Lost in translation
how easy problems become hard due to bad encodings

Vampire Workshop 2015

Leonardo de Moura
Microsoft Research
I wanted to give the following talk

http://leanprover.github.io/
Automated Reasoning Tools
as a service

- Software/Hardware Verification Tools
- Test-case generators
- Static Analyzers
- ...
The “dream”

Automated reasoning tools as black boxes

- SAT Solvers
- SMT Solvers
- ATP
Example 1: SAT solvers and Tseitin encoding

- Most SAT solvers expect the input formula to be in CNF.
- In practice, it is not feasible to convert formulas into CNF using equivalences such as:

<table>
<thead>
<tr>
<th>eliminate $\Rightarrow$</th>
<th>$A \Rightarrow B \equiv \neg A \lor B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce the scope of $\neg$</td>
<td>$\neg(A \lor B) \equiv \neg A \land \neg B$, $\neg(A \land B) \equiv \neg A \lor \neg B$</td>
</tr>
<tr>
<td>apply distributivity</td>
<td>$A \lor (B \land C) \equiv (A \lor B) \land (A \lor C)$, $A \land (B \lor C) \equiv (A \land B) \lor (A \land C)$</td>
</tr>
</tbody>
</table>
Example 1:
SAT solvers and Tseitin encoding

However, there is a *linear time* translation to CNF that produces an *equisatisfiable* formula. Replace the distributivity rules by the following rules:

\[
\frac{F[l_i \ op \ l_j]}{F[x], \ x \Leftrightarrow l_i \ op \ l_j}
\]

\[
\frac{x \Leftrightarrow l_i \lor l_j}{-x \lor l_i \lor l_j, -l_i \lor x, -l_j \lor x}
\]

\[
\frac{x \Leftrightarrow l_i \land l_j}{-x \lor l_i, -x \lor l_j, -l_i \lor -l_j \lor x}
\]

(\text{*}) \quad x \text{ must be a fresh variable.}
Example 1:
SAT solvers and Tseitin encoding

- Tseitin encoding is easy to implement.
- However, there are several important improvements.
  - Example: detect common sub-formulas.
  - SAT preprocessors (such as SatELite) “fix” naive CNF encodings before invoking the actual SAT solver
  - Good: preprocessors are reused by different research groups
Example 2: Finite model finding & symmetry breaking

• Given a first-order logic formula F, find a finite model M for it

• Procedures: reduce to SAT (or SMT), reduce to EPR

• MACE-style reduction
  • Fix domain D = \{1, \ldots, n\}
  • Create propositional variables for each predicate P and argument vector \( (d_1, \ldots, d_k) \)
    where \( k \) is the rarity of P and \( d_i \) in D
  • Similarly, one proposition \( p_{f,v,r} \) application for each function f argument vector \( v = (d_1, \ldots, d_k) \) and “result” \( r \) in D
  • Convert F into CNF, instantiate, and add
    • function definition constraint: (not \( p_{f,v,r} \)) or (not \( p_{f,v,r'} \))
    • totality constraints: \( (p_{f,v,1} \text{ or } \ldots \text{ or } p_{f,v,n}) \)
Example 2: Finite model finding & symmetry breaking

- The encoding into SMT is simpler. Example: we can use the theory of uninterpreted functions and avoid function definition and totality constraints.

- Symmetry reduction is a very important optimization (in both cases).

  - The MACE-style encoding implies that for each model, all of its isomorphic valuations (obtained by permuting the domain elements) are also models.

- Idea: add symmetry breaking constraints that force that the model we are looking for has a certain canonical form.
Example 2: Finite model finding & symmetry breaking

• Suppose the problem encoder did not include the symmetry breaking problems.

• Now, to achieve good performance the solver developer must try to infer the symmetries (a much harder problem). See “SyMT: finding symmetries in SMT formulas”, by Carlos Areces, David Deharbe, Pascal Fontaine and Ezequiel Orbe

• SMT-LIB has as huge set of finite model finding (QF_UF) problems where symmetry breaking constraints have not been added.

• Consequently, many SMT solvers (e.g., CVC4, veriT, Yices, Z3) do implement SyMT-like procedures to be able to solve these problems efficiently.
Example 3: Sledgehammer

• Sledgehammer is a very successful tool available in the Isabelle Proof assistant

• It converts HOL into FOL and invokes many ATPs (Vampire) and SMT solvers (Z3)

• A lot is lost in the translation.

• Sledgehammer may fail in very simple queries because they are higher-order, but it will succeed once the user has, for example, manually unfolded some definitions.

• We need provers/solvers that can understand HOL and perform proofs by induction. Even if it is just thin layer.
Example 4: Nonlinear arithmetic solvers

- Nonlinear (real polynomial) arithmetic is decidable

  \[ x^2 - 4x + y^2 - y + 8 < 1 \]
  \[ xy - 2x - 2y + 4 > 1 \]

- Expensive decision procedure

- Most efficient complete solvers are based on Cylindrical Algebraic Decomposition (CAD)

- Perform computations with real algebraic numbers
Example 4: Nonlinear arithmetic solvers

- Real algebraic numbers

\[
\sqrt{\frac{1}{9}} - \sqrt{\frac{2}{9}} + \sqrt{\frac{4}{9}} = \sqrt[3]{\sqrt{2} - 1}
\]

\[
x^9 + 3x^6 + 3x^3 - 1, (0,1)
\]
Example 4: Nonlinear arithmetic solvers

- **Bad application**: object placement in 3D virtual world (constraints of the form \(\text{distance}(a, b) < n\))
  - Precision is not important: algebraic numbers are an overkill for this kind of application

- Avoiding real-algebraic numbers
  - replace \( p = 0 \) with \(-\delta < p < \delta\) (for a small \(\delta\))
  - replace \( p \leq 0 \) with \( p < \delta\)

- The resulting problem is satisfiable iff it has a rational model. This trick only works if the solver takes the property into account.

- Remark: we should not apply this transformation to linear equalities since they can be easily eliminated using variable substitution
  - Example, given \(x+y+2=0\), replace \(x\) with \(-y-2\) “everywhere” and delete equation.
Example 4:
Nonlinear arithmetic solvers

- **Bad application**: object placement in 3D virtual world (constraints of the form $\text{distance}(a, b) < n$)
  - Precision is not important: algebraic numbers are an overkill for this kind of application

- **Avoiding real-algebraic numbers**
  - replace $p = 0$ with $-\delta < p < \delta$ (for a small $\delta$)
  - replace $p \leq 0$ with $p < \delta$
  - The resulting problem is satisfiable iff it has a rational model. This trick only works if the solver takes the property into account.

- **Remark**: we should not apply this transformation to linear equalities since they can be easily eliminated using variable substitution
  - Example, given $x+y+2=0$, replace $x$ with $-y-2$ “everywhere” and delete equation.
Example 4: Nonlinear arithmetic solvers

- **Bad idea**: convert nonlinear real arithmetic into nonlinear integer arithmetic (using fixed point encoding).
  - Replace $x$ with $10^k y$ where $y$ is a fresh integer variable and $k$ is the number of decimal places.
- This approximation also avoids algebraic numbers.
- The resulting problem is in an undecidable fragment (Hilbert's 10th problem).
- This encoding was used by a Z3 user.
- **Lesson**: users must have a rough idea on how the solver works.
Example 5: Proof checking in dependent type theory

• Proof assistants based on dependent type theory (e.g., Agda, Coq and Lean) have a builtin notion of reduction.

  • Beta-reduction \((\lambda x, f \, x \, x) \, (g \, a) \rightarrow f \, (g \, a) \, (g \, a)\)

  • Eta-reduction \((\lambda x, f \, x) \rightarrow f\)

  • Iota-reduction

    \[
    \text{nat.primrec} \, c \, f \, 0 \rightarrow c
    \]

    \[
    \text{nat.primrec} \, c \, f \, (\text{succ} \, n) \rightarrow f \, n \, (\text{nat.primrec} \, c \, f \, n)
    \]

• In these systems, we say that \(t\) and \(s\) are definitionally equal if there is an \(r\) such that \(t \rightarrow r \leftarrow s\)

• Zero-step proofs: we can use reflexivity to prove that definitionally equal terms are equal. Example: \((\text{refl} \, 4)\) is a proof for \(2+2 = 4\) since \(2+2\) is convertible to 4.
Example 5: Proof checking in dependent type theory

- A naive definitional equality checker for $t$ and $s$ will simply compute the normal forms for $t$ and $s$ and check whether they are syntactically equal or not.

- In practice, the naive checker will fail in examples such as

  \[ \text{fact} (99+1) \text{ and } \text{fact} 100. \]

- Most proof assistants use the following heuristics for checking whether $(f\ s)$ is definitionally equal to $(f\ t)$.
  
  - If $s$ and $t$ are definitionally equal, then return yes.
  
  - Otherwise, unfold $f$ and try again.

- $(\text{refl} (\text{fact} 100))$ is a compact proof for $\text{fact} (99+1) = \text{fact} 100$, but it is only feasible to check it if the type/proof checker implements an optimization like the one above.
Flexible solvers and provers

We need more flexible tools.

Customized solutions should be easy to build.

Reuse preprocessors and problem encoders.

Solvers should not be big monolithic black boxes, but a collection of tools and procedures.

Open source tools is a must have.

Efforts such as TPTP and SMT-Lib are fundamental
The strategy challenge

To build theoretical and practical tools allowing users to exert strategic control over core heuristic aspects of high performance prover and solvers.
What is a strategy?

Theorem proving as an exercise of combinatorial search.

Strategies are adaptations of general search mechanisms which reduce the search space by tailoring its exploration to a particular class of formulas.
Different strategies for different domains

From timeout to 0.05secs

QBVF = Quantifiers + Bit-vectors + uninterpreted functions

Hardware Fixpoint Checks.
Given: $I[x]$ and $T[x, x']$
\[ \forall x, x' . I[x] \land T^k[x, x'] \to \exists y, y' . I[y] \land T^{k-1}[y, y'] \]

Ranking function synthesis.
Hardware fixpoint checks
Ranking function synthesis
Why is Z3 fast in these benchmarks?

Z3 is using a custom strategy that combines:

• rewriting, SAT, model based quantifier instantiation
Combining Strategies
Main inspiration: LCF-approach
Combining Strategies
Main inspiration: LCF-approach
Combining Strategies
Main inspiration: LCF-approach
Combining Strategies
Main inspiration: LCF-approach
Combining Strategies
Main inspiration: LCF-approach

Proof Builder

Proof Builder

Proof Builder

thm in LCF terminology

proof in LCF terminology
Tactical: combinators

then( Tactic , Tactic ) = Tactic

orelse( Tactic , Tactic ) = Tactic

repeat( Tactic ) = Tactic
SMT Tactic

- Goal
- Tactic
- Subgoals
- Proof builder
- Model builder
SMT Tactic

goal = formula sequence × attribute sequence

proofconv = proof sequence → proof
modelconv = model × nat → model
trt = sat model
| unsat proof
| unknown goal sequence × modelconv × proofconv
| fail

tactic = goal → trt
SMT Tactic

goal = formula sequence × attribute sequence

proofconv = proof sequence → proof
modelconv = model × nat → model
trt = sat model
| unsat proof
| unknown goal sequence × modelconv × proofconv
| fail
tactic = goal → trt

end-game tactics:
never return unknown(sb, mc, pc)
SMT Tactic

\[
\begin{align*}
goal & \quad = \text{formula sequence} \times \text{attribute sequence} \\
proofconv & \quad = \text{proof sequence} \to \text{proof} \\
modelconv & \quad = \text{model} \times \text{nat} \to \text{model} \\
trt & \quad = \text{sat} \ \text{model} \bigg| \ \text{unsat} \ \text{proof} \\
& \quad \ | \quad \ \text{unknown} \ \text{goal sequence} \times \text{modelconv} \times \text{proofconv} \\
& \quad \ | \quad \ \text{fail} \\
tactic & \quad = \text{goal} \to \text{trt}
\end{align*}
\]

non-branching tactics:
\(sb\) is a singleton in unknown\((sb, mc, pc)\)
Trivial goals

Empty goal [ ] is trivially satisfiable

False goal [ ..., false, ...] is trivially unsatisfiable
SMT Tactic: example

\[ a = b + 1, \ (a < 0 \lor a > 0), \ b > 3 \]

Tactic: elim-vars

Proof builder

\[ (b + 1 < 0 \lor b + 1 > 0), \ b > 3 \]

Model builder
SMT Tactic: example

\[ a = b + 1, \ (a < 0 \lor a > 0), \ b > 3 \]
SMT Tactic: example

\[ a = b + 1, (a < 0 \lor a > 0), b > 3 \]

Tactic: split-or

[ \[ a = b + 1, a < 0, b > 3 \] ]
[ \[ a = b + 1, a > 0, b > 3 \] ]

Proof builder

Model builder
### SMT Tactic

<table>
<thead>
<tr>
<th>simplify</th>
<th>propagate-bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnf</td>
<td>propagate-values</td>
</tr>
<tr>
<td>cnf</td>
<td>split-ineqs</td>
</tr>
<tr>
<td>tseitin</td>
<td>split-eqs</td>
</tr>
<tr>
<td>lift-if</td>
<td>rewrite</td>
</tr>
<tr>
<td>bitblast</td>
<td>p-cad</td>
</tr>
<tr>
<td>gb</td>
<td>sat</td>
</tr>
<tr>
<td>vts</td>
<td>solve-eqs</td>
</tr>
</tbody>
</table>
then : \((tactic \times tactic) \rightarrow tactic\)
then\((t_1, t_2)\) applies \(t_1\) to the given goal and \(t_2\) to every subgoal produced by \(t_1\).

then* : \((tactic \times tactic\ sequence) \rightarrow tactic\)
then*\((t_1, [t_{2,1}, ..., t_{2,n}])\) applies \(t_1\) to the given goal, producing subgoals \(g_1, ..., g_m\).
If \(n \neq m\), the tactic fails. Otherwise, it applies \(t_{2,i}\) to every goal \(g_i\).

orelse : \((tactic \times tactic) \rightarrow tactic\)
orelse\((t_1, t_2)\) first applies \(t_1\) to the given goal, if it fails then returns the result of \(t_2\) applied to the given goal.

par : \((tactic \times tactic) \rightarrow tactic\)
par\((t_1, t_2)\) executes \(t_1\) and \(t_2\) in parallel.
then(skip, \( t \)) = \text{then}(t, \text{skip}) = t

\text{orelse}(\text{fail}, t) = \text{orelse}(t, \text{fail}) = t
repeat : tactic → tactic
  Keep applying the given tactic until no subgoal is modified by it.
repeatupto : tactic × nat → tactic
  Keep applying the given tactic until no subgoal is modified by it, or the maximum number of iterations is reached.
tryfor : tactic × seconds → tactic
  tryfor(t, k) returns the value computed by tactic t applied to the given goal if this value is computed within k seconds, otherwise it fails.
Features/Measures

Probing structural features of formulas
Features/Measures: Yices 1.0 strategy

- diff logic?
  - no: simplex
  - yes: atom/dim < k
    - no: simplex
    - yes: floyd warshall
Features/Measures: Yices 1.0 strategy

\[ \text{orelse(then(failif(diff } \text{ and } \frac{\text{atom}}{\text{dim}} > k), \text{ simplex}), \text{ floydwarsall})} \]

Fail if condition is not satisfied. Otherwise, do nothing.
**Features/Measures**

bw: Sum total bit-width of all rational coefficients of polynomials in case.
diff: True if the formula is in the difference logic fragment.
linear: True if all polynomials are linear.
dim: Number of arithmetic constants.
atoms: Number of atoms.
degree: Maximal total multivariate degree of polynomials.
size: Total formula size.
Tacticals: syntax sugar

\[
\text{if}(c, \ t_1, \ t_2) = \text{orelse}(\text{then}(\text{failif}(\neg c), \ t_1), \ t_2) \\
\text{when}(c, \ t) = \text{if}(c, \ t, \ \text{skip})
\]
Abstraction/Refinement

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \]

\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]
Abstraction/Refinement

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \]

\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]

SAT Solver
Abstraction/Refinement

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka "naming" atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \]
\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]

Assignment

\[ p_1, \ p_2, \ \neg p_3, \ p_4 \]
Abstraction/Refinement

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]

SAT Solver

Assignment

\[ p_1, \ p_2, \neg p_3, \ p_4 \]

\[ x \geq 0, \ y = x + 1, \]
\[ \neg (y > 2), \ y < 1 \]
Abstraction/Refinement

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \quad p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]

SAT Solver

Assignment

\[ p_1, \ p_2, \ \neg p_3, \ p_4 \]

\[ x \geq 0, \ y = x + 1, \neg(y > 2), \ y < 1 \]

Unsatisfiable

\[ x \geq 0, \ y = x + 1, \ y < 1 \]

Theory Solver
Abstraction/Refinement

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \]

\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]

Assignment

\[ p_1, \ p_2, \ \neg p_3, \ p_4 \]

\[ x \geq 0, \ y = x + 1, \ \neg (y > 2), \ y < 1 \]

New Lemma

\[ \neg p_1 \lor \neg p_2 \lor \neg p_4 \]

Unsatisfiable

\[ x \geq 0, \ y = x + 1, \ y < 1 \]

Theory Solver
Design engines as tacticals

then(preprocess, smt(finalcheck))

Apply “cheap” propagation/pruning steps; and then apply complete “expensive” procedure
Conclusions

• Flexible solver/prover architectures
• “Good encodings” are solver/prover dependent
• Transparency (open source) is essential
• Separation of concerns: problem encoders x solvers
• “Orchestrating smaller/simpler procedures”