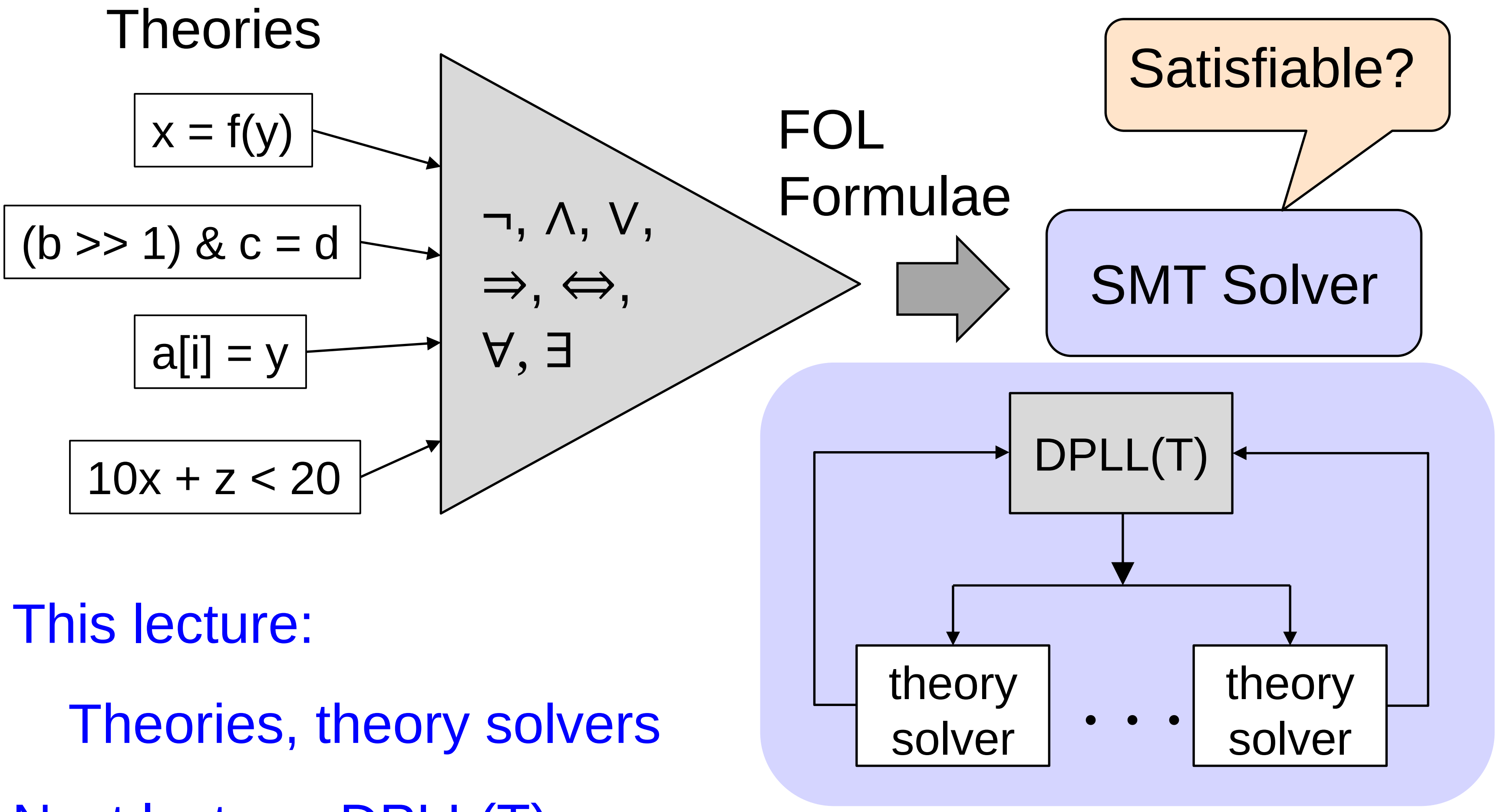
CS510 Software Engineering

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: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$
  - Quantifiers: $\forall$, $\exists$
- **Non-logical symbols**
  - Variables: x, y, z
  - N-ary functions: f, g
  - N-ary predicates: p, q
  - Constants: a, b, c

-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:

$\forall x \forall y\ x>0 \wedge y>0 \Rightarrow add(x,y)>0$

First-order theory *T* is defined by:

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

Examples of FO theories

$x = f(y)$

Equality (and uninterpreted functions)

- = stands for the usual equality
- f is not interpreted in T-model

$(b >> 1) \& c = d$

Fixed-width bitvectors

- >> is shift operator (function)
- & is bit-wise-and operator (function)
- 1 is a constant

$10x + z < 20$

Linear arithmetic (over R and Z)

- + is arithmetic plus (function)
- < is less-than (predicate)
- 10 and 20 are constants

$a[i] = y$

Arrays

- a[i] can be viewed as *select(a, i)* that selects the i-th element in array a

Satisfiability Modulo Theory

First-order theory T is defined by:

- **Signature Σ_T**
  - set of constant, function, and predicate symbols
- **Set of *T-Models***
  - models that fix the interpretation of symbols of Σ_T
  - alternately, can use Axioms $\boldsymbol{A_T}$ *(closed Σ_T formulae)* to provide meaning to symbols of Σ_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 Σ_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. $\forall x,y,z.\ x=y \wedge y=z \rightarrow x=z$ (transitivity)
4. for each n-ary function symbol f,
   $\forall x_1,\ldots,x_n,y_1,\ldots,y_n.\ \bigwedge_i (x_i=y_i) \rightarrow$
   $f(x_1,\ldots,x_n) = f(y_1,\ldots,y_n)$ (function congruence)
5. for each n-ary predicate symbol p,
   $\forall x_1,\ldots,x_n,y_1,\ldots,y_n.\ \bigwedge_i (x_i=y_i) \rightarrow$
   $(p(x_1,\ldots,x_n) \leftrightarrow p(y_1,\ldots,y_n))$ (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: $O(n \log n)$

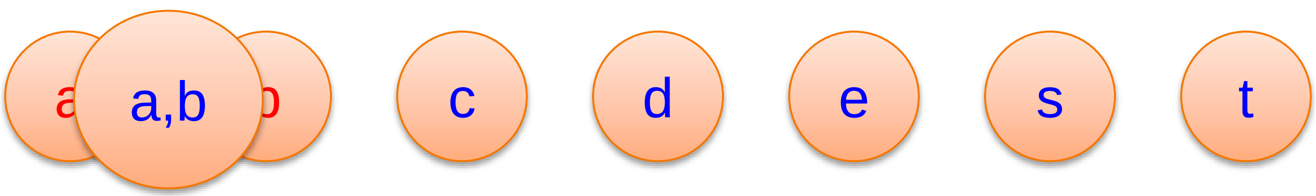
Deciding Equality

$a = b, b = c, d = e, b = s, d = t, a \neq e, a \neq s$

Note: Quantifier-free, Conjunctive

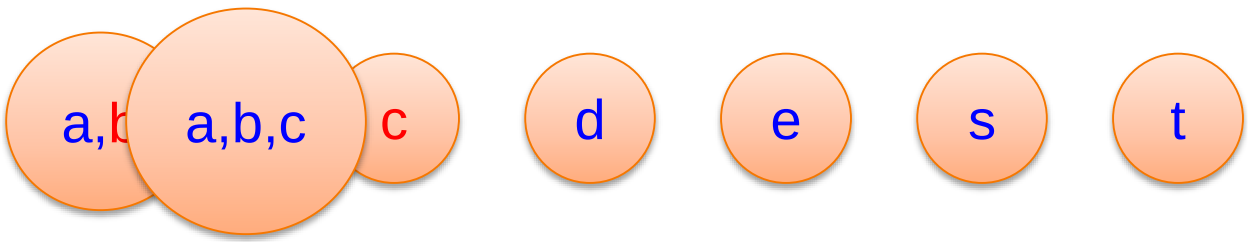
Deciding Equality

$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ a \neq e,\ a \neq s$



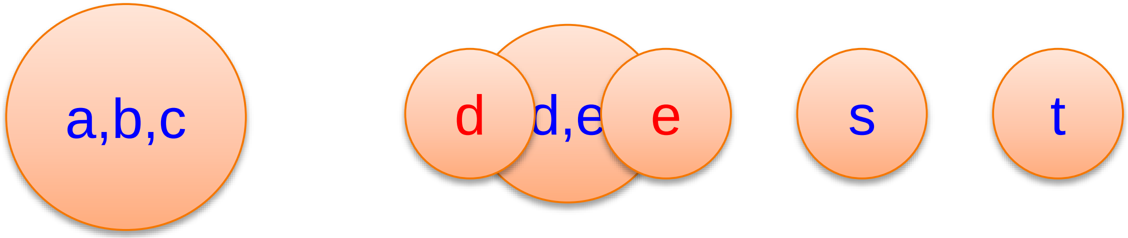
Deciding Equality

$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ a \neq e,\ a \neq s$



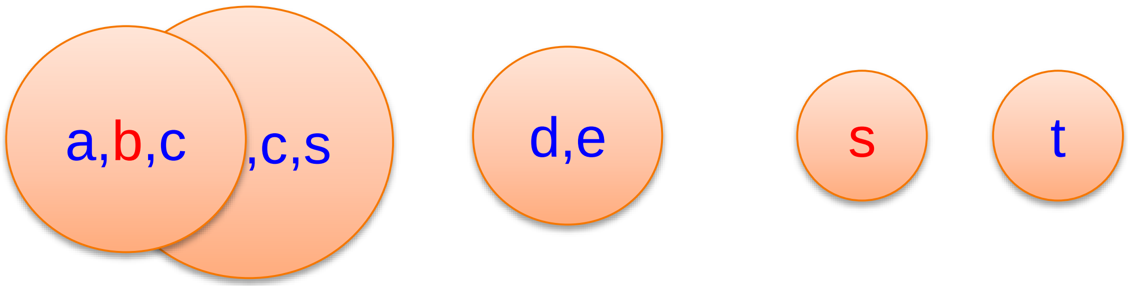
Deciding Equality

$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ a \neq e,\ a \neq s$



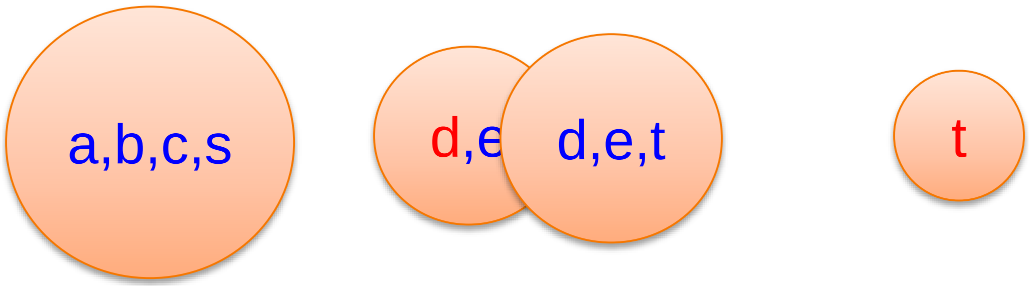
Deciding Equality

$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ a \neq e,\ a \neq s$



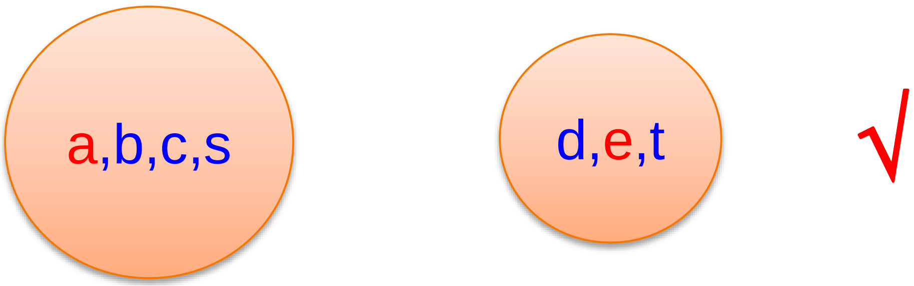
Deciding Equality

$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ a \neq e,\ a \neq s$



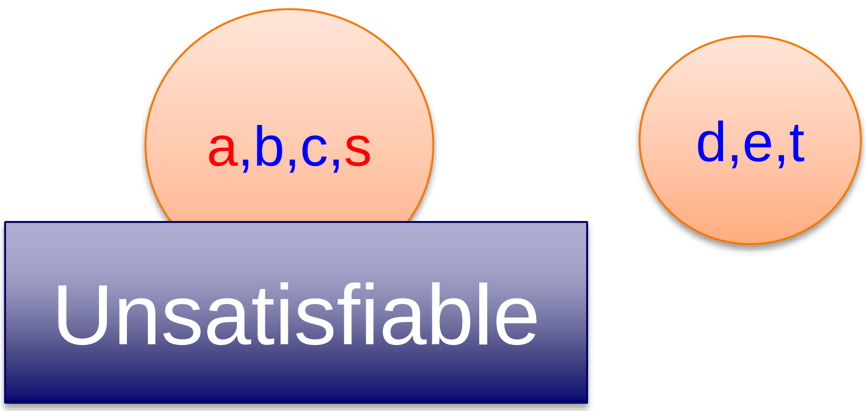
Deciding Equality

$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ a \neq e,\ a \neq s$



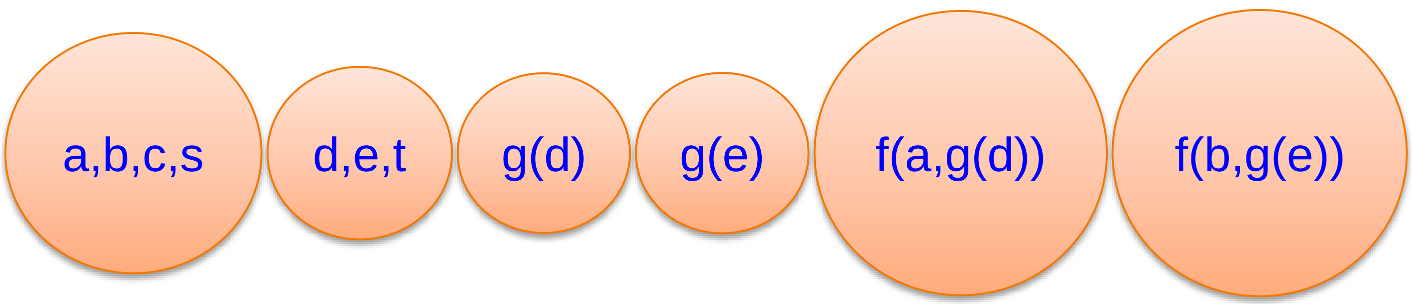
Deciding Equality

$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ a \neq e,\ a \neq s$



Deciding Equality + (uninterpreted) Functions

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

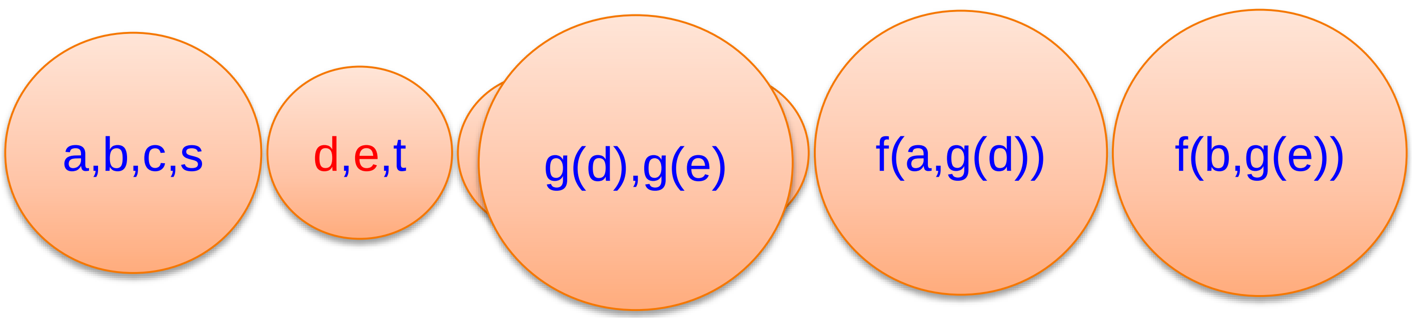


Congruence Rule:

$x_1 = y_1, \ldots, x_n = y_n$ implies $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$

Deciding Equality + (uninterpreted) Functions

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

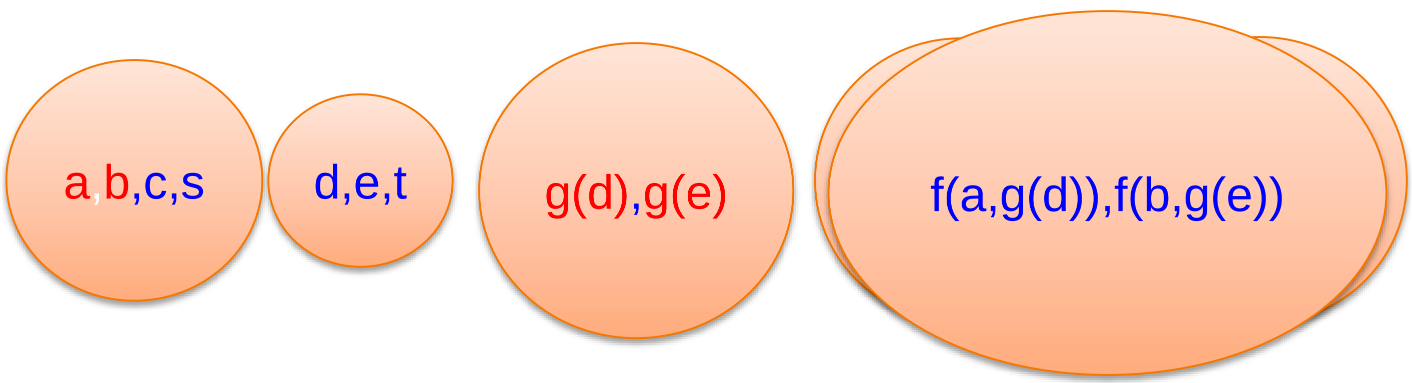


Congruence Rule:

$x_1 = y_1, \ldots, x_n = y_n$ implies $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$

Deciding Equality + (uninterpreted) Functions

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

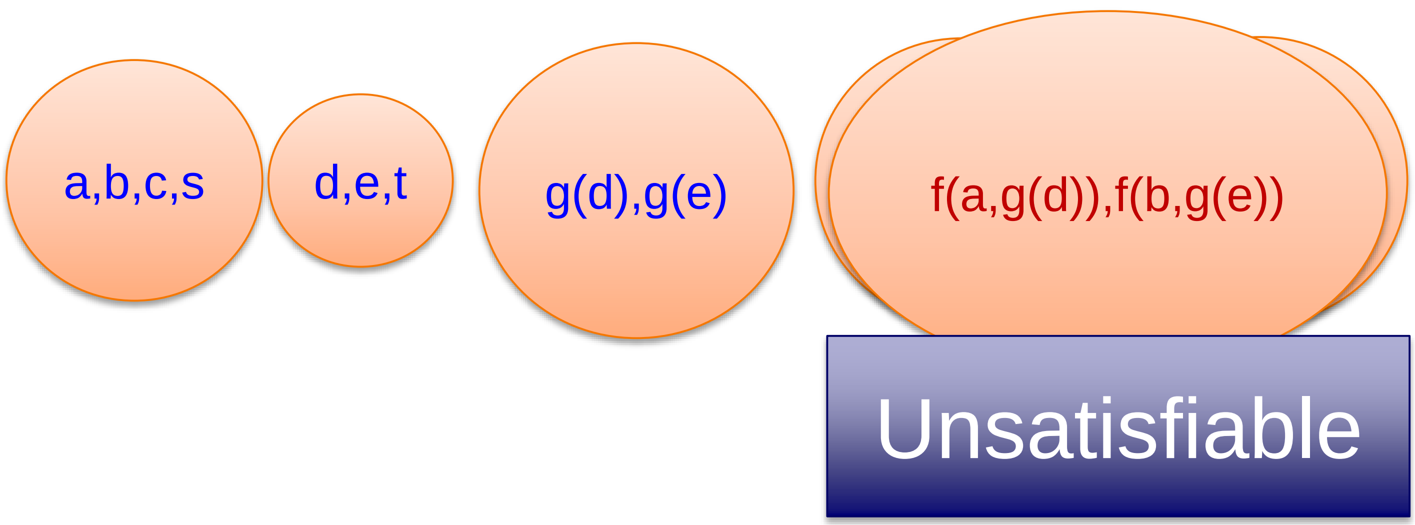


Congruence Rule:

$x_1 = y_1, \ldots, x_n = y_n$ 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

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

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

T_E formula that is satisfiable iff programs are not equivalent:

$(z1 = y0 \wedge y1 = x0 \wedge x1 = z1 \wedge r1 = x1{*}x1) \wedge$
$(r2 = y0{*}\,y0) \wedge$
$\neg(r2 = r1)$

quantifier-free conjunctive fragment

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

Example: program equivalence

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

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

T_E formula that is satisfiable iff programs are not equivalent:

(z1 = y0 ∧ y1 = x0 ∧ x1 = z1 ∧ r1 = **sq**(x1) ∧
(r2 = **sq**(y0)) ∧
¬(r2 = r1)

uninterpreted function symbol **sq** (abstraction of *)

Using T_E (with uninterpreted functions), SMT solver proves unsat in a fraction of a second.

Example: program equivalence

```
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 ^ (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

$$
\begin{array}{rcl}
\mathit{formula} & : & \mathit{formula} \vee \mathit{formula} \mid \neg\mathit{formula} \mid \mathit{atom} \\
\mathit{atom} & : & \mathit{term}\ \mathit{rel}\ \mathit{term} \mid \mathit{Boolean\text{-}Identifier} \mid \mathit{term}[\,\mathit{constant}\,] \\
\mathit{rel} & : & = \ \mid\ < \\
\mathit{term} & : & \mathit{term}\ \mathit{op}\ \mathit{term} \mid \mathit{identifier} \mid\ \sim \mathit{term} \mid \mathit{constant} \mid \\
 & & \mathit{atom}\,\mathtt{?}\,\mathit{term}\,\mathtt{:}\,\mathit{term} \mid \\
 & & \mathit{term}[\,\mathit{constant} : \mathit{constant}\,] \mid \mathit{ext}(\,\mathit{term}\,) \\
\mathit{op} & : & + \mid - \mid \cdot \mid / \mid \ll \mid \gg \mid \& \mid \mathtt{|} \mid \oplus \mid \circ
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{formula} & : & \mathit{formula} \vee \mathit{formula} \mid \neg \mathit{formula} \mid \mathit{atom} \\
\mathit{atom} & : & \mathit{term}\ \mathit{rel}\ \mathit{term} \mid \mathit{Boolean\text{-}Identifier} \mid \mathit{term}[\,\mathit{constant}\,] \\
\mathit{rel} & : & = \mid < \\
\mathit{term} & : & \mathit{term}\ \mathit{op}\ \mathit{term} \mid \mathit{identifier} \mid \sim \mathit{term} \mid \mathit{constant} \mid \\
 & & \mathit{atom}?\mathit{term}:\mathit{term} \mid \\
 & & \mathit{term}[\,\mathit{constant} : \mathit{constant}\,] \mid \mathit{ext}(\,\mathit{term}\,) \\
\mathit{op} & : & + \mid - \mid \cdot \mid / \mid << \mid >> \mid \& \mid \text{|} \mid \oplus \mid \circ
\end{array}
$$

$\sim x$: bit-wise negation of x

$ext(x)$: sign- or zero-extension of x

$x << d$: left shift with distance d

$x \circ y$: 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*'

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?

What constraints do we generate for a given atom

This is easy for the bit-wise operators.

Example for $t = a \mid b$

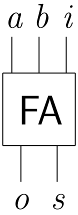
$$\bigwedge_{i=0}^{l-1} t_i = (a_i \vee b_i))$$

What about x=y

How to flatten $s=a+b$

How to flatten *s=a+b*

$\longrightarrow$ we can build a *circuit* that adds them!



Full Adder

$$s \equiv (a+b+i\,)\bmod 2 \equiv a \oplus b \oplus i$$

$$o \equiv (a+b+i\,)\operatorname{div} 2 \equiv a \cdot b + a \cdot i + b \cdot i$$

The full adder in CNF:

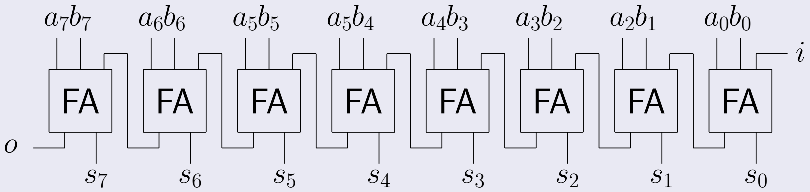
$$(a \vee b \vee \neg o) \wedge (a \vee \neg b \vee i \vee \neg o) \wedge (a \vee \neg b \vee \neg i \vee o) \wedge$$
$$(\neg a \vee b \vee i \vee \neg o) \wedge (\neg a \vee b \vee \neg i \vee o) \wedge (\neg a \vee \neg b \vee o)$$

Flattening bit-vector arithmetic

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

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:

$$a \cdot b = c \wedge b \cdot a \neq c \wedge x < y \wedge x > y$$

CNF: About 11000 variables, unsolvable for current SAT solvers

Similar problems with division, modulo

Multipliers result in very hard formulas

Example:

$$a \cdot b = c \wedge b \cdot a \neq c \wedge x < y \wedge x > y$$

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 $\boldsymbol{T_{PA}}$

- 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 $T_{\mathbb{Z}}$

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

where

- …,-2,-1,0,1,2,… are constants
- …,-3*,-2*,2*,3*,… 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

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

```
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:

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

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

Σ_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:

  $select(a,i) = e \rightarrow \forall j.\ select(\ store(a,i,e), j) = select(a,j)$

  is T_A-valid

…and not for whole arrays

- Example:

  $select(a,i)=e \rightarrow store(a,i,e)=a$

  is not T_A-valid

A program:

A[1]=-1;
A[2]=1;
k=unknown( );
if (A[k]==1) ...

Summary of Decidability Results

[BM Ch. 3, Page 90]

Theory		Quantifiers Decidable	QFF Decidable
T_E	Equality	NO	YES
T_{PA}	Peano Arithmetic	NO	NO
$T_{\mathbb{N}}$	Presburger Arithmetic	YES	YES
$T_{\mathbb{Z}}$	Linear Integer Arithmetic	YES	YES
$T_{\mathbb{R}}$	Real Arithmetic	YES	YES
$T_{\mathbb{Q}}$	Linear Rationals	YES	YES
T_A	Arrays	NO	YES

QFF: Quantifier Free Fragment

Summary of Complexity Results

[BM Ch. 3, Pages 90, 91]

Theory		Quantifiers	QF Conjunctive
T_E	Equality	–	$O(n \log n)$
$T_{\mathbb{N}}$	Presburger Arithmetic	$O(2^{\wedge}2^{\wedge}2^{\wedge}(kn))$	NP-complete
$T_{\mathbb{Z}}$	Linear Integer Arithmetic	$O(2^{\wedge}2^{\wedge}2^{\wedge}(kn))$	NP-complete
$T_{\mathbb{R}}$	Real Arithmetic	$O(2^{\wedge}2^{\wedge}(kn))$	$O(2^{\wedge}2^{\wedge}(kn))$
$T_{\mathbb{Q}}$	Linear Rationals	$O(2^{\wedge}2^{\wedge}(kn))$	PTIME
T_A	Arrays	–	NP-complete

n – input formula size; k – some positive integer

Note: Quantifier-free Conjunctive fragments look good!

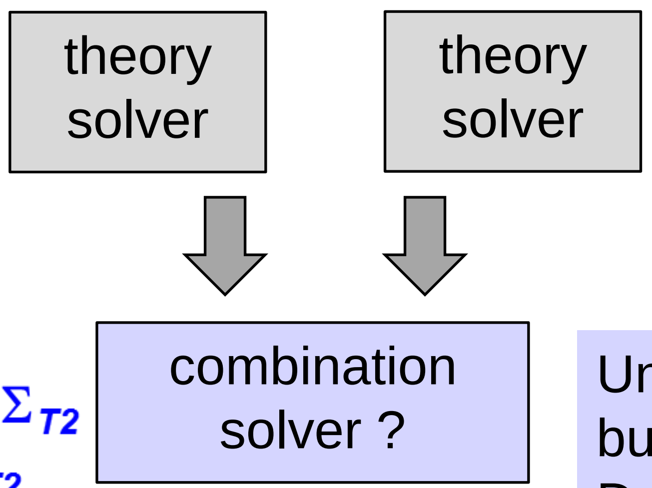
Combination of Theories

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 T1 and T2, can we decide satisfiability of formulae in $T1 \cup T2$?

Theory $T1$
Signature: Σ_{T1}
Axioms: $\boldsymbol{A_{T1}}$



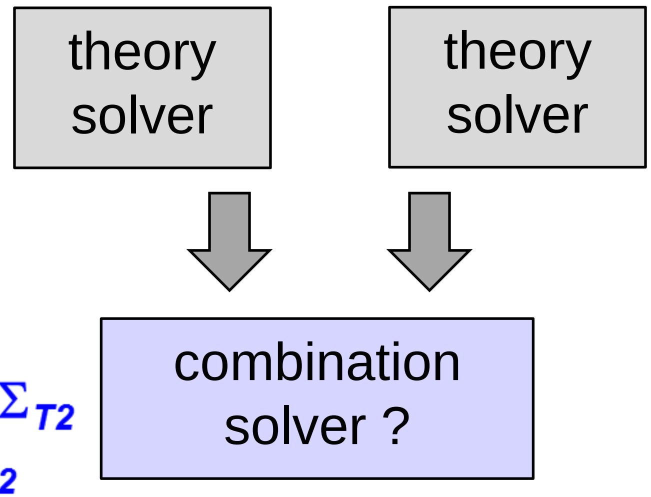
Theory $T2$
Signature: Σ_{T2}
Axioms: $\boldsymbol{A_{T2}}$

Theory $T1 \cup T2$
Signature: $\Sigma_{T1} \cup \Sigma_{T2}$
Axioms: $\boldsymbol{A_{T1}} \cup \boldsymbol{A_{T2}}$

Undecidable for arbitrary T1, T2 but
Decidable under Nelson-Oppen restrictions

Decision Procedure for Combination of Theories

Theory $T1$
Signature: Σ_{T1}
Axioms: $\boldsymbol{A}_{T1}$



Theory $T2$
Signature: Σ_{T2}
Axioms: $\boldsymbol{A}_{T2}$

Theory $T1 \cup T2$
Signature: $\Sigma_{T1} \cup \Sigma_{T2}$
Axioms: $\boldsymbol{A}_{T1} \cup \boldsymbol{A}_{T2}$

Nelson-Oppen Procedure for deciding satisfiability

- If both T1 and T2 are *quantifier-free (conjunctive) fragments*
- If “=“ is the only symbol common to their signatures
- If T1 and T2 meet certain other technical restrictions

Note: Quantifier-free Conjunctive fragments look good!

Theories in SMT Solvers

Modern SMT solvers support many useful theories

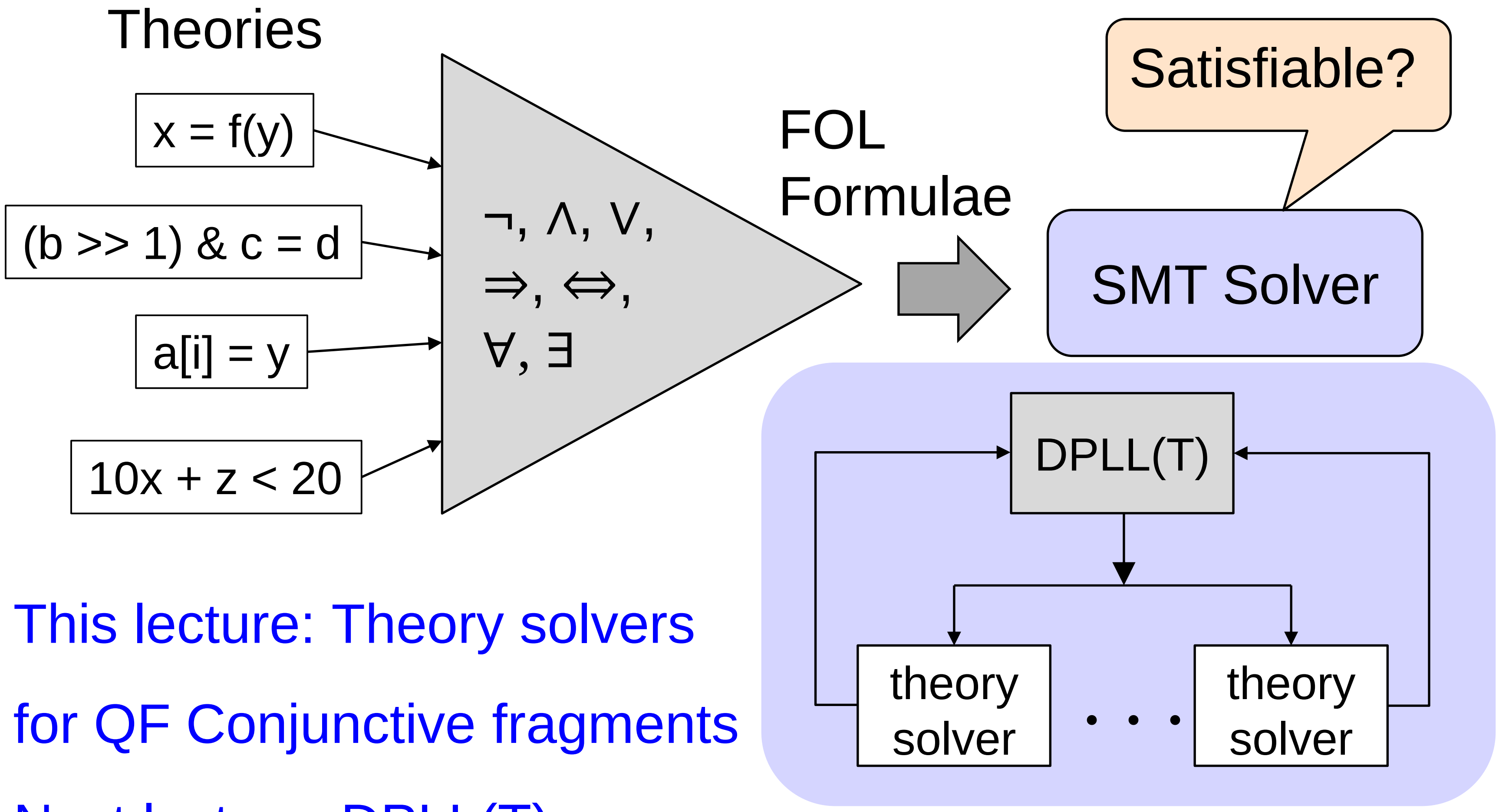
- QF_UF: Quantifier-free Equality with Uninterpreted Functions
- QF_LIA: Quantifier-free Linear Integer Arithmetic
- QF_LRA: Quantifier-free Linear Real Arithmetic
- QF_BV: Quantifier-free Bit Vectors (fixed-width)
- QF_A: Quantifier-free Arrays

… and many combinations

Check out more info at

http://smtlib.cs.uiowa.edu/index.shtml
http://smtlib.cs.uiowa.edu/logics.shtml

summary



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`

SMT solvers: DPLL(T)

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

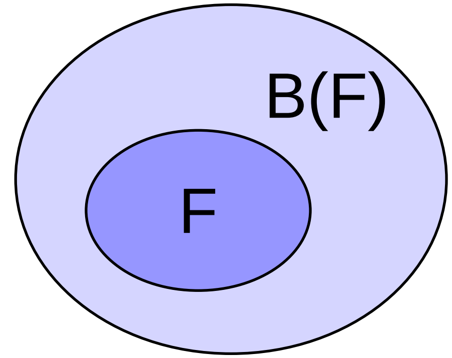
- Treats each atomic formula as a propositional variable
- Resulting formula is called a *Boolean abstraction (B)*

Example

$$F:\ \underbrace{(x=z)}_{b1} \wedge ((\underbrace{y=z}_{b2} \wedge \underbrace{x = z+1}_{b3}) \vee \neg\ \underbrace{(x=z)}_{b1})$$

- B(F): $b1 \wedge ((b2 \wedge b3) \vee \neg b1)$
- Boolean abstraction (B) defined inductively over formulas
- B is a bijective function, B^{-1} also exists
  - $B^{-1}(b1 \wedge b2 \wedge b3)$: $(x=z) \wedge (y=z) \wedge (x=z+1)$
  - $B^{-1}(b1 \vee b2')$: $(x=z) \vee \neg(y=z)$

DPLL(T): main idea



Example

$$F: \underbrace{(x=z)}_{b1} \wedge ((\underbrace{y=z}_{b2} \wedge \underbrace{x = z+1}_{b3}) \vee \neg \underbrace{(x=z)}_{b1})$$

- B(F): $b1 \wedge ((b2 \wedge b3) \vee \neg b1)$

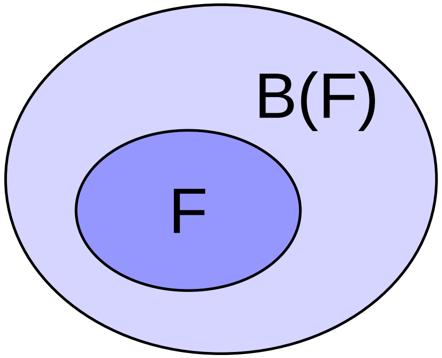
B(F) is an *over-approximation* of F

- 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, F may still be Unsat

Example

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

DPLL(T): main idea



Example

F: (x=z) ∧ ((y=z ∧ x = z+1) ∨ ¬ (x=z))

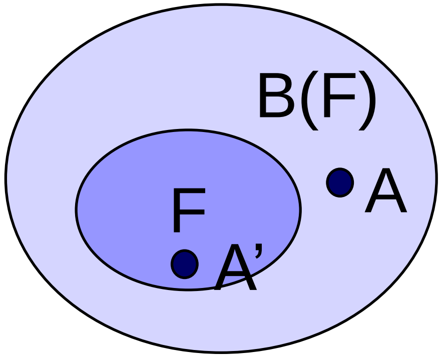
b1 b2 b3 b1

- B(F): b1 ∧ ((b2 ∧ b3) ∨ ¬b1)

B(F) is an *over-approximation* of 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 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 ¬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)

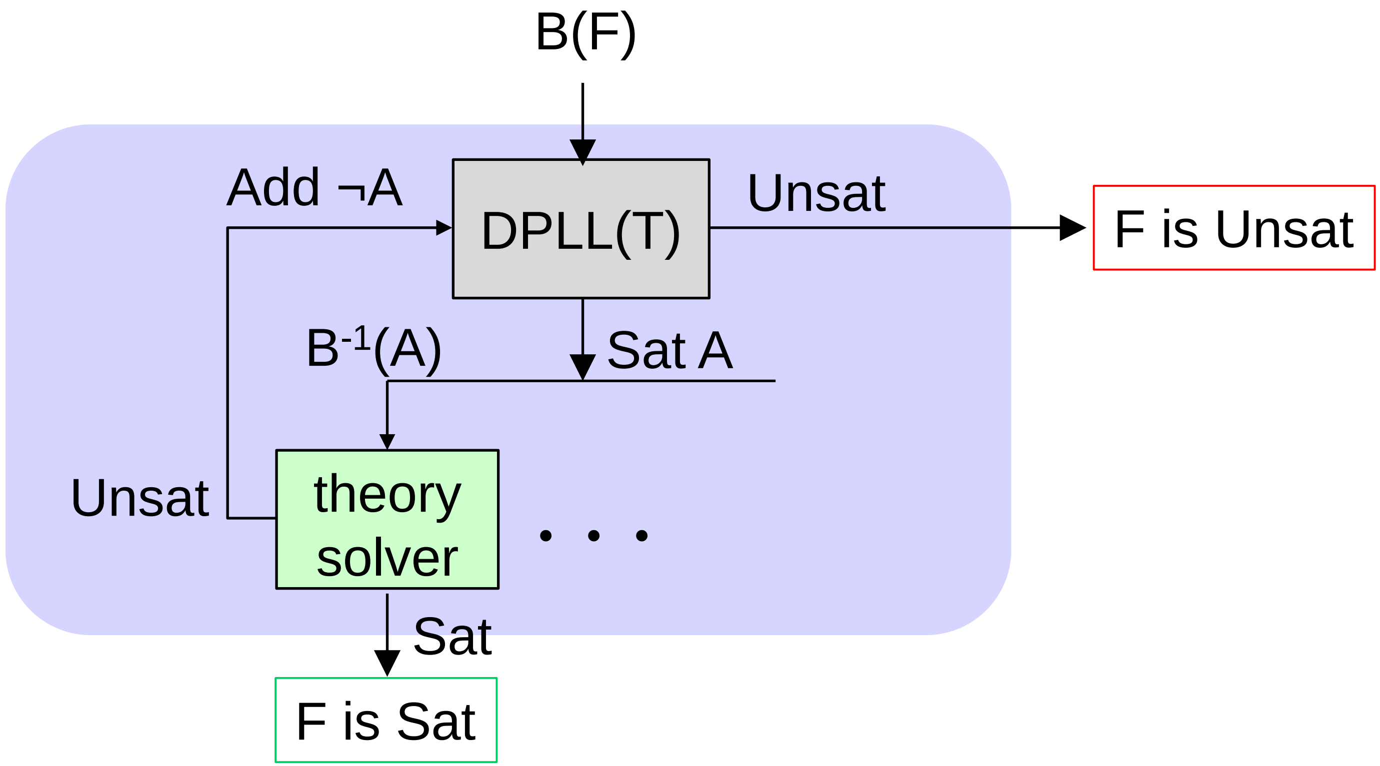
DPLL(T): simple version recap

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

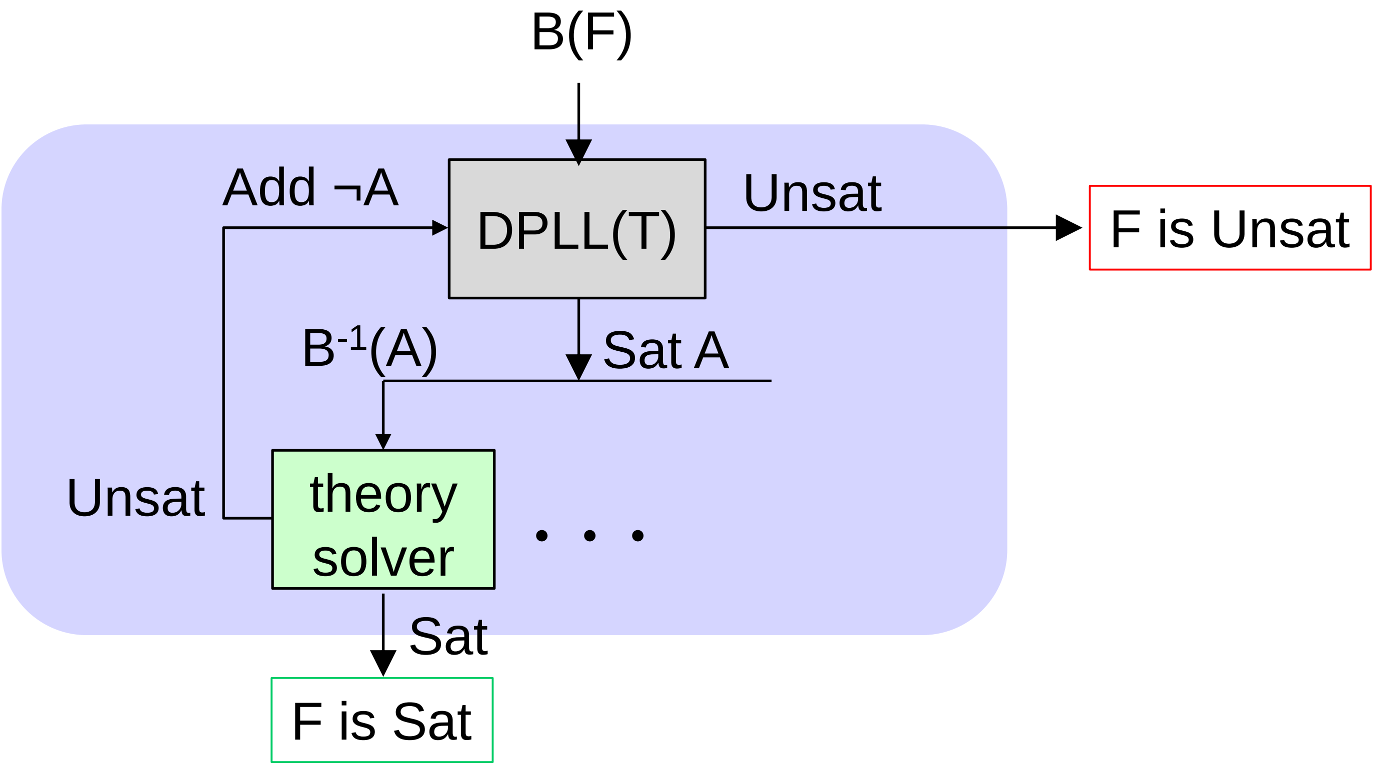
Repeat (2, 3) until there are no more satisfying assignments

DPLL(T): simple version example



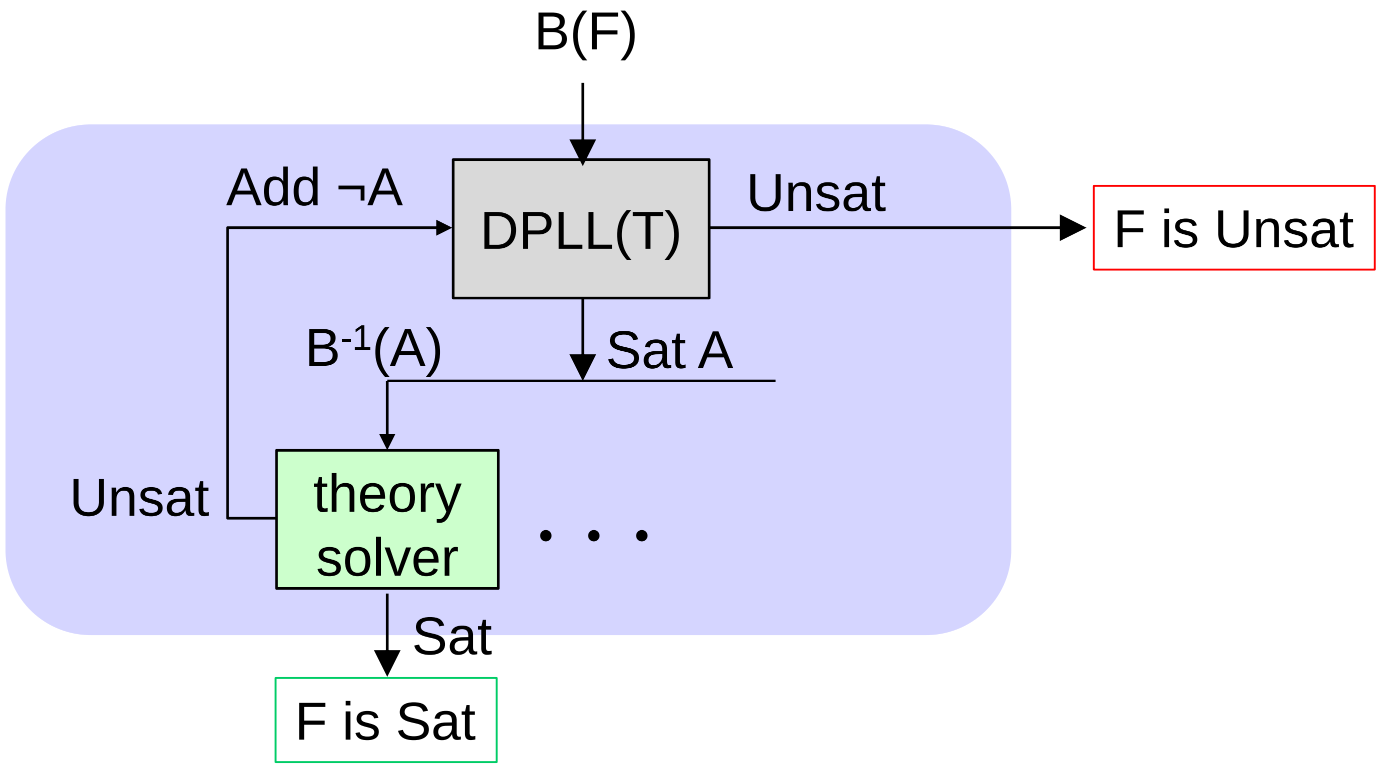
- Example F: (x=z) ∧ ((y=z ∧ x = z+1) ∨ ¬ (x=z))
  - B(F): b1 ∧ ((b2 ∧ b3) ∨ ¬b1)
  - DPLL finds A: b1 ∧ b2 ∧ b3, B^{-1}(A): (x=z) ∧ (y=z) ∧ (x=z+1)
  - Theory solver checks B^{-1}(A), this is unsat modulo T, therefore add ¬*A*
  - DPLL finds B(F) ∧ ¬*A:* b1 ∧ ((b2 ∧ b3) ∨ ¬b1) ∧ (b1' + b2' + b3') is 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) ∧ ¬*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 ¬*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=getstr();
2. n=getstr();
3. i=getint();
4. x=strcat(“abc”,m)
5. if (strlen(m)+i>5)
6.   y=“abcd”
7. else
8.   y=strcat(“efg”,n);
9. if (x==y) …

$Path1 : assert(e_1 \wedge !e_2)$
$e_1 : x = concat("abc", m)$ $e_2 : strlen(m) + i > 5$ $e_3 : y = concat("efg", n)$ $e_4 : y = "abcd"$ $e_5 : x = y$

An Example of Symbolic Analysis and DPLL(T)