model theory [17], and Margetson and Ridge's formalized completeness and cut elimination theorems [24] are among the first results in this area. Recently, SAT solvers have been formalized in proof assistants. Marić [25, 26] 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. 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 [22] 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 [39]. Most of this work was done by Vaucher during a two-month internship, an impressive achievement. Finally, Oe et al. [34] 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.

7 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 [31, 36]. This paper, like its predecessors [7, 9], 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 certified 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 [20, 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.

Acknowledgment Stephan Merz made this work possible. Dmitriy Traytel remotely cosupervised Fleury's M.Sc. thesis and provided ample advice on using Isabelle (as opposed to developing it). Andrei Popescu gave us his permission to reuse, in a slightly adapted form, the succinct description of locales he cowrote on a different occasion [7]. Simon Cruanes, Anders Schlichtkrull, Mark Summerfield, and Dmitriy Traytel suggested textual improvements.

References

[1] Ballarin, C.: Locales: A module system for mathematical theories. J. Autom. Reasoning 52(2), 123–153 (2014)
[2] 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 (1996)
[3] Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
[4] Blanchette, J.C., Böhme, S., Fleury, M., Smolka, S.J., Steckermeier, A.: Semi-intelligible Isar proofs from machine-generated proofs. J. Autom. Reasoning (to appear)
[5] Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT solvers. J. Autom. Reasoning 51(1), 109–128 (2013)
[6] 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 (2011)
[7] 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 (2013)
[8] Blanchette, J.C., Fleury, M., Schlichtkrull, A., Traytel, D.: IsaFoL: Isabelle Formalization of Logic, `https://bitbucket.org/jasmin_blanchette/isafol`
[9] Blanchette, J.C., Popescu, A., Traytel, D.: Unified classical logic completeness: A coinductive pearl. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 46–60. Springer (2014)
[10] 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 (2010)
[11] Church, A.: A formulation of the simple theory of types. J. Symb. Log. 5(2), 56–68 (1940)

[12] Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM 5(7), 394–397 (1962)
[13] Fleury, M.: Formalisation of ground inference systems in a proof assistant, https://www.mpi-inf.mpg.de/fileadmin/inf/rg1/Documents/fleury_master_thesis.pdf
[14] Fleury, M., Blanchette, J.C.: Formalization of Weidenbach's *Automated Reasoning—The Art of Generic Problem Solving*, https://bitbucket.org/jasmin_blanchette/isafol/src/master/Weidenbach_Book/README.md
[15] Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation, LNCS, vol. 78. Springer (1979)
[16] 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 (2010)
[17] Harrison, J.: Formalizing basic first order model theory. In: Grundy, J., Newey, M. (eds.) TPHOLs '98. LNCS, vol. 1479, pp. 153–170. Springer (1998)
[18] Heule, M., Hunt Jr., W.A., Wetzler, N.: Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. Softw. Test. Verif. Reliab. 24(8), 593–607 (2014)
[19] 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 (1999)
[20] Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability. Addison-Wesley (2015)
[21] Krauss, A.: Partial recursive functions in higher-order logic. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS, vol. 4130, pp. 589–603. Springer (2006)
[22] Lescuyer, S.: Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. Ph.D. thesis (2011)
[23] Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. Inf. Process. Lett. 47(4), 173–180 (1993)
[24] Margetson, J., Ridge, T.: Completeness theorem. vol. 2004. http://afp.sf.net/entries/Completeness.shtml, Formal proof development
[25] Marić, F.: Formal verification of modern SAT solvers. Archive of Formal Proofs 2008, http://afp.sf.net/entries/SATSolverVerification.shtml, Formal proof development
[26] Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theor. Comput. Sci. 411(50), 4333–4356 (2010)
[27] Marques-Silva, J.P., Sakallah, K.A.: GRASP—A new search algorithm for satisfiability. In: ICCAD '96. pp. 220–227. IEEE Computer Society Press (1996)
[28] Matuszewski, R., Rudnicki, P.: Mizar: The first 30 years. Mechanized Mathematics and Its Applications 4(1), 3–24 (2005)
[29] 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 (2001)
[30] 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)
[31] 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 (2012)
[32] Nipkow, T., Klein, G.: Concrete Semantics: With Isabelle/HOL. Springer (2014)
[33] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)

[34] 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 (2012)

[35] 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)

[36] 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 (2009)

[37] 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 (2014)

[38] Shankar, N.: Metamathematics, Machines, and Gödel's Proof, Cambridge Tracts in Theoretical Computer Science, vol. 38. Cambridge University Press (1994)

[39] Shankar, N., Vaucher, M.: The mechanical verification of a DPLL-based satisfiability solver. Electr. Notes Theor. Comput. Sci. 269, 3–17 (2011)

[40] Sternagel, C., Thiemann, R.: An Isabelle/HOL formalization of rewriting for certified termination analysis, `http://cl-informatik.uibk.ac.at/software/ceta/`

[41] 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 (2014)

[42] 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 (2015)

[43] 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)

[44] Woodcock, J., Banach, R.: The verification grand challenge. J. Univers. Comput. Sci. 13(5), 661–668 (2007)