Verifying solvers:
How much do you want to prove?

Mathias Fleury

2022/10/13

How do we make SAT solvers correct?

Proofs See talk by Armin and the next one by Yong, but:

- requires to check the proof for each file
- not all techniques can be represented by current proof formats

Program Verification This talk!

- works for every input, so no overhead
- does not crash even if run the program for a year

How do we make SAT solvers correct?

Proofs See talk by Armin and the next one by Yong, but:

- requires to check the proof for each file
- not all techniques can be represented by current proof formats

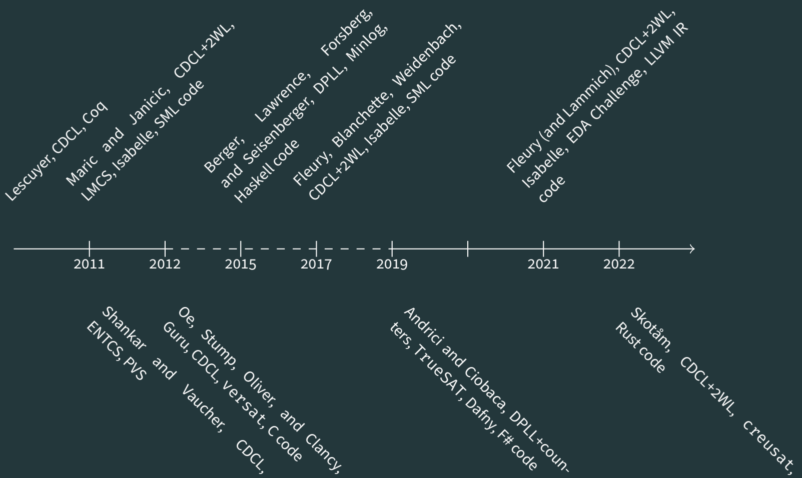
Program Verification This talk!

- works for every input, so no overhead
- does not crash even if run the program for a year

Restrictions

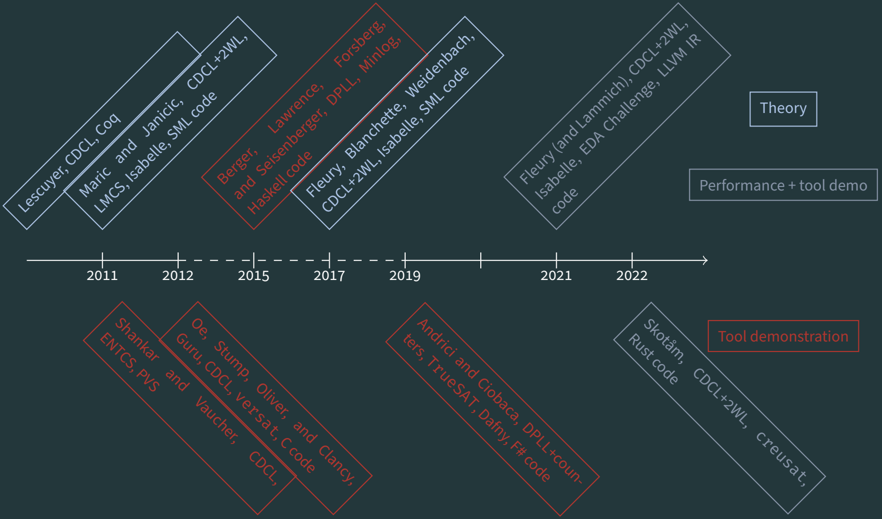
- Parsing is always trusted CakeML has some modelisation of file systems, but don't try "`grep aaa file >> file`" at home
- Printing the answer is trusted too
- The resulting SAT solvers live outside of their system
  - you cannot use them in the system you do the proofs

A Personal History of Solver Verification



incomplete especially because the bottom-up approach is a good master thesis

A Personal History of Solver Verification



What far can you go with one Master thesis?

Fleury: functional code, DPLL, no restarts, propagation by going over all clauses, decision by going over all clauses. but it terminates and is complete
Solves no problem from the SAT Comp

Skotåm: imperative code, CDCL, restarts, watch literals, decision heuristic.
Solves > 150.

Top: Some theory expressed in your tool

?

Bottom Some (hopefully fast) code

All full verifications go top-down.

seL4 kernel is mixed:

```
Specification -> Haskell <- C
```

Most partial verifications go bottom-up.

Most natural for each tool!

Top: Some theory expressed in your tool

?

Bottom Some (hopefully fast) code

All full verifications go top-down.

seL4 kernel is mixed:

```
Specification -> Haskell <- C
```

Most partial verifications go bottom-up.

Most natural for each tool!

Warning: Related Work is hard (TrueSAT)

Table 1. Summary of existing verified SAT solvers.

Solver	Algorithm	Proof Assistant	Downside
versat [27]	CDCL	Guru	not fully verified
Marić [30]	DPLL	Isabelle/HOL	not imperative
Berger et al. [26]	DPLL	Minlog	DPLL-only, not imperative
IsaSAT [31]	CDCL	Isabelle/HOL	not imperative
TrueSAT (this work)	DPLL	Dafny	DPLL-only

Incorrect representation of related work

Only bottom-up work that also proves completeness and termination.

Cheating: DPLL without statefull heuristics.

Warning: Related Work is hard (TrueSAT)

Table 1. Summary of existing verified SAT solvers.

Solver	Algorithm	Proof Assistant	Downside
versat [27]	CDCL	Guru	not fully verified
Marić [30]	DPLL	Isabelle/HOL	not imperative
Berger et al. [26]	DPLL	Minlog	DPLL-only, not imperative
IsaSAT [31]	CDCL	Isabelle/HOL	not imperative
TrueSAT (this work)	DPLL	Dafny	DPLL-only

Incorrect representation of related work

Only bottom-up work that also proves completeness and termination.

Cheating: DPLL without statefull heuristics.

The Theory Inside the TP

Express within the TP:

- shallow embedding (reuse from the TP) rare
- or: deep embedding redefine clauses as multiset, models

Express theory within TP

- each transformation must fit within the theory

The theory is what you make out of it! for PAC checker: talk about polynomials, not about multiplier

SAT: What is the theory?

Proofs from the SAT point of view:

- The bottom-up approach: Resolutions
- The top-down approach: (CDCL via) models

What happens if we try something more complicated?

Other Verified Solvers

SAT Checkers: (see next talk, by construction no completeness)

Ordered Resolution Solver: project to prove feasibility no advanced feature, purely functional code

What has been tried?

CAD issues already expressing the definitions for the algorithms

Other Verified Solvers

SAT Checkers: (see next talk, by construction no completeness)

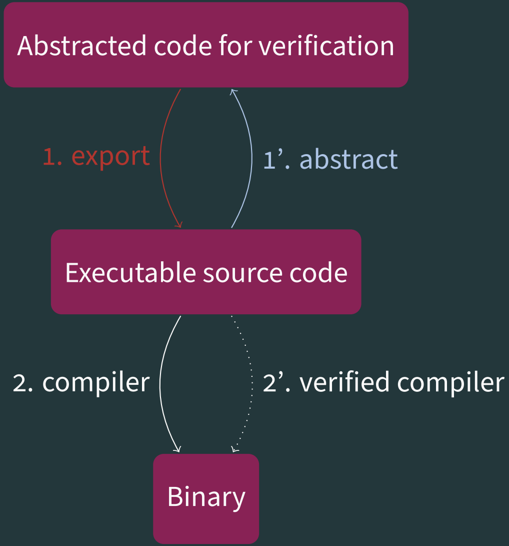
Ordered Resolution Solver: project to prove feasibility no advanced feature, purely functional code

What has been tried?

CAD issues already expressing the definitions for the algorithms

Bottom-Up Or the Art of Proving very little

Organisation



Translation from Rust to why3 (unverified) [Denis, Jourdan, Marché, ICFEM’21]

1’ transalation from Guru to C [Stump et al, PLPV’09]

2’: only used in a SAT checker

Key Idea

Implicit Checker The checker = the verification

Every approach I am aware of: checker = resolution checker

Theorem (Correctness)

Deriving $\bot$ implies that the problem is UNSAT.

Key Idea

Implicit Checker The checker = the verification

Every approach I am aware of: checker = resolution checker

Theorem (Correctness)

Deriving ⊥ implies that the problem is UNSAT.

Key Idea

Implicit Checker The checker = the verification

Every approach I am aware of: checker = resolution checker

Theorem (Correctness)

Deriving $\bot$ implies that the problem is UNSAT.

Some Invariants of a SAT Solver

Deriving the empty clause: input problem unsat

Conflicts on current level: runtime assertion

Termination: Unknown

No conflict+all assigned: checking of the model

No crash: depends on the approach

What do you have to prove?

Well-behaved: no read past end of array

Assignments: consistent and propagations are entailed

Clauses: not modified except by resolution

- But: non trivial for minimization where the resolution is implicit

Making the Solver more Complex: Adding Restart?

Assume you already have a working CDCL.

Adding restarts means:

1. call backtrack to level 0

That is all

except for heuristics, performance debugging, ...

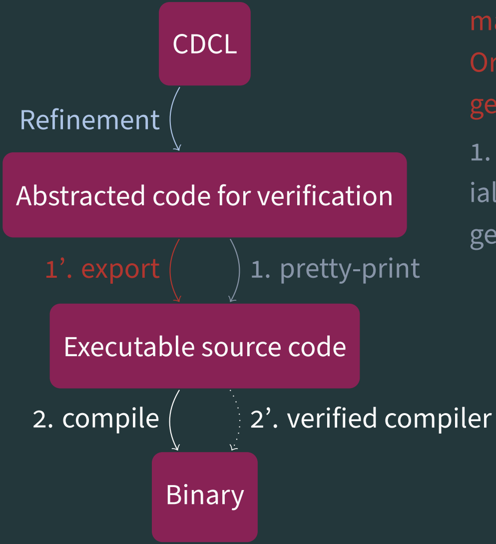
Challenges

What is hard?

- Usually relies on automatic provers, which must be able to handle the specification
  Skotåm: swapping literals
- No termination
  ITP don't like non-termination
- Closer to programs written by hand
  easier to try different strategies

Top-Down Approach: Proving Too much

Organisation



1'. Hupel: use semantics from 2'.
Or Lammich: LLVM generation

1. trusted as trivial translation (SML generation)

Key Idea

Abstract Correctness (Pragmatic) CDCL is fully correct

Theorem (Total Correctness[1])

Deriving ⊥ iff the probem is UNSAT. No conflict + total assignment = SAT. Termination.

Theorem (Total Correctness IsaSAT-LLVM)

If the answer is not unknown, it is either SAT with a model or UNSAT.

IsaSAT-SML had full correction SML semantics does not forbid arrays $\geq 2^{64}$, no compiler support

[1] At some point, memory representation can cause also aborts.

Key Idea

Abstract Correctness (Pragmatic) CDCL is fully correct

Theorem (Total Correctness[1])

Deriving ⊥ iff the probem is UNSAT. No conflict + total assignment = SAT. Termination.

Theorem (Total Correctness IsaSAT-LLVM)

If the answer is not unknown, it is either SAT with a model or UNSAT.

IsaSAT-SML had full correction SML semantics does not forbid arrays $\geq 2^{64}$, no compiler support

[1] At some point, memory representation can cause also aborts.

Key Idea

Abstract Correctness (Pragmatic) CDCL is fully correct

Theorem (Total Correctness[1])

Deriving ⊥ iff the probem is UNSAT. No conflict + total assignment = SAT. Termination.

Theorem (Total Correctness IsaSAT-LLVM)

If the answer is not unknown, it is either SAT with a model or UNSAT.

IsaSAT-SML had full correction SML semantics does not forbid arrays $\geq 2^{64}$, no compiler support

[1] At some point, memory representation can cause also aborts.

Key Idea

Abstract Correctness (Pragmatic) CDCL is fully correct

Theorem (Total Correctness[1])

Deriving ⊥ iff the probem is UNSAT. No conflict + total assignment = SAT. Termination.

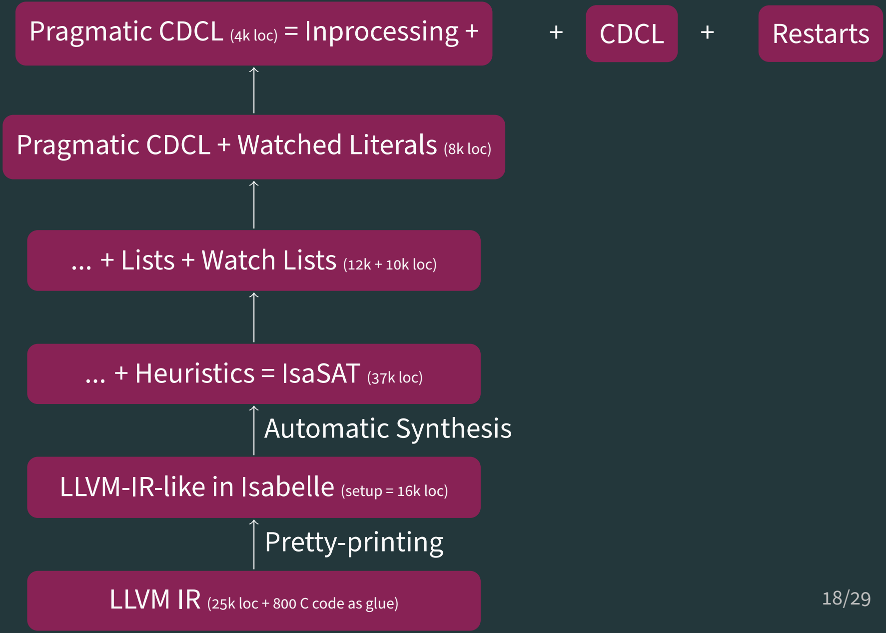
Theorem (Total Correctness IsaSAT-LLVM)

If the answer is not unknown, it is either SAT with a model or UNSAT.

IsaSAT-SML had full correction SML semantics does not forbid arrays $\geq 2^{64}$, no compiler support

[1] At some point, memory representation can cause also aborts.

Refinement in IsaSAT



Deriving the empty clause: unsat (OR: derive conflict at level 0)

Conflicts on current level: completeness of propagations

Termination: Yes IsaSAT can answer unknown

if too many clauses $\sum_{c \in \text{clauses}} 5 + |c| \approx |\text{clause\_memory}| \geq 2^{63}$

No crash: yes (up to the assumptions on memory) allocation does not fail

Making the Solver more Complex: Adding Restart?

Assume you already have a working CDCL.

Adding restarts means:

1. change your CDCL (to include a counter to increase restart interval)
2. change the refinement to be based on the extended CDCL
3. add restarts with the counter. Make sure that it does not overflow.

That is all

except for heuristics, performance debugging, ...

Challenges

What is hard?

- you have to prove everything lots of code
- limited by the speed of your tools bring Isabelle to its knees
- hard to find people Isabelle and code synthesis can be seen as two different systems

Refinements

In retrospect over the entire project:

- Many components that are not independent   everything is parametrized by the set of variables...

  Watch list can be indexed by every literal in the set of clauses

- Mistakes have been made: too much coupling   ... that is not duplicated

  Better: watch lists are defined over a set of literals that is the same as the set of clauses   also moving up proof that index valid

- But: refactoring takes time.

Refinements

In retrospect over the entire project:

- Many components that are not independent   everything is parametrized by the set of variables...

  Watch list can be indexed by every literal in the set of clauses

- Mistakes have been made: too much coupling   ... that is not duplicated

  Better: watch lists are defined over a set of literals that is the same as the set of clauses   also moving up proof that index valid

- But: refactoring takes time.

Refinements

In retrospect over the entire project:

- Many components that <u>are not independent</u> everything is parametrized by the set of variables...

  Watch list can be indexed by every literal in the set of clauses
- Mistakes have been made: too much coupling ... that is not duplicated

  Better: watch lists are defined over a set of literals that is the same as the set of clauses also moving up proof that index valid
- But: refactoring takes time.

Refinements

In retrospect over the entire project:

- Many components that are not independent everything is parametrized by the set of variables...

  Watch list can be indexed by every literal in the set of clauses
- Mistakes have been made: too much coupling ... that is not duplicated

  Better: watch lists are defined over a set of literals that is the same as the set of clauses also moving up proof that index valid
- But: refactoring takes time.

Refinements

In retrospect over the entire project:

- Many components that are not independent everything is parametrized by the set of variables...

  Watch list can be indexed by every literal in the set of clauses

- Mistakes have been made: too much coupling ... that is not duplicated

  Better: watch lists are defined over a set of literals that is the same as the set of clauses also moving up proof that index valid

- But: refactoring takes time.

Refinements

In retrospect over the entire project:

- Many components that are not independent everything is parametrized by the set of variables...
  Watch list can be indexed by every literal in the set of clauses
- Mistakes have been made: too much coupling ... that is not duplicated
  Better: watch lists are defined over a set of literals that is the same as the set of clauses also moving up proof that index valid
- But: refactoring takes time.

Refinements

In retrospect over the entire project:

- Testing new features hard Some I implemented and proved things that did not work and I removed.
- Testing improvement for code generation structure was forced, not a choice. Pointers

What can you not express?

- aliasing

```
struct ISASAT {
    TRAIL trail;
    CLAUSES clauses; ....
};

struct ISASAT solver;
isasat->trail = assign(lit, solver->trail);
```

- pointers are complicated IsaSAT: I tried to use a pointer to a state and never managed to make it less than 10 times slower

What can you not express?

- aliasing

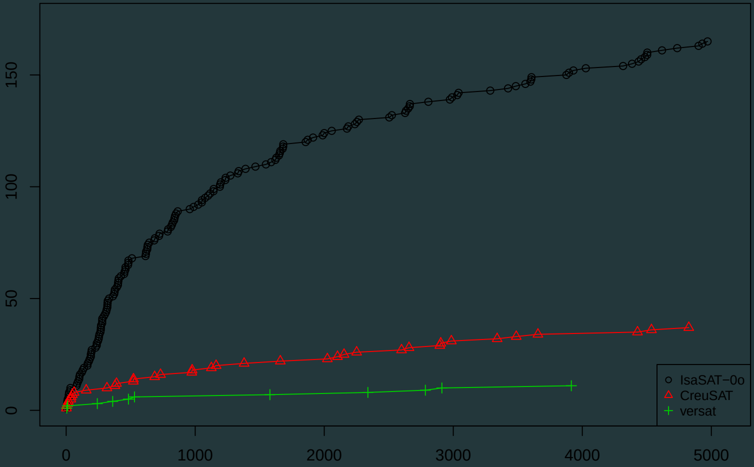
```
struct ISASAT {
    TRAIL trail;
    CLAUSES clauses; ....
};

struct ISASAT solver;
isasat->trail = assign(lit, solver->trail);
```

- pointers are complicated IsaSAT: I tried to use a pointer to a state and never managed to make it less than 10 times slower

The Code

How do they perform?



CDF of various verified solvers on the SC2022 (7 GB, 5500 s)

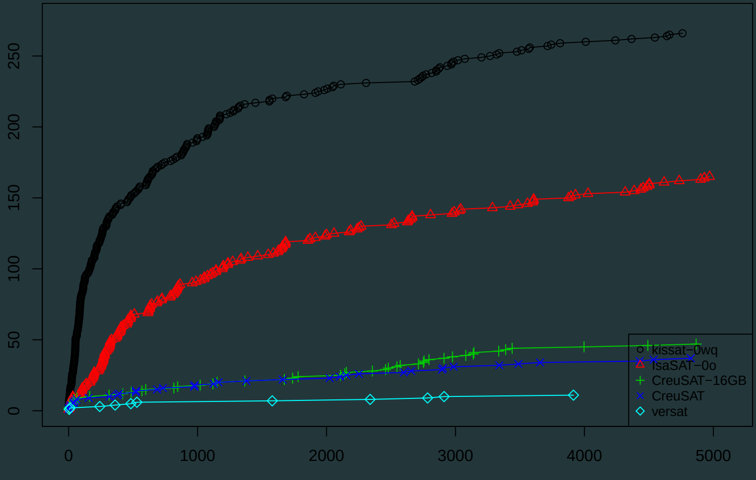
many out-of-memory for CreuSAT

How do they perform?

Solver	SAT	UNSAT
IsaSAT	175	130
Creusat	145	79
versat	60	62

Table 1: Results on the SC2015 according to Skotåm (24 GB, 1800 s)

How do they perform?



CDF of various solvers on the SC2022 (7 GB, 5500 s)

many out-of-memory for CreuSAT

How good is the code? Guru

```
void * gpropagate_h(int gnv_24, int gdl_4, void * gas_37, void * gws_17) {
 start_gpropagate_h: {
{/* match with exactly one case: gassign_state */
void * gpa_13;
void * gwhy_6;
void * gdls_6;
void * ghist_6;
int ghist_cur_4;
int ghist_end_4;
void * gcarraway_tmp_119;
gpa_13 = ginit_unique_unique(guwarray, gas_37, ((gAssignState_gassign_state *)gas_37)->gpa_2)
;
gwhy_6 = ginit_unique_unique(gwarray, gas_37, ((gAssignState_gassign_state *)gas_37)->gwhy_2)
;
gdls_6 = ginit_unique_unique(guwarray, gas_37, ((gAssignState_gassign_state *)gas_37)->gdls_2
;
ghist_6 = ginit_unique_unique(guwarray, gas_37, ((gAssignState_gassign_state *)gas_37)->ghist_
;
[...]
switch ((int)gcarraway_tmp_120) {

case op_gff: {

fprintf(stderr,"abort at /Users/kain/Projects/versat/old_versions/0.6/src/unitprop.g, line 76
```

How good is the code? IsaSAT

```
define ISASAT_STATE @unit_propagation_outer_loop_wl_D(ISASAT_STATE %x) #0 {

  start:
    %x1 = call i8 @IsaSAT_Profile_PROPAGATE ()
    call void @IsaSAT_Profile_LLVM_start_profile (i8 %x1)
    br label %while_start

  while_start:
    %s = phi ISASAT_STATE [ %x3, %while_body ], [ %x, %start ]
    %x2 = call i1 @literals_to_update_wl_empty_fast_code (ISASAT_STATE %s)
    br i1 %x2, label %while_body, label %while_end

  while_body:
    %xb = call { ISASAT_STATE, i32 } @select_and_remove_from_literals_to_update_wl(ISASAT_STA
    %a1 = extractvalue { ISASAT_STATE, i32 } %xb, 0
    %a2 = extractvalue { ISASAT_STATE, i32 } %xb, 1
    %x3 = call ISASAT_STATE @unit_propagation_inner_loop_wl_D (i32 %a2, ISASAT_STATE %a1)
    br label %while_start

  while_end:
    %xc = call i8 @IsaSAT_Profile_PROPAGATE ()
    call void @IsaSAT_Profile_LLVM_stop_profile (i8 %xc)
    ret ISASAT_STATE %s
}
```

(only edit: ISASAT_STATE is unfolded in the code and remove prefix from function names)

How good is the code? CreuSAT

```
#[cfg_attr(feature = "trust_unit", trusted)]
#[ensures(f.equisat(^f))]
pub fn unit_propagate(f: &mut Formula, trail: &mut Trail, watches: &mut Watches) -> Result<()
    let mut i = trail.curr_i;
    let old_trail: Ghost<&mut Trail> = ghost! { trail };
    let old_f: Ghost<&mut Formula> = ghost! { f };
    let old_w: Ghost<&mut Watches> = ghost! { watches };
    #[invariant(trail_inv, trail.invariant(*f))]
    while i < trail.trail.len() {
        let lit = trail.trail[i].lit;
        match propagate_literal(f, trail, watches, lit) {
            Ok(_) => {}
            Err(cref) => {
                return Err(cref);
            }
        }
        i += 1;
    }
    trail.curr_i = i;
    Ok(())
}
```

(only edit: remove some invariants and ensures)

Conclusion

Comparison: How different are there really?

- Removing assertions from bottom-up means being more top-down and requires more proofs where automation struggles
- Very hard to remove proofs from top-down
- Link top-down with concrete code? Currently has not been tried but I am trying to find a student

Conclusion

- Only application of verified SAT solvers: finishing last at SAT Competition, getting Masters, or PhDs
- But: do you have applications where proof checking is not possible?