Mathematical Logic 2016

Lecture 20: Satisfiability modulo theory (SMT) solving

Instructor: Ashutosh Gupta

TIFR, India

Compile date: 2016-10-20

Where are we and where are we going?

We have seen

- definition of theories in FOL

We will see the design of solvers that solve quantifier free formulas of decidable theories

- DPLL($\mathcal{T}$)
- Theory propagation implementation
- Example theory propagation implementation

DPLL($\mathcal{T}$)

DPLL solves(i.e. checks satisfiability) QF propositional formulas

DPLL($\mathcal{T}$) solves QF formulas in theory $\mathcal{T}$,

- separates the boolean and theory reasoning,
- proceeds like DPLL, and
- needs support of a $\mathcal{T}$-solver $DP_{\mathcal{T}}$, i.e., a decision procedure for conjunction of literals of $\mathcal{T}$

The tools that are build using DPLL($\mathcal{T}$) are called satisfiablity modulo theory solvers (SMT solvers)

DPLL($\mathcal{T}$) - some notation

Let $\mathcal{T}$ be a FO-theory with signature $\mathbf{S}$.

We assume input formulas are from $\mathcal{T}$, QF, and in CNF.

Definition 20.1

For a QF $\mathcal{T}$ formula F, let atoms(F) denote the set of atoms appearing in F.

Example 20.1

- $f(x) \approx g(h(x, y))$ *is a formula in QF_EUF.*
- $x > 0 \vee y + x \approx 3.5z$ *is a formula in QF_LRA.*

Free variables vs. constants

If we have a QF formula and we are interested in satisfiability. Then, we may assume that all variables in the formula are existentially quantified.

If we apply skolemization on such a formula then we obtain a formula with fresh constants and no variables.

In QF satisfiability checking, variables and constants play the same role.

To have consistent notation with FOL clauses, we may assume that in the current case there are fresh constants from **par** (recall completeness of FOL) instead of variables.

Boolean encoder

For a formula F, let boolean encoder e be a partial map from $atoms(F)$ to fresh boolean variables.

For a term t, let $e(t)$ denote the term obtained by replacing each atom a by $e(a)$ if $e(a)$ is defined.

Example 20.2

Let $F = x < 2 \vee (y > 0 \vee x \geq 2)$
and $e = \{x < 2 \mapsto x_1, y > 0 \mapsto x_2\}$
$e(F) = x_1 \vee (x_2 \vee \neg x_1)$

Partial model

Definition 20.2

For a boolean encoder e, a partial model m is an ordered partial map from $range(e)$ to $\mathcal{B}$.

Example 20.3

partial models $m_1 = (x \mapsto 0, y \mapsto 1)$ and $m_1 = (y \mapsto 1, x \mapsto 0)$ are not same.

Definition 20.3

For a partial model m of e, let
$e^{-1}(m) \triangleq \{e^{-1}(x) | x \mapsto 1 \in m\} \cup \{\neg e^{-1}(x) | x \mapsto 0 \in m\}$

DPLL($\mathcal{T}$)

Algorithm 20.1: DPLL($\mathcal{T}$)

Input: CNF F, boolean encoder e

AddClauses($e(F)$); m := UnitPropagation(); $dl := 0$; $dstack := \lambda x.0$;

do

  // backtracking

  while $\exists x\ \{x \mapsto 0, x \mapsto 1\} \subseteq m$ **do**

    if $dl = 0$ **then return** *unsat*;

    (C, dl) := AnalyzeConflict(m) ; // clause learning

    $m.resize(dstack(dl))$; AddClauses($\{C\}$); m := UnitPropagation();

  // Boolean decision

  if *m is partial* **then**

    $dstack(dl) := m.size()$;

    $dl := dl + 1$; m := Decide(); m := UnitPropagation() ;

  // Theory propagation

  if $\forall x\ \{x \mapsto 0, x \mapsto 1\} \not\subseteq m$ **then**

    (Cs, dl') := TheoryDeduction($\bigwedge e^{-1}(m)$);

    if $dl' < dl$ **then** $\{dl = dl'$; $m.resize(dstack(dl))$; $\}$;

    AddClauses($e(Cs)$); m := UnitPropagation();

while *m is partial or* $\exists x\ \{x \mapsto 0, x \mapsto 1\} \subseteq m$;

return *sat*

stands for decision level

dstack records history for backtracking

returns a clause set and a decision level

Theory propagation

TheoryDeduction looks at the atoms assigned so far and checks

- if they are mutually unsatisfiable
- if not, are there other literals from F that are implied by the current assignment

Any implementation must comply with the following goals

- Correctness: boolean model is consistent with $\mathcal{T}$
- Termination: unsat partial models are never repeated

TheoryDeduction

TheoryDeduction solves conjunction of literals and returns a set of clauses and a decision level.

$$(Cs, dl') := \text{TheoryDeduction}(\bigwedge e^{-1}(m))$$

Cs may contain the clauses of the form

$$(\bigwedge L) \Rightarrow \ell$$

where $\ell \in lits(F) \cup \{\bot\}$ and $L \subseteq e^{-1}(m)$.

Note: The RHS need not be a single literal

Requirement form TheoryDeduction

The output of TheoryDeduction must satisfy the following conditions

- If $\bigwedge e^{-1}(m)$ is unsat in $\mathcal{T}$ then Cs must contain a clause with $\ell = \bot$.
- if $\bigwedge e^{-1}(m)$ is sat then $dl' = dl$.
  Otherwise, dl' is the decision level immediately after which the unsatisfiablity occurred (clearly stated shortly).

Example : DPLL($\mathcal{T}$)

Consider $F = (x = y \vee y = z) \wedge (y \neq z \vee z = u) \wedge (z = x)$
$e(F) = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge x_4$

After $\textsc{AddClauses}(e(F)); m := \textsc{UnitPropagation}()$
$m = \{x_4 \mapsto 1\}$

After $m := \textsc{Decide}();$
$m = \{x_4 \mapsto 1, x_2 \mapsto 0\}$

After $m := \textsc{UnitPropagation}()$
$m = \{x_4 \mapsto 1, x_2 \mapsto 0, x_1 \mapsto 1\}$

After $(Cs, dl') := \textsc{TheoryDeduction}(x = y \wedge y \neq z \wedge z = x)$
$Cs = \{x \neq y \vee y = z \vee z \neq x\}, dl' = 0, e(Cs) = \{\neg x_1 \vee x_2 \vee \neg x_4\}$

After $\textsc{AddClauses}(e(Cs)); m := \textsc{UnitPropagation}()$
$m = \{x_4 \mapsto 1, x_2 \mapsto 0, x_1 \mapsto 1, x_1 \mapsto 0\} \leftarrow$ conflict

Topic 20.1

Theory propagation implementation

Theory propagation implementation - Incremental theory solver

Typically, theory propagation is implemented using incremental/online solvers.

Incremental/online solver $DP_{\mathcal{T}}$

- takes input constraints as a sequence of literals,
- maintains a data structure that defines the solver state and satisfiability of constraints seen so far.
- provides a stack like interface
  - push(ℓ) - adds literal ℓ in "constraint store"
  - pop() - removes last pushed literal from the store
  - checkSat() - checks satisfiability of current store
  - unsatCore() - returns the set of literals that caused unsatisfiablity

Note: We assume that push and pop call checkSat() at the end of their execution. Therefore, explicit calls to checkSat() are not necessary. However, practical tools allow users to choose the policy of calling checkSat() - lazy vs. eager

Theory propagation implementation

Algorithm 20.2: TheoryDeduction

Input: Set of literals Ls

Read only input: m partial model, $dstack$ decision depths, dl current decision level

foreach $\ell \in Ls$ **do**
  $DP_{\mathcal{T}}.push(\ell)$

if $DP_{\mathcal{T}}.checkSat() == unsat$ **then**
  $Ls' := DP_{\mathcal{T}}.unsatCore()$; // minimize clause
  $dl' := \max\{dl'' | \exists \ell \in Ls', i.\ dstack(dl'') < i \wedge m[i] = e(\ell) \mapsto \_\}$;
  return $(\{\neg \bigwedge Ls'\}, dl')$
else
  //implied clauses
  $Cs := \emptyset$;
  foreach $\ell \in Lits(F)$ **do**
    $DP_{\mathcal{T}}.push(\neg\ell)$;
    if $DP_{\mathcal{T}}.checkSat() == unsat$ **then**
      $Ls' := DP_{\mathcal{T}}.unsatCore()$; // ℓ is called implied model and $\neg\ell \in Ls'$
      $Cs := Cs \cup \{\neg \bigwedge Ls'\}$;
    $DP_{\mathcal{T}}.pop()$;
  return *(Cs,dl)*

Topic 20.2

Example theory propagation implementation

Theory of Equality and function symbols (EUF)

EUF syntax: quantifier-free first order formulas with signature $\mathbf{S} = (\mathbf{F}, \emptyset)$, i.e., countably many function symbols and no predicates.

The theory axioms include

1. $\forall x.\ x \approx x$
2. $\forall x, y.\ x \approx y \Rightarrow y \approx x$
3. $\forall x, y, z.\ x \approx y \wedge y \approx z \Rightarrow x \approx z$
4. for each $f/n \in \mathbf{F}$,

$$\forall x_1, .., x_n, y_1, .., y_n.\ x_1 \approx y_1 \wedge .. \wedge x_n \approx y_n \Rightarrow f(x_1, .., x_n) \approx f(y_1, .., y_n)$$

Since the axioms are valid in FOL with equality, the theory is sometimes referred as the base theory.

Note: Predicates can be easily added if desired

DP_{EUF}

Decides conjunction of literals with interface push,pop,checkSat and unsatCore

General idea: maintain equivalence classes among terms

Algorithm 20.3: $DP_{EUF}.push(t_1 \bowtie t_2)$

globals: set of terms $Ts := \emptyset$, set of pair of classes $DisEq := \emptyset$, bool $conflictFound := 0$

$Ts := Ts \cup subTerms(t_1) \cup subTerms(t_2)$;

$C_1 := getClass(t_1)$; $C_2 := getClass(t_2)$; // if t_1 and t_2 are seen first time, create new classes

if $\bowtie$ = "$\approx$" **then**

  if $C_1 = C_2$ **then return** ;

  $C = mergeClasses(C_1, C_2)$; $parent(C) := (C_1, C_2, t_1 \approx t_2)$;

  if $(C_1, C_2) \in DisEq$ **then** { $conflictFound := 1$; **return**; } ;

  foreach $f(r_1, \ldots, r_n), f(s_1, \ldots, s_n) \in Ts \wedge \forall i \in 1..n.\ \exists C.\ r_i, s_i \in C$ **do**

    $DP_{EUF}.push(f(r_1, \ldots, r_n) \approx f(s_1, \ldots, s_n))$;

else

  // $\bowtie$ = "$\not\approx$"

  $DisEq := DisEq \cup (C_1, C_2)$;

  if $C_1 = C_2$ **then** $conflictFound := 1$; **return** ;

Exercise 20.1

a. Run $DP_{EUF}.push$ on $x \approx f(x) \wedge f(f(f(x))) \not\approx f(f(x))$.

b. Give a fix in push such that it works on the above example.

c. $f^3(a) = a \wedge f^5(a) = a \wedge f(a) \neq a$

checkSat and pop

- $DP_{EUF}.checkSat()$ { **return** *conflictFound*; }
- $DP_{EUF}.pop()$ is implemented by recording the time stamp of pushes and undoing all the mergers happened in after last push.

Unsat core

Definition 20.4

For an unsat set of formulas Σ, an unsat core of Σ is a subset (preferably minimal) of Σ that is unsat.

Algorithm 20.4: $DP_{EUF}.unsatCore()$

---

assume($conflictFound = 1$);
Let $(t_1 \not\approx t_2)$ be the dis-equality that was violated;
return $\{t_1 \not\approx t_2\} \cup getReason(t_1, t_2)$;

Algorithm 20.5: $getReason(t_1, t_2)$

Let $(t_1' \approx t_2')$ be the merge operation that placed t_1 and t_2 in same class;
if $t_1' = f(s_1, .., s_k) \approx f(u_1, ...u_k) = t_2'$ *was derived due to congruence* **then**
  $reason := \bigcup_i getReason(s_i, u_i)$
else
  $reason := \{t_1' \approx t_2'\}$
$Set = getReason(t_1, t_1') \cup reason \cup getReason(t_2', t_2)$

Completeness of $DP_{EUF}.push$

Theorem 20.1

Let $\Sigma = \{\ell_1, .., \ell_n\}$ be a set of literals in $\mathcal{T}_{EUF}$.
$DP_{EUF}.push(\ell_1); ...; DP_{EUF}.push(\ell_n)$; finds conflict iff Σ is unsat.

Proof.

Since $DP_{EUF}.push$ uses only sound proof steps of the theory, it cannot find conflict if Σ is sat.

Assume Σ is unsat and there is a proof for it.

Since $DP_{EUF}.push$ applies congruence only if the resulting terms appear in Σ, we show that there is a proof that contains only such terms.

Since Σ is unsat, there is $\Sigma' \cup \{s \not\approx t\} \subseteq \Sigma$ s.t. $\Sigma' \cup \{s \not\approx t\}$ is unsat and Σ' contains only positive literals (why?).

Exercise 20.2

Show the last claim holds.

Completeness of $DP_{EUF}.push$(contd.)

Proof(contd.)

Therefore, we must have a proof step s.t.

$$\frac{u_1 \approx u_2 \quad .. \quad u_{n-1} \approx u_n}{s \approx t},$$

where premises have proofs from Σ', $u_1 = s$, and $u_n = t$.

Now we show that we can transform the proof via induction over height of congruence proof steps.

Wlog, we assume the premises either occur in Σ' or derived from congruence.

Furthermore, if $u_i \approx u_{i+1}$ is derived from congruence then the top symbols are same in u_i and u_{i+1}.

Exercise 20.3

Justify the "wlog" claim.

Completeness of $DP_{EUF}.push$(contd.)

Proof(contd.)

claim: If s and u occurs in Σ, then any proof of $s \approx u$ can be turned into proof that contains terms from Σ

base case: If no congruence is used to derive $s \approx u$ then certainly no fresh term was invented.

induction step: We need not worry about $u_i \approx u_{i+1}$ that are coming from Σ. Only, along chain of equalities due to congruences may have new terms.

Let $f(u_{11}, .., u_{1k}) \approx f(u_{21}, .., u_{2k}) \quad .. \quad f(u_{(j-1)1}, .., u_{(j-1)k}) \approx f(u_{j1}, .., u_{jk})$ be such a sub-chain in the last proof step for $s \approx u$.

We know $f(u_{11}, .., u_{1k})$ and $f(u_{j1}, .., u_{jk})$ occur in Σ'.

Exercise 20.4

Justify the last claim.

Completeness of $DP_{EUF}.push$(contd.)

Proof(contd.)

We can rewrite the proof in the following form.

$$\frac{\dfrac{u_{11}\approx u_{21} \quad .. \quad u_{(j-1)1}\approx u_{j1}}{u_{11}\approx u_{j1}} \quad .. \quad \dfrac{u_{1k}\approx u_{2k} \quad .. \quad u_{(j-1)k}\approx u_{jk}}{u_{1k}\approx u_{jk}}}{f(u_{11},..,u_{1k}) \approx f(u_{j1},..,u_{jk})},$$

For each $i \in 1..k$, u_{1i} and u_{ji} occur in Σ.

Due to induction hypothesis, $u_{11} \approx u_{j1}$ has a proof with the desired restriction. □

Topic 20.3

Optimizations

Theory propagation strategies

- Exhaustive or Eager :
  Cs contains all possible clauses
- Minimal or Lazy :
  Cs only contains the clause that refutes current m
- Somewhat Lazy :
  Cs contains only easy to deduce clauses

Implied literals without implied clauses

Bottleneck: There may be too many implied clauses.

Observation: Very few of the implied clauses are useful, i.e., contribute in early detection of conflict.

Optimization: apply implied literals, without adding implied clauses.

Optimization overhead: If an implied model is used in conflict then recompute the implied clause for the implication graph analysis.

Relevancy

Bottleneck: All the assigned literals are sent to the theory solver.

Observation: However, *DPLL* only needs to send those literals to the solver that make unique clauses satisfiable.

Optimization:

- Each clause chooses one literal that makes it sat under current model.
- Those clause that are not sat under current model do nothing.
- If a literal is not chosen by any clause then it is not passed on to $\mathcal{T}$-solver.

Patented: US8140459 by Z3 guys(the original idea is more general than stated here)

Optimization overhead: Relevant literal management

Exercise 20.5

Suggest a scheme for relevant literal management.

Effect of optimizations

Only experiments can tell if these are good ideas!

Topic 20.4

SMT Solvers

Rise of SMT solvers

- In early 2000s, stable SMT solvers started appearing. e.g., Yiecs
- SMT competition(SMT-comp) became a driving force in their every increasing efficiency
- Formal methods community quickly realized their potential
- Z3, one of the leading SMT solver, has about 3000 citations (375 per year)(June 2016)

Input format for SMT solvers

SMT2 is a standard input format for SMT solvers.

`http://smtlib.cs.uiowa.edu/language.shtml`

- Formulas are written in infix notation

```
(>= (* 2 x) (+ y z))
```

- There is a simple type system
- Solver interacts like a stack

File format

An SMT file has four distinct parts

1. Preamble declarations
2. Type/Variable declarations
3. Asserting formulas
4. Solving commands

Preamble declaration

- Set configurations of the solvers

```
; setting Theory/Logic
(set-logic QF_UFLIA)

; enable proof generation in case of unsat formula
(set-option :produce-proofs true)
```

Sort/Variable declaration

- Declare new sorts of the variables

```
(declare-sort symbol numeral)
```

- Declare variables and uninterpreted functions that may be used in the formulas

```
(declare-fun symbol (sort*) sort)
```

Example 20.4

```
(declare-sort U 0)     ; new sort with no parameters
(declare-fun x () Int)
(declare-fun f (Int) Int)
(declare-fun g (Int Int) Int)
(declare-fun h ((Array U Int) Int) Int)
```

Asserting formulas

- Formulas are asserted in a sequence

Example 20.5

```
(assert (>= (* 2 x) (+ y z)))
(assert (< (f x) (g x x)))
(assert (> (f y) (g x x)))
```

Commands

- Commands are the actions that solver needs to do

Example 20.6

```
(check-sat) ; checks if the conjunction of asserted formula
            ; is sat
(get-model) ; returns a model if the formulas are sat
```

Stack interaction

The standard is designed to be interactive

- Asserted formulas are pushed in the stack of the solver
- `(push)` command places marker on the stack
- `(pop)` removes the formulas upto the last marker

Example 20.7

```
(push)
(assert (= x y))
(check-sat)
(pop)
```

The full example

```
(set-logic QF_UFLIA)
(set-option :produce-proofs true)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(declare-fun f (Int) Int)
(declare-fun g (Int Int) Int)
(assert (>= (* 2 x) (+ y z)))
(assert (< (f x) (g x x)))
(assert (> (f y) (g x x)))
(check-sat)
(get-model)
(push)
(assert (= x y))
(check-sat)
(pop)
(exit)
```

Demo

http://rise4fun.com/z3

API

The solvers also provide an API for using the solvers.

Leading tools

The following are some of the leading SMT solvers

- Z3
- CVC4
- MathSAT
- Boolector

Topic 20.5

Problems

Run SMT solvers

Exercise 20.6

- *Find a satisfying assignment of the following formula using SMT solver*

$$(x > 0 \vee y < 0) \wedge (x + y > 0 \vee x - y < 0)$$

Give the model generated by the SMT solver.

- *Prove the following formula is valid using SMT solver*

$$(x > y \wedge y > z) \Rightarrow x > z$$

Give the proof generated by the SMT solver.

Please do not simply submit the output and write the answer in mathematical notation.

Knapsack problem

Exercise 20.7

Write a program for solving the knapsack problem that requires filling a knapsack with stuff with maximum value. For more information look at the following.

`https://en.wikipedia.org/wiki/Knapsack_problem`

The output of the program should be the number of solutions that have value more than 95% of the best value.

Download Z3 from the following webpage:
`https://github.com/Z3Prover/z3`

We need a tool to feed random inputs to your tool. Write a tool that generates random instances, similar to what was provided last time.

Evaluate the performance on reasonably sized problems. You also need to

End of Lecture 20