CS257: Introduction to Automated Reasoning

DPLL(T): Combining T-Solvers with SAT
Theory of Uninterpreted Functions: $\mathcal{T}_{\text{EUF}}$

Given a signature $\Sigma$ with equalities, the most unrestricted theory would include the class of all $\Sigma$-models.

This family of theories parameterized by the signature, is known as the theory of **Equality with Uninterpreted Functions (EUF)** or the **empty theory**, since it imposes no restrictions on its models.

$\text{QF}_\text{UF}$ (conjunctions of $\mathcal{T}_{\text{EUF}}$-literals) can be decided with **congruence closure** procedure.

**Example:** $(f(a) = a) \land (g(a) \neq f(a))$

**Note:** For simplicity, assume we only consider equality over 1 sort.
Congruence Closure: Definitions

Consider a set $S$ and a binary relation $R$.

$R$ is an **equivalence relation** iff it is reflexive, symmetric, and transitive.

An equivalence relation $R$ is a **congruence relation** iff for every $n$-ary function $f$,

\[ \forall x_1, \ldots, x_n. \forall y_1, \ldots, y_n. (\bigwedge_{i=1}^{n} R(x_i, y_i)) \rightarrow R(f(x_1, \ldots, x_n), f(y_1, \ldots, y_n)). \]

Is $=$ an congruence relation?
Congruence Closure: Definitions

Given a relation $R$, its **equivalence closure** $R^E$ is the **smallest** relation that
- contains $R$;
- is a equivalent relation.

Given a relation $R$, its **congruence closure** $R^C$ is the **smallest** relation that
- contains $R$;
- is a congruence relation.
Congruence Closure: Algorithm

Given a $\Sigma$-formula $\alpha$, define its **subterm set** $S_\alpha$ as the set that contains precisely the subterms of $\alpha$ that do not contain equality symbols.

Example: $\alpha := f(f(a)) = a \land f(f(f(a))) = a \land g(a) \neq g(f(a))$

$$S_\alpha := \{ a, f(a), f(f(a)), f(f(f(a))), g(a), g(f(a)) \}$$

High-level idea:

1. Partition the literals into a set of equalities $E$ and a set of inequalities $D$
2. Construct the congruence closure of $E$ over $S_\alpha$
3. Unsatisfiable iff there exists $t_1 \neq t_2 \in D$ and $(t_1, t_2) \in E^C$
Congruence Closure: Algorithm

\[ \alpha := f(f(a)) = a \land f(f(f(a))) = a \land g(a) \neq g(f(a)) \]

\[ S_\alpha := \{a, f(a), f(f(a)), f(f(f(a))), g(a), g(f(a))\} \]

**Step 1:** place each subterm of \( \alpha \) into its own congruence class:

\[ \{a\}, \{f(a)\}, \{f(f(a))\}, \{f(f(f(a)))\}, \{g(a)\}, \{g(f(a))\} \]
Congruence Closure: Algorithm

Step 2: For each positive literal $t_1 = t_2$ in $\alpha$
  - **merge** the congruence classes for $t_1$ and $t_2$
  - **propagate** the resulting congruences

\[
\alpha := f(f(a)) = a \land f(f(f(a))) = a \land g(a) \neq g(f(a))
\]

\[
\{a, f(a), f(f(a)), f(f(f(a)))\}, \{g(a), g(f(a))\}
\]
Congruence Closure: Algorithm

\[ \alpha := f(f(a)) = a \land f(f(f(a))) = a \land g(a) \neq g(f(a)) \]
\[ \{ a, f(a), f(f(a)), f(f(f(a))) \}, \{ g(a), g(f(a)) \} \]

Step 3: \( \alpha \) is \( T_{EUF} \)-unsatisfiable, iff \( \alpha \) has a negative literal \( t_1 \neq t_2 \), where \( t_1 \) and \( t_2 \) are in the same congruence class.

Note: This Algorithm can be implemented efficiently with a union-find data structure (CC. Chap. 9.1-9.3).
Congruence Closure: still an active research problem

What if we have disjunctions?

The congruence closure checks the satisfiability of conjunctions of $T_{EUF}$-literals.

What about

$$g(a) = c \land (f(g(a)) \neq f(c) \lor g(a) = d) \land c \neq d$$

**Theorem**: The $T$-satisfiability of quantifier-free formulas is decidable iff the $T$-satisfiability of conjunctions/sets of literals is decidable.

Convert the formula to DNF and check if any of its disjuncts is $T$-satisfiable. Very inefficient!

A better solution: exploit propositional satisfiability technology
Lifting SAT Technology to SMT

Two main approaches:

1. “Eager”
   - translate into an equisatisfiable propositional formula
   - feed it to any SAT solver

2. “Lazy”
   - abstract the input formula to a propositional one
   - feed it to a (CDCL-based) SAT solver
   - use a theory decision procedure to refine the formula and guide the SAT solver
   - Notable systems: Bitwuzla, cvc5, MathSAT, Yices, Z3
Lazy Approach for SMT

Given a quantifier-free \( \Sigma \) -formula \( \phi \), for each atomic formula \( \alpha \) in \( \phi \), we associate a unique propositional variable \( e(\alpha) \).

The **Boolean skeleton** of a formula \( \phi \) is a propositional logic formula, where each atomic formula \( \alpha \) in \( \phi \) is replaced with \( e(\alpha) \).

Example:

\[
\phi := (x < 0 \lor (x + y < 1 \land \neg(x < 0))) \rightarrow y < 0
\]

Let \( e(x < 0) = p_1 \), \( e(x + y < 1) = p_2 \), \( e(y < 0) = p_3 \)

What is the **Boolean skeleton** of \( \phi \)?
(Very) Lazy Approach for SMT – Example

\[ g(a) = c \land f(g(a)) \neq f(c) \lor g(a) = d \land c \neq d \]

Simplest setting:

- Off-line SAT solver
- Non-incremental **theory solver** for conjunctions of equalities and disequalities
- Theory atoms (e.g., \( g(a) = c \)) abstracted to propositional atoms (e.g., \( 1 \))
(Very) Lazy Approach for SMT – Example

\[ g(a) = c \quad \land \quad f(g(a)) \neq f(c) \quad \lor \quad g(a) = d \quad \land \quad c \neq d \]

\[ \begin{array}{l}
1 \quad -
\hline
\end{array} \]
\[ \begin{array}{l}
-2
\hline
\end{array} \]
\[ \begin{array}{l}
3
\hline
\end{array} \]
\[ \begin{array}{l}
-4
\hline
\end{array} \]

- Send \( \{1, \neg 2 \lor 3, \neg 4\} \) to SAT solver.
- SAT solver returns model \( \{1, \neg 2, \neg 4\} \).
  Theory solver finds (concretization of) \( \{1, \neg 2, \neg 4\} \) unsat.
- Send \( \{1, \neg 2 \lor 3, \neg 4, \neg 1 \lor 2 \lor 4\} \) to SAT solver.
- SAT solver returns model \( \{1, 3, \neg 4\} \).
  Theory solver finds \( \{1, 3, \neg 4\} \) unsat.
- Send \( \{1, \neg 2 \lor 3, \neg 4, \neg 1 \lor 2, \neg 1 \lor \neg 3 \lor 4\} \) to SAT solver.
- SAT solver finds \( \{1, \neg 2 \lor 3, \neg 4, \neg 1 \lor 2 \lor 4, \neg 1 \lor \neg 3 \lor 4\} \) unsat.

Done: the original formula is unsatisfiable in \( T_{EUF} \).
Eager Approach for SMT – Example

\[ f(b) = a \lor f(a) \neq a \]

Step 1: eliminate all function applications (Ackermann’s encoding)

- introduce a constant symbol \( f_x \) to replace function application \( f(x) \);
- for each pair of introduced variables \( f_x, f_y \), add the formula \( x = y \rightarrow f_x = f_y \).

\[
\begin{align*}
 f(b) & \Rightarrow f_b \\
 f(a) & \Rightarrow f_a \\
 (f_b = a \lor f_a \neq a) & \land (a = b \rightarrow f_a = f_b)
\end{align*}
\]

Now, atomic formulas are equalities between constants/variables
Eager Approach for SMT – Example

Rename $f_b$ as $c$ and $f_a$ as $d$:

$$(f_b = a \lor f_a \neq a) \land (a = b \rightarrow f_a = f_b)$$

becomes

$$(c = a \lor d \neq a) \land (a = b \rightarrow d = c)$$

Step 2: eliminate all equalities.

- replace each pair of constants $x$, $y$ with a unique propositional variable $p_{x,y}$
- add facts about reflexivity, symmetry, transitivity

$$(p_{c,a} \lor \neg p_{d,a}) \land (p_{a,b} \rightarrow p_{d,c})$$

$$\land p_{a,a} \land p_{b,b} \land p_{c,c} \land p_{d,d}$$

$$\land (p_{a,b} \leftrightarrow p_{b,a}) \land (p_{a,c} \leftrightarrow p_{c,a}) \land (p_{a,d} \leftrightarrow p_{d,a}) \ldots$$

$$\land ((p_{a,b} \land p_{b,c}) \rightarrow p_{a,c}) \land ((p_{a,c} \land p_{c,d}) \rightarrow p_{a,d}) \ldots$$

The resulting propositional formula is equi-satisfiable with the original $T_{EUF}$-formula.

Note: not all the transitivity cases are needed.
“Eager”
- translate into an equisatisfiable propositional formula
- feed it to any SAT solver

“Lazy”
- abstract the input formula to a propositional one
- feed it to a (CDCL-based) SAT solver
- use a theory decision procedure to refine the formula and guide the SAT solver

What are the pros and cons of the eager approach and the lazy approach?
Submit your answers to

https://pollev.com/andreww095
Lazy Approach – Enhancements

Several enhancements are possible to increase efficiency:

- Check $T$-satisfiability only of full propositional model.
- Check $T$-satisfiability of partial assignment $M$ as it grows.
- If $M$ is $T$-unsatisfiable, add $\neg M$ as a clause.
- If $M$ is $T$-unsatisfiable, identify a $T$-unsatisfiable subset $M_0$ of $M$ and add $\neg M_0$ as a clause.
- If $M$ is $T$-unsatisfiable, add clause and restart.
- If $M$ is $T$-unsatisfiable, backtrack to some point where the assignment was still $T$-satisfiable.
Lazy Approach – Main Benefits

• Every tool does what it is good at:
  - SAT solver takes care of Boolean information
  - Theory solver takes care of theory information

• The theory solver works only with conjunctions of literals

• Modular approach:
  - SAT and theory solvers communicate via a simple API
  - SMT for a new theory only requires new theory solver
  - An off-the-shelf SAT solver can be embedded in a lazy SMT system with a few new lines of code
An Abstract Framework for Lazy SMT

Several variants and enhancements of lazy SMT solvers exist.

They can be modeled with an abstract calculus.
Review: CDCL

States: Fail or \( (M, \Delta, C) \)

Initial state:

- \( ((), \Delta_0, no) \), where \( \Delta_0 \) is to be checked for satisfiability

Expected final states:

- Fail if \( \Delta_0 \) is unsatisfiable
- \( (M, G, no) \) otherwise, where
  - \( G \) is equivalent to \( \Delta_0 \) and
  - \( M \) satisfies \( G \)
We are going to extend this abstract framework to lazy SMT
From SAT to SMT

Same states and transitions but

- $\Delta$ contains quantifier-free clauses in some theory $T$
- CDCL Rules operates on the Boolean skeleton of $\Delta$ (assume a mapping from theory literal to propositional literal)
- $M$ is a sequence of theory literals (i.e., atomic formulas or their negations) and decision points
- the CDCL system is augmented with rules

$T$-Conflict, $T$-Propagate
SMT-level Rules

Fix a theory $T$

At SAT level:

$$
\begin{align*}
C &= \text{no} \\
\{l_1, \ldots, l_n\} &\in \Delta \\
\neg l_1, \ldots, \neg l_n &\in M
\end{align*}
\Rightarrow
(C)$$

C := \{l_1, \ldots, l_n\}

At SMT level:

$$
\begin{align*}
C &= \text{no} \\
l_1 \land \ldots \land l_n &\models_T \bot \\
l_1, \ldots, l_n &\in M
\end{align*}
\Rightarrow
(T-Conflict)
\Rightarrow
C := \{-l_1, \ldots, -l_n\}
$$

If the conjunction of a set of literals in the trail are unsatisfiable modulo $T$, the negation of the set of literals constitutes a conflict clause.

November 6, 2023 CS257 23 / 35
SMT-level Rules

At SAT level:

\[
\{l_1, \ldots, l_n, l\} \in \Delta \quad \neg l_1, \ldots, \neg l_n \in M \quad l, \neg l \notin M \quad \frac{}{M := M \upharpoonright l} \quad \text{(Propagate)}
\]

At SMT level:

\[
l \in \text{Lits}(\Delta) \quad M \models T \quad l, \neg l \notin M \quad \frac{}{M := M \upharpoonright l} \quad \text{(T-Propagate)}
\]

If the current partial assignment logically entails some literal \( l \) in \( T \), extend the trail with \( l \).
SMT-level Rules

At SAT level:

\[ C = \{ l \} \cup D \quad \{ l_1, \ldots, l_n, \neg l \} \in \Delta \quad \neg l_1, \ldots, \neg l_n, \neg l \in M \quad \neg l_1, \ldots, \neg l_n <_M \neg l \]

\[ C := \{ l_1, \ldots, l_n \} \cup D \]

(Explain)

At SMT level:

\[ C = \{ l \} \cup D \quad \neg l_1 \land \ldots \land \neg l_n \models_T \neg l \quad \neg l_1, \ldots, \neg l_n <_M \neg l \]

\[ C := \{ l_1, \ldots, l_n \} \cup D \]

(T-Explain)

There is a literal \( l \) in the conflict clause, and \( \neg l \) is logically entailed by some literals assigned before it. We can derive a new conflict clause by performing a resolution.
**Modeling the Very Lazy Theory Approach**

_T-Conflict_ is enough to model the naive integration of SAT solvers and theory solvers seen in the earlier EUF example

\[
g(a) = c \quad \land \quad f(g(a)) \neq f(c) \quad \lor \quad g(a) = d \quad \land \quad c \neq d
\]

<table>
<thead>
<tr>
<th>M</th>
<th>Δ</th>
<th>C</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>–2 v 3, –4</td>
<td>no</td>
<td>by <strong>Propagate</strong>⁺</td>
</tr>
<tr>
<td>1</td>
<td>–2 v 3, –4</td>
<td>no</td>
<td>by <strong>Decide</strong></td>
</tr>
<tr>
<td>1</td>
<td>–2 v 3, –4</td>
<td>no</td>
<td>by <strong>T-Conflict</strong></td>
</tr>
<tr>
<td>1</td>
<td>–2 v 3, –4, –1 v 2 v 4</td>
<td>–1 v 2 v 4</td>
<td>by <strong>Learn</strong></td>
</tr>
<tr>
<td>1</td>
<td>–2 v 3, –4, –1 v 2 v 4</td>
<td>no</td>
<td>by <strong>Restart</strong></td>
</tr>
<tr>
<td>1</td>
<td>–2 v 3, –4, –1 v 2 v 4</td>
<td>no</td>
<td>by <strong>Propagate</strong>⁺</td>
</tr>
<tr>
<td>1</td>
<td>–2 v 3, –4, –1 v 2 v 4, –1 v –3 v 4</td>
<td>–1 v –3 v 4</td>
<td>by <strong>T-Conflict, Learn</strong></td>
</tr>
<tr>
<td>Fail</td>
<td>–2 v 3, –4, –1 v 2 v 4, –1 v –3 v 4</td>
<td>–1 v –3 v 4</td>
<td>by <strong>Fail</strong></td>
</tr>
</tbody>
</table>
A Better Lazy Approach

The very lazy approach can be improved considerably with

- An **on-line** SAT engine,
  which can accept new input clauses on the fly

- an **incremental and explicating** $T$-solver,
  which can:
  1. check the $T$-satisfiability of $M$ as it is extended and
  2. identify a small $T$-unsatisfiable subset of $M$ once $M$ becomes $T$-unsatisfiable
A Better Lazy Approach

\[ g(a) = c \land f(g(a)) \neq f(c) \lor g(a) = d \land c \neq d \]

<table>
<thead>
<tr>
<th></th>
<th>( M )</th>
<th>( \Delta )</th>
<th>( C )</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( 1 )</td>
<td>( \neg 2 \lor 3, \neg 4 )</td>
<td>no</td>
<td>by Propagate\+</td>
</tr>
<tr>
<td>1</td>
<td>( 1 )</td>
<td>( \neg 2 \lor 3, \neg 4 )</td>
<td>no</td>
<td>by Decide</td>
</tr>
<tr>
<td>1</td>
<td>( 1 )</td>
<td>( \neg 2 \lor 3, \neg 4 )</td>
<td>( \neg 1 \lor 2 )</td>
<td>by T-Conflict</td>
</tr>
<tr>
<td>1</td>
<td>( 1 )</td>
<td>( \neg 2 \lor 3, \neg 4 )</td>
<td>no</td>
<td>by Backjump</td>
</tr>
<tr>
<td>1</td>
<td>( 1 )</td>
<td>( \neg 2 \lor 3, \neg 4 )</td>
<td>no</td>
<td>by Propagate</td>
</tr>
<tr>
<td>1</td>
<td>( 1 )</td>
<td>( \neg 2 \lor 3, \neg 4 )</td>
<td>( \neg 1 \lor \neg 3 \lor 4 )</td>
<td>by T-Conflict</td>
</tr>
<tr>
<td>Fail</td>
<td>( 1 )</td>
<td>( \neg 2 \lor 3, \neg 4 )</td>
<td>( \neg 1 \lor \neg 3 \lor 4 )</td>
<td>by Fail</td>
</tr>
</tbody>
</table>
Lazy Approach – Strategies

Ignoring **Restart** (for simplicity), a **common strategy** is to apply the rules using the following priorities:

1. If a clause is falsified by the current assignment \( M \), apply **Conflict**
2. If \( M \) is \( T \)-unsatisfiable, apply **\( T \)-Conflict**
3. Apply **Fail** or **Explain** + **Learn** + **Backjump** as appropriate
4. Apply **Propagate**
5. Apply **Decide**

**Note:** Depending on the cost of checking the \( T \)-satisfiability of \( M \), Step (2) can be applied with lower frequency or priority
With \textbf{T-Conflict} as the only theory rule, the theory solver is used just to validate the choices of the SAT engine.

With \textbf{T-Propagate} and \textbf{T-Explain}, it can also be used to guide the engine’s search.

\[
\frac{l \in \text{Lits}(\Delta) \quad M \models_T l \quad l, \lnot l \notin M}{M := M / l} \quad (T\text{-Propagate})
\]

\[
C = \{l\} \cup D \quad \lnot l_1 \land \ldots \land \lnot l_n \models_T \lnot l \quad \lnot l_1, \ldots, \lnot l_n \prec_M \lnot l \quad \frac{C := \{l_1, \ldots, l_n\} \cup D}{C} \quad (T\text{-Explain})
\]
Theory Propagation Example

\[ g(a) = c \quad \land \quad f(g(a)) \not\equiv f(c) \quad \lor \quad g(a) = d \quad \land \quad c \not\equiv d \]

1

<table>
<thead>
<tr>
<th>M</th>
<th>Δ</th>
<th>C</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \neg 2 \lor 3, \neg 4 )</td>
<td>no</td>
<td>(1) ( \not\models_T 2 ) (by Propagate+)</td>
</tr>
<tr>
<td>1 ( \neg 4 ) 2</td>
<td>( \neg 2 \lor 3, \neg 4 )</td>
<td>no</td>
<td>(1, ( \neg 4 \not\models_T \neg 3 )) (by T-Propagate)</td>
</tr>
<tr>
<td>1 ( \neg 4 ) 2 ( \neg 3 )</td>
<td>( \neg 2 \lor 3, \neg 4 )</td>
<td>no</td>
<td>(1, ( \neg 2 \not\models_T \neg 3 )) (by T-Propagate)</td>
</tr>
<tr>
<td>1 ( \neg 4 ) 2 ( \neg 3 )</td>
<td>( \neg 2 \lor 3, \neg 4 )</td>
<td>( \neg 2 \lor 3 )</td>
<td>(by Conflict)</td>
</tr>
<tr>
<td>Fail</td>
<td>( \neg 2 \lor 3, \neg 4 )</td>
<td>by Fail</td>
<td></td>
</tr>
</tbody>
</table>

Note: \( T \)-propagation eliminates search altogether in this case
no applications of Decide are needed
Modeling Modern Lazy SMT Solvers

At the core, current lazy SMT solvers are implementations of the transition system with rules:

1. **Propagate, Decide, Conflict, Explain, Backjump, Fail**

2. **T-Conflict, T-Propagate, T-Explain**

3. **Learn, Forget, Restart**

Basic DPLL Modulo Theories $\overset{\text{def}}{=} (1) + (2)$

DPLL Modulo Theories $\overset{\text{def}}{=} (1) + (2) + (3)$
Correctness

Updated terminology:

**Irreducible state**: state to which no *Basic DPLL modulo Theories* rules apply

**Execution**: sequence of transitions allowed by the rules and starting with $M = \emptyset$ and $C = \text{no}$

**Exhausted execution**: execution ending in an irreducible state

**Proposition**: (Termination) Every execution in which
(a) *Learn/Forget* are applied only finitely many times and
(b) *Restart* is applied with increased periodicity
is finite.

**Lemma**: Every exhausted execution ends with either $C = \text{no}$ or *Fail*.

**Proposition** (Soundness) For every exhausted execution starting with $\Delta = \Delta_0$ and ending with *Fail*, the clause set $\Delta_0$ is *$T$-unsatisfiable*. 
DPLL($T$) Architecture

The approach formalized so far can be implemented with a simple architecture named DPLL($T$)

$$\text{DPLL} (T) = \text{DPLL} (X) \text{ engine } + \ T\text{-solver}$$

\textbf{DPLL}(X):
- Very similar to a SAT solver, enumerates Boolean models
- Not allowed: pure literal
- Required: incremental addition of clauses
- Desirable: partial model detection

\textbf{$T$}-solver:
- Checks the $T$-satisfiability of conjunctions of literals
- Computes theory propagations
- Produces explanations of $T$-unsatisfiability/propagation
- Must be incremental and backtrackable
SAT Solver
- Only sees Boolean skeleton of problem
- Builds partial model by assigning truth values to literals
- Sends these literals to the core as assertions

Core
- Sends each assertion to the appropriate theory
- Sends deduced literals to other theories/SAT solver
- Handles theory combination

Theory Solvers
- Decide $T$-satisfiability of a conjunction of theory literals
- Incremental
- Backtrackable
- Conflict Generation
- Theory Propagation

Arithmetic
Arrays
UF
Bit-Vectors

SAT Solver
DPLL