A Verified SAT Solver with Two Watched Literals Using Imperative HOL

Mathias Fleury[1(✉)], Jasmin Christian Blanchette[2,1], and Peter Lammich[3]

[1] Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrücken, Germany
{mathias.fleury,jasmin.blanchette}@mpi-inf.mpg.de
[2] Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
j.c.blanchette@vu.nl
[3] Technische Universität München, Munich, Germany
lammich@in.tum.de

Abstract. Based on our earlier formalization of conflict-driven clause learning (CDCL) in Isabelle/HOL, we refine the CDCL calculus further to add an important optimization: two watched literals. We formalize the data structure and the invariants. Then we refine the calculus to an executable SAT solver in Standard ML: Through a chain of refinements, we target Imperative HOL using the Isabelle Refinement Framework and extract imperative Standard ML code. Although our solver is not yet competitive, it offers acceptable performance for some applications, and custom heuristics can easily be added to improve it further.

1 Introduction

SAT solvers are programs that decide the Boolean satisfiability (SAT) problem. Other NP-complete problems are often reduced to SAT, to exploit efficient SAT solvers. For example, the termination prover $\mathsf{T_TT_2}$ [14] compares multisets of terms using the multiset extension of the subterm order [28]. Checking these proofs, or certificates, is NP-complete. The certification is performed by a trusted SAT solver. Employing a verified SAT solver would reduce the trusted code base, especially when $\mathsf{T_TT_2}$ is used in conjunction with the CeTA checker [29], which is formalized in Isabelle/HOL.

We recently used Isabelle/HOL to formalize a calculus for conflict-driven clause learning (CDCL) [4], based on Nieuwenhuis, Olivetti, and Tinelli's presentation [25]. CDCL is the core calculus implemented in most SAT solvers. It is a generalization of the Davis–Putnam–Logemann–Loveland (DPLL) procedure [8] with clause learning and nonchronological backjumping. We also formalized a variation by Weidenbach [31], which explores the first unique implication point to derive interesting clauses to learn.

An important optimization in SAT solvers is the two-watched-literal [24] data structure. It allows for efficient unit propagation and conflict detection—the core CDCL operations. In this paper, we present a verified CDCL-based SAT solver with watched literals. We use stepwise refinement: Our existing formalization of CDCL is connected, in several correctness-preserving steps, to an imperative SAT solver implementation.

In the first step, we specify an abstract version of the two-watched-literal scheme, resulting in a refined calculus called TWL (Section 3). We follow the description from Weidenbach's textbook draft, tentatively titled *Automated Reasoning—The Art of Generic*

Problem Solving. Formalizing the two-watched-literal optimization reveals some mistakes and suggests the addition of missing invariants in the book.

The TWL calculus captures a SAT solver as a system of nondeterministic transition rules. The next refinement step implements the rules of the calculus in a more algorithmic way, using the nondeterministic programming language provided by the Isabelle Refinement Framework [16] (Section 4). The resulting algorithm is still close to the abstract calculus: It is nondeterministic and uses an abstract mathematical representation of the state. The next step refines the data structure: Lists are used instead of multisets, and clauses are implemented by indices into a list of clauses (Section 5).

An essential ingredient for an efficient implementation of watched literals is a data structure called *watched lists*. These index the clauses by the literals they watch—literals that can influence the truth value of the clauses they belong to, in the current state of the solver. Watched lists are introduced in a straightforward data refinement step (Section 6).

Finally, we use the Sepref tool [17] to refine the functional program to an imperative one. Sepref automatically synthesizes an imperative program and a refinement proof from a functional program, replacing abstract functional data structures by concrete imperative implementations, while leaving the algorithmic structure of the program unchanged. For this step, we implement specialized imperative data structures: The clauses are stored in a dynamic array of arrays, and the current partial model is represented as a list that holds the currently set literals and an array that associates atoms with their truth values. Isabelle/HOL's code generator is used to extract a self-contained SAT solver in imperative Standard ML (Section 7).

We compare the performance of our program to existing solvers: MiniSat [9]; the state-of-art solver Glucose [1]; the OCaml-based DPT [11] solver; and the only other verified SAT solver we know of, `versat` [26] (Section 8). Our solver's performance is similar to `versat`'s, but more heuristics must be implemented before it can compete with the state of the art (Section 8). Compared with `versat`, the hallmark of our solver is its modularity. Heuristics can easily be incorporated, and further refinement steps can be performed if desired. Furthermore, our solver is guaranteed to terminate.

We recently submitted an extended version of our earlier paper [4], on the formalization of various CDCL calculi, to a journal. To make that article self-contained, we added a section that briefly describes the derivation of an imperative program [3, Section 6]. Clearly, that section is no substitute for the current 10-section paper. Here, we explain each step of the refinement in detail and put a particular emphasis on the methodology and the proving technology used. In addition, we present an empirical evaluation of the solver's performance. Some of the background material contained in Sections 2 and 3 is based on the article's text. Our formalization is available online as part of the Isabelle Formalization of Logic (IsaFoL) repository.[1] The theorems referenced in this paper are labeled with their Isabelle names for convenience.

2 The CDCL Calculus

We define literals as a datatype $'v$ *lit* with two constructors: Given an atom A of propositional logic, of type $'v$ ("variable"), Pos A and Neg A are literals. The negation of a literal

[1] `https://bitbucket.org/isafol/isafol/src/ITP2017/Weidenbach_Book/`

is defined by $-\,\mathsf{Pos}\,A = \mathsf{Neg}\,A$ and $-\,\mathsf{Neg}\,A = \mathsf{Pos}\,A$. A clause is a multiset of literals and has type *'v clause*; following Isabelle conventions, all multisets are finite. We use logical symbols for clauses to ease reading when there is no risk of confusion, writing $\bot$, L, $C \vee D$ for $\emptyset$, $\{L\}$, and $C \uplus D$, respectively. A consistent set of literals I *entails* a clause C ($I \models C$) if and only if I and C share a literal. I entails a (multi)set of clauses N if and only if I entails every clause in N.

In our previous paper [4] and in the corresponding journal submission [3], we presented two families of CDCL calculi and connected them through various refinement steps. Here, we briefly describe one of the calculi, which will serve as the starting point for the refinement chain. The calculus, which we call W because it follows Weidenbach's CDCL variant [31], operates on states (M, N, U, D), where

- M is the partial model under construction, or *trail*;
- N is the multiset of initial clauses;
- U is the multiset of learned clauses;
- D is a conflict clause, or the special value $\top$ if no conflict has been detected.

The multiset of initial clauses does not change during the execution of the calculus. In contrast, the multiset of learned clauses grows monotonically starting from $\emptyset$.

The trail M consists of a list of annotated literals, of type $('v, 'v\ \textit{clause})\ \textit{ann\_lit}$. The literals L in M can be decisions, written $L^\dagger$, or they are the result of a unit propagation, in which case they are annotated with the clause C that caused the propagation, written L^C. We use the infix operator $\cdot$ to denote both the Cons list constructor and list concatenation. In accordance with Isabelle conventions, the trail is extended on the left.

The calculus assumes that N contains no duplicate literals and never produces clauses containing duplicates. We write $S \Longrightarrow_{\mathsf{W}} T$ if the calculus makes a step from state S to state T. Steps are derived by the following rules:

Propagate $\quad (M, N, U, \top) \Longrightarrow_{\mathsf{W}} (L^{C \vee L} \cdot M, N, U, \top)$
if $C \vee L \in N \uplus U$, $M \models \neg C$, and L is undefined in M (i.e., neither $L \in M$ nor $-L \in M$)

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

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

Backjump $\quad (M' \cdot K^\dagger \cdot M, N, U, C) \Longrightarrow_{\mathsf{W}} (L^{D \vee L} \cdot M, N, U \uplus \{D \vee L\}, \top)$
where $D \vee L$ is derived in a specific way from the clauses and the conflict, $M \models \neg D$, and L is undefined in M

In Weidenbach's formulation and in our formalization, the Backjump rule takes the form of several fine-grained rules, whose collective effect amounts to Backjump. These rules explicitly derive the clause $D \vee L$. They are described in our earlier paper [4] under the name CDCL_W. The details are not essential for the current paper.

The W calculus is used as follows: We start with an initial state $(\epsilon, N, \emptyset, \top)$ and repeatedly apply the rules until we reach a normal form. From this normal form, we hope to read off a solution to the SAT problem. We call a state (M, N, U, D) *conclusive* if $D = \top$ and $M \models N$ or if $D = \bot$ and N is unsatisfiable. Given a conclusive state, the solution to the SAT problem can easily be extracted.

The W calculus always terminates, but there are no guarantees that the final state is conclusive. If we want this property, we need a strategy to restrict rule applications.

Weidenbach's *reasonable* strategy prefers Propagate and Conflict over all other rules. We call the calculus restricted by this strategy W+stgy.

Given a relation $\Longrightarrow$, we write $S \Longrightarrow^{!} T$ if T is a normal form reachable from S.

Theorem 1 (Correctness [10, *full_cdcl$_W$_stgy_final_state_conclusive_from_init_state*]) *If $(\epsilon, N, \emptyset, \top) \Longrightarrow^{!}_{\mathsf{W+stgy}} S'$ and N contains no clauses with duplicate literals, then S' is a conclusive state.*

3 Two Watched Literals

The two-watched-literal (2WL or TWL) scheme [24] is a data structure that makes it possible to efficiently identify candidate clauses for unit propagation or conflict. In each nonunit clause, we distinguish two *watched* literals—the other literals are *unwatched*. Initially, any of a nonunit clause's literals can be chosen to be watched. The solver maintains the following *2WL invariant* for each clause:

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

This is the invariant given by Weidenbach. It is inspired by MiniSat. As a consequence of this invariant, setting an unwatched literal will never yield a candidate for propagation or conflict. This can dramatically reduce the number of candidates to consider.

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

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

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

To illustrate how the solver maintains the 2WL invariant, we consider the small problem shown in Figure 1. 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. The following scenario is possible:

1. We start with an empty trail and the clauses shown in Figure 1(a). We decide to make A true. The trail becomes $A^{\dagger}$. We need to consider every clause where $\neg A$ is watched, i.e., clauses 3 and 4, in any order.
2. We first consider clause 4 for $\neg A$. We propagate B from it. The trail becomes $B \cdot A^{\dagger}$. We still need to consider clause 3 for $\neg A$ and the clauses for $\neg B$.

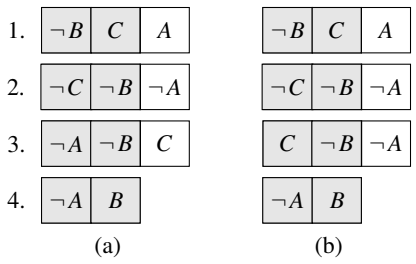

Fig. 1. Evolution of the 2WL data structure on an example

3. We consider clause 3 for $\neg A$. The literal C is unwatched and non-false: We swap C and $\neg A$, resulting in the clauses shown in Figure 1(b). We must still consider clauses 1, 2, and 3 for $\neg B$.
4. We consider clause 3 for $\neg B$: We propagate C. The trail becomes $C \cdot B \cdot A^{\dagger}$. We still need to update the clauses 1 and 2 for $\neg B$ and the clauses associated with $\neg C$.
5. We consider clause 2. All its literals are false—a conflict. Thanks to the invariant's last condition ("or a conflict has been found"), we do not need to update clause 1 nor the clauses associated with $\neg C$ to maintain the invariant.

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

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

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

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

The $\mathsf{state}_{\mathsf{W}}\_\mathsf{of}$ function converts a TWL state to the corresponding W state:

definition $\mathsf{state}_{\mathsf{W}}\_\mathsf{of} :: {'v\ state}_{\mathsf{TWL}} \Rightarrow {'v\ state}_{\mathsf{W}}$ **where**
$\mathsf{state}_{\mathsf{W}}\_\mathsf{of}\ (M, N, U, D, NP, UP, WS, Q) =$
$(M, \mathsf{image}\ \mathsf{clause}_{\mathsf{W}}\_\mathsf{of}\ N \uplus NP, \mathsf{image}\ \mathsf{clause}_{\mathsf{W}}\_\mathsf{of}\ U \uplus UP, D)$

where $\mathsf{clause}_{\mathsf{W}}\_\mathsf{of}\ (\mathsf{Clause}_{\mathsf{TWL}}\ W\ UW) = W \uplus UW$.

The first two TWL rules have direct counterparts in W:

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

Conflict $(M, N, U, \top, NP, UP, \{(L, C)\} \uplus WS, Q) \Longrightarrow_{\mathsf{TWL}} (M, N, U, C, NP, UP, \emptyset, \emptyset)$
if 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 the 2WL invariant trivially holds for C while an update on C is pending.

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

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

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

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

As in W+stgy, we postpone decisions. This is achieved by requiring that WS and Q are empty in the Decide rule. And due to the distinction between unit and nonunit clauses, we need two rules for nonchronological backjumping:

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

Backjump_Nonunit $(M' \cdot K^{\dagger} \cdot M, N, U, C, NP, UP, \emptyset, \emptyset) \Longrightarrow_{\mathsf{TWL}}$
$(L^{D \vee L} \cdot M, N, U \uplus \{D \vee L\}, \top, NP, UP, \emptyset, \{L\})$
if $D \neq \bot$ and the conditions on Backjump are satisfied for $D \vee L$

Backjump_Unit $(M' \cdot K^{\dagger} \cdot M, N, U, C, NP, UP, \emptyset, \emptyset) \Longrightarrow_{\mathsf{TWL}}$
$(L^{L} \cdot M, N, U, \top, NP, UP \uplus \{L\}, \emptyset, \{L\})$
if the conditions on Backjump are satisfied for L

Theorem 2 (Invariant [10, *cdcl_twl_stgy_twl_struct_invs*]) *If the state S satisfies the 2WL invariant and $S \Longrightarrow_{\mathsf{TWL}} T$, then T satisfies the 2WL invariant.*

Theorem 3 (Refinement [10, *full_cdcl_twl_stgy_cdcl$_W$_stgy*]) *Let S be a state that satisfies the 2WL invariant. If $S \Longrightarrow^{!}_{\mathsf{TWL}} T$, then $\mathsf{state}_{\mathsf{W}}\_\mathsf{of}\ S \Longrightarrow^{!}_{\mathsf{W}} \mathsf{state}_{\mathsf{W}}\_\mathsf{of}\ T$.*

TWL refines W+stgy's end-to-end behavior and produces final states that are also final states for W. We can apply Theorem 1 to establish partial correctness. Termination of TWL is a direct consequence of the termination of W.

4 Refining to an Algorithm

Our goal is to obtain a SAT solver implementation from the calculus. For this purpose, we refine the calculus in multiple consecutive steps to an implementation.

The Isabelle Refinement Framework [16, 17, 19] provides a tool chain for program development via stepwise refinement. It is based on the *nondeterminism monad* over the datatype $'a\ nres = \mathsf{FAIL} \mid \mathsf{RES}\ 'a$. If the program has an execution that diverges or

triggers an assertion, its result is FAIL; otherwise, the result is RES X, where X is the set of possible return values. The function RETURN x, which abbreviates RES $\{x\}$, returns the value x; bind m f nondeterministically chooses a return value from m and applies f to it; and REC F x executes the recursive function with body F and initial argument x. Based on these constructs and Isabelle/HOL's standard 'if–then–else' and 'case' expressions, the Refinement Framework defines higher-level constructs such as 'while' and 'for each' loops. The Haskell-style 'do' monadic notation is also supported.

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

```
definition
  PCUI_algo :: 'v lit × 'v clause ⇒ 'v state_TWL ⇒ 'v state_TWL
where
  PCUI_algo LC S = do {
    let (M,N,U,D,NP,UP,WS,Q) = S;
    let (L,C) = LC;
    L' ← RES {L'. L' ∈ watched C − {L}};
    if L' ∈ M then
      RETURN S                                              (* Ignore *)
    else
      if ∀L ∈ unwatched C. − L ∈ M
        if − L' ∈ M
          RETURN (M,N,U,C,NP,UP,∅,∅)                        (* Conflict *)
        else
          RETURN (L'^C · M,N,U,D,NP,UP,WS,{−L'} ⊎ Q)        (* Propagate *)
      else do {
        K ← RES {K. K ∈ unwatched C ∧ −K ∉ M}
        (N',U') ← RES {(N',U'). update_clauses (N,U) C L K (N',U')}
        RETURN (M,N',U',D,NP,UP,WS,Q)                       (* Update *)
      }
  }
```

In the above definition, the predicate update_clauses (N,U) C L K (N',U') updates the clause C by exchanging the watched literal L and the unwatched literal K in C. The clause is updated in N and U to get N' and U'. Since propagations are always performed immediately, WS never contains unit clauses. This algorithm still contains abstract, nondeterministic parts. For example, in the Update part, we leave the choice of the new watched literal K underspecified.

To specify refinement between two programs, the Refinement Framework defines a partial order $\leq$ on $'a\ nres$ by lifting the subset relation with FAIL being the greatest element—i.e., RES $X \leq$ RES Y if and only if $X \subseteq Y$ and $r \leq$ FAIL for all r. We also use this order to state that a program is correct: The statement $P\ x \Longrightarrow f\ x \leq$ RES $\{y.\ Q\ y\}$

specifies the total correctness of program f with precondition P and postcondition Q. For the four-rule program presented above, we have the following refinement theorem:

Lemma 4 (Refinement [10, *unit_propagation_inner_loop_body_add*]) *If the 2WL invariant holds for all clauses occurring in the NU component of S, then*

$$\mathsf{PCUI}_{\mathsf{algo}}\ LC\ S \leq \mathsf{RES}\ \{T.\ \mathsf{add\_to}_{WS}\ LC\ S \Longrightarrow_{\mathsf{PCUI}} T\}$$

PCUI is the fragment of TWL consisting of the Propagate, Conflict, Update, and Ignore rules. The function $\mathsf{add\_to}_{WS}\ LC\ S$ returns the state obtained from S by adding LC to the WS component. For the entire SAT solver, we proved the following theorem:

Theorem 5 (Refinement [10, *cdcl_twl_stgy_prog_spec*]) *If the 2WL invariant holds for all clauses occurring in the NU component of S, then*

$$\mathsf{TWL}_{\mathsf{algo}}\ S \leq \mathsf{RES}\ \left\{T.\ S \Longrightarrow^{!}_{\mathsf{TWL}} T\right\}$$

The state returned by the program is a final state for TWL. From Theorem 3, we deduce that it is also a final state for W+stgy. Hence, the program $\mathsf{TWL}_{\mathsf{algo}}$ is a SAT solver by Theorem 1.

5 Representing Clauses as Lists

The program presented in the previous section uses the same abstract state as the calculus. The next refinement step changes the state's representation: We store the initial and learned clauses in a list and use indices to refer to the clauses. States are tuples $(M, NU, u, D, NP, UP, WS', Q)$:

- NU is the list of all nonunit clauses. It simultaneously refines N and U. The initial clauses occupy indices 1 to $u-1$, and the learned clauses start at index u. The first element of the list is not used to keep index 0 as a null clause reference.
- M is the trail, where the annotations are replaced by an index. For nonunit clauses, the annotation is the index in NU: L^i is used instead of L^C if $NU\,!\,i = C$, where $NU\,!\,i$ is the $(i+1)$st element of NU. When annotating literals with unit clauses, which are not present in NU, we use index 0—i.e., we put L^0 instead of L^L on the trail.
- In WS', we implement a pair (L, C) by the index of clause C. The literal L, which is the same for all pairs in WS, is stored locally in the refined propagation algorithm.

From now on, we will use the letter C to refer to clause indices and will not insist on the distinction between a clause and the index that refers to it.

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

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

```
definition
  PCUIlist :: 'v lit ⇒ 'v clause_idx ⇒ 'v statelist ⇒ 'v statelist
where
  PCUIlist L C S = do {
    let (M,NU,u,D,NP,UP,WS,Q) = S;
    let i = (if (NU ! 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,u,D,NP,UP,WS,Q)                          (* Ignore *)
    else
      case find_unwatched M (NU ! C) of
        None ⇒
          if pol' = Some False then
            RETURN (M,NU,u,NU ! C,NP,UP,∅,∅)                (* Conflict *)
          else                                              (* Propagate *)
            RETURN (L'^C · M,NU,u,D,NP,UP,WS,{−L'} ⊎ Q)
      | Some j ⇒ do {
          let NU' = list_update NU C (list_swap NU ! C i j);
          RETURN (M,NU',u,D,NP,UP,WS,Q)                     (* Update *)
        }
  }
```

Refinement between a concrete program f and an abstract program g is expressed by a statement $\forall x\, y.\ (x,y) \in R \Longrightarrow f\ x \leq \Downarrow_S\ g\ y$, where R is a relation between concrete and abstract arguments, and S is a relation between concrete and abstract results. The function $\Downarrow_R$ maps an abstract result to the largest concrete result whose values are related, by R, to abstract values. Two edge cases are $\Downarrow_R$ FAIL = FAIL and $\Downarrow_{\mathsf{Id}}\ r = r$. The former implies that a failing assertion is refinable by any program, so we can assume in the refinement proof that assertions on the abstract side never fail. The latter makes correctness a special case of refinement, the abstract program being of the form RES $\{x.\ Q\ x\}$.

The Refinement Framework includes a verification condition generator. Hoare-logic-style rules are used to prove correctness goals. For other refinement goals, a heuristic tries to align the concrete and abstract program structure and generates refinement goals for corresponding statements of the two programs. In the refinement proof of our PCUI algorithm, the goal for the definition of the other watched literal L' is $(NU\,!\,C)\,!\,(1-i) \in$ watched $C - \{L\}$. Intuitively, this holds because for $i \in \{0,1\}$, the expression $1-i$ returns the index of the other watched literal.

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

6 Storing Clauses Watched by a Literal: Watched Lists

In the Next_Literal rule, the set of all clauses that watch a given literal is calculated. The next refinement step eliminates this gratuitous inefficiency: Instead of iterating over all clauses, we maintain a map from literals to clauses that watch these literals. States are of the form $(M, NU, u, D, NP, UP, Q, W)$, where $W :: 'v\ lit \Rightarrow clause\_idx\ list$ maps a literal to its watched list.

For the current literal L, the abstract state stores all clauses that watch L and still require processing in the *WS* component. In the concrete algorithm, we use a local index variable w as iterator over the watched list. After processing a clause, there are two cases. If the clause still watches L (rules Propagate, Conflict, and Ignore), we increment w to consider the next clause. Otherwise, the clause no longer watches L (rule Update). We exchange the element at index w with the last element of the watched list and shorten the list by one (function delete_index_and_swap). As the iteration order on the watched list is irrelevant, this is an efficient way to delete an element in constant time based on arrays. This technique is found in many SAT solvers.

The refined PCUI algorithm is presented below, where $W(a := b)$ denotes the function that maps a to b and otherwise coincides with W:

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

In general, when performing a chain of refinements, we want to reuse information from the earlier refinement steps to carry out the later refinement steps. Consider a func-

tion α mapping concrete states to abstract states and an invariant I_{abs} on abstract states. The invariant I on concrete states usually consists of a part I_{new} specific to the newly introduced concrete structure and a part I_{old} inherited from higher abstraction levels—i.e., we have $I_{\text{abs}}\ (\alpha\ s) \Longrightarrow I_{\text{old}}\ s$. Refinement between a concrete function f and an abstract function g is expressed as $f\ x \leq \Downarrow_{\{(s,\alpha\ s)\,|\,s.\ I\ s\}}\ g\ y$. However, since g preserves the abstract invariant I_{abs}, it is sufficient to prove the weaker goal $f\ x \leq \Downarrow_{\{(s,\alpha\ s)\,|\,s.\ I_{\text{new}}\ s\}}\ g\ y$. Because we frequently needed this technique for large invariants, we wrote a tactic in the Eisbach language [23] to synthesize a candidate I_{cand} for I given I_{new} and I_{abs}. It transforms the goal into $I_{\text{cand}}\ s \Longrightarrow I\ s$, which can usually be discharged using standard Isabelle tactics. Eisbach was very useful for such tedious but straightforward tasks.

7 Generating Code

In a next refinement step, we introduce imperative data structures. Some of the required data structures are already available in the Imperative Collections Framework [18], while others must be developed specifically for this project. Then we use the Sepref tool [17] to automatically replace the operations in the abstract algorithm by their concrete counterparts. The result is a deterministic program in the heap monad of Imperative HOL [5] and a refinement theorem linking it to the original abstract program.

The last step of our development is to use the Isabelle/HOL code generator [12] to generate Standard ML code from the Imperative HOL program. The ML program is extended with a simple unverified parser for SAT problems in conjunctive normal form. We call the resulting solver IsaSAT.

So far, our programs have computed the polarity of an atom by traversing the trail, a list of literals. This is very inefficient. In contrast, efficient SAT solvers employ a map from atoms to their polarity. We integrate this optimization into the imperative data structure used for the trail. To develop this new data structure, we again use stepwise refinement. This refinement is isolated from the rest of the development, which only relies on its final result: an efficient implementation of the trail and its operations. As Lammich observed before [18], this kind of modularity is invaluable when designing complex data structures.

In a first refinement step, we restrict the type of atoms to natural numbers and add a list of polarities (of type *bool option*), such that the $(i+1)$st element gives the polarity of atom i. The polarity function introduced in Section 5 is then implemented as follows:

definition $\mathsf{polarity}_{\mathsf{list\_pair}}$
  :: *nat lit* $\Rightarrow$ (*nat*, *clause_idx*) *ann_lit list* $\times$ *bool option list* $\Rightarrow$ *bool option*
where
  $\mathsf{polarity}_{\mathsf{list\_pair}}\ L\ (M, Ls) =$ case Ls ! $\mathsf{atm\_of}\ L$ of
    None $\Rightarrow$ None
  | Some $b \Rightarrow$ Some (if $\mathsf{is\_pos}\ L$ then b else $\neg b$)

The Sepref tool is based on a separation logic with assertion type *assn*. It can be used to express relations $'a \Rightarrow 'b \Rightarrow$ *assn* between plain values, of type $'a$, and data structures on the heap, of type $'b$. The tool generates code using the imperative data structures according to a refinement relation specified by the user.

In a subsequent refinement step, we use Sepref to implement the list of polarities by an array and atoms by 32-bit integers. Accordingly, we define two auxiliary relations:

- The relation lit_assn :: $(nat, clause\_idx)\, ann\_lit \Rightarrow (uint32, clause\_idx)\, ann\_lit \Rightarrow assn$ refines an annotated literal with natural number atoms by an annotated literal with a 32-bit unsigned integer (type *uint32*). The clause indices are kept as unbounded natural numbers.
- The relation trail_list_pair_assn :: $(nat, clause\_idx)\, ann\_lit\ list \times bool\ option\ list \Rightarrow (uint32, clause\_idx)\, ann\_lit\ list \times bool\ option\ array \Rightarrow assn$ refines the trail data structure. The list of Booleans is refined by an array with the same content.

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

$$\begin{aligned}&(\mathsf{polarity\_code}, \mathsf{RETURN} \circ \mathsf{polarity}_{\mathsf{list\_pair}})\\ &\quad \in [\lambda(M,L).\ L \in N_1]\ \mathsf{trail\_list\_pair\_assn}^{\mathrm{k}} \times \mathsf{lit\_assn}^{\mathrm{k}} \to \mathsf{id\_assn}\end{aligned}$$

The term in square brackets specifies the precondition for the refinement: The literal must be among the valid literals for the current formula (N_1). This is required to ensure that the array indexing is in bounds. The term between the closing square bracket and the arrow states the refinements for the parameters, i.e., the trail and the literal. The $^{\mathrm{k}}$ ("keep") annotations indicate that the imperative program will not modify the arguments on the heap. The term after the arrow is the refinement for the result, which is trivial here because the data structure remains *bool option*.

Combining the two refinement steps yields the following theorem, which is used by Sepref to refine the programs that use trails:

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

where trail_assn combines the two refinement relations for trails.

Once we have refined TWL to an imperative algorithm and combined it with a function initializing the data structure from a list of clauses, we define the complete imperative SAT solver as a function IsaSAT_code in Imperative HOL. It takes as arguments a list of clauses and returns a Boolean indicating whether the clauses are satisfiable. By combining the refinement theorems for all refinement steps, we obtain the correctness theorem for the complete solver.

Theorem 6 (End-to-End Correctness [10, *SAT_wl_code_full_correctness*]) *The imperative SAT solver returns whether its input is satisfiable:*

$$\begin{aligned}&(\mathsf{IsaSAT\_code}, \mathsf{RETURN} \circ \mathsf{is\_satisfiable})\\ &\quad \in [\mathsf{no\_duplicate\_no\_false\_no\_tautology}]\ \mathsf{clauses\_assn}^{\mathrm{k}} \to \mathsf{bool\_assn}\end{aligned}$$

Here, clauses_assn refines a multiset of multisets of literals to a list of lists.

Finally, we invoke the code generator to translate IsaSAT_code to Standard ML. To give a flavor of the output, the code generated for the solver's main loop is presented below (in a slightly modified form to increase readability):

```
fun TWL_code n_0 initial_state = fn () =>
  let val (_, final_state) =
    heap_WHILET (fn (finished, _) => fn () => not finished)
      (fn (_, T) => analyze_or_decide_code n_0
         PCUI_and_Next_Literal T ()) ())
      (false, initial_state) ()
  in final_state end
```

The parameter `n_0` is a list of literals such that every atom of the problem appears at least once The list `n_0` is used to make the Decide rule deterministic. The ML idiom `(fn () => ...)` is inserted by the code generator to ensure that side effects of functions occur in the intended order.

8 Evaluation

To find out how efficient IsaSAT is, we compared it with the verified solver `versat` [26]; the fastest imperative OCaml solver we know of, DPT [11]; the well-known MiniSat [9]; and the state-of-the-art solver Glucose [1].

`versat`, by Oe et al. [26], is specified and verified using the Guru proof assistant [30], which can generate C code. Optimized data structures are used, including for watched literals and conflict analysis. However, termination is not guaranteed, and model soundness is proved trivially by means of a run-time check of the models; if this check fails, the solver's outcome is "unknown." Regrettably, neither `versat` nor Guru appear to be available anymore. To compare our solver with `versat`, we had to proceed indirectly: We chose the same benchmarks as Oe et al. and the PicoSAT [2] version they used (936) and extrapolated from the figures we measured on modern hardware. To reduce the impact of PicoSAT's nondeterminism, we computed the average of eight runs.

The problems that either IsaSAT or `versat` solves are listed below. The solving time is given in seconds; a dash (–) indicates the solver did not return within 3600 s.

Problem	IsaSAT	`versat`	DPT	PicoSAT	MiniSat	Glucose
`itox_vc965`	0.5	2	2.5	0.2	0.1	0.2
`eq.atree.braun.7.unsat`	12.7	17	12.7	2.8	2.5	1.7
`eq.atree.braun.8.unsat`	131.7	270	291.1	17.9	19.3	12.4
`eq.atree.braun.9.unsat`	990.5	–	1763.0	74.5	96.5	48.6
`dspam_dump_vc973`	–	3195	–	0.5	6.7	1.1
`total-5-11-u`	–	461	–	23.7	9.3	5.8
`AProvE07-15`	813.2	–	837.7	52.8	25.0	6.9
`manol-pipe-c10nidw_s`	–	313	279.9	10.3	10.5	8.9

The performance of our solver is similar to `versat`'s, but the results are inconclusive due to the lack of benchmarks.

In addition, we ran all solvers except `versat` on the 116 problems classified easy or medium from the SAT Competition 2009, with a time limit of 900 s. Glucose solves all 116 problems in 52 s on average. MiniSat solves 114 problems, spending 88 s on average per problem it solves; DPT solves 61 problems in 197 s on average. IsaSAT

solves 19 problems in 70 s on average. These tests were carried out on a Xeon E5-4640 with 512 GB of memory, with Intel Turbo Boost deactivated. They show that Glucose, MiniSat, and PicoSAT are much faster than the other solvers and that DPT is substantially faster than IsaSAT and solves more instances.

There are several reasons explaining why IsaSAT is slower. First, it does not have restarts nor conflict minimization. Restarts would allow it to explore another part of the search space, and conflict minimization produces smaller learned clauses. Moreover, heuristics are crucial: We use a static ordering of the variables to do decisions, whereas most SAT solvers use the VSIDS (variable state independent decaying sum) heuristic [24], which builds a dynamic ordering.

Another issue is that Isabelle/HOL can only generate code in (impure) functional languages, whereas unverified SAT solvers are usually written in C++. To ensure memory safety, functional languages check array bounds at run time. Also other features, such as the arbitrary precision arithmetic used for Isabelle/HOL programs, tend to be less efficient than their C++ counterparts.

To reduce these effects, we implemented literals by 32-bit integers, which required some extra work to prove absence of overflows. This increased the speed of our solver by a factor between two and four. Nevertheless, profiling data indicates that our solver still spends up to 10% of its time on operations related to arbitrary precision integers.

9 Related Work

The closest formalizations to ours are `versat` by Oe et al. [26] and Marić's [21, 22]. Marić verified a CDCL-based solver in Isabelle/HOL, including two watched literals, as a purely functional program. His methodology is quite different from ours, as it does not include refinement. While he was able to generate code, the export does not work anymore. We expect the performance to be substantially worse than ours, since he uses lists instead of arrays. Beyond Oe et al. and Marić, there are several formalizations of CDCL that do not include watched literals, including Lescuyer's in Coq [20] and Shankar and Vaucher's in PVS [27], and also some formalizations of DPLL.

Instead of verifying a SAT solver, another way to obtain highly trustworthy solutions is to have the solver produce a certificate, which can be processed by a checker. While certificates for satisfiable formulas are simply a valuation of the variables and can be easily generated and checked, certificates for unsatisfiable formulas are more complicated. The de facto standard format is DRAT (deletion resolution asymmetric tautology) [13], which can be easily generated by solvers. The standard DRAT certificate checker [33] is, however, an unverified C program. There is also a verified DRAT checker [32], but it does not scale to large problems. There is some unpublished research [6,7,15], including by Lammich, indicating that it is possible to have efficient verified checkers.

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

10 Conclusion

We have extended our Isabelle/HOL framework for CDCL with a verified imperative SAT solver. Refinement-based development is flexible and makes later changes in the development easier. We expect that many heuristics and optimizations can be added with only small modular changes to the existing proofs. For example, an initial version of our SAT solver did not include the trail optimization presented in Section 7. Adding it required changes only to the last refinement step.

The refinement steps are an interesting case study of the Refinement Framework, Imperative HOL, and the code generator. In future work, we plan to add restarts and a more sophisticated decision heuristic. We also want to formalize CDCL(T), the theory behind satisfiability-modulo-theories (SMT) solvers, and other extensions such as MaxSAT.

Acknowledgement Max Haslbeck, Anders Schlichtkrull, and Mark Summerfield suggested 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] Audemard, G., Simon, L.: Glucose 2.1: Aggressive—but reactive—clause database management, dynamic restarts. In: Workshop on the Pragmatics of SAT 2012 (2012)
[2] Biere, A.: PicoSAT essentials. JSAT 4(2-4), 75–97 (2008)
[3] Blanchette, J.C., Fleury, M., Lammich, P., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality, `http://matryoshka.gforge.inria.fr/pubs/sat_article.pdf`, Submitted
[4] 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 (2016)
[5] 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 (2008)
[6] Cruz-Filipe, L., Heule, M., Jr., W.A.H., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. CoRR abs/1612.02353 (2016), `http://arxiv.org/abs/1612.02353`
[7] Cruz-Filipe, L., Marques-Silva, J., Schneider-Kamp, P.: Efficient certified resolution proof checking. CoRR abs/1610.06984 (2016), `http://arxiv.org/abs/1610.06984`
[8] Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM 5(7), 394–397 (1962)
[9] Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer (2003)
[10] 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
[11] Goel, A., Grundy, J.: Decision Procedure Toolkit, `http://dpt.sourceforge.net/`
[12] 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)

[13] 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)
[14] Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean Termination Tool 2. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 295–304. Springer (2009)
[15] Lammich, P.: GRAT—Efficient formally verified sat solver certification toolchain, http://www21.in.tum.de/~lammich/grat/
[16] Lammich, P.: Automatic data refinement. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 84–99. Springer (2013)
[17] Lammich, P.: Refinement to Imperative/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 253–269. Springer (2015)
[18] Lammich, P.: Refinement based verification of imperative data structures. In: Avigad, J., Chlipala, A. (eds.) CPP 2016. pp. 27–36. ACM (2016)
[19] Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Beringer, L., Felty, A.P. (eds.) ITP 2012. LNCS, vol. 7406, pp. 166–182. Springer (2012)
[20] Lescuyer, S.: Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. Ph.D. thesis, Université Paris-Sud (2011)
[21] Marić, F.: Formal verification of modern SAT solvers. Archive of Formal Proofs (2008), http://isa-afp.org/entries/SATSolverVerification.shtml, Formal proof development
[22] Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theor. Comput. Sci. 411(50), 4333–4356 (2010)
[23] Matichuk, D., Murray, T.C., Wenzel, M.: Eisbach: A proof method language for Isabelle. J. Autom. Reasoning 56(3), 261–282 (2016)
[24] 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)
[25] 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)
[26] 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)
[27] Shankar, N., Vaucher, M.: The mechanical verification of a DPLL-based satisfiability solver. Electr. Notes Theor. Comput. Sci. 269, 3–17 (2011)
[28] Sternagel, C.: The generalized subterm criterion in TTT_2. CoRR abs/1609.03432 (2016), http://arxiv.org/abs/1609.03432
[29] Sternagel, C., Thiemann, R.: An Isabelle/HOL formalization of rewriting for certified termination analysis, http://cl-informatik.uibk.ac.at/software/ceta/
[30] Stump, A., Deters, M., Petcher, A., Schiller, T., Simpson, T.W.: Verified programming in Guru. In: Altenkirch, T., Millstein, T.D. (eds.) PLPV 2009. pp. 49–58. ACM (2009)
[31] 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)
[32] Wetzler, N., Heule, M.J.H., Hunt, W.A.: Mechanical verification of SAT refutations with extended resolution. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 229–244. Springer (2013)
[33] Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 422–429. Springer (2014)