Optimizing a Verified SAT Solver

Mathias Fleury[1,2]

[1] Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrücken, Germany
`mathias.fleury@mpi-inf.mpg.de`
[2] Saarbrücken Graduate School of Computer Science, Saarland Informatics Campus, Saarbrücken, Germany

Abstract. In previous work, I verified a SAT solver with dedicated imperative data structures, including the two-watched-literal scheme. In this paper, I extend this formalization with four additional optimizations. The approach is still based on refining an abstract calculus to a deterministic program. In turn, an imperative version is synthesized from the latter, which is then exported to Standard ML. The first optimization is the extension with blocking literals. Then, the memory management is improved in order to implement the heuristics necessary to implement search restart and forget, which were subsequently implemented. This required changes to the abstract calculus. Finally, the solver uses machine words until they overflow before switching to unbounded integers. Performance has improved and is now closer to MiniSAT without preprocessing.

1 Introduction

SAT solvers are highly optimized programs full of tricks. This makes them an interesting case study for verification, both for the calculi and the data structures involved. Since SAT solvers are a prototypical example of highly optimized programs, it is interesting to see to what extent verification is feasible.

A common approach to increasing the trustworthiness of SAT solvers is to make them return independently verifiable *proofs* that certify the correctness of their answers. Such proofs were successfully produced by tools that solved long-standing open problems such as the *Pythagorean Triples Problem* [20] or *Schur Number Five* [19]. However, the production of proofs does not provide total correctness guarantees: Although a correct proof guarantees that a solver produced a correct result, it is not guaranteed that the solver will be able to produce a proof in the first place. Moreover, proof checkers and SAT solvers share similar techniques and data structures. They, thus, face similar efficiency challenges, and the techniques presented here are applicable to checkers too.

In previous work with Blanchette, Lammich, and Weidenbach, I developed a SAT solver, called IsaSAT [9], which I verified in Isabelle [34]. The first functional implementation, IsaSAT-0, could not solve any problem on a collection of problems from the SAT competitions. To improve performance, I extended IsaSAT with *watched literals* [15]. The resulting version, IsaSAT-17, could solve 390 problems. Watched literals are a well-known optimization [24] but there is more to a modern SAT solver. In this article, I present four additional optimizations.

IsaSAT is specified using stepwise refinement, starting from a non-deterministic transition system [9] that is refined [15] in several steps using the *Isabelle Refinement Framework* [27–29]. Each layer refines and restricts the possible behavior until the program is fully deterministic. After that, *Sepref* [28] synthesizes an imperative version of the functions which can be exported to Haskell, OCaml, Scala, or Standard ML by Isabelle's code generator. Each layer also inherits properties from previous layers; for example, termination of the executable solver is derived from the termination of the initial transition system (Section 3).

Because some idioms made the proofs hard to maintain and slow to process, I first refactored the Isabelle formalization (Section 4). The first optimization is the use of *blocking literals* [12] to improve Boolean constraint propagation (Section 5). The idea is to cache a literal for each clause—if the literal is true in the current partial model of the solver, the clause can be ignored (saving a likely cache miss by not accessing the clause).

To avoid focusing on hard parts of the search space, the search of a SAT solver is heuristically restarted and the search direction changed. Clauses that are deemed useless are also forgotten. However, the standard heuristics rely on the presence of meta-information in clauses that can be efficiently accessed. To make this possible, I redesigned the clause representation, which also allowed me to implement the *position saving* [16] heuristic (Section 6). Extending the SAT solver with *restart* and *forget* required the extension of the calculus with watched literals: Both behaviors were already present in my abstract calculus but were not implemented in the next refinement step. Heuristics are critical and easy to verify, but hard to implement in a way that improves performance (Section 7).

Using machine integers instead of unbounded integers is another useful optimization. The new IsaSAT thus uses machine integers until the numbers don't fit in them anymore, in which case unbounded integers are used to maintain completeness (theoretically, IsaSAT could have to learn more than 2^{64} clauses before reaching the conclusion, which would overflow clause counters). The code is duplicated in the solver but specified only once (Section 8).

I analyze the importance of the different features and compare IsaSAT with state-of-the-art solvers (Section 9). Even though the new features improve IsaSAT significantly, much more work is required to match the best unverified solvers. The formalization is available online[3] and is part of the *Isabelle Formalization of Logic* (IsaFoL) effort [3]. The results presented here were briefly mentioned in Blanchette's invited talk at CPP 2019 [7, Section 3].

2 The Isabelle Refinement Framework

The Isabelle Refinement Framework is at the center of my approach. Several refinement layers are used and each layer inherits properties from previous steps. Each step can change data structures and restrict the behavior of the program.

[3] `https://bitbucket.org/isafol/isafol/src/master/Weidenbach_Book/`

The framework allows me to express programs in a non-determinism monad. A program can either fail if any execution fails (FAIL); otherwise it returns a set of all possible results (RES X where any element of X is a possible outcome). RETURN x is a special case that returns the single value x; i.e., RES $\{x\}$. The bind function bind $m\,f$ applies f to every outcome of m and is most of the time written with the Haskell-style 'do' notation $\mathsf{do}\ \{a \leftarrow m;\ f\,a\}$. Then higher-level constructs are defined such as 'while' loops.

The framework provides a way to express refinement relations between two programs. First, a program can restrict the behavior of another program. The framework provides a partial order $\leq$ such that $\mathsf{RES}\,X \leq \mathsf{RES}\,Y$ if and only if $X \subseteq Y$ and FAIL is the top element (for all programs r, $r \leq \mathsf{FAIL}$). Second, data structures can also be refined. Given a relation R, $g \leq \Downarrow_R f$ means that every outcome of g is also an outcome of f up to conversion by R. To reason on program refinement, the framework provides tactics that heuristically map or *align* one instruction of the refined program to one instruction of the refining one; for example, they can align $\mathsf{RETURN}\,x$ and $\mathsf{RES}\,X$, yielding the goal $x \in X$.

Finally, the framework provides the Sepref tool [28], which can synthesize a deterministic program with imperative data structures in Imperative HOL [10] from a non-deterministic program. For example, it can refine lists to arrays if all accesses are proven valid. Once synthesized, Isabelle's code generator [18] can be used to export the code to Haskell, OCaml, Scala, and Standard ML.

Code generation in Isabelle is built around a mapping from Imperative HOL operations to concrete code in the target language. This mapping is composed of *code equations* translating code and the correctness of the mapping cannot be verified in Isabelle. For example, accessing the n-th element of an Imperative HOL array is mapped to accessing the n-th elements of the target language (e.g., `Array.sub` in Standard ML). These equations are the *trusted code base*.

3 IsaSAT

The IsaSAT solver, which this work extends, is organized in several refinement layers. Each one restricts the behavior or refines the data structures.

The most abstract layer [9], called CDCL, describes a conflict-driven clause learning (CDCL) transition system with dedicated transitions for restarts and forget. CDCL builds a candidate model, called the *trail* or M. Each time a clause is not satisfied by the trail, CDCL analyzes the clause to adapt the trail.

The second layer is a non-determinism transition system, called TWL, for two watched literals, and is expressed using an inductive predicate. It is connected to the previous calculus but restricts the behavior by forbidding restarts and forgets. Each clause has two literals called *watched*; the others are *unwatched*. The calculus operates on states $(M, N, U, D, \mathit{NP}, \mathit{UP}, \mathit{WS}, Q)$, where M is the trail; N and U are the set of clauses of length greater than one; D is the conflict that is analyzed or $\top$; NP and UP are sets of clauses of length one; WS is a multiset of pairs (L, C) in the clause $C \in N + U$ such that L is a literal watched; Q is a multiset of literals. The SAT solver must visit each clause once after one of its watched literal has been set, i.e. the clause C in (L, C) of WS. Each visit results in either a change of one watched literal in order to maintain the two-watched-literal invariant or no change. The Ignore rule describes the latter:

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

Informally, if the other watched literal L' is true, then no change of the watched literals of the clause C is required.

The third layer, called Algo, is expressed using the non-determinism monad of the Refinement Framework. Compared with TWL,, the non-deterministic program fixes the order of rules, restricting its behavior.

In the first three layers, clauses are represented by multisets. In the fourth layer, called List, clauses become lists that are accessed by indices. This layer mostly features invariants stating that accesses using indices are in bounds. In the fifth layer, called WList, watch lists are added. They keep a mapping from a literal to all the clauses that are watching it. This mapping is critical for performance (recalculating them when required is too costly), but it is easier to introduce watch lists separately. In previous refinement steps, the mapping was recalculated when required. In a sixth layer, we add some additional invariants.

All heuristics are defined in the seventh and last layer, called Heur, leading to fully deterministic functions. Sepref is used to synthesize an imperative version of the code. Following the DIMACS format used in the SAT Competition, the generated code uses 32-bit machine words for the literals. Finally, Isabelle's code generator is used to export code in Standard ML, where it is combined with a trusted parser to get an executable program. IsaSAT is correct:

Theorem 1 (End-to-End Correctness) *If the literals in the input clauses fit in 32-bits and the input clauses do no contain duplicate literals, then IsaSAT returns a model if its input is satisfiable, or none if it is unsatisfiable.*

4 Refactoring IsaSAT

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

Proof Style. The original and most low-level proof style is the apply script: It is a forward style and each tactic creates subgoals. It is ideal for proof exploration and simple proofs. It is, however, hard to maintain. A more readable style states explicit statements of properties in Isar [42]. The styles can be combined: each intermediate step can be recursively justified by apply scripts or Isar. For robustness, I use Isar where possible.

The tactics aligning goals are inherently apply style, but I prefer Isar. I will show the difference on the example of the refinement of $\mathsf{PCUI}_{\mathsf{Algo}}$ (Figure 1a) by $\mathsf{PCUI}_{\mathsf{List}}$ (Figure 1b). Assume the arguments of the function are related by

```
definition $\mathsf{PCUI_{Algo}}$ where
  $\mathsf{PCUI_{Algo}}\ LC\ S = \text{do}$ {
    let $(L, C) = LC$;
    $L' \leftarrow \mathsf{RES}\ (\mathsf{watched}\ C - \{L\})$;
    if $L' \in \mathsf{trail_{List}}\ S$ then
      $\mathsf{RETURN}\ S$
    else ...
  }
```

(a) Ignore rule after refactoring

```
definition $\mathsf{PCUI_{WList}}$ where
  $\mathsf{PCUI_{List}}\ LC\ S = \text{do}$ {
    let $(L, C) = LC$;
    $L' \leftarrow \mathsf{RES}\ (\mathsf{watched}\ C - \{L\})$;
    if $L' \in \mathsf{trail_{List}}\ S$ then
      $\mathsf{RETURN}\ S$
    else ...
  }
```

(b) Ignore rule after refactoring

Fig. 1. Comparison of the code of Ignore rule in Algo before and after refactoring

the relation $((LC, S), (LC', S')) \in R_{\text{state}}$. The first two goals stemming from aligning $\mathsf{PCUI_{Algo}}$ with $\mathsf{PCUI_{List}}$ are

$$\forall L'\ L\ C\ C'.\ ((LC, S), (LC', S')) \in R_{\text{state}} \land LC = (L, C) \land LC' = (L', C') \rightarrow (LC, LC') \in R_{\text{watched}} \tag{1}$$

$$\forall L'\ L\ C\ C'.\ ((LC, S), (LC', S')) \in R_{\text{state}} \land LC = (L, C) \land LC' = (L', C') \land (LC, LC') \in R_{\text{watched}} \rightarrow \mathsf{RES}\ (\mathsf{watched}\ C - \{L\}) \leq\Downarrow R_{\text{other watched}}(\text{RES}\ (\mathsf{watched}\ C' - \{L'\})) \tag{2}$$

where equation (1) relates the two lets, equation (2) the two RES, and the relations R_{watched} and $R_{\text{other watched}}$ are two schematic variables that have to be instantiated during the proof (e.g., by the identity). Although I strive to use sensible variable names, they are lost when aligning the programs, making the goals harder to understand.

A slightly modified version of Haftmann's `explore` tool [17] transforms the goals into Isar statements. The workflow to use it is the following. First, use Sepref's tactic to align two programs. Then, `explore` prints the structured statements. Finally, those statements can be inserted in the theory, before the goal. Figure 2a shows the output: equations (1) and (2) corresponds to the two `have` statements, where **have** $R\,x$ **if** $P\,x$ **and** $Q\,x$ **for** x stands for the unstructured goal $\forall x. (P\,x \land Q\,x \longrightarrow R\,x)$. Each goal can be named and used to solve one proof obligations arising from the alignment of the two programs.

`explore` does not change the goals and hence, variables and assumptions are not shared between proof steps, leading to duplication across goals. I later expanded the `explore` to preprocess the goals before printing them: It uses **context**s (Figure 2b) that introduces blocks sharing variables and assumptions. These proofs are now faster to check and write and minor changes are easier to do. There is no formal link between the statements and the goal obligations: If the goal obligations changes, the Isar statements have to be updated by hand. After big changes in the refined functions, it can be easier to regenerate the new statements, re-add them to the theory, and reprove them than to adapt the

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

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

```
context
  fixes $L'\ L\ C\ C'\ C\ C'$
  assumes $((LC, S), (LC', S')) \in R_{\text{state}}$
    and $LC = (L, C)$ and
    $LC' = (L', C')$
begin
lemma $(LC, LC') \in R_{\text{watched}}$
  sorry
lemma RES (watched $C - \{L\}$)
    $\le\Downarrow R_{\text{other watched}}$
      (RES (watched $C' - \{L'\}$))
  sorry
end
```

(b) Proof with contexts as generated `explore_context`, with sharing.

Fig. 2. Different ways of writing the proof that $\mathsf{PCUI}_{\mathsf{List}}$ from Figure 1a refines $\mathsf{PCUI}_{\mathsf{Algo}}$

old one. Thanksfully, this only happens a few times, usually when significantly changing the function anyway, which also significantly changes the proof.

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

1. use specification and abstract data structure in Heur (e.g., the conflict clause is an optional multiset);
2. map the operations on abstract to concrete functions (e.g., the function converting a clause to a conflict clause is refined to a specific function converting a clause to a lookup table);
3. discharge the preconditions from step 2 with Sepref (e.g., no duplicate literal).

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

$1'$. introduce the heuristics and data structures in Heur (e.g., the conflict is a lookup table);
$2'$. add assertions for preconditions on code generation to Heur.

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

5 Adding Blocking Literals

Blocking literals [12] are an extension of the two-watched-literal scheme and are composed of two parts: a relaxed invariant and the caching of a literal. Most SAT solvers implement both aspects. Blocking literals reduce the number of memory accesses (and, therefore, of cache misses).

Invariant. IsaSAT-17's version of the two-watched-literal scheme is inspired by MiniSAT 1.13. The key invariant is the following [15]:

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

I now relax the condition by replacing "the other watched literal" by "any other literal". This weaker version means that there are fewer changes to the watched literals to do: If there is a true literal, no change is required. Accordingly, the side conditions of the Ignore rule of TWL can be relaxed from $L' \in \mathsf{watched}\ C$ to $L' \in C$. Adapting the proof of correctness was relatively easy. The proofs are easy to fix (after adding some key lemmas) thanks to Sledgehammer [8], a tool that uses automatic theorem provers to find proofs.

The generalized Ignore rule is refined to the non-determinism monad (Figure 3a). Since the calculus has only been generalized, no change in the refinement would have been necessary. In the code, the rule can be applied in three different ways: Either L', the other watched literal L'', or another literal from the clause is true (the last case is not shown in Figure 3). Any literal (even the false watched literal L) can be chosen for L'.

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

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

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

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

Fig. 3. Refinement of the rule Ignore with blocking literals from Algo to WList

Caching of a literal. Most SAT solvers contain an second part: When visiting a clause, it is often sufficient to visit a single literal [37]. Therefore, to avoid a likely cache miss, a literal per clause, called *blocking literal*, is cached in the watch lists. If it is true, no additional work is required; otherwise, the clause is visited: If a true literal is found, this literal is elected as new blocking literal, requiring no update of the watch lists.

In the refinement step WList, the choice is fixed to the cached literal from the watch list (Figure 3b). The identity "let $L' = L'$;" helps the tactics of the Refinement Framework to recognize L' as the choice for $\mathsf{RES}\ \{L' \mid L' \in C\}$, i.e. yielding the goal obligation $L' \in \mathsf{RES}\ \{L' \mid L' \in C\}$.

IsaSAT's invariant on the blocking literal forces the blocking literal to be *different* from the associated watch literal (corresponding to the condition $L \neq L'$ in Figure 3). This is not necessary for correctness but offers better performance (since L is always false) and enables special handling of binary clauses: No memory access is necessary to know the content of the clause. IsaSAT's watched lists contain an additional Boolean indicating whether the clause is binary.

6 Improving Memory Management

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

However, none of these two tricks can be directly obtained by refinement and Isabelle offers no control over the memory allocator. Therefore, I implemented both optimizations at once, similarly to the implementation in CaDiCaL [4]. The implementation uses a large array, the *arena*, to allocate each clause one after the other, with the metadata before the clauses (Figure 4): The lengths (here 4 and 5) precede the clause. Whereas the specifications allow the representation to contain holes between clauses, the concrete implementation avoids it.

In IsaSAT-17, the clauses were a list of clauses, each one being a list of literals (both list being refined to arrays). This representation could not be refined to an arena. Moreover, it was not compatible with removing clauses without shifting the positions. For example, if the first clause was removed from the list $[A \vee B \vee C;\ \neg A \vee \neg B \vee C \vee D]$, then the position of the second clause changed. This was a problem as the indices are used in the trail. Therefore, I first changed the representation from a list of lists to a mapping from natural numbers

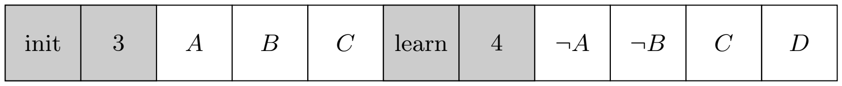

Fig. 4. Example of arena module with two clauses $A \vee B \vee C$ (initial clause, 'init') and $\neg A \vee \neg B \vee C \vee D$ (learned clause, 'learn')

to clauses. Then, every element of the domain was mapped to a clause in the arena with the same index (for example, in Figure 4, the clause 2 is $A \vee B \vee C$; 7 is $\neg A \vee \neg B \vee C \vee D$; there are no other clauses).

Introducing arenas requires some subtle changes to the existing code base. First, the arena contains natural numbers (clause length) and literals (clause content). Therefore, I use a datatype (as a tagged union) that contains either a literal or a natural number. Both types are refined to the same type, a 32-bits word and the datatype is removed when synthesizing code. An invariant on the whole arena describes its content. Moreover, because literals are refined to 32-bit machine words, the length has to fit in 32 bits. However, as the input problems can contain at most 2^{16} different atoms and duplicate-free tautologies, the maximum length of a clause is 2^{32}. To make it possible to represent all clauses including those of size 2^{32}, the arena actually keeps the number of unwatched literals (i.e., the length minus 2), unlike Figure 4.

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

Position Saving. I implemented a heuristic called *position saving* [16], which requires an additional metadata. It considers a clause as a circular buffer: When looking for a new literal, the search starts from the last searched position instead of starting from the first non-watched literal of the clause. The position is saved as a metadata of the clause. Similarly to CaDiCaL [4], the heuristic is only used for long clauses (length larger than four). Otherwise, the position field is not allocated in the arena (i.e., the size of the metadata depends on the clause size). Incorporating the heuristic was easy thanks to non-determinism. For example, to apply the Ignore rule, finding a true literal is sufficient, *how* it is found is not specified. This makes it easy to verify a different search algorithm.

Although there exist some benchmarks showing that this technique improve the performance of solvers [5], only CaDiCaL and Lingeling [4] implement it and I did not know if it would improve IsaSAT: The generated code is hardly readable and hard to change in order to test such techniques. However, it was easy to add and it improves performance on most problems (see Section 9).

7 Implementing Restarts and Forgets

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

Completeness is not guaranteed anymore if restart and forget are applied too often. To keep completeness, I delay them more and more. TWL does not propagate clauses of length 1, because they do not fit in the two-watched-literal scheme. These clauses are propagated during the initialization are cannot be removed from the trail. However, such clauses will always be repropagated by CDCL. Therefore, a TWL restart corresponds to a CDCL restart and some propagations. If decisions are also kept, then IsaSAT can reuse parts of the trail [36]. This technique avoids redoing some work after a restart. The trail could even be entirely reused if the decision heuristics would do the same decisions.

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

After clauses have been marked as deleted, the watch lists are not garbage collected. Instead, before accessing a clause, IsaSAT tests if the clause has been deleted or not. However, this is an implementation-specific detail I don't want to mirror in Algo. To address this, I changed Algo in a less intrusive way. Before Algo was iterating over WS. After the change, a finite number of no-ops is added to the while loop (Figure 5). When aligning the two programs, an iteration over a deleted clause is mapped to a no-op. More precisely, there are two tests: whether the blocking literal is true and whether the clause is marked as deleted. If the blocking literal is true, the state does not change (whether the clause is deleted or not). Otherwise, the clause has to be accessed. If the clause is deleted, it is removed from the watch list.

IsaSAT uses the EMA-14 heuristic [6], which is based on two exponential moving averages of scores, implemented using fixed-points numbers: a "slow" average measuring the long-term tendency of the scores and a "fast" one for the local tendency. If the fast average is worse than the slow one, the heuristic is triggered. Then, depending on the number of clauses, either restart or reduce is triggered. The heuristic follows the unpublished implementation of CaDiCaL [4], with fixed-point calculations. This is easier to implement than Glucose's queue for scores. Due to programming errors, it took several iterations to get EMA-14

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

Fig. 5. Skipping deleted clauses during iteration over the watch list

right: The first version never restarted while the second did as soon as possible. Although both versions were complete, the last version performed better.

8 Using Machine Integers

When I started to work on IsaSAT, it was natural to use unbounded integers to index clauses in the arena (refined from Isabelle's natural numbers). First, they are the only way to write lists accesses in Isabelle (further refined to array accesses). Second, they are also required for completeness to index the clauses and there was also no code-generation setup for array accesses with machine words. Finally, the Standard ML compiler I use, MLton [41], efficiently implements numbers first as machine words and then as unbounded GMP integers. However, profiling showed that subtractions and additions took among them around 10% of the time.

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

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

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

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

9 Evaluation

I evaluated IsaSAT-30 on preprocessed problems from the SAT Competitions 2009 to 2017 and from the SAT Race 2015 using a timeout of 1800 s. The hardware was an Intel Xeon E5620, 2.40 GHz, 4 cores, 8 threads. Each instance was limited to 10 GB of RAM. The problems were preprocessed by CryptoMiniSat [38]. The motivation behind this is that preprocessing can significantly simplify the problem. Detailed results can be found on the companion web page[4].

State-of-the-art solvers solve more problems than IsaSAT with the default options (Figure 6). Since the instances have already been preprocessed, the difference comes from a combination of simplifications (pre- and inprocessing), better heuristics, and a better implementation. To assess the difference, I have also benchmarked the solvers without simplification (third column of Figure 6). Heule's MicroSAT [21] aims at being very short (240 lines of code including comments). Compared with IsaSAT, it has neither position saving nor blocking literals but is highly optimized and its heuristics work well together. The version without the four presented optimizations differs from IsaSAT-17 by various minor optimizations. IsaSAT performs better than the only other verified SAT solver with efficient data structures I know of, `versat`.

I compared the impact of reduction, restart, position saving, and machine words (Figure 7). Since Standard ML is garbage-collected, the peak memory usage depends on the system's available memory. The results show that restarts and machine words have a significant impact on the number of solved problems. The results are less clear for the other features. Position saving mostly has a positive impact. The negative influence of reduction hints at a bad heuristic: I later tuned the heuristic by keeping clauses involved in the conflict analysis and the results improved from 749 to 801 problems. The fact that garbage collection of the arena is not implemented could also have an impact, as memory is wasted.

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

SAT solver	Default options Solved	Default options Average time (s)	No simplification Solved	No simplification Average time (s)
CryptoMiniSat	1774	349	1637	349
Glucose	1703	320	1696	303
CaDiCaL	1677	361	1602	346
MiniSAT	1388	326	1373	317
MicroSAT	1018	310	N/A	
IsaSAT-30 fixed heuristic	801	359	N/A	
IsaSAT-30 without the four optimizations	433	301	N/A	
IsaSAT-17	393	220	N/A	
`versat` [35]	368	224	N/A	

Fig. 6. Performance of some SAT solvers (N/A if no simplification is done by default)

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

Fig. 7. Benchmarks of variants of IsaSAT-30 before fixing the forget heuristic

10 Discussion and Related Work

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

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

Continuous testing also turned out to be important. It can catch performance regression before any change in the search behavior is done, allowing me to debug them. One extreme example was the special handling of binary clauses: A Boolean was added to every element of the watch list, changing the type from `word64 * word32` to `word64 * (word32 * bool)`. This change in the critical spot of any SAT solver caused a performance loss of around 20% due to 3.5 times as many cache misses. Since the search behavior had not changed, I took a single problem and tried to understand where the regression came from. First, `word64 * (word32 * bool)` is less efficient than `word64 * word32 * bool` as it requires a pointer for `word32 * bool`. This can be alleviated by using a single constructor datatype (the code generator generates the later version and the single constructor is optimized away). However, there is a second issue: The tuple uses three 64-bit words, whereas only two would be used in the equivalent C structure. I added code equations to merge the `word32 * bool` into a single `word64` (with 31 unused bits), solving the regression. Developers of non-verified SAT solvers face similar issues[5] but they are more tools for C and C++.

While working on the SAT solver, I added several code equations to the trusted code base. The additional code equations are either trying to avoid conversions to unbounded integers (`IntInf`) and back (as would happen by default when accessing arrays) or related to printing statistics during the execution. Whether or not the equations are safe is not always obvious. For example, the code equations to access arrays *without* converting the numbers to unbounded integers and back[6] are safe as long as the array bounds are checked.

However, IsaSAT is compiled with an option that deactivates array-access bound checks. When accessing elements outside of an array, the behavior is undefined. As long as I am using Sepref and the assumptions of Theorem 1 hold, validity of the memory accesses is proved. Without the custom code equations and with bound checks, only 536 problems are solved, instead of 749.

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

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

Related Work. This work is related to other verification attempts of fast code, like Lammich's GRAT toolchain [26,30]. One of the differences is that he uses a C++ program to preprocess the certificates in order to be able to check them more efficiently later. However, like a SAT solver, a checker uses many arrays and therefore would likely benefit from machine words.

Unlike the top-down approach used here, the verification of the seL4 microkernel [25] relies on abstracting the program to verify. An abstract specification in Isabelle is refined to an Haskell program. Then, a C program is abstracted and connected to the Haskell program. Unbounded integers are not supported in C and therefore achieving completeness of a SAT solver would not be possible.

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

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

Other techniques to abstract programs exist, like Chargueraud's characteristic formulas [11]. Another option is Why3 [14] or a similar verification condition generator like Dafny [31]. Some meta-arguments in Why3 (for example, incrementing a 64-bit machine integer initialized with 0 will not overflow in a reasonable amount of time; therefore, machine integers are safe [13]) would simplify the generation of efficient code. In any case, refinement helps to verify a large program.

Isabelle's code generator does not formally connect the generated code to the original function. On the one hand, Hupel's verified compiler [23] from Isabelle to the semantics of the verified Standard ML compiler CakeML could bridge the gap. However, code export from Imperative HOL is not yet supported. On the other hand, HOL4 in conjunction with CakeML makes it possible to bridge this gap and also to reason about input and output like parsing the input file and printing the answer [22]. There is, however, no way to eliminate the array-access checks. Moreover, CakeML uses boxed machine words unlike MLton, which probably leads to a significant slowdown.

Marić has developed another verified SAT solver [33] in Isabelle without refinement, making his formalization impossible to extend. Moreover, a different version of watched literals, no efficient data structures (only lists), nor heuristics are used. Oe et al. use a different verification approach without refinement for `versat`. The Guru proof assistant [39] is used to generate C code. Termination or correctness of the generated model is not proven. Similarly to IsaSAT, `versat` uses machine words—it relies on `int` to be 32 bits, which is not guaranteed in C—but cannot solve larger instances. The SAT competition includes such problems which usually can be solved easily if the decision heuristic initially makes literals false. There is no bound checking for arrays. `versat` features a different flavor of watched literals but neither blocking literals nor restart or forget.

Among SAT solvers, there are two main lines of research: Solvers derived from MiniSAT, like Glucose [2] and MapleSAT [32], focus on improving CDCL (and especially the heuristics) whereas solvers like CaDiCaL [4], CryptoMiniSat [38] and Lingeling [4] also feature inprocessing.

11 Conclusion

I have extended a verified SAT solver, IsaSAT, with four additional optimizations to improve performance and I have verified those extensions. Even if the refinement approach is helpful, adding these optimizations is a significant effort. Lammich is currently working on generating LLVM code which could give more control on the generated code (e.g., the tuples representation is more efficient).

I now plan to extend my calculus to be able to represent CDCL($\mathcal{T}$), the calculus behind SMT solvers. The theory of linear arithmetic has already been implemented by Thiemann [40].

Acknowledgment. Jasmin Blanchette discussed several earlier drafts with me. This work would not have been possible without Christoph Weidenbach and Peter Lammich. Marijn Heule, Benjamin Kiesl, Peter Lammich, Hans-Jörg Schurr, Petar Vukmirović, and the anonymous reviewers suggested many textual improvements.

References

[1] Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) IJCAI 2009. pp. 399–404. Morgan Kaufmann Publishers Inc. (2009), `http://ijcai.org/Proceedings/09/Papers/074.pdf`

[2] Audemard, G., Simon, L.: Glucose 2.1: Aggressive—but reactive—clause database management, dynamic restarts. In: Workshop on the Pragmatics of SAT 2012 (2012)

[3] Becker, H., Bentkamp, A., Blanchette, J.C., Fleury, M., From, A.H., Jensen, A.B., Lammich, P., Larsen, J.B., Michaelis, J., Nipkow, T., Peltier, N., Popescu, A., Robillard, S., Schlichtkrull, A., Tourret, S., Traytel, D., Villadsen, J., Petar, V.: IsaFoL: Isabelle Formalization of Logic, `https://bitbucket.org/isafol/isafol/`

[4] Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT Competition 2017. In: Balyo, T., Heule, M., Järvisalo, M. (eds.) SAT Competition 2017: Solver and Benchmark Descriptions, pp. 14–15. University of Helsinki (2017)

[5] Biere, A.: Deep bound hardware model checking instances, quadratic propagations benchmarks and reencoded factorization problems. In: Balyo, T., Heule, M., Järvisalo, M. (eds.) SAT Competition 2017: Solver and Benchmark Descriptions, pp. 37–38. University of Helsinki (2017)

[6] Biere, A., Fröhlich, A.: Evaluating CDCL restart schemes. In: Proceedings POS-15. Sixth Pragmatics of SAT workshop (2015)

[7] Blanchette, J.C.: Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In: Mahboubi, A., Myreen, M.O. (eds.) CPP 2019. pp. 1–13. ACM (2019), `https://doi.org/10.1145/3293880.3294087`

[8] Blanchette, J.C., Böhme, S., Fleury, M., Smolka, S.J., Steckermeier, A.: Semi-intelligible isar proofs from machine-generated proofs. J. Autom. Reasoning 56(2), 155–200 (2016), `https://doi.org/10.1007/s10817-015-9335-3`

[9] 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), `https://doi.org/10.1007/978-3-319-40229-1_4`

[10] 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), `https://doi.org/10.1007/978-3-540-71067-7_14`

[11] Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: ICFP. pp. 418–430. ACM (2011), `https://doi.org/10.1145/2034773.2034828`

[12] Chu, G., Harwood, A., Stuckey, P.J.: Cache conscious data structures for Boolean satisfiability solvers. JSAT 6(1-3), 99–120 (2009)

[13] Clochard, M., Filliâtre, J., Paskevich, A.: How to avoid proving the absence of integer overflows. In: VSTTE. LLNCS, vol. 9593, pp. 94–109. Springer (2015), `https://doi.org/10.1007/978-3-319-29613-5_6`

[14] Filliâtre, J., Paskevich, A.: Why3—where programs meet provers. In: ESOP. LLNCS, vol. 7792, pp. 125–128. Springer (2013), `https://doi.org/10.1007/978-3-642-37036-6_8`

[15] Fleury, M., Blanchette, J.C., Lammich, P.: A verified SAT solver with watched literals using Imperative HOL. In: CPP. pp. 158–171. ACM (2018), `https://doi.org/10.1145/3167080`

[16] Gent, I.P.: Optimal implementation of watched literals and more general techniques. J. Artif. Intell. Res. 48, 231–251 (2013), `https://doi.org/10.1613/jair.4016`

[17] Haftmann, F.: Draft toy for proof exploration (August 2013), `www.mail-archive.com/isabelle-dev@mailbroy.informatik.tu-muenchen.de/msg04443.html`

[18] 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), `https://doi.org/10.1007/978-3-642-12251-4_9`

[19] Heule, M.J.H.: Schur number five. In: McIlraith, S.A., Weinberger, K.Q. (eds.) Proceedings of AAAI 18. pp. 6598–6606. AAAI Press (2018), `https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16952`

[20] Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer. In: Creignou, N., Berre, D.L. (eds.) SAT 2016. LNCS, vol. 9710, pp. 228–245. Springer (2016), `https://doi.org/10.1007/978-3-319-40970-2_15`

[21] Heule, M.: microsat (2014), `https://github.com/marijnheule/microsat`

[22] Ho, S., Abrahamsson, O., Kumar, R., Myreen, M.O., Tan, Y.K., Norrish, M.: Proof-producing synthesis of CakeML with I/O and local state from monadic HOL functions. In: IJCAR. LNCS, vol. 10900, pp. 646–662. Springer (2018), `https://doi.org/10.1007/978-3-319-94205-6_42`

[23] Hupel, L., Nipkow, T.: A verified compiler from Isabelle/HOL to CakeML. In: ESOP. LNCS, vol. 10801, pp. 999–1026. Springer (2018), `https://doi.org/10.1007/978-3-319-89884-1_35`

[24] Katebi, H., Sakallah, K.A., Marques-Silva, J.P.: Empirical study of the anatomy of modern SAT solvers. In: SAT 2011. LNCS, vol. 6695, pp. 343–356. Springer (2011), `https://doi.org/10.1007/978-3-642-21581-0_27`

[25] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an operating-system kernel. Commun. ACM 53(6), 107–115 (2010), `https://doi.org/10.1145/1743546.1743574`

[26] Lammich, P.: GRAT—Efficient formally verified SAT solver certification toolchain, `http://www21.in.tum.de/~lammich/grat/`

[27] Lammich, P.: Automatic data refinement. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 84–99. Springer (2013), `https://doi.org/10.1007/978-3-642-39634-2_9`

[28] Lammich, P.: Refinement to Imperative/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 253–269. Springer (2015), `https://doi.org/10.1007/978-3-319-22102-1_17`

[29] Lammich, P.: Refinement based verification of imperative data structures. In: Avigad, J., Chlipala, A. (eds.) CPP 2016. pp. 27–36. ACM (2016), `https://doi.org/10.1145/2854065.2854067`

[30] Lammich, P.: Efficient verified (UN)SAT certificate checking. In: CADE. LNCS, vol. 10395, pp. 237–254. Springer (2017), `https://doi.org/10.1007/978-3-319-63046-5_15`

[31] Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR (Dakar). LLNCS, vol. 6355, pp. 348–370. Springer (2010), `https://doi.org/10.1007/978-3-642-17511-4_20`

[32] Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. pp. 123–140. LNCS, Springer International Publishing, Cham (2016), `https://doi.org/10.1007/978-3-319-40970-2_9`

[33] Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theor. Comput. Sci. 411(50), 4333–4356 (2010), `https://doi.org/10.1016/j.tcs.2010.09.014`

[34] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)

[35] 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), `https://doi.org/10.1007/978-3-642-27940-9_24`

[36] Ramos, A., van der Tak, P., Heule, M.: Between restarts and backjumps. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 216–229. Springer (2011), `https://doi.org/10.1007/978-3-642-21581-0_18`

[37] Ryan, L.: Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University (2004)

[38] Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) SAT 2009, LNCS, vol. 5584, pp. 244–257. Springer (2009), `https://doi.org/10.1007/978-3-642-02777-2_24`

[39] 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), `https://doi.org/10.1145/1481848.1481856`

[40] Thiemann, R.: Extending a verified simplex algorithm. In: Barthe, G., Korovin, K., Schulz, S., Suda, M., Sutcliffe, G., Veanes, M. (eds.) LPAR-22 Workshop and Short Paper Proceedings. Kalpa Publications in Computing, vol. 9, pp. 37–48. EasyChair (2018), `https://easychair.org/publications/paper/6JF3`

[41] Weeks, S.: Whole-program compilation in MLton. In: ML. p. 1. ACM (2006), `https://doi.org/10.1145/1159876.1159877`

[42] 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)