A More Pragmatic CDCL for IsaSAT and Targetting LLVM (Short Paper)

Mathias Fleury[1,2,3(✉)] and Peter Lammich[4,5]

[1] University Freiburg, Freiburg, Germany
fleury@cs.uni-freiburg.de
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[3] Johannes Kepler University Linz, Linz, Austria
[4] University of Manchester, Manchester, UK
[5] University of Twente, Twente, Netherlands
p.lammich@utwente.nl

Abstract. IsaSAT is the most advanced verified SAT solver, but it did not yet feature inprocessing (to simplify and strengthen clauses). In order to improve performance, we enriched the base calculus to not only do CDCL but also inprocess clauses. We also replaced the target of our code synthesis by Isabelle/LLVM. With these improvements, we can solve 4 times more SAT Competition 2022 problems than the original IsaSAT version, and 4.5 times more problems than any other verified SAT solver we are aware of. Additionally, our changes significantly reduce the trusted code base of our verification.

1 Introduction

SAT solving is a very important tool that has been extensively used in various applications like mathematics or cryptography. To ensure the correctness of the answer provided by a SAT solver, there are two approaches: either producing a certificate that can be checked independently or verifying a SAT solver. The first approach has been extensively studied and works very well in practice [19,26,28] – only checked proofs are counted in the SAT Competition [2].

The second approach, i.e., verifying a whole SAT solver is orders of magnitudes more complex than checking a certificate. To this end, the goal of the IsaFoL (Isabelle Formalization of Logic) [3] effort is to develop methodology and libraries for formalizing modern research in automated reasoning. In this context, we have verified a CDCL calculus (conflict-driven clause learning) and a two-watched literals data structure (Sect. 2). To show that they are useful, we have developed the verified SAT solver IsaSAT [8], which we later optimized [12]. To our surprise, it won the EDA Challenge 2021 defeating all the non-verified solvers, but, as expected, it finished last at the SAT Competition 2022 [2]. However, the former used a much shorter timeout (200 s, not announced before the competition) whereas the latter uses 5000 s.

In this paper, we present our new developments in IsaSAT, which make our solver arguably the most advanced formally verified SAT solver to date: inprocessing and verifying fast LLVM code [20] rather than slow functional code.

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 207–219, 2023.
https://doi.org/10.1007/978-3-031-38499-8_12

Inprocessing is a critical feature of modern SAT solvers (e.g., every winner of the SAT Competition since 2013 includes it). In order to use it in our formally verified solver, we had to extend our verified CDCL calculus: Our new PCDCL calculus includes features to encompass various inprocessing techniques, even if we have not yet implemented every possible technique (Sect. 3).

We generate IsaSAT by exporting a model in the interactive theorem prover Isabelle [22] to executable code. Earlier we used Isabelle's default code generator to export to Standard ML (SML). However, the performance was not sufficient – especially memory consumption was very high. Thus, we switched to Isabelle/LLVM [18], which generates LLVM intermediate representation (LLVM IR). Apart from allowing faster imperative code, it also reduced the trusted code base (Sect. 4), replacing the rather niche MLton [27] compiler by only the backend of the widely used LLVM.

Porting our entire development to Isabelle/LLVM required some changes and some cleanup. Moreover, when we implemented and verified inprocessing, we realized that some design decisions need to be improved. In Sect. 5, we report on our experiences and lessons learned while porting and extending IsaSAT.

Finally, we have benchmarked IsaSAT on the problems from the SAT Competition 2022. We show that just porting IsaSAT from SML to Isabelle/LLVM significantly improved the performance, and the new inprocessing techniques combined with heuristic improvements give us another significant increase, demonstrating the usefulness of our PCDCL calculus (Sect. 6).

This presentation is an extended version of our (non-peer-reviewed) system description from the EDA Challenge 2021 [13] and the SAT Competition 2022 [6]. Compared to that version, we have provided much more details on PCDCL, our experience porting the development to LLVM, and performance tests.

2 Preliminaries

CDCL. CDCL is a procedure that builds a partial assignment called *a trail* either by guessing (called *deciding*) or propagating information. If the partial assignment is a model, the algorithm stops. If there is a conflict between the partial assignment and a clause, the partial assignment is repaired and a new clause is learned. For more details (beyond the scope of this paper), we refer the reader to the *Handbook of Satisfiability* [7].

We use a transition system for our formalization of CDCL [8]. Its state consists of the trail M, the (multi)sets of initial and learned clauses (N and U), and the conflict clause to analyze (or None if there is none). We show one rule, `decide`, that adds L to the current assignment M:

```
inductive decide :: 'st ⇒ 'st ⇒ bool where
  undefined_lit M L ⟹ |L| ∈ |N| ⟹
  decide (M, N, U, None) (L · M, N, U, None)
```

If no conflict has been found so far (`None`), we add the new literal L at the beginning of the trail M. We prove that our set of rules is terminating and correct [8].

Code Synthesis. To generate the IsaSAT code, we start from the abstract rules like `decide` and gradually refine it to some deterministic functions using the Refinement Framework [16]. Then, we rely on Sepref [17] to synthesize code: It takes an (Isabelle) function and synthesizes a new version, replacing functional data structures (like lists) by imperative data structures (like arrays). There are two versions of the tool. The older version, which we used before [8,12], uses Imperative HOL [9] and Isabelle's standard (trusted) code generator [14] to export code into various functional languages. We used Standard ML (SML) with the compiler MLton [27], because it offers (by far) the best performance for our use case. The new Sepref is part of the Isabelle/LLVM library (developed by the second author) and generates LLVM IR from a model of LLVM IR inside the theorem prover. The code generator interprets a shallow embedding of Isabelle/LLVM as equivalent to similar looking LLVM code. This reduces the trusted code base in two ways: first, the trusted pretty printer is simpler, and, second, instead of the rather niche full compiler MLton, we use only the backend of the widely used LLVM [20].

The biggest difference is that Imperative HOL allows arbitrary large arrays and integers, whereas Isabelle/LLVM is more realistic, requiring integers (in particular array offsets, see Sect. 5.1) to have a fixed bit-width.

Related Work. Our goal is to produce a fully verified SAT solver, without any runtime checks, that both *terminates* and returns a *correct model* while using efficient data structures. No other solver achieves all three goals. The SAT solver TrueSAT from Andrici and Ciobaca [1] relies on the original DPLL and uses less efficient data structures (including counters instead of watch lists), but it terminates. Historically, this would roughly correspond to SAT solver from the early 90s. However, it only uses stateless heuristics, and it is not clear if the approach can be extended to CDCL (where the solver learns and keeps new clauses) or to stateful heuristics (like VSIDS [21]). The solvers versat [23] and Creusat [25] go into a similar direction with CDCL instead of DPLL, but prove a weaker correctness property: they only show that an UNSAT result is correct, while a SAT result requires an additional check. Also, termination is not proved. Only proving this partial property makes many proofs considerably easier, in particular adding restarts. Oe et al's solver versat [23] was the first partially verified solver that could run benchmarks from the SAT Competition. More recently, Skotåm [25] has verified in his Master's thesis the SAT solver CreuSAT using the Creusot framework (relying on Why3 internally). While CreuSAT is much faster than versat in our tests, its correctness relies on (trusted) SMT solvers, and the proofs are not checked by a small kernel like our Isabelle code. However, the verification also takes much less time (a few minutes compared to several hours).

Modern SAT solvers use inprocessing to make the subsequent CDCL run heuristically faster [15]. In particular, clauses are strengthened and global transformation (e.g., to remove variables) are applied. Two techniques (that we do not support), variable elimination and addition, slowly change the models of the formula by changing the set of variable. The SAT solver then reconstructs a

model of the original formula at the end. Fazekas et al. [11] made it compatible with incremental SAT solving. All others inprocessing technique fit into our extended CDCL described in the next section.

3 Pragmatic CDCL for Inprocessing

SAT solvers nowadays apply a combination of CDCL (most of the time) and inprocessing (sometimes). Therefore, we extended our calculus similarly. At the core, we have our terminating CDCL. We also allow for formula transformation and restarts. We call the combination *pragmatic CDCL* or PCDCL.

Splitting the Clause Set. Inprocessing makes it possible to strengthen and simplify clauses. However, we want models from the final set of clauses to remain models from the initial set of clauses. Deleting clauses is not possible: if we start with the clauses $A \vee C$ and $B \vee \neg B$, removing the tautology means that the model A of $A \vee C$ is not a model of the initial clause set anymore. Hence we want to keep the literal B without considering the tautology for propagation/conflict.

To solve the issue we split our set of clauses into two parts: clauses that are useful for propagation and clauses that can be ignored but are kept for their literals. Thus we keep the set of all literals $\mathcal{A}$ constant. For our proof of refinement to the original CDCL, we have to make sure that the new behavior is also possible in the original calculus – in particular we do not miss propagations or conflicts. In the case of tautologies, this is simple (they are never used). If we consider subsumption, like $A \vee B$ subsumes $A \vee B \vee C$, whenever the latter propagates, then the former is a conflict. Therefore, the behavior is compatible.

While the idea of splitting our clauses seems surprising, the additional clause sets are only required for the connection to our CDCL transition system, and we entirely remove them when generating the code. Moreover, the refinement is easier as we do not have to update our heuristics to remove literals (and potentially shorten arrays). Finally, this is similar to the behavior of SAT solvers like Kissat [4]: while the clauses are removed, all literals of the problem are set.

In our original refinement, we have split the clauses to distinguish between clauses of length 1 (where we cannot distinguish two distinct literals and thus they cannot fit into our two-watched literals data structures) and longer clauses, but the aim was only distinguishing on the length.

One important point to notice is that the role of our clause sets changes. In our original CDCL, N was the (immutable) set of initial clauses and U contains the redundant clauses that can be removed at any point: N ensures that we do gain new models during our transformations. Now, the set changes: strengthening an irredundant clause from N also shortens the clause that is in there. Therefore, a naive version could remove literals.

Overall we have 4 sets of clauses: the irredundant clauses N and the redundant U clauses, and each one is divided into the active clauses (N_a and U_a) and the inactive (discarded) clauses (N_d and U_d). For example, tautologies or subsumed clauses are discarded, but remain in N, so literals are never removed. In our development there are actually three sets (containing a literal set at level 0

or tautologies, subsumed clauses, and false clauses) to reduce the number of case distinction in some proofs. We never demote irredundant clauses to redundant ones, but we can promote them.

Inprocessing Rules. Our aim when picking the rule is to be general (like we can learn any useful clause) and then we specialize rules to specific techniques. We will show this with the example of subsumption-resolution [7]. When doing subsumption-resolution, we resolve two clauses together if the conclusion is shorter. Then we can remove either one or both of the antecedents. For example, resolving $A \vee B \vee C$ with $A \vee \neg C$ produces the clause $A \vee B$ with subsumes the former clause. If the latter clause was $A \vee B \vee \neg C$, the resolved clauses would actually subsume both clauses.

One of the most important inprocessing rule learns any possible clause. To simplify the presentation, we will only give the rules operating on the learned clauses, but similar rules exists for the initial set of clauses.

$$\begin{aligned}&\texttt{inductive}\ \mathsf{cdcl\_learn\_clause} :: \textit{'prag\_st} \Rightarrow \textit{'prag\_st} \Rightarrow \textit{bool}\ \texttt{where}\\ &\quad |C| \subseteq |N + N_d| \Longrightarrow \mathsf{count\_decided}\, M = 0 \Longrightarrow\\ &\quad N \wedge N_d \vDash C \Longrightarrow \neg\mathsf{tautology}\, C \Longrightarrow \mathsf{distinct}\, C \Longrightarrow\\ &\quad \mathsf{cdcl\_learn\_clause}\ (M, N, U, \mathsf{None}, N_d, U_d)\\ &\qquad\qquad (M, N, U \wedge C, \mathsf{None}, N_d, U_d)\end{aligned}$$

The side conditions not only include that the clause is entailed and duplicate-free, but also the clause is not a tautology and we do not break CDCL invariants ($\mathsf{count\_decided}\, M = 0$). Then we can deactivate subsumed clauses:

$$\begin{aligned}&\texttt{inductive}\ \mathsf{cdcl\_subsumed} :: \textit{'prag\_st} \Rightarrow \textit{'prag\_st} \Rightarrow \textit{bool}\ \texttt{where}\\ &\quad C \subseteq D \Longrightarrow \mathsf{count\_decided}\, M = 0 \Longrightarrow\\ &\quad \mathsf{cdcl\_subsumed}\ (M, N, U \wedge C \wedge D, \mathsf{None}, N_d, U_d)\\ &\qquad\qquad (M, N, U \wedge C, \mathsf{None}, N_d, D \wedge U_d)\end{aligned}$$

We combine these rules to express subsumption-resolution: We first learn the clause obtained by resolution. Then we can remove the antecedents. If either antecedent is in N, we also have promoted the conclusion from N to U. The advantage of our approach is that we can express other inprocessing techniques without adding new rules, only by specializing them.

Overall we have 9 rules with some overlap with CDCL (propagation and conflict), but mostly simplification of clauses (removing true clauses and false literals from clauses) and pure literal deletion: When a literal always appears positively (or always negatively), we can set this literal to be true unconditionally (later removing all clauses containing it): every model after adding the clause is also a model of the original set of clauses but not the opposite. This is the first transformation that does not preserve models in IsaSAT or any other verified SAT solvers.

Refinement of Subsumption-Resolution. While the definition of subsumption resolution is very simple, the refinement to code was challenging.

We verified forward subsumption [7] following CaDiCaL [5] (unbounded however, so all clauses selected heuristically are checked). We sort clauses by size and check if the current candidate is subsumed by one of the smaller clauses. Because we use two-watched literals, we need to distinguish between the binary clauses (than can produce new units) and the other clauses. At the end, we implemented two forward subsumption passes: one for binary clauses only and the other for larger clauses.

To subsume the candidates, we build occurrence lists and populate them with binary clauses, whereas Kissat [5] reuses watch lists. Moreover, for efficiency, we need a new marking data structure for efficient detection of subsuming-resolution.

4 Correctness of the Code and Completeness

Our specification model_if_satisfiable takes the multiset of clauses and returns a model (if there is one) or None if the clauses are unsatisfiable. Our implementation $\mathsf{IsaSAT}_{\mathsf{SML}}$ opts takes an array containing the clauses and returns an optional array containing the assignment, assuming that the clauses do not contain duplicated literals or the empty clause (precondition proper_lits_no_dups_⊥). The additional argument opts activates and deactivates certain techniques for solving. The following theorem states that our implementation refines the specification:

Theorem 1 (SML End-to-End Correctness). *The following refinement relation holds:*

$$\begin{aligned}&(\mathsf{IsaSAT}_{\mathsf{SML}}\ \mathsf{opts},\ \mathsf{model\_if\_satisfiable})\\ \in\ &[\mathsf{proper\_lits\_no\_dups\_}\bot]\ \mathsf{clauses\_assn} \to \mathsf{option\_model\_assn}\end{aligned}$$

The LLVM version is nearly the same. It can handle duplicated literals and the empty clause. Moreover, the new specification model_if_satisfiable_bounded allows for an *unknown* result if arrays would grow larger than the size permitted by the fixed bit-width. While this limit does not exist in Imperative_HOL, it exists in practice as no machine supports arrays that large. Therefore, we technically weakened our theorem, but did not change practical guarantees on the generated code. For $\mathsf{IsaSAT}_{\mathsf{SML}}$ we start [12] with 64-bit unsigned integers and only switch to GMP integers if the arrays grow too large.

Theorem 2 (LLVM End-to-End Correctness). *The following refinement relation holds:*

$$\begin{aligned}&(\mathsf{IsaSAT}_{\mathsf{LLVM}}\ \mathsf{opts},\ \mathsf{RETURN} \circ \mathsf{model\_if\_satisfiable\_bounded})\\ \in\ &[\mathsf{proper\_lits}]\ \mathsf{clauses\_assn} \to \mathsf{option\_model\_assn}\end{aligned}$$

Moreover, the change from SML to LLVM reduces the trusted code base: The Isabelle/LLVM model is closer to the actual LLVM, such that the trusted

pretty-printer is simpler. LLVM is also more low-level, such that fewer parts of the compiler have to be trusted. Finally, the LLVM compiler is more widely used and tested than the rather niche MLton compiler we used before.

5 Experience Porting the Development to LLVM

We report on the challenges we faced when updating the huge IsaSAT formalization (Sect. 5.1). Moreover, we report on the unverified parts of IsaSAT (Sect. 5.2), and finally compile some lessons learned (Sect. 5.3).

5.1 Required Changes

Before porting the development to LLVM, we removed our only remaining source of unbounded integers: the clause indices during the garbage collection. As garbage collection does not happen very often, we did not expect this to make a difference. Surprisingly, it turns out to have a performance impact.

Isabelle/LLVM is an entire tool set, including a fork of the original Sepref tool. While related to the original Sepref tool, there are different libraries, and the development of the two versions has diverged.

Initially, we tried to support both versions of Sepref. We ended up with two sets of files for code synthesis, and duplication of some libraries (to provide constants defined in Isabelle/LLVM but not in $\text{Sepref}_{\text{SML}}$). This significantly complicated our refinement approach, although we made it conceptually cleaner during the porting. Then, we realized that $\text{IsaSAT}_{\text{LLVM}}$ was much faster than $\text{IsaSAT}_{\text{SML}}$ (we observed a factor 2 on our test files), and decided to discontinue the SML backend.

With this, also some workarounds for SML specific performance issues (like the tuple `uint32 * bool * uint64` being much less efficient than combining the uint32 and the Boolean into a single 64-bit number) became obsolete.

Compilation. We have experimented with compilation flags before to improve performance. We know from the SML code that we need to increase the level of inlining, because many small functions make the verification easier. The same applies for LLVM and the easiest solution is to use link-time optimization that increases the inlining level as a side effect. However, this makes profiling impossible – exactly like the SML code. So there is no regression here.

Tuples. In 2021, we observed a major performance regression of the synthesis, caused by a new feature in $\text{Sepref}_{\text{LLVM}}$: pointer-equality tracking caused quadratic behaviour for case-splits of tuples. As our solver state is a large tuple, synthesis became impossible (several dozen minutes for simple functions).

To avoid the issue, we decided to work around on the abstract level, using getter and setter functions for the state's components, rather than case splitting. Now, every function on the state would first get the required components, update them, and then put them back. For example:

```
definition rescore_conflict :: clause_index ⇒ isasat ⇒ isasat where
  rescore_conflict C S = do{
    let (M, S) = extract_trail S;
    ... (*reads the trail M and can change it*) ...
    let S = update_trail M S;
    RETURN S
  }
```

This makes synthesis much faster. However, the ownership model of Sepref does not allow aliasing, nor do our refinement relations allow leaving a 'gap' in the state where we moved out an element. As an easy work-around, we resorted to placing dummy-values, like empty lists, in the state, hoping that LLVM would optimize away the allocations and deallocations for these values. However, this did not happen: In the hot-spot of the SAT solver, the propagation loop, the dummy value for the trail was recreated and freed each time. Thus, we locally resorted to unfolding our code to make sure that we need only one free in the inner propagation loop. We leave a more principled solution of this problem (possibly changing Sepref) to future work.

We even attempted to go one step further (as the state-of-the-art SAT solver Kissat [4] does) and simply passing a pointer to the state structure as argument. Once we had already changed our refinement with accessors, we simply had to change them to work on a pointer. However, we never managed to make the synthesized code efficient. We observed a factor of 10 slower code. Hand-optimizing the accessors (basically making sure that LLVM understands that we care only about one component) reduced this to factor 2 slower. Once we realized that the LLVM optimizer was replacing the pointer by the structure passed directly as argument, we gave up on that approach.

5.2 Unverified Parts

In the generated SAT solver, there are some parts that we cannot verify. First, the parser is not verified, because the file system has no model in Isabelle (unlike CakeML, where conditions apply however). To this end, we link the verified code with an unverified C program, which provides the parser and command line interface.

Second, Isabelle/LLVM does not support any output (like statistics, or the DRAT proofs [28] required for the SAT Competition). For the SML version, we could use a feature of Isabelle's code generator to (axiomatically) implement a function by some external function (e.g. a function that does nothing in the model, by a printing function). As Isabelle/LLVM does not yet have such a feature, we resorted to post-processing the generated code (i.e., a function that does nothing in the model, is replaced by a printing function or even a function storing some literals for DRAT proofs). Note that this post-processing is not required for IsaSAT to work (but it won't print DRAT proofs).

5.3 Lessons Learned

Lesson 1: Embrace Duplication. We have already highlighted the importance of the set of all possible literals $\mathcal{A}$, in particular to establish a bound on the size of various arrays. At first, we tried to avoid duplicating this set across the different components on the specification side. This, however, resulted in a closer coupling of the various refinement proofs, impeding modularity: data structures that, conceptually, are just a small part of the whole state, have to be formalized on the whole state, just to have the set $\mathcal{A}$ available. We solved this problem by duplicating the set $\mathcal{A}$ on the abstract level for all new data structures. Note that this duplication is removed in a later refinement stage.

Lesson 2: The Limits are Isabelle Files. Checking our Isabelle files takes nearly two hours. This can be explained by three factors: 1. the heuristic and code synthesis amounts to 91 000 loc, making it a very large formalization; 2. the synthesis is single-threaded (for technical reasons); 3. Sepref encourages a style that is not very parallel: every refinement starts with a call to a tactic `refine_vcg` that generates the goals (meaning that all successive tactics have to wait). To improve performance we have attempted [12] to generate more standard proofs in Isar (by generating the text corresponding to the theorems to prove), but it is not clear that this style is faster as huge number of variables are generated (this style is required for more complicated proofs, however).

In order to improve Isabelle's performance and speed-up the testing of new heuristics in $\mathsf{IsaSAT}_{\mathsf{LLVM}}$, we have split the files into three parts: the shared definitions of the functions to refine, the (single-threaded) synthesis, and the correctness proof of the refinement. Even with these optimizations, proof checking still takes 2 h. There is also no clear improvement path. The old SML version has a similar problem, but it is overall faster because it has fewer features, making it less critical.

Lesson 3: Performance Bugs exist. In order to improve performance, we need to measure and observe performance. To solve that problem, IsaSAT prints statistics and produces some timing information. The statistics during the run made identifying scheduling bugs for the different techniques possible – we accidentally ran some techniques way too often or barely ever. Especially because we increase the interval between two inprocessing rounds geometrically, a simple statistics at the end of the run is not sufficient. One interesting performance bug we found was that we accidentally inverted reducing clauses (marking them as removed) and garbage collection (physically removing them). Therefore, we would nearly always physically delete clauses. We never saw this issue, because we also printed the statistics inverted. To help debugging performance, we produce some timing information by measuring time in the C program:

```
c propagate           : 83.48% (581.66 s)
c reduce              : 0.12% (0.82 s)
c subsumption         : 0.06% (0.39 s)
c pure_lits           : 0.05% (0.33 s)
```

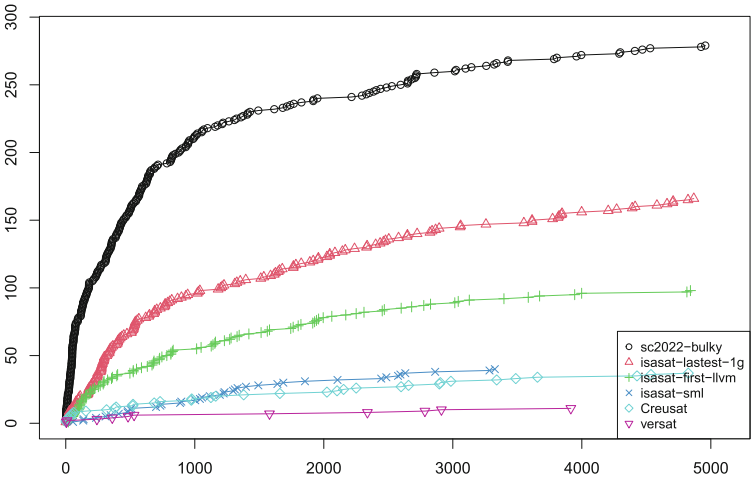

Fig. 1. CDF of the performance of SAT solvers

```
c binary_simp          : 0.02% (0.15 s)
c GC                   : 0.16% (1.10 s)
```

This helps to identify bottlenecks but also outliers where one technique is particularly slow and requires some limits or a change in the scheduling to avoid slowing down the solver too much. This makes it possible to identify errors like allocations in loops. The overall timing matches what we expect from other SAT solvers (although usually they spend more time on inprocessing and less on propagation).

6 Performance

In order to study the performance we have run 3 different IsaSAT versions: the original SML solver (using MLton with the LLVM backend), the first port of the IsaSAT solver, and the current version with inprocessing and various other improvements on heuristics that do not require any change on our PCDCL calculus, notably rephasing and target phases [10] (but no local search) and the alternation between aggressive restarts (heuristically seems better for UNSAT) and few restarts (seems better for SAT) following the ideas of Chanseok Oh [24].

We run all the benchmarks from the SAT Competition 2022 on an Intel Xeon E5-2620 v4 CPU at 2.10 GHz (with turbo-mode disabled) with a memory limit of 7 GB and a timeout of 5000 s. For comparison, we have included VERSAT [23] and CreuSAT [25]. For completeness, we have included Kissat [6] (more precisely the bulky version submitted for the anniversary track).

The results are given in Fig. 1 as a CDF (the higher the curve, the more solved problems). The first surprise is that CreuSAT performs similarly to $\text{IsaSAT}_{\text{SML}}$ (37 vs 40 solved problems), worse than expected given the results reported in the

Master's thesis [25] that tested on the 2015 benchmarks. We suspect that is due to the garbage collection and the fact that problems from the SAT Competition have become harder.

There is a clear improvement when going from the SML version to the LLVM version (98 solved), while the latest version solves 166. The SML version produces 335 out-of-memory errors (OOMs), the base LLVM version is more memory efficient (23 OOMs) like the latest IsaSAT version (19 OOMs) or CaDiCaL that has the same memory layout (17 OOMs). However, there is still a large gap to reach the performance level of Kissat and its inprocessing techniques.

7 Conclusion

We have reported on updating our verified SAT solver IsaSAT to a more powerful base calculus (our pragmatic CDCL) which can express inprocessing, and to the more efficient Isabelle/LLVM backend. We have also compiled important lessons learned from proof-engineering and maintaining large formalizations like IsaSAT (∼200 kloc of proofs).

Our changes made IsaSAT solve 4 times more problems (166/40), making it the most efficient verified SAT solver. At the same time, our verification is more complete than the next fastest verified solvers.

Most techniques (including the two most important, vivification and probing) either fit into our new PCDCL base calculus or do not require any change (like random walk [10] that is conjectured to be the reason for the major performance improvement in 2020). One major technique that we cannot currently express is variable elimination, because models are changed and need to be fixed. We leave the required extensions to our PCDCL for future work.

Acknowledgments. The work presented here was done over several years and several work places. The first author was supported for some time by Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), and the LIT AI Lab funded by the State of Upper Austria. We thank the anonymous reviewers for their detailed comments.

References

1. Andrici, C.C., Ciobâcă, S.: A verified implementation of the DPLL algorithm in Dafny. Mathematics **10**(13), 1–26 (2022). https://ideas.repec.org/a/gam/jmathe/v10y2022i13p2264-d850381.html
2. Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.): Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, Department of Computer Science, University of Helsinki, Finland (2022)
3. Becker, H., et al.: IsaFoL: Isabelle formalization of logic. https://bitbucket.org/isafol/isafol/
4. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proceedings of SAT

Competition 2020 - Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
5. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2021. In: Proceedings of the SAT Competition 2021 - Solver and Benchmark Descriptions (2021). submitted
6. Biere, A., Fleury, M.: Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In: Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proceedings of the SAT Competition 2022 - Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2022-1, pp. 10–11. University of Helsinki (2022)
7. Biere, A., Järvisalo, M., Kiesl, B.: Preprocessing in SAT solving. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications. 2nd edn, vol. 336, pp. 391–435. IOS Press (2021). https://doi.org/10.3233/FAIA200992
8. Blanchette, J.C., Fleury, M., Lammich, P., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. J. Autom. Reason. **61**(1–4), 333–365 (2018). https://doi.org/10.1007/s10817-018-9455-7
9. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_14
10. Cai, S., Zhang, X., Fleury, M., Biere, A.: Better decision heuristics in CDCL through local search and target phases. J. Artif. Intell. Res. **74**, 1515–1563 (2022). https://doi.org/10.1613/jair.1.13666
11. Fazekas, K., Biere, A., Scholl, C.: Incremental inprocessing in SAT solving. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 136–154. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_9
12. Fleury, M.: Optimizing a Verified SAT Solver. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2019. LNCS, vol. 11460, pp. 148–165. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-20652-9_10
13. Fleury, M.: IsaSAT and Kissat entering the EDA Challenge 2021 (2021). https://www.eda-ai.org/results/, system description accepted at the EDA Challenge 2021. https://m-fleury.github.io/ox-hugo/Fleury-EDA-Challenge-2021.pdf
14. 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, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12251-4_9
15. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 355–370. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_28
16. Lammich, P.: Automatic data refinement. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 84–99. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39634-2_9
17. Lammich, P.: Refinement to imperative HOL. J. Autom. Reason. **62**(4), 481–503 (2017). https://doi.org/10.1007/s10817-017-9437-1
18. Lammich, P.: Generating verified LLVM from Isabelle/HOL. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving, ITP 2019, 9–12, September 2019, Portland, OR, USA. LIPIcs, vol. 141, pp. 22:1–22:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.22

19. Lammich, P.: Efficient verified (UN)SAT certificate checking. J. Autom. Reason. **64**(3), 513–532 (2019). https://doi.org/10.1007/s10817-019-09525-z
20. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004, pp. 75–88. IEEE (2004). https://doi.org/10.1109/cgo.2004.1281665
21. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, 18–22 June 2001, pp. 530–535. ACM (2001). https://doi.org/10.1145/378239.379017
22. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
23. 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, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27940-9_24
24. Oh, C.: Between SAT and UNSAT: the fundamental difference in CDCL SAT. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 307–323. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_23
25. Skotåm, S.H.: CreuSAT - using rust and Creusot to create the world's fastest deductively verified SAT solver. Master's thesis, University of Oslo (2022). https://www.duo.uio.no/handle/10852/96757
26. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: `cake_lpr`: verified propagation redundancy checking in CakeML. In: TACAS 2021. LNCS, vol. 12652, pp. 223–241. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_12
27. Weeks, S.: Whole-program compilation in MLton. In: Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, 16 September 2006, p. 1. ACM Press (2006). https://doi.org/10.1145/1159876.1159877
28. 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, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_31