One Thousand and One Refinement: From CDCL to a Verified SAT Solver

Mathias Fleury

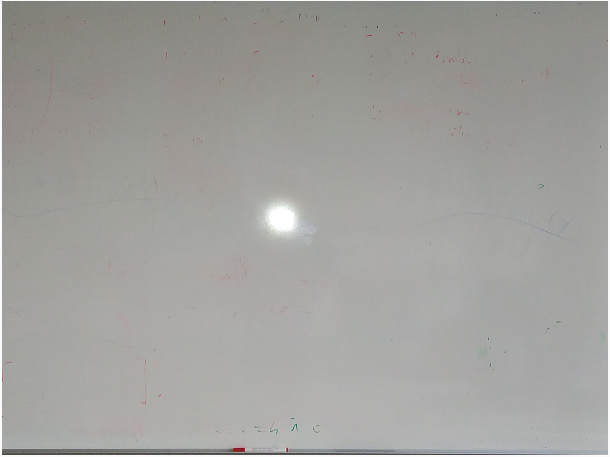
2020/01/28

SAARLAND UNIVERSITY
SAARBRÜCKEN GRADUATE SCHOOL OF COMPUTER SCIENCE

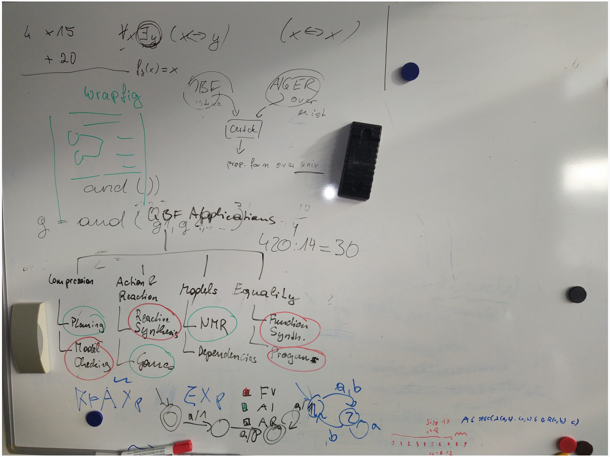
SIC Saarland Informatics Campus

max planck institut informatik

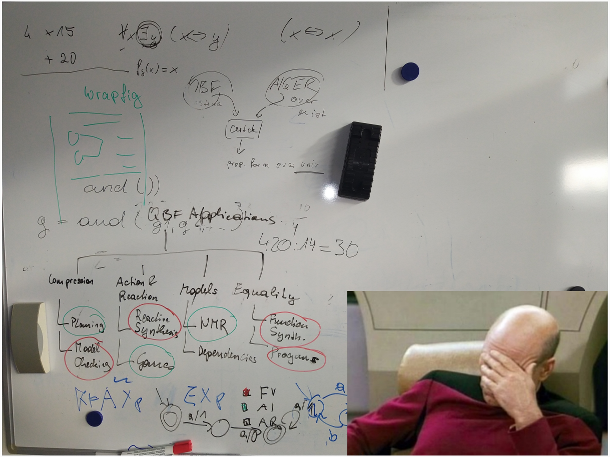
When you start your proof



After a few days…



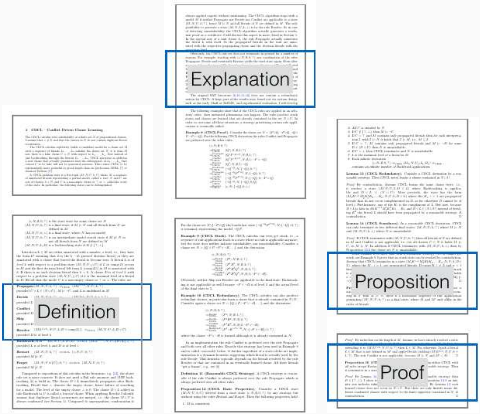
After a few days… Mistake!



Then you write your paper

Paper accepted = Proof correct

Then you extend your paper



Paper accepted = Proof correct

What about ITPs?

When you start...

What about ITPs?

When you start... Before you finish

Paper proofs vs proof assistants

IsaFoL project

Isabelle Formalisation of Logic

The IsaFoL project: motivation

Eat your own dog food

- case study for proof assistants and automatic provers

Build state-of-the-art libraries

- Automated Reasoning: The Art of Generic Problem Solving
  (ongoing textbook project by Christoph Weidenbach)

Focus on meta-theorems

- reuse proofs
- be general

The IsaFoL project: content

Excerpts of the IsaFoL project:

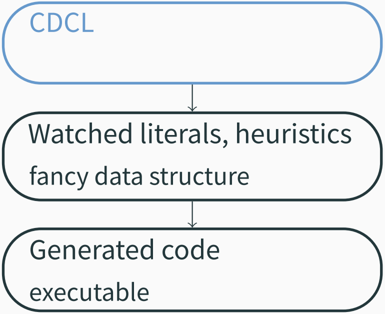
- Resolution, ordered resolution, and prover by Schlichtkrull et al. [ITP'16, IJCAR'18, CPP'19]
- Superposition by Peltier [AFP'16]
- UNSAT Checker by Lammich [CADE 27]
- CDCL and SAT solver [IJCAR'16, JAR'16, IJCAI'17, CPP'19, NFM'19]

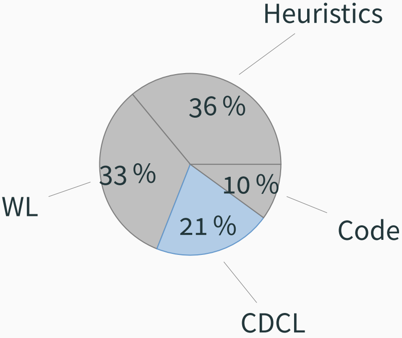
The IsaFoL project: content

Excerpts of the IsaFoL project:

- Resolution, ordered resolution, and prover by Schlichtkrull et al. [ITP'16, IJCAR'18, CPP'19]
- Superposition by Peltier [AFP'16]
- UNSAT Checker by Lammich [CADE 27]
- **CDCL and SAT solver [IJCAR'16, JAR'16, IJCAI'17, CPP'19, NFM'19]**

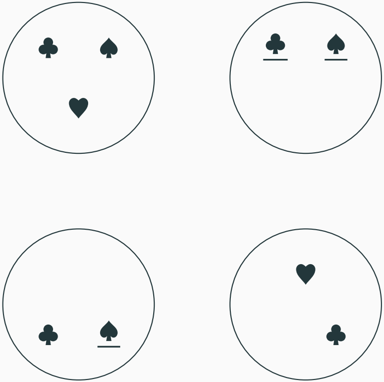
CDCL





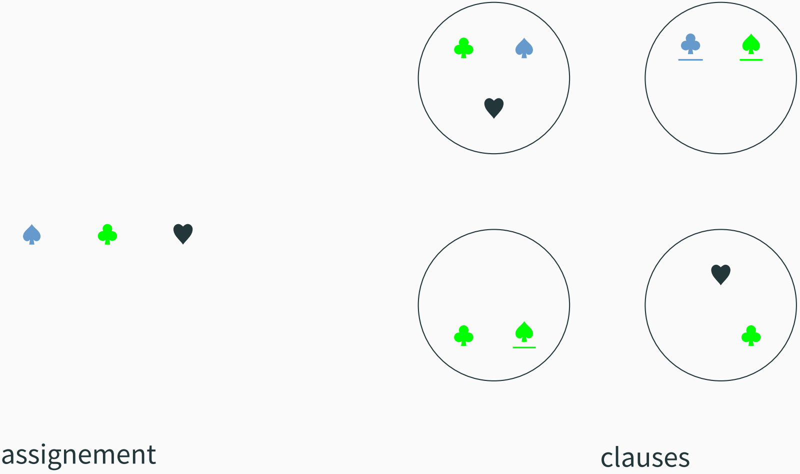
Formalisation length (total: 78 000 lines of code)

CDCL explanation

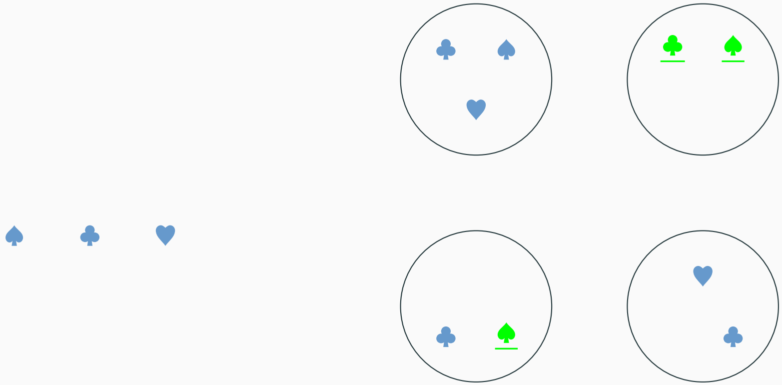


clauses

CDCL explanation



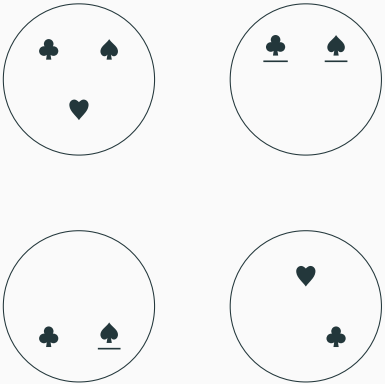
CDCL explanation



assignement

clauses

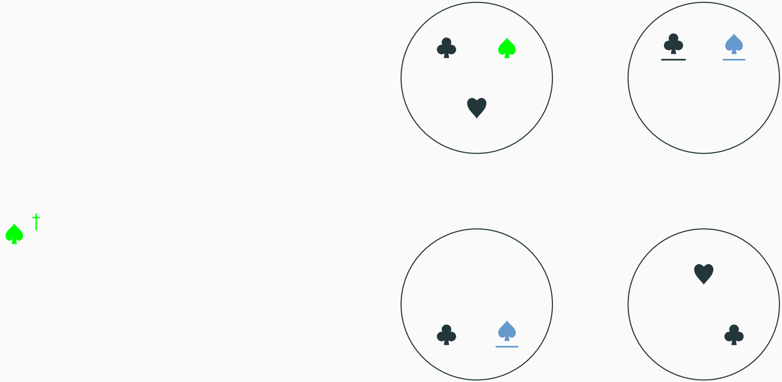
CDCL explanation



assignement = trail

clauses

CDCL explanation



assignement = trail

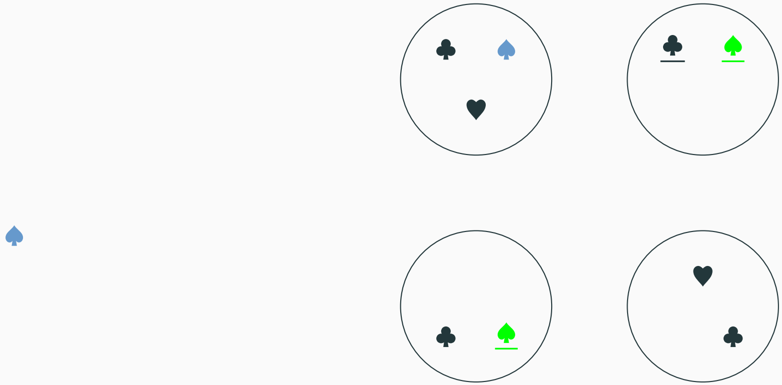
clauses

CDCL explanation

assignement = trail

clauses

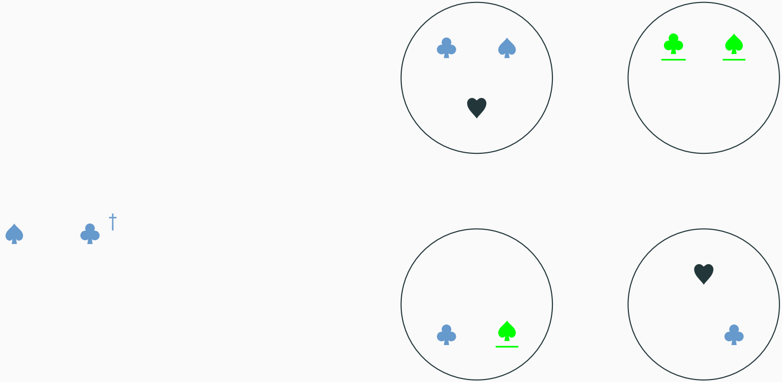
CDCL explanation



assignement = trail

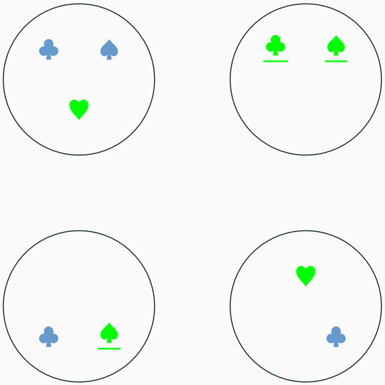
clauses

CDCL explanation



assignement = trail

clauses



assignement = trail

clauses

Refinement by specialisation

Core of CDCL is DPLL+BJ

back to some decision

DPLL+BJ =	Propagate +	Decide +	Backjump
			UI
DPLL =	Propagate +	Decide +	Backtrack

back to latest decision

Refinement by specialisation

Core of CDCL is DPLL+BJ

back to some decision

DPLL+BJ =	Propagate +	Decide +	Backjump
			⊒
DPLL =	Propagate +	Decide +	Backtrack

back to latest decision

How to maximize reuse?

Backtrack = Parametrised Backjump (Backtrack_cond)

Backjump on paper vs. in Isabelle

Backjump on paper

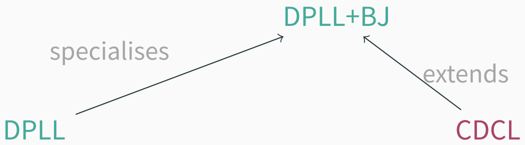
if $C \in N$ and $M \models \neg C$ and there is a C' such that...
then $(M, N) \Rightarrow_{CDCL} (M'L, N)$.

Definition (Parametrised Backjump in Isabelle)

if $C \in N$ and $M \models \neg C$ and there is a C' such that...
and $BJ\_cond\ C'$
then $(M, N) \Rightarrow_{CDCL} (M'L, N)$.

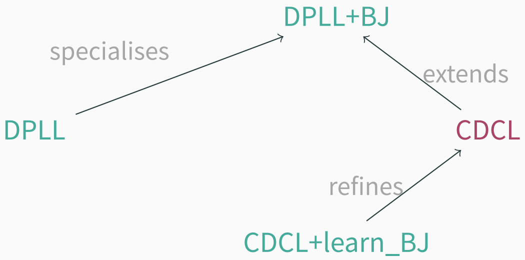

```
sublocale DPLL < DPLL+BJ where
  DPLL+BJ_Cond = DPLL_Cond
```

Development hierarchy



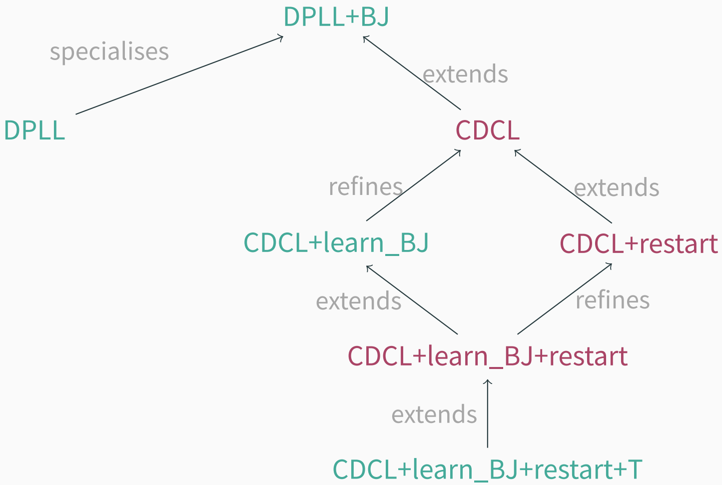
CDCL = DPLL+BJ + Learn + Forget

Development hierarchy



Strategy used in most implementations:
learn only backjump clause

Development hierarchy



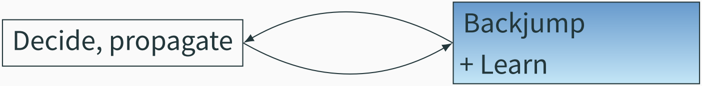
Definition (Parametrised Backjump (BJ_cond))

if $C \in N$ and $M \models \neg C$ and there is a C' such that...
and $BJ\_cond\ C'$
then $(M, N) \Rightarrow_{CDCL} (L^{\dagger}M', N)$.

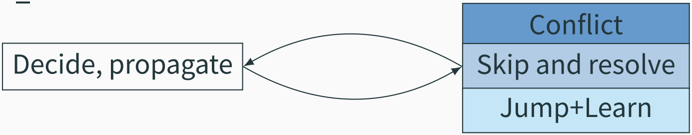
How to get a suitable C'?

Refinement by inclusion

CDCL_learn_BJ



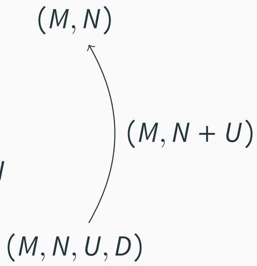
CDCL_W



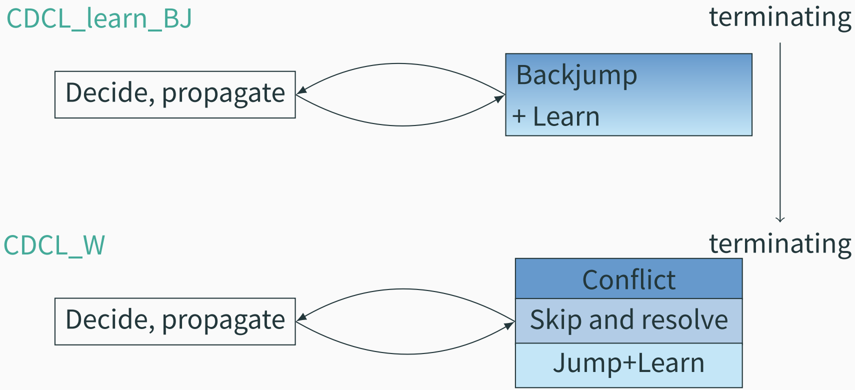
Refinement by inclusion

CDCL_learn_BJ

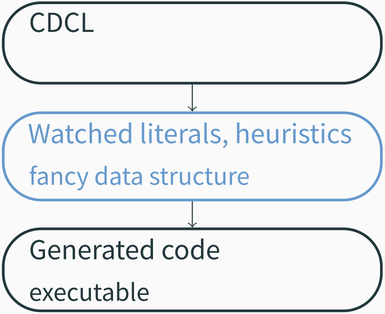
CDCL_W

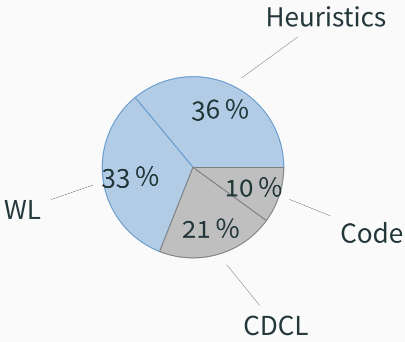


Refinement by inclusion



Refining Data Structures

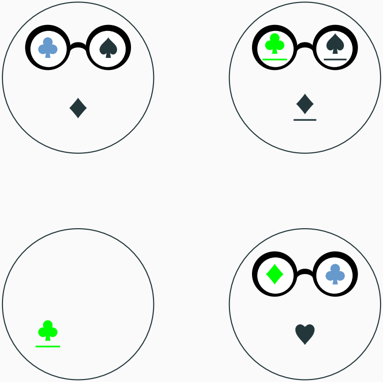




Formalisation length (total: 78 000 lines of code)

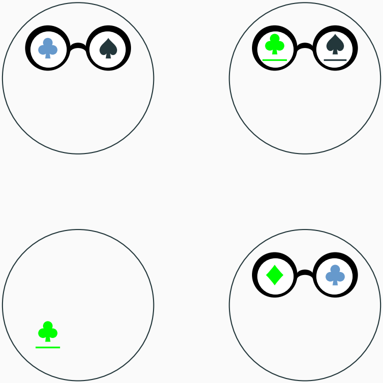
Watched literals explanation

Watched literals = sophisticated data structure to identify propagations and conflicts.



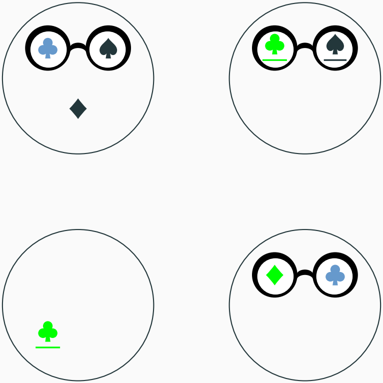
Watched literals explanation

Watched literals = sophisticated data structure to identify propagations and conflicts.



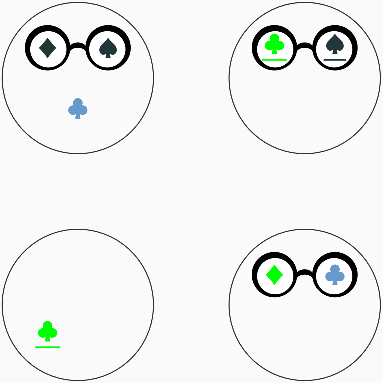
Watched literals explanation

Watched literals = sophisticated data structure to identify propagations and conflicts.



Watched literals explanation

Watched literals = sophisticated data structure to identify propagations and conflicts.



Watched literals

First formalisation attempt failed.

Development done in two steps:

Watched literals

First formalisation attempt failed.

Development done in two steps:

1. watched literals...
2. ... extended with blocking literals

Watched literals

First formalisation attempt failed.

Development done in two steps:

1. watched literals...
2. ... extended with blocking literals

My Approach non-deterministic transition system

Refinement in the non-determinism monad

Then we enter the non-determinism monad:

- closer to programs
- preserves non-determinism

Refinement in the non-determinism monad

Then we enter the non-determinism monad:

- closer to programs
- preserves non-determinism

Abstract level:

$$\text{OBTAIN should\_restart such that}$$
$$\text{should\_restart} \implies \#\text{conflict} > \text{threshold}$$

Refinement in the non-determinism monad

Then we enter the non-determinism monad:

- closer to programs
- preserves non-determinism

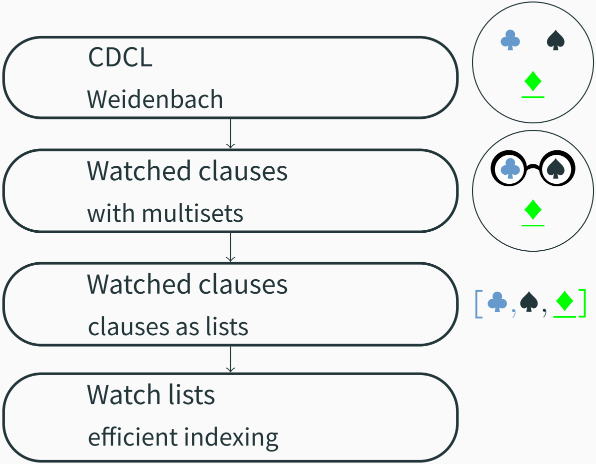
Abstract level:

$$\mathrm{OBTAIN}\ \mathrm{should\_restart}\ \text{such that}$$
$$\mathrm{should\_restart} \implies \#\mathrm{conflict} > \mathrm{threshold}$$

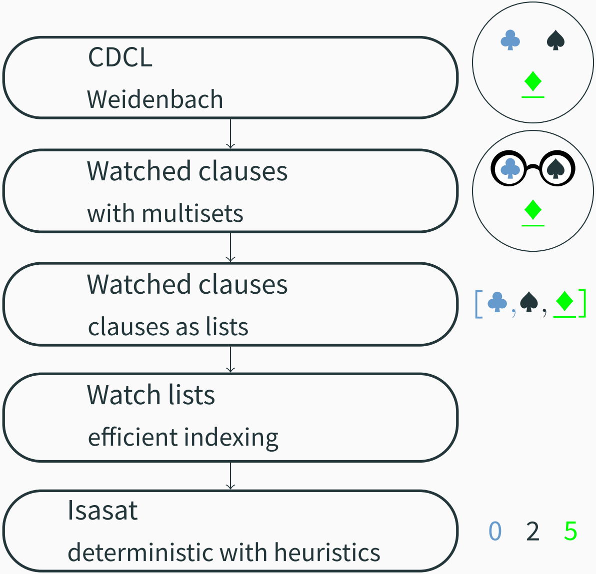
Concrete level:

$$\mathrm{should\_restart} \leftarrow \mathrm{RETURN}(\#\mathrm{conflict} > \mathrm{threshold} \land \mathrm{heuristic})$$

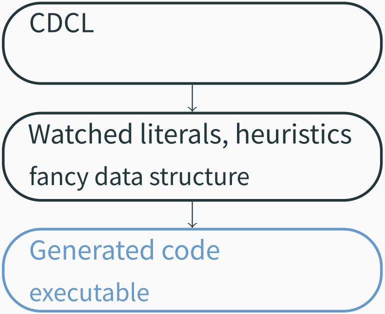
Refinement to keep abstractions

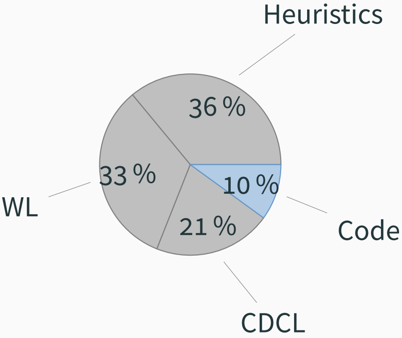


Refinement to keep abstractions



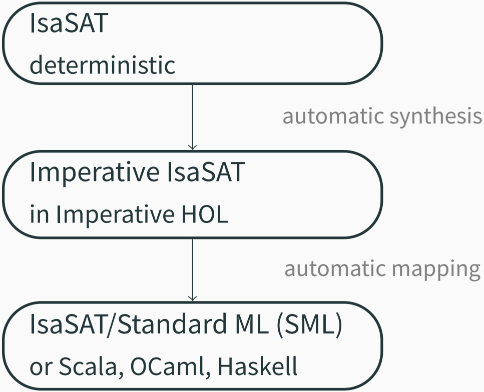
Generating Code





Formalisation length (total: 78 000 lines of code)

What is the imperative code?



Code synthesis and generation

Abstract code:

```
ASSERT(i < length xs);
RETURN(xs!i);
```

Code synthesis and generation

Abstract code:

```
ASSERT(i < length xs);
RETURN(xs!i);
```

After synthesis by Sepref in Imperative HOL:

```
Array.nth xs i
```

Code synthesis and generation

Abstract code:

```
ASSERT(i < length xs);
RETURN(xs!i);
```

After synthesis by Sepref in Imperative HOL:

```
Array.nth xs i
```

After printing in SML, via code equations and printing:

```
Array.sub(xs, i)
```

Code synthesis and generation

Abstract code:

```
ASSERT(i < length xs);
RETURN(xs!i);
```

After synthesis by Sepref in Imperative HOL:

```
Array.nth xs i
```

After printing in SML, via code equations and printing:

```
Array.sub(xs, i)
```

A native array

Code synthesis and generation

Abstract code:

```
ASSERT(i < length xs);
RETURN(xs!i);
```

After synthesis by Sepref in Imperative HOL:

```
Array.nth xs i
```

After printing in SML, via code equations and printing:

```
if i < Array.size xs
then xs[i]
else raise OutOfBound
```

Code synthesis and generation

Abstract code:

```
ASSERT(i < length xs);
RETURN(xs!i);
```

Information is lost during translation

After synthesis by Sepref in Imperative HOL:

```
Array.nth xs i
```

After printing in SML, via code equations and printing:

```
if i < Array.size xs
then xs[i]
else raise OutOfBound
```

In IsaSAT removed by a compiler flag...

Code synthesis and generation

Abstract code:

```
ASSERT(i < length xs);
RETURN(xs!i);
```

After synthesis by Sepref in Imperative HOL:

```
Array.nth xs i
```

In the nice Isabelle world GMP integer

After printing in SML, via code equations and printing:

```
if i < Array.size xs
then xs[i]
else raise OutOfBound
```

Code synthesis and generation

Abstract code:

```
ASSERT(i < length xs);
RETURN(xs!i);
```

In IsaSAT, uint64 integer until it does not fit

After synthesis by Sepref in Imperative HOL:

```
Array.nth xs i        Array.nth_uint64 xs i
```

After printing in SML, via code equations and printing:

```
if i < Array.size xs
then xs[i]                 Array.nth_uint64(xs, i)
else raise OutOfBound
```

Correctness theorem

Theorem

If the input is well formed and UNSAT (resp. SAT), then IsaSAT terminates and it returns UNSAT (resp. SAT with a model).[1]

[1] if the Standard ML compiler is able to allocate large enough arrays

Correctness theorem

Theorem

If the input is well formed and UNSAT (resp. SAT), then IsaSAT terminates and it returns UNSAT (resp. SAT with a model).[1]

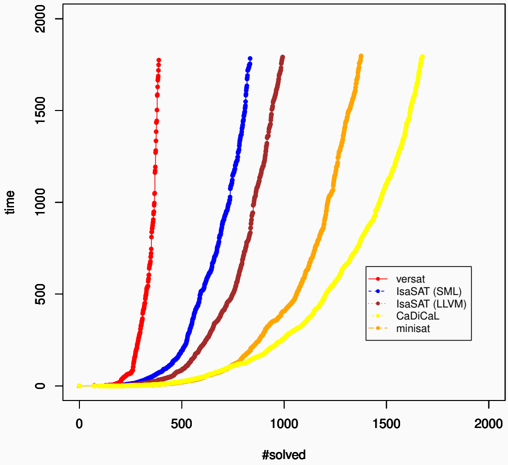
And the only other efficient verified solver

Theorem (Correctness versat)

If the input is well formed and the solver returns UNSAT, then the problem is UNSAT.

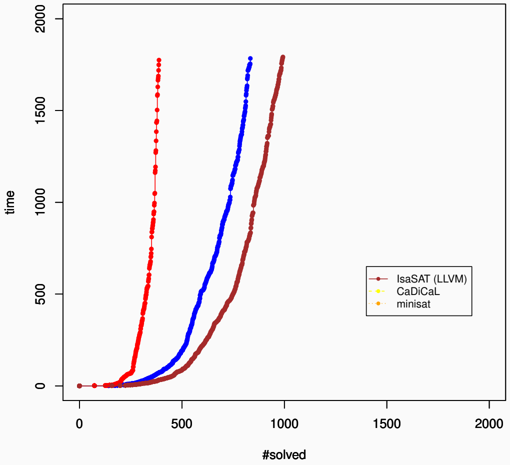
[1] if the Standard ML compiler is able to allocate large enough arrays

Performance



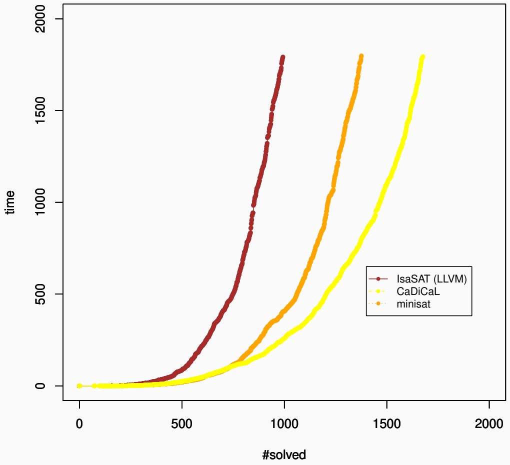
Comparison of various SAT solvers on preprocessed instances

Performance



Comparison of various SAT solvers on preprocessed instances

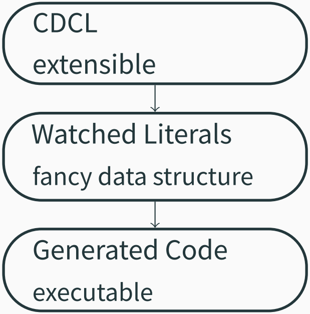
Performance



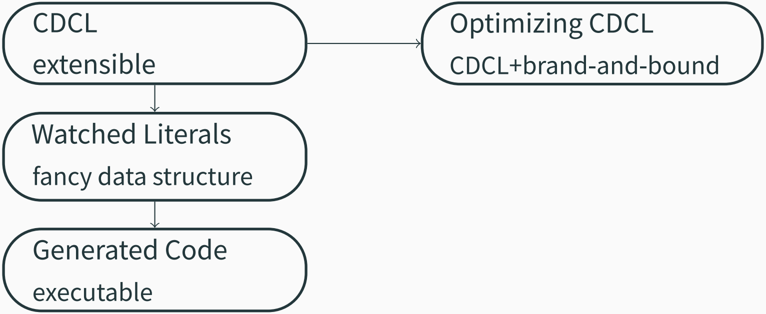
Comparison of various SAT solvers on preprocessed instances

Conclusion

Conclusion


CDCL
extensible
Watched Literals
fancy data structure
Generated Code
executable

Conclusion


CDCL
extensible
Optimizing CDCL
CDCL+brand-and-bound
Watched Literals
fancy data structure
Generated Code
executable

O Captain! My Captain!
Now comes the appendix, go back to the previous slide

Appendix Outline

What is hard?

Why is it so hard?

Size

Mostly about definitions

Formalisation part	Length (kloc)	
CDCL Libraries	3	Entailment
CDCL	17	
Refinement Libraries	6	Setup for machine words, arrays of arrays
Refinement except last layer	26	
Heuristics	35	code synthesis, lots of code

Size

Mostly about definitions

Formalisation part	Length (kloc)	
CDCL Libraries	3	Entailment
CDCL	17	
Refinement Libraries	6	Setup for machine words, arrays of arrays
Refinement except last layer	26	
Heuristics	35	code synthesis, lots of code

Size

Mostly about definitions

Aliasing and ownership

Formalisation part	Length (kloc)	
CDCL Libraries	3	Entailment
CDCL	17	
Refinement Libraries	6	Setup for machine words, arrays of arrays
Refinement except last layer	26	
Heuristics	35	code synthesis, lots of code

Size

Mostly about definitions

Aliasing and ownership

Formalisation part	Length (kloc)	
CDCL Libraries	3	Entailment
CDCL	17	
Refinement Libraries	6	Setup for machine words, arrays of arrays
Refinement except last layer	26	
Heuristics	35	code synthesis, lots of code

Size

Mostly about definitions

Aliasing and ownership

Single threaded

Formalisation part	Length (kloc)	
CDCL Libraries	3	Entailment
CDCL	17	
Refinement Libraries	6	Setup for machine words, arrays of arrays
Refinement except last layer	26	
Heuristics	35	code synthesis, lots of code

Refinement

Refinement in the non-determinism monad: Data structure

Abstract level:

$$\mathrm{OBTAIN}\, L \text{ s.t. } L \in C$$

Concrete level:

$$\mathrm{blit} \leftarrow \mathrm{RETURN}(\mathrm{watcher.blit})$$

Correctness and Trust

Correctness

And IsaSAT/LLVM:

Theorem (Correctness IsaSAT/LLVM)

If the input is a valid input and the solver returns SAT (UNSAT), then the problem is SAT (UNSAT).

Isabelle protects of:

- programming errors (out-of-bound)
- correctness errors (SAT instead of UNSAT)

But not of:

- performance bugs (restarts)

What do you trust?

IsaSAT/SML	IsaSAT/LLVM	CaDiCaL
Parser	Parser	The parser CDCL
Code equations	Isabelle's LLVM Semantics	Implementation
Compiler	LLVM ~2 faster than SML, ~10 times less memory	Compiler

What do you trust?

IsaSAT/SML	IsaSAT/LLVM	CaDiCaL
Parser	Parser	The parser
		CDCL
Code equations	Isabelle's LLVM Semantics	Implementation
Compiler	LLVM	Compiler
	$\sim$2 faster than SML, $\sim$10 times less memory	

There is no bug that happens after two years of calculation because you wrote `uint64_max - 4` instead of `uint64_max - 5`

Features

Techniques in IsaSAT

VMTF decision heuristic	Critical
Conflicts as hash-table and array	Critical
Recursive conflict minimization	Critical
Arena-based memory	I never saw a difference
Blocking literals + position saving	Helps a lot
EMA-14 restarts + trail reuse	Helps, but I still don't understand what CaDiCaL does
Special handling of binary clauses	I never saw a difference

Missing Features

Missing Features

Two trivial but key features

- deletion of true clauses
- removal of false literals

Solution: “pragmatic CDCL” with resolution rules to simplify clauses set

CDCL

Is Weidenbach's CDCL the right CDCL?

Easy to add:

Definition (Conflict Minimisation)

Learn a clause $D' \vee L' \subseteq D \vee L$ if $N \models D' \vee L'$.

Impossible to add (it breaks invariants):

Definition (Inprocessing)

An irredundant clause is subsumed by a learned clause: make the latter irredundant.

but!

If we go with

$$(M, N, N_{\text{subsumed}}, U, U_{\text{subsumed}}, D)$$

and do not consider subsumed clauses, CDCL can see

$$(M, N + N_{\text{subsumed}}, U + U_{\text{subsumed}}, D)$$

and everything will work as expected.

Complexity

Complexity

As for SAT implementations,

Never-ending task there is always one more heuristic or one more technique to implement...

No tooling ... makes it even harder

Testing a heuristic is hard

Complexity

On the proof side

Proving Correctness time consuming (overflow problems), Isabelle is slow

Side conditions of CDCL

Property (CDCL Invariant)

The set of all literals you consider is exactly the set of literals in the set of clauses.

Evaluator	Performance	
MLton	2.5 s	includes parsing
PolyML	43 s	
value	?	requires 64-bit PolyML
nbe, simp	⊥	do not know about Imperative HOL

What makes refinement hard?

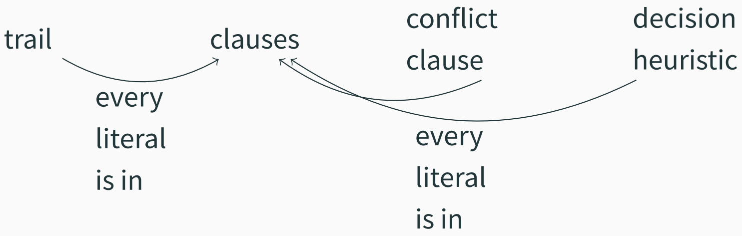
Refinement is easy when:

- you can ignore the result of operations
- i.e., reduce interdependency between components of the state

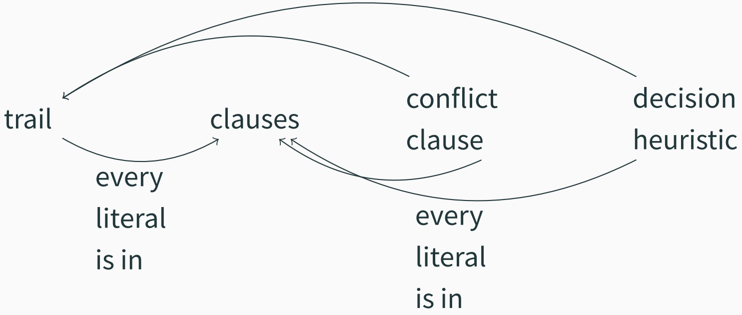
```
M <- RETURN (Decided L .  M)
```

What is the impact on the other components?

What makes refinement hard?



What makes refinement hard?



Importing Correctness in Isabelle

Idea

Abstract code:

```
ASSERT(i < length xs);
RETURN (xs ! i);
```

After synthesis, done automatically by Sepref:

```
return xs[i]
```

Can we run it in Isabelle?

- result cannot be extracted from the `return` (imperative monad)...
- ... but we can generate a purely functional version...
- ... which is what I optimised for

Evaluator	Performance	
MLton	2.5 s	includes parsing
PolyML	43 s	
value	?	requires 64-bit PolyML
nbe, simp	$\bot$	do not know about Imperative HOL, so you cannot allocate arrays

IsaSAT/LLVM vs IsaSAT/MLton

LLVM is better and has an easier job

- LLVM has more man-power: MLton's LLVM backend produces slightly better code
- LLVM's IR is the target for tools vs target for humans (Isabelle's code generator produces terrible and unreadable code)
- LLVM's input is the code you would expect

LLVM has more freedom to do a good job

- The code is not functional at all and contains barely any datatype
- ML enforces sharing, which is good until is not
  1. $\lambda(\#\textit{props}, \textit{stats}).\,(\#\textit{props} + 1, \textit{stats})$ reallocates
  2. `clause_ref * (bool * literal)`[2] needs more memory than `struct {clause_ref; struct {bool; literal};}` (cache problems!)
- Array access and conversions are checked[3]

[2] Isabelle is not able to generate `clause_ref * bool * literal` and using a tuple made things worse

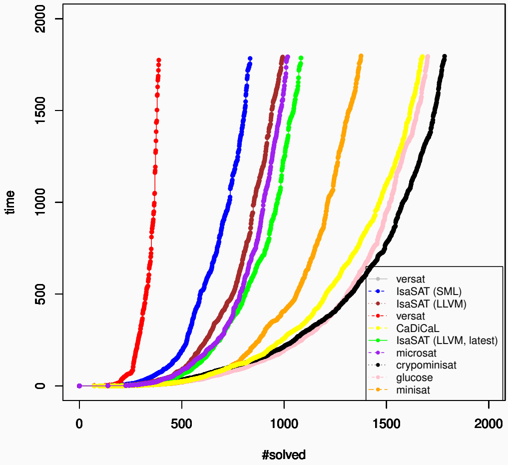
[3] although I deactivate these checks

Memory is not cheap

- IsaSAT/ML uses 10 times more memory
- IsaSAT/ML uses the GC... but I have no idea why: IsaSAT uses base types (or with in-place operations) and arrays resizing (freeing the old one is enough)

Performance

Performance



Comparison of various SAT solvers on preprocessed instances

OCDCL

Conjecture

OCDCL+stgy performs at most 3^n Backtrack steps.

Lemma (verified in Isabelle)	**Conjecture**
ODPLL+stgy performs at most 3^n *Backtrack steps.*	OCDCL+stgy performs at most 3^n Backtrack steps.

Lemma (verified in Isabelle)

ODPLL+stgy performs at most 3^n Backtrack steps.

Conjecture

OCDCL+stgy performs at most 3^n Backtrack steps.

Proof.

- trails are not repeated
- trails have a certain form
- and they are such 3^n such trail

□

Lemma (verified in Isabelle)

ODPLL+stgy performs at most 3^n Backtrack steps.

Proof.

- trails are not repeated
- trails have a certain form
- and they are such 3^n such trail

□

Conjecture

OCDCL+stgy performs at most 3^n Backtrack steps.

Proof.

- ~~trails are not repeated~~
- trails have a certain form
- and they are such 3^n such trail

□

Problem: backjump is nearly a restart.

Related Work

Related Work

	Marić 2008 Isabelle	Lescuyer 2011 Coq	Schankar et al 2011 PVS	Oe et al 2012 Guru
Backjumping				
Learning	-	*		
Soundness				
Compeleteness				-
Implementation			-	
Termination				-
Restart+Forget	-	-	-	-
WL	~	-	-	~