SATISFIABILITY MODULO THEORIES

Presenters:

Aaron Gorenstein

Erin Brady

Why are we here?

- FOL is pretty expressive, many utilities
- Determining if a FOL example is SAT is hard
- Propositional SAT is (comparatively) easy
- Perhaps we can meet SAT halfway
- Limit ourselves to “theories”

Two Directions

- Eager: Aaron
  - Translate necessary theories into SAT, and solve
- Lazy: Erin (sorry)
  - Solve SAT, and then see if it also works with theory

Theories?

- Constraints over FOL
- Example:

$$X < Y \wedge \sim(X < Y + 0)$$

- Σ (signature)

Theories? Cont'd

- **Equality** (Needs no theories!)
- **Integer and Real Arithmetic** (no multiplication – why?)
- **Arrays**
- Fixed-width bit vectors
- Inductive data types

The Importance of Being Eager

- Idea: given statement, “translate” sufficient facts from theory to derive “equisatisfiable” SAT clause
- Tricky part: translation!
  - Correctness
  - Speed
  - Size

Beginning Translation

- Where do we start?
  - Integer arithmetic, equality, and enhance with “limited lambdas.”
- Path to SAT
  - Eliminate (expand) lambdas, then functions and predicates, and then integer to boolean form.

  - Wait, lambdas?

String Replacement for Translation

- Eliminate Lambdas
  - Straightforward
- Function/Predicate Elimination
  - Naïve: replace every f(a, b, … z) with atom xF
  - Issue: if f(a, b) appears twice?

From Arithmetic to Boolean

- Where are we?
- $\sum_{j=1}^{n} a_{i,j}\, x_j \geq\; b_j$
- Integer linear programming!
- Simpler: direct encoding
  - Replace each unique constraint in the linear arithmetic formula with a fresh Boolean variable (creates F_{bvar})
  - Generate a Boolean formula F_{cons} that encodes constraints to maintain validity of formula
  - AND and SAT!
- So wait, how do we encode constraints?
  - Equality, difference, and arbitrary!

Translation 4

- Small-domain encodings
  - A satisfying assignment is bounded by m, n, length of a, length of b
  - General solver would deal with solution size
    - O(log m + log b + m (log m + log a))
    - Problem: that m log m term – may have thousands of constraints!
  - Equality
  - Difference
  - Sum constraints of form $(x_i + x_j)$ R b_t
  - Mostly-difference constraints with sparse (few vars per) constraint

Lazy Approach

- Lazy SMT T-Solvers are the alternative to the eager approach
- Start with an efficient SAT solver
- Integrate with decision procedures for first-order theories (Theory, or T-Solvers)

Integrating SAT Solver and T-Solver

- Offline schema
  - Uses DPLL and T-Solver as two separate parts
  - Give boolean version (φ^P) of input formula (φ)
  - Input to DPLL
    - φ^P unsatisfiable? Then φ is T-unsatisfiable
    - φ^P satisfied by μ^P? Input μ into T-solver
      - If μ is T-consistent, then φ is T-consistent
      - If μ is T-inconsistent, add $\sim\mu^P$ to φ^P and restart DPLL

Integrating SAT Solver and T-Solver

- Online schema:
  - Modifies DPLL to enumerate truth assignments that are checked by a T-Solver

```
SatValue T-DPLL (T-formula φ, T-assignment & μ) {
   if (T-preprocess(φ, μ) == Conflict);
      return Unsat;
   φ^p = T2B(φ); μ^p = T2B(μ);
   while (1) {
      T-decide_next_branch(φ^p, μ^p);
      while (1) {
         status = T-deduce(φ^p, μ^p);
         if (status == Sat) {
            μ = B2T(μ^p);
            return Sat; }
         else if (status == Conflict) {
            blevel = T-analyze_conflict(φ^p, μ^p);
            if (blevel == 0)
               return Unsat;
            else T-backtrack(blevel, φ^p, μ^p);
         }
         else break;
} } }
```

How DPLL and T-DPLL Differ

- T-DPLL extends DPLL concepts of:
  - **Literal deduction**: check for new literal assignments by using the boolean formulas, but also by using the theory
  - **Conflict deduction**: check for boolean conflicts or theory conflicts that entail {[]}

Enhancements to T-DPLL

- Normalize T-atoms
- Static learning
- Early pruning
- T-propagation
- T-backjumping/T-learning
- Generating partial assignments
- Pure-literal filtering

Abstract Theory

Propagate : $\mu \parallel \varphi, c \vee l \implies \mu\, l \parallel \varphi, c \vee l$ **if** $\begin{cases} \mu \models_p \neg c \\ l \text{ is undefined in } \mu \end{cases}$

Decide : $\mu \parallel \varphi \implies \mu\, l^{\bullet} \parallel \varphi$ **if** $\begin{cases} l \text{ or } \neg l \text{ occurs in } \varphi \\ l \text{ is undefined in } \mu \end{cases}$

Fail : $\mu \parallel \varphi, c \implies \mathit{Fail}$ **if** $\begin{cases} \mu \models_p \neg c \\ \mu \text{ contains no decision literals} \end{cases}$

Restart : $\mu \parallel \varphi \implies \emptyset \parallel \varphi$

$\mathcal{T}$-Propagate : $\mu \parallel \varphi \implies \mu\, l \parallel \varphi$ **if** $\begin{cases} \mu \models_{\mathcal{T}} l \\ l \text{ or } \neg l \text{ occurs in } \varphi \\ l \text{ is undefined in } \mu \end{cases}$

$\mathcal{T}$-Learn : $\mu \parallel \varphi \implies \mu \parallel \varphi, c$ **if** $\begin{cases} \text{each atom of } c \text{ occurs in } \mu \parallel \varphi \\ \varphi \models_{\mathcal{T}} c \end{cases}$

$\mathcal{T}$-Forget : $\mu \parallel \varphi, c \implies \mu \parallel \varphi$ **if** $\{ \varphi \models_{\mathcal{T}} c$

$\mathcal{T}$-Backjump :

$$\mu\, l^{\bullet}\, \mu' \parallel \varphi, c \implies \mu\, k \parallel \varphi, c \text{ if } \begin{cases} \mu\, l^{\bullet}\, \mu' \models_p \neg c, \text{ and there is} \\ \text{some clause } c' \vee l' \text{ such that:} \\ \quad \varphi, c \models_{\mathcal{T}} c' \vee l' \text{ and } \mu \models_p \neg c', \\ \quad l' \text{ is undefined in } \mu, \text{ and} \\ \quad l \text{ or } \neg l \text{ occurs in } \mu\, l^{\bullet}\, \mu' \parallel \varphi \end{cases}$$

Fair Rule Application Strategy

Termination: Starting from a state $\emptyset \parallel \varphi_0$, the strategy generates only finite derivations.

Soundness: If φ_0 is $\mathcal{T}$-satisfiable, every exhausted derivation of $\emptyset \parallel \varphi_0$ generated by the strategy ends with a state of the form $\mu \parallel \varphi$ where μ is a ($\mathcal{T}$-consistent) total, satisfying assignment for φ.

Completeness: If φ_0 is not $\mathcal{T}$-satisfiable, every exhausted derivation of $\emptyset \parallel \varphi_0$ generated by the strategy ends with *Fail*.

Properties of T-Solvers

- **Input**: Collection of T-literals μ
- **Output**: T-SAT or T-UNSAT for μ
- Typically involve a specific design procedure that was developed with the background theory in mind

Features of T-Solvers

- **Model generation**: for a T-consistent set μ, the T-solver can generate a T-model ∫ such that ∫ $|=_T$ μ
- **Conflict set generation**: for a T-inconsistent set μ, the T-solver can find a subset n – the <u>theory conflict set</u> - which has caused the inconsistency

Features of T-Solvers

- **Incrementality**: the T-solver can remember previous calls – so, if μ_1 is T-satisfiable and the T-Solver is called for $\mu_1 \cup \mu_2$, it does not restart the computation from scratch
- **Backtrackability**: the T-solver can undo steps to return to a previous state efficiently

Features of T-Solvers

- **Deduction of unassigned literals**: if the T-solver is given a T-consistent set, it can also find and decide literals from unassigned atoms in the original formula
- **Deduction of interface equalities**: when returning SAT, the T-solver can deduce equalities between the variables/terms in μ

Theory of Equality

- No restrictions on interpretation of function/predicate symbols
- Given a signature Σ, the theory that includes all possible models is T_{ε}
- Also known as the “empty theory” or the “theory of equality with uninterpreted functions”

Shostak's Method

- General method to combine theory of equality with other appropriate theories
- Important definitions:
  - solved form S: Each lefthand side appears only once
  - $y_F(G)$ means that there will be no conflicts that occur from replacing variables

Shostak Theory

- A consistent theory T with signature ∑ is a Shostak theory if:
  - ∑ has no predicate symbols
  - There is a canonizer function (∑-terms -> ∑-terms) such that $|=_T$ s=t iff canon(s) == canon(t)
  - There is a solver function (∑-eqs -> formula sets):
    - If $|=_T$ s≠t , then solve(s=t) == { [] }
    - Else, solve(s = t) returns a set S of equations in solved form such that $|=_T$ s=t <-> $y_{s=t}$(S).

Splitting on Demand

- T-solvers can demand that the DPLL continue to split before passing anything to the T-solver
- Literals could be unknown to DPLL, or contain fresh constant symbols
- Must allow new symbols to be added to the list of clauses
- Allows the T-solver, with it’s knowledge of the background theory, to dictate in which direction DPLL should go

Layered Theory Solvers

- T-solvers are “layered” by their complexity levels
- If a solution is not found by a simple T-solver, move on

Citations

- Topic, materials, and figures from
C. Barrett, R. Sebastiani, S. A. Seshia, & C. Tinelli, Satisfiability Modulo Theories, in A. Biere, H. van Maaren, M. Heule and Toby Walsh, Eds.,*Handbook of Satisfiability*, IOS Press, 2009

- Thanks to Professor Kautz for discussion clarifying concepts!