CS510 Software Engineering
Satisfiability Modulo Theories (SMT)

Slides modified from those by Aarti Gupta

Textbook: The Calculus of Computation by A. Bradley and Z. Manna
Satisfiability Modulo Theory (SMT)

This lecture:

Theories, theory solvers

Next lecture: DPLL(T)
First-Order Theories

Software manipulates structures
  • Numbers, arrays, lists, bitvectors,…

Software (and hardware) verification
  • Involve reasoning about such structures

First-order theories
  • Formalize structures to enable reasoning about them
  • Validity is sometimes decidable

  • Note: Validity of FOL is undecidable
First-order theories

Recall: FOL

- Logical symbols
  - Connectives: \(\neg, \land, \lor, \Rightarrow, \Leftrightarrow\)
  - Quantifiers: \(\forall, \exists\)

- Non-logical symbols
  - Variables: \(x, y, z\)
  - N-ary functions: \(f, g\)
  - N-ary predicates: \(p, q\)
  - Constants: \(a, b, c\)

First-order theory \(T\) is defined by:

- Signature \(\Sigma_T\)
  - set of constant, function, and predicate symbols

- Set of \(T\)-Models
  - models that fix the interpretation of symbols of \(\Sigma_T\)
  - alternately, can use Axioms \(A_T\) (closed \(\Sigma_T\) formulae) to provide meaning to symbols of \(\Sigma_T\)

- Every dog has its day
- Some dogs have more days than others
- All cats have more days than dogs
- Triangle length theory

Interpretation of a FOL formula:
\[\forall x \forall y \; x > 0 \land y > 0 \Rightarrow \text{add}(x, y) > 0\]
Examples of FO theories

Equality (and uninterpreted functions)
- \( = \) stands for the usual equality
- \( f \) is not interpreted in T-model

Fixed-width bitvectors
- \( >> \) is shift operator (function)
- \( & \) is bit-wise-and operator (function)
- \( 1 \) is a constant

Linear arithmetic (over \( R \) and \( Z \))
- \( + \) is arithmetic plus (function)
- \( < \) is less-than (predicate)
- 10 and 20 are constants

Arrays
- \( a[i] \) can be viewed as \( \text{select}(a, i) \) that selects the \( i \)-th element in array \( a \)
Satisfiability Modulo Theory

First-order theory $T$ is defined by:

- **Signature $\Sigma_T$**
  - set of constant, function, and predicate symbols
- **Set of $T$-Models**
  - models that fix the interpretation of symbols of $\Sigma_T$
  - alternately, can use Axioms $A_T$ (closed $\Sigma_T$ formulae) to provide meaning to symbols of $\Sigma_T$

A formula $F$ is **T-satisfiable** *(satisfiable modulo $T$)* iff $M \vDash F$ for some T-model $M$.

A formula $F$ is **T-valid** *(valid modulo $T$)* iff $M \vDash F$ for all T-models $M$.

Theory $T$ is **decidable** if *validity modulo $T$* is decidable for every $\Sigma_T$-formula $F$.

There is an algorithm that always terminates with “yes” if $F$ is $T$-valid, and “no” if $F$ is $T$-invalid.
Fragment of a Theory

Fragment of a theory $T$

is a syntactically restricted subset of formulae of the theory

Example

- *Quantifier-free fragment* (QFF) of theory $T$ is the set of formulae without quantifiers
- Quantifier-free *conjunctive* fragment of theory $T$ is the set of formulae without quantifiers and *disjunction*

Fragments

- can be decidable, even if the full theory isn’t
- can have a decision procedure of lower complexity than for full theory
Theory of Equality $T_E$

Signature

$\Sigma_E : \{=, a, b, c, \ldots, f, g, h, \ldots, p, q, r, \ldots\}$

consists of

- a binary predicate “=“ that is interpreted using axioms
- constant, function, and predicate symbols
Axioms of $T_E$

1. $\forall x. \, x=x$  
   (reflexivity)

2. $\forall x, y. \, x=y \rightarrow y=x$  
   (symmetry)

3. $\forall x, y, z. \, x=y \land y=z \rightarrow x=z$  
   (transitivity)

4. for each n-ary function symbol $f$,
   $\forall x_1, \ldots, x_n, y_1, \ldots, y_n. \, \land_i (x_i=y_i) \rightarrow$
   $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$  
   (function congruence)

5. for each n-ary predicate symbol $p$,
   $\forall x_1, \ldots, x_n, y_1, \ldots, y_n. \, \land_i (x_i=y_i) \rightarrow$
   $(p(x_1, \ldots, x_n) \leftrightarrow p(y_1, \ldots, y_n))$  
   (predicate congruence)
Decidability of \( T_E \)

Bad news
- \( T_E \) is undecidable

Good news
- Quantifier-free fragment of \( T_E \) is decidable
- Very efficient algorithms for QFF conjunctive fragment
  - Based on congruence closure
Theory solver for $T_E$

- In 1954 Ackermann showed that the theory of equality and uninterpreted functions is decidable.
- In 1976 Nelson and Oppen implemented an $O(m^2)$ algorithm based on congruence closure computation.
- Modern implementations are based on the union-find data structure (data structures again!)
- Efficient: $O(n \log n)$
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]

Note: Quantifier-free, Conjunctive
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[ a = b, \quad b = c, \quad d = e, \quad b = s, \quad d = t, \quad a \neq e, \quad a \neq s \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]

\[ a, b, c, s \quad \text{and} \quad d, e, t \]

\[ \checkmark \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ f(a, g(d)) \neq f(b, g(e)) \]

Congruence Rule:
\[ x_1 = y_1, \ ..., \ x_n = y_n \text{ implies } f(x_1, ..., x_n) = f(y_1, ..., y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \quad b = c, \quad d = e, \quad b = s, \quad d = t, \quad f(a, g(d)) \neq f(b, g(e)) \]

Congruence Rule:
\[ x_1 = y_1, \ldots, x_n = y_n \text{ implies } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ f(a, g(d)) \neq f(b, g(e)) \]

Congruence Rule:
\[ x_1 = y_1, \ldots, x_n = y_n \text{ implies } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
Deciding Equality + (uninterpreted) Functions

\[
a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ f(a, g(d)) \neq f(b, g(e))
\]

Efficient implementation using Union-Find data structure
Example: program equivalence

```c
int fun1(int y) {
    int x, z;
    z = y;
    y = x;
    x = z;
    return x*x;
}

int fun2(int y) {
    return y*y;
}
```

The formula that is satisfiable iff programs are not equivalent:

\[
(z_1 = y_0 \land y_1 = x_0 \land x_1 = z_1 \land r_1 = x_1 \times x_1) \land \\
(r_2 = y_0 \times y_0) \land \\
\neg(r_2 = r_1)
\]

Using 32-bit integers, and interpreting * as multiplication, a SAT solver fails to return an answer in 1 minute.
Example: program equivalence

```c
int fun1(int y) {
    int x, z;
    z = y;
    y = x;
    x = z;
    return x*x;
}

int fun2(int y) {
    return y*y;
}
```

The formula that is satisfiable iff programs are not equivalent:

\[(z_1 = y_0 \land y_1 = x_0 \land x_1 = z_1 \land r_1 = \text{sq}(x_1) \land (r_2 = \text{sq}(y_0)) \land \neg(r_2 = r_1)\]

Using \(T_E\) (with uninterpreted functions), SMT solver proves unsat in a fraction of a second.
Example: program equivalence

```c
int fun1(int y) {
    int x, z;
    x = x ^ y;
    y = x ^ y;
    x = x ^ y;
    return x*x;
}

int fun2(int y) {
    return y*y;
}
```

Is the uninterpreted function abstraction going to work in this case?

No, we need the theory of fixed-width bitvectors to reason about \(^ (\text{xor})\).

Theory of fixed-width bitvectors $T_{BV}$

**Signature**
- constants
- fixed-width words (bitvectors) for modeling machine ints, longs, etc.
- arithmetic operations (+, -, *, /, etc.) (functions)
- bitwise operations (&, |, ^, etc.) (functions)
- comparison operators (<, >, etc.) (predicates)
- equality (=)

**Theory of fixed-width bitvectors is decidable**
- Bit-flattening to SAT: NP-complete complexity
Bit-Vector Logic: Syntax

\[
\begin{align*}
\text{formula} & : \text{formula} \lor \text{formula} \mid \neg \text{formula} \mid \text{atom} \\
\text{atom} & : \text{term} \ \text{rel} \ \text{term} \mid \text{Boolean-Identifier} \mid \text{term}[\text{constant}] \\
\text{rel} & : = \mid < \\
\text{term} & : \text{term} \ \text{op} \ \text{term} \mid \text{identifier} \mid \sim \text{term} \mid \text{constant} \mid \text{atom}?\text{term}\text{term} \\
\text{op} & : + \mid - \mid \cdot \mid / \mid << \mid >> \mid \& \mid \mid \mid \oplus \mid \circ
\end{align*}
\]
Bit-Vector Logic: Syntax

formula : formula ∨ formula | ¬formula | atom
atom : term rel term | Boolean-Identifier | term[ constant ]
rel : = | <
term : term op term | identifier | ∼ term | constant |
atom?term:term |
term[ constant : constant ] | ext( term )
op : + | − | · | / | ◦ | << | >> | & |  | ⊕ | ◦

∼ x: bit-wise negation of x
ext(x): sign- or zero-extension of x
x <<< d: left shift with distance d
x ◦ y: concatenation of x and y
A simple decision procedure

Transform Bit-Vector Logic to Propositional Logic
Most commonly used decision procedure
Also called 'bit-blasting'
A simple decision procedure

Transform Bit-Vector Logic to **Propositional Logic**
Most commonly used decision procedure
Also called ’*bit-blasting’*

### Bit-Vector Flattening

1. Convert propositional part as before
2. Add a *Boolean variable for each bit* of each sub-expression (term)
3. Add *constraint* for each sub-expression

We denote the new Boolean variable for *i* of term *t* by \( t_i \)
What constraints do we generate for a given term?

Example for a $| l b$:

$l - 1 \land i = 0 ($\(\mu(t)\) $i = (a_i \lor b_i)$)

(read $x = y$ over bits as $x \iff y$)

We can transform this into CNF using Tseitin's method.
What constraints do we generate for a given atom

This is easy for the bit-wise operators.

Example for \( t = a \land b \)

\[
\bigwedge_{i=0}^{l-1} t_i = (a_i \lor b_i)
\]

What about \( x = y \)
How to flatten $s = a + b$
Flattening bit-vector arithmetic

How to flatten $s = a + b$

→ we can build a circuit that adds them!

```
\begin{array}{ccc}
a & b & i \\
\hline
\text{FA} & \text{FA} & \text{FA} \\
o & s & s
\end{array}
```

Full Adder

- $s \equiv (a + b + i) \mod 2 \equiv a \oplus b \oplus i$
- $o \equiv (a + b + i) \div 2 \equiv a \cdot b + a \cdot i + b \cdot i$

The full adder in CNF:

\[
(a \lor b \lor \neg o) \land (a \lor \neg b \lor i \lor \neg o) \land (a \lor \neg b \lor \neg i \lor o) \land \\
(\neg a \lor b \lor i \lor \neg o) \land (\neg a \lor b \lor \neg i \lor o) \land (\neg a \lor \neg b \lor o)
\]
Flattening bit-vector arithmetic

Ok, this is good for one bit! How about more?
Flattening bit-vector arithmetic

Ok, this is good for one bit! How about more?

8-Bit ripple carry adder (RCA)

Also called carry chain adder

Adds $l$ variables

Add: $10^* l$ clauses 6 for $o$, 4 for $s$
Multipliers result in very hard formulas

Example:

\[ a \cdot b = c \land b \cdot a \neq c \land x < y \land x > y \]

CNF: About 11000 variables, unsolvable for current SAT solvers

Similar problems with division, modulo
Multipliers result in very hard formulas

Example:

\[ a \cdot b = c \land b \cdot a \neq c \land x < y \land x > y \]

CNF: About 11000 variables, unsolvable for current SAT solvers

Similar problems with division, modulo
Theories for Arithmetic

Natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$
Integers $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$

Three theories: (Axioms in [BM Ch. 3])

Peano arithmetic $T_{PA}$
- Natural numbers with addition (+), multiplication (*), equality (=)
- $T_{PA}$-satisfiability and $T_{PA}$-validity are undecidable

Presburger arithmetic $T_{\mathbb{N}}$
- Natural numbers with addition (+), equality (=)
- $T_{\mathbb{N}}$-satisfiability and $T_{\mathbb{N}}$-validity are decidable

Theory of integers $T_{\mathbb{Z}}$
- Integers with addition (+), subtraction (-), comparison (>), equality (=), multiplication by constants
- $T_{\mathbb{Z}}$-satisfiability and $T_{\mathbb{Z}}$-validity are decidable
Theory of Integers $T_\mathbb{Z}$

$\Sigma_\mathbb{Z} : \{\ldots, -2, -1, 0, 1, 2, \ldots, -3^*, -2^*, 2^*, 3^*, \ldots, +, -, =, >\}$

where

• $\ldots, -2, -1, 0, 1, 2, \ldots$ are constants
• $\ldots, -3^*, -2^*, 2^*, 3^*, \ldots$ are unary functions
  (intended meaning: $2^* x$ is $x + x$, $-3^* x$ is $-x - x - x$)
• $+, -, >, =$ have the usual meaning

$T_\mathbb{N}$ and $T_\mathbb{Z}$ have the same expressiveness

• Every $\Sigma_\mathbb{Z}$-formula can be reduced to $\Sigma_\mathbb{N}$-formula
• Every $\Sigma_\mathbb{N}$-formula can be reduced to $\Sigma_\mathbb{Z}$-formula
Example: compiler optimization

```java
for (i=1; i<=10; i++) {
    a[j+i] = a[j];
}
```

```java
int v = a[j];
for (i=1; i<=10; i++) {
    a[j+i] = v;
}
```

A $T_\mathbb{Z}$ formula that is satisfiable iff this transformation is invalid:

$$(i \geq 1) \land (i \leq 10) \land (j + i = j)$$

quantifier-free conjunctive fragment
Theory of Reals $T_{\mathbb{R}}$ and Theory of Rationals $T_{\mathbb{Q}}$

$\Sigma_{\mathbb{R}} : \{0, 1, +, -, *, =, \geq \}$ with multiplication

$\Sigma_{\mathbb{Q}} : \{0, 1, +, -, =, \geq \}$ without multiplication

Both are decidable
- High time complexity

Quantifier-free fragment of $T_{\mathbb{Q}}$ is efficiently decidable
Theory of Arrays $T_A$

$\Sigma_A : \{select, store, =\}$

where

- $select(a,i)$ is a binary function:
  - read array $a$ at index $i$
- $store(a,i,v)$ is a ternary function:
  - write value $v$ to index $i$ of array $a$

Axioms of $T_A$

1. $\forall a, i, j. \ i = j \rightarrow select(a,i) = select(a,j)$ (array congruence)
2. $\forall a, v, i, j. \ i = j \rightarrow select(store(a,i,v),j) = v$ (select-store 1)
3. $\forall a, v, i, j. \ i \neq j \rightarrow select(store(a,i,v),j) = select(a,j)$ (select-store 2)

$T_A$ is undecidable

Quantifier-free fragment of $T_A$ is decidable
Note about $T_A$

Equality (=) is only defined for array elements...

- Example:
  \[ \text{select}(a,i) = e \rightarrow \forall j. \text{select}(\text{store}(a,i,e), j) = \text{select}(a,j) \]
  is $T_A$-valid

...and not for whole arrays

- Example:
  \[ \text{select}(a,i) = e \rightarrow \text{store}(a,i,e) = a \]
  is not $T_A$-valid

A program:
\[
A[1] = -1; \\
A[2] = 1; \\
k = \text{unknown}(); \\
\text{if} (A[k] == 1) ...
\]
# Summary of Decidability Results

[BM Ch. 3, Page 90]

<table>
<thead>
<tr>
<th>Theory</th>
<th>Quantifiers Decidable</th>
<th>QFF Decidable</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_E$ Equality</td>
<td>NO ✓</td>
<td>YES</td>
</tr>
<tr>
<td>$T_{PA}$ Peano Arithmetic</td>
<td>NO ✓</td>
<td>NO ✓</td>
</tr>
<tr>
<td>$T_N$ Presburger Arithmetic</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>$T_Z$ Linear Integer Arithmetic</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>$T_R$ Real Arithmetic</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>$T_Q$ Linear Rationals</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>$T_A$ Arrays</td>
<td>NO ✓</td>
<td>YES</td>
</tr>
</tbody>
</table>

QFF: Quantifier Free Fragment
### Summary of Complexity Results

[BM Ch. 3, Pages 90, 91]

<table>
<thead>
<tr>
<th>Theory</th>
<th>Quantifiers</th>
<th>QF Conjunctive</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_E$ Equality</td>
<td>–</td>
<td>O($n \log n$)</td>
</tr>
<tr>
<td>$T_N$ Presburger Arithmetic</td>
<td>O($2^2^2^2^{(kn)}$)</td>
<td>NP-complete</td>
</tr>
<tr>
<td>$T_Z$ Linear Integer Arithmetic</td>
<td>O($2^2^2^2^{(kn)}$)</td>
<td>NP-complete</td>
</tr>
<tr>
<td>$T_R$ Real Arithmetic</td>
<td>O($2^2^2^2^{(kn)}$)</td>
<td>O($2^2^2^2^{(kn)}$)</td>
</tr>
<tr>
<td>$T_Q$ Linear Rationals</td>
<td>O($2^2^2^2^{(kn)}$)</td>
<td>PTIME</td>
</tr>
<tr>
<td>$T_A$ Arrays</td>
<td>–</td>
<td>NP-complete</td>
</tr>
</tbody>
</table>

$n$ – input formula size; $k$ – some positive integer

**Note:** Quantifier-free Conjunctive fragments look good!
Combination of Theories

Many applications require reasoning over a combination of theories

- Example: \( f(x) + 10 = g(y) \) belongs to \( T_E \cup T_\mathbb{Z} \)

Given decision procedures for theories \( T_1 \) and \( T_2 \), can we decide satisfiability of formulae in \( T_1 \cup T_2 \) ?

Theory \( T_1 \)
- Signature: \( \Sigma_{T_1} \)
- Axioms: \( A_{T_1} \)

Theory \( T_2 \)
- Signature: \( \Sigma_{T_2} \)
- Axioms: \( A_{T_2} \)

Theory \( T_1 \cup T_2 \)
- Signature: \( \Sigma_{T_1} \cup \Sigma_{T_2} \)
- Axioms: \( A_{T_1} \cup A_{T_2} \)

Undecidable for arbitrary \( T_1, T_2 \) but
Decidable under Nelson-Oppen restrictions
Decision Procedure for Combination of Theories

Theory $T_1$
Signature: $\Sigma_{T_1}$
Axioms: $A_{T_1}$

theory solver

theory solver

Theory $T_2$
Signature: $\Sigma_{T_2}$
Axioms: $A_{T_2}$

combination solver ?

Theory $T_1 \cup T_2$
Signature: $\Sigma_{T_1} \cup \Sigma_{T_2}$
Axioms: $A_{T_1} \cup A_{T_2}$

Nelson-Oppen Procedure for deciding satisfiability

- If both $T_1$ and $T_2$ are quantifier-free (conjunctive) fragments
- If “=” is the only symbol common to their signatures
- If $T_1$ and $T_2$ meet certain other technical restrictions

Note: Quantifier-free Conjunctive fragments look good!
Modern SMT solvers support many useful theories

- QF_UF: Quantifier-free Equality with Uninterpreted Functions
- QF_LIA: Quantifier-free Linear Integer Arithmetic
- QF_LRA: Quantifier-free Linear Real Arithmetic
- QF_BV: Quantifier-free Bit Vectors (fixed-width)
- QF_A: Quantifier-free Arrays

... and many combinations

Check out more info at

http://smtlib.cs.uiowa.edu/index.shtml
http://smtlib.cs.uiowa.edu/logics.shtml
This lecture: Theory solvers for QF Conjunctive fragments
Next lecture: DPLL(T)
Z3 SMT Solver

http://rise4fun.com/z3/

Input format is an extension of SMT-LIB standard

Commands

- `declare-const` – declare a constant of a given type
- `declare-fun` – declare a function of a given type
- `assert` – add a formula to Z3’s internal stack
- `check-sat` – determine if formulas currently on stack are satisfiable
- `get-model` – retrieve an interpretation
- `exit`
Main idea: combine DPLL SAT solving with theory solvers

- DPLL-based SAT over the *Boolean structure* of the formula
- theory solver handles the *conjunctive fragment*

- Recall: SAT solvers use many techniques to prune the exponential search space

This is called DPLL(T)

- T could also be a combination of theories (using Nelson-Oppen)
DPLL(T): main idea

SAT solver handles Boolean structure of the formula
- Treats each atomic formula as a propositional variable
- Resulting formula is called a *Boolean abstraction* (*B*)

Example

\[ F: (x=z) \land ((y=z \land x = z+1) \lor \neg (x=z)) \]

\[ \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \]

\[ b_1 \quad b_2 \quad b_3 \quad b_1 \]

- \( B(F): b_1 \land ((b_2 \land b_3) \lor \neg b_1) \)
- Boolean abstraction (*B*) defined inductively over formulas
- \( B \) is a bijective function, \( B^{-1} \) also exists
  - \( B^{-1} (b_1 \land b_2 \land b_3): (x=z) \land (y=z) \land (x=z+1) \)
  - \( B^{-1} (b_1 \lor b_2'): (x=z) \lor \neg(y=z) \)
DPLL(T): main idea

Example

F: \((x=z) \land ((y=z \land x = z+1) \lor \neg (x=z))\)

\[\downarrow \quad \downarrow \quad \downarrow \quad \downarrow\]

b1  b2  b3  b1

- B(F): \(b1 \land ((b2 \land b3) \lor \neg b1)\)

- Use DPLL SAT solver to decide satisfiability of B(F)
  - If B(F) is Unsat, then F is Unsat
  - If B(F) has a satisfying assignment A, F may still be Unsat

Example

- SAT solver finds a satisfying assignment A: \(b1 \land b2 \land b3\)
- But, \(B^{-1}(A)\) is unsatisfiable modulo theory
  - \((x=z) \land (y=z) \land (x=z+1)\) is not satisfiable
DPLL(T): main idea

Example

\[ F: (x=z) \land ((y=z \land x = z+1) \lor \neg (x=z)) \]

- \[ B(F): b_1 \land ((b_2 \land b_3) \lor \neg b_1) \]

• Use DPLL SAT solver to decide satisfiability of \( B(F) \)
  • If \( B(F) \) is Unsat, then \( F \) is Unsat
  • If \( B(F) \) has a satisfying assignment \( A \)

• Use theory solver to check if \( B^{-1}(A) \) is satisfiable modulo \( T \)
  • Note \( B^{-1}(A) \) is in conjunctive fragment
  • If \( B^{-1}(A) \) is satisfiable modulo theory \( T \), then \( F \) is satisfiable
DPLL(T): simple version

- Generate a Boolean abstraction $B(F)$
- Use DPLL SAT solver to decide satisfiability of $B(F)$
  - If $B(F)$ is Unsat, then $F$ is Unsat
  - If $B(F)$ has a satisfying assignment $A$
- Use theory solver to check $B^{-1}(A)$ is satisfiable modulo $T$
  - If $B^{-1}(A)$ is satisfiable modulo theory $T$, then $F$ is satisfiable
  - Because $A$ satisfies the Boolean structure, and is consistent with $T$
- What if $B^{-1}(A)$ is unsatisfiable modulo $T$? Is $F$ Unsat?
  - No!
    - There may be other assignments $A'$ that satisfy the Boolean structure and are consistent with $T$
- *Add* $\neg A$ to $B(F)$, and *backtrack* in DPLL SAT to find other assignments
  - Until there are no more satisfying assignments of $B(F)$

Like a conflict clause (due to a theory conflict)
DPLL(T): simple version recap

1. Generate a Boolean abstraction $B(F)$

2. Use DPLL SAT solver to decide satisfiability of $B(F)$
   - If $B(F)$ is Unsat, then $F$ is Unsat
   - Otherwise, find a satisfying assignment $A$

3. Use theory solver to check if $B^{-1}(A)$ is satisfiable modulo $T$
   - If $B^{-1}(A)$ is satisfiable modulo theory $T$, then $F$ is satisfiable
   - Otherwise, $B^{-1}(A)$ is unsatisfiable modulo $T$
     Add $\neg A$ to $B(F)$, and backtrack in DPLL SAT

Repeat (2, 3) until there are no more satisfying assignments
DPLL(T): simple version example

- Example F: \((x=z) \land ((y=z \land x = z+1) \lor \neg (x=z))\)
  - B(F): \(b_1 \land ((b_2 \land b_3) \lor \neg b_1)\)
  - DPLL finds A: \(b_1 \land b_2 \land b_3\), \(B^{-1}(A)\): \((x=z) \land (y=z) \land (x=z+1)\)
  - Theory solver checks \(B^{-1}(A)\), this is unsat modulo T, therefore add \(\neg A\)
  - DPLL finds \(B(F) \land \neg A\): \(b_1 \land ((b_2 \land b_3) \lor \neg b_1) \land (b_1' + b_2' + b_3')\) is Unsat
  - Therefore, F is Unsat
DPLL(T): simple version

- **Correctness**
  - When it says “F is Sat”, there is an assignment that satisfies the Boolean structure *and* is consistent with theory.
  - When it says “F is Unsat”, the formula is unsatisfiable because $B(F) \land \neg A$ is also an over-approximation of F.
  - $B^{-1}(A)$ is not consistent with T, i.e., $B^{-1}(\neg A)$ is T-valid.
DPLL(T): simple version

- **Termination**
  - B(F) has only a finite number of satisfying assignments
  - When \( \neg A \) is added to B(F), the assignment A will never be generated again
  - Either some satisfying assignment of B(F) is also T-satisfiable (F is SAT), or all satisfying assignments of B(F) are not T-satisfiable (F is Unsat)
An Example of Symbolic Analysis and DPLL(T)

1. \( m = \text{getstr}(); \)
2. \( n = \text{getstr}(); \)
3. \( i = \text{getint}(); \)
4. \( x = \text{strcat}(\text{“abc”}, m) \)
5. \( \text{if } (\text{strlen}(m) + i > 5) \)
6. \( y = \text{“abcd”} \)
7. \( \text{else} \)
8. \( y = \text{strcat}(\text{“efg”}, n); \)
9. \( \text{if } (x == y) \ldots \)

Path 1: \( \text{assert}(e_1 \land \neg e_2) \)

| \( e_1 \) | \( x = \text{concat}(\text{“abc”}, m) \) |
| \( e_2 \) | \( \text{strlen}(m) + i > 5 \) |
| \( e_3 \) | \( y = \text{concat}(\text{“efg”}, n) \) |
| \( e_4 \) | \( y = \text{“abcd”} \) |
| \( e_5 \) | \( x = y \) |
An Example of Symbolic Analysis and DPLL(T)

Path 2: $\text{assert}(e_1 \land e_2 \land e_4 \land e_5)$

- $e_1 : x = \text{concat}("abc", m)$
- $e_2 : \text{strlen}(m) + i > 5$
- $e_3 : y = \text{concat}("efg", n)$
- $e_4 : y = "abcd"
- $e_5 : x = y$

**Add:**
- $e_5 : m = "d"
  - $\text{concat}("abc", m) = "abcd" \rightarrow e_5$
  - $e_5 \rightarrow \text{strlen}(m) = 1$