Computer-Aided Reasoning for Software

CSE507

Satisfiability Modulo Theories

courses.cs.washington.edu/courses/cse507/18sp/

Emina Torlak
emina@cs.washington.edu

Today

Last lecture

- Practical applications of SAT and the need for a richer logic

Today

- Introduction to Satisfiability Modulo Theories (SMT)
- Syntax and semantics of (quantifier-free) first-order logic
- Overview of key theories

topics

Satisfiability Modulo Theories (SMT)

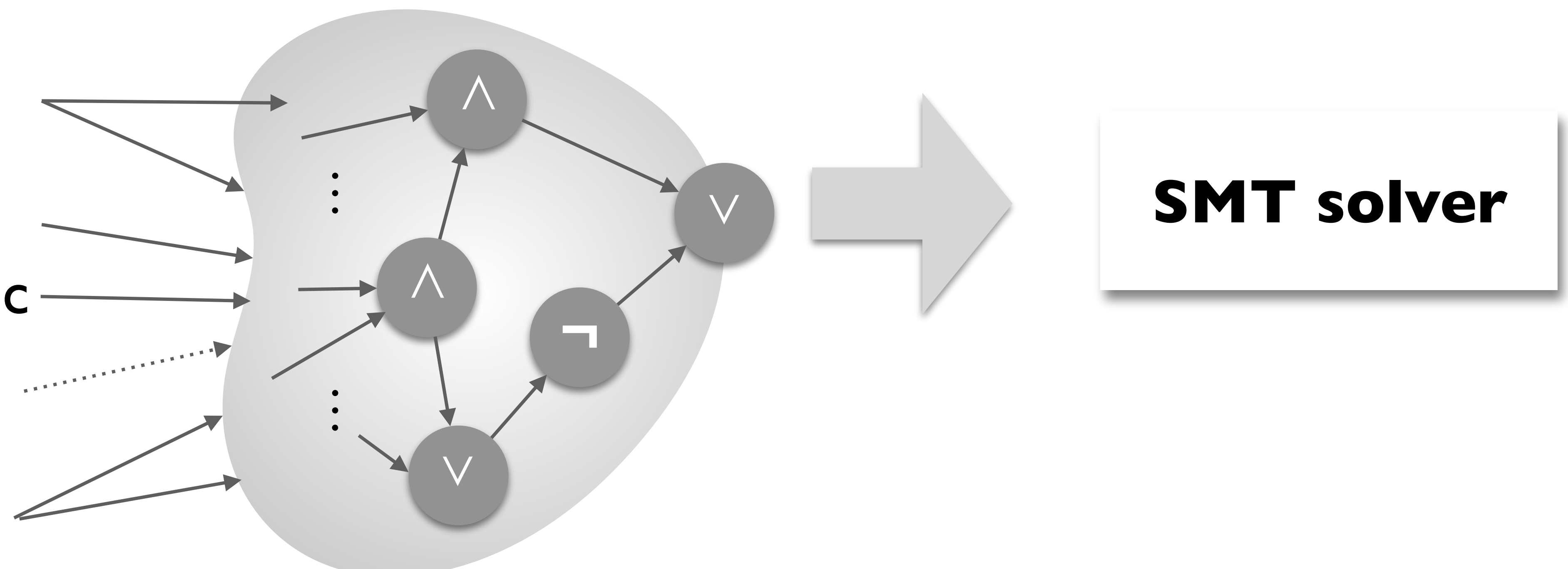
$x = g(y)$

$2x + y \leq 5$

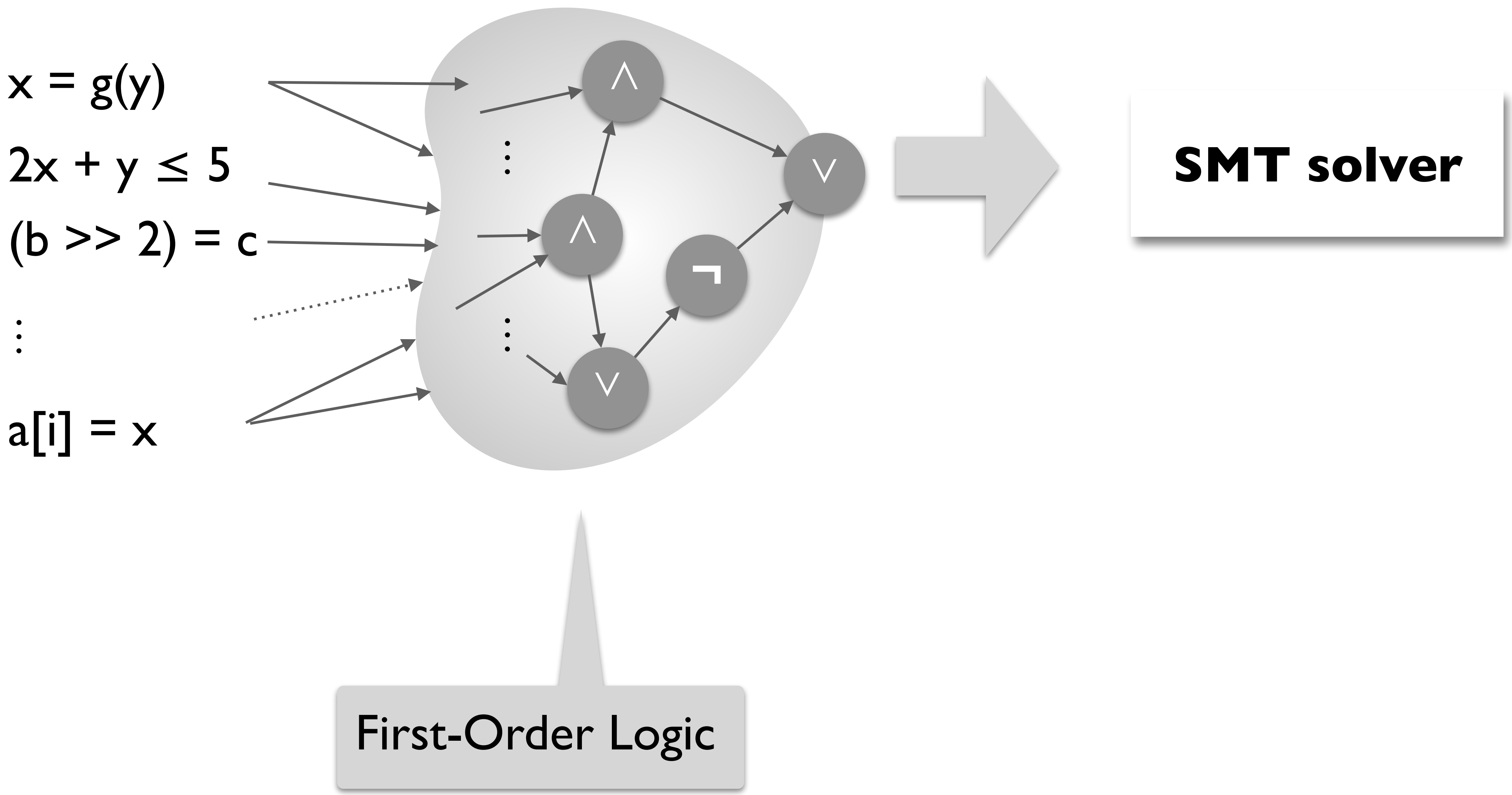
$(b >> 2) = c$

⋮

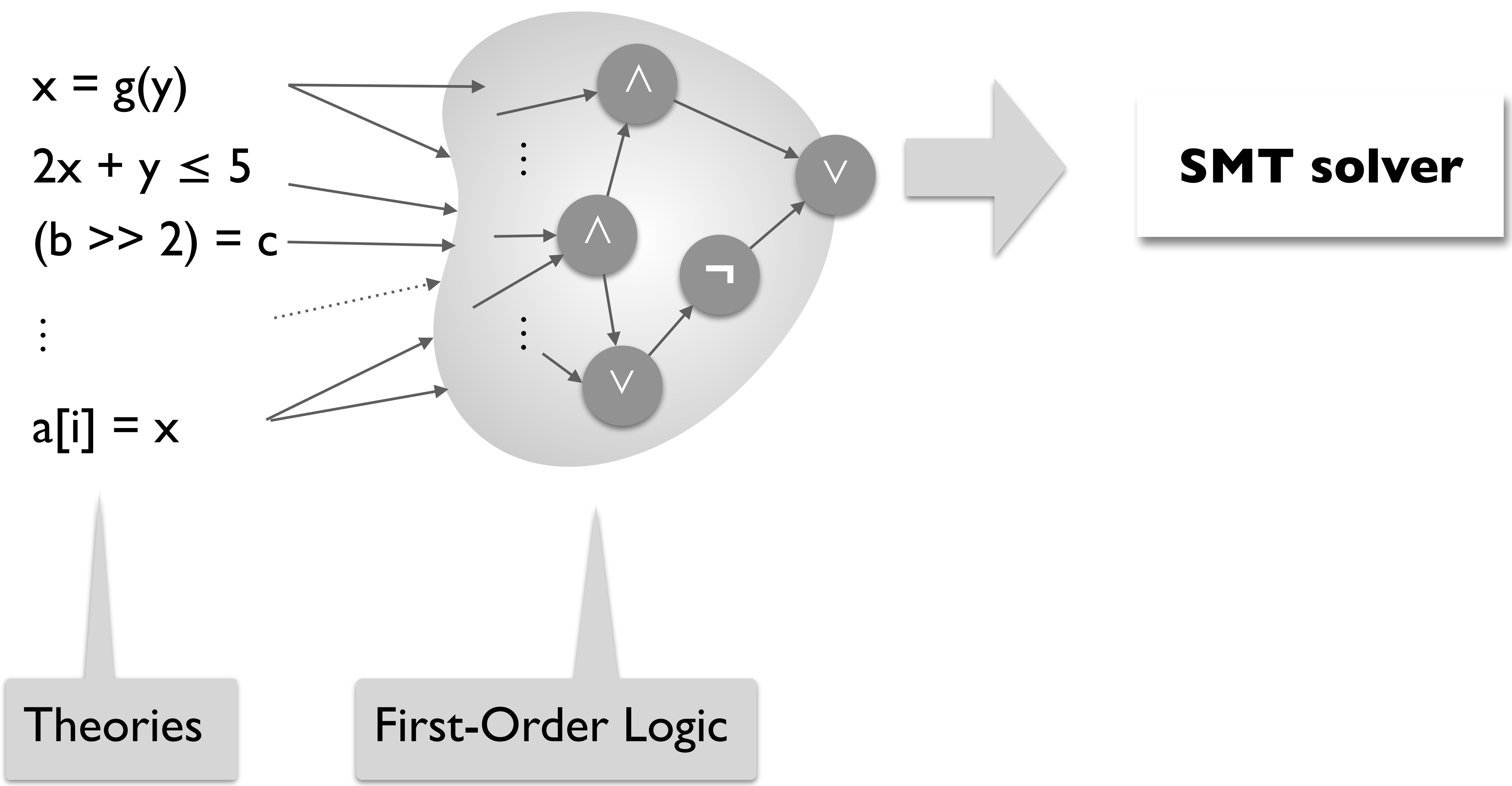
$a[i] = x$



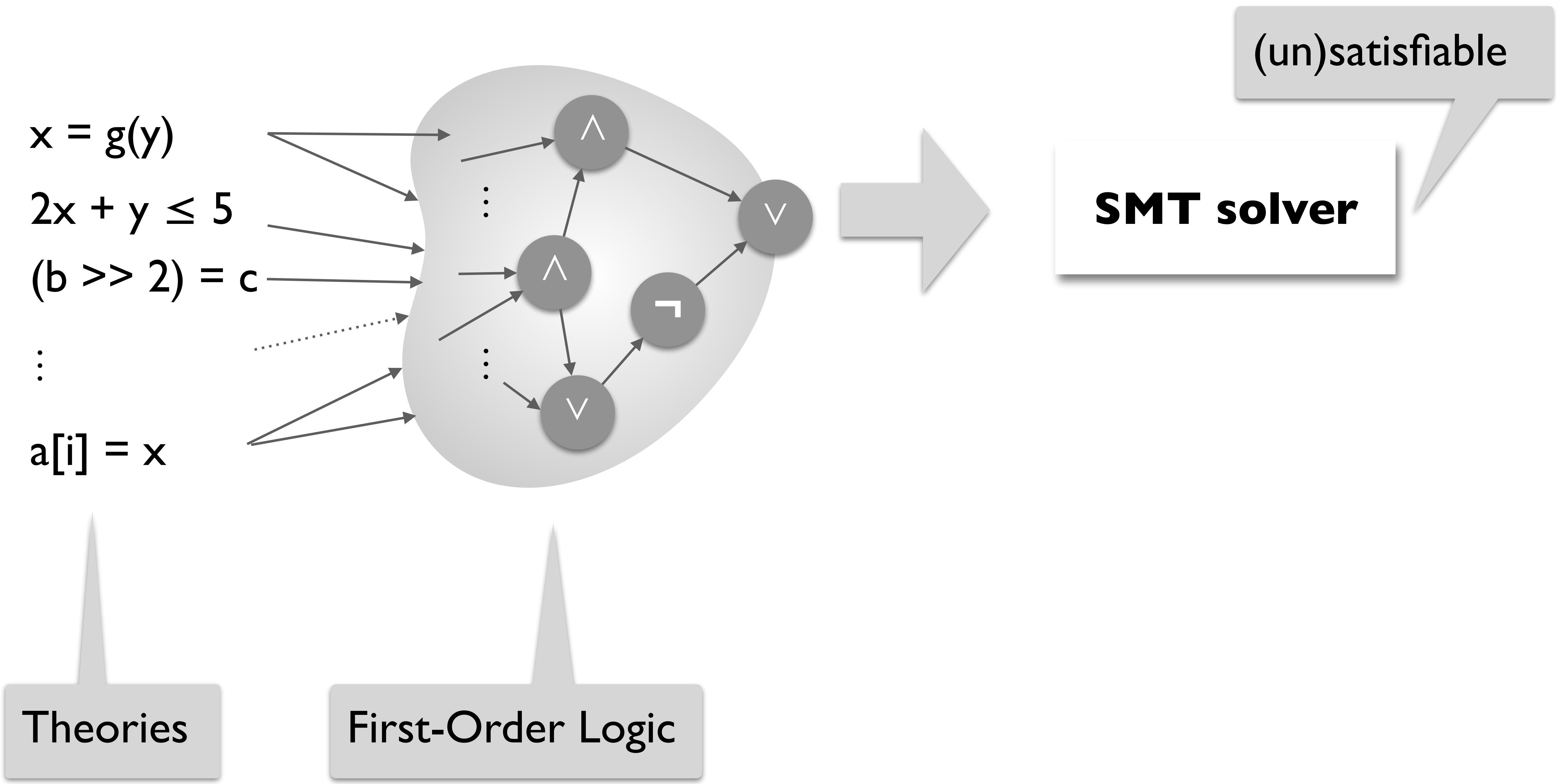
Satisfiability Modulo Theories (SMT)



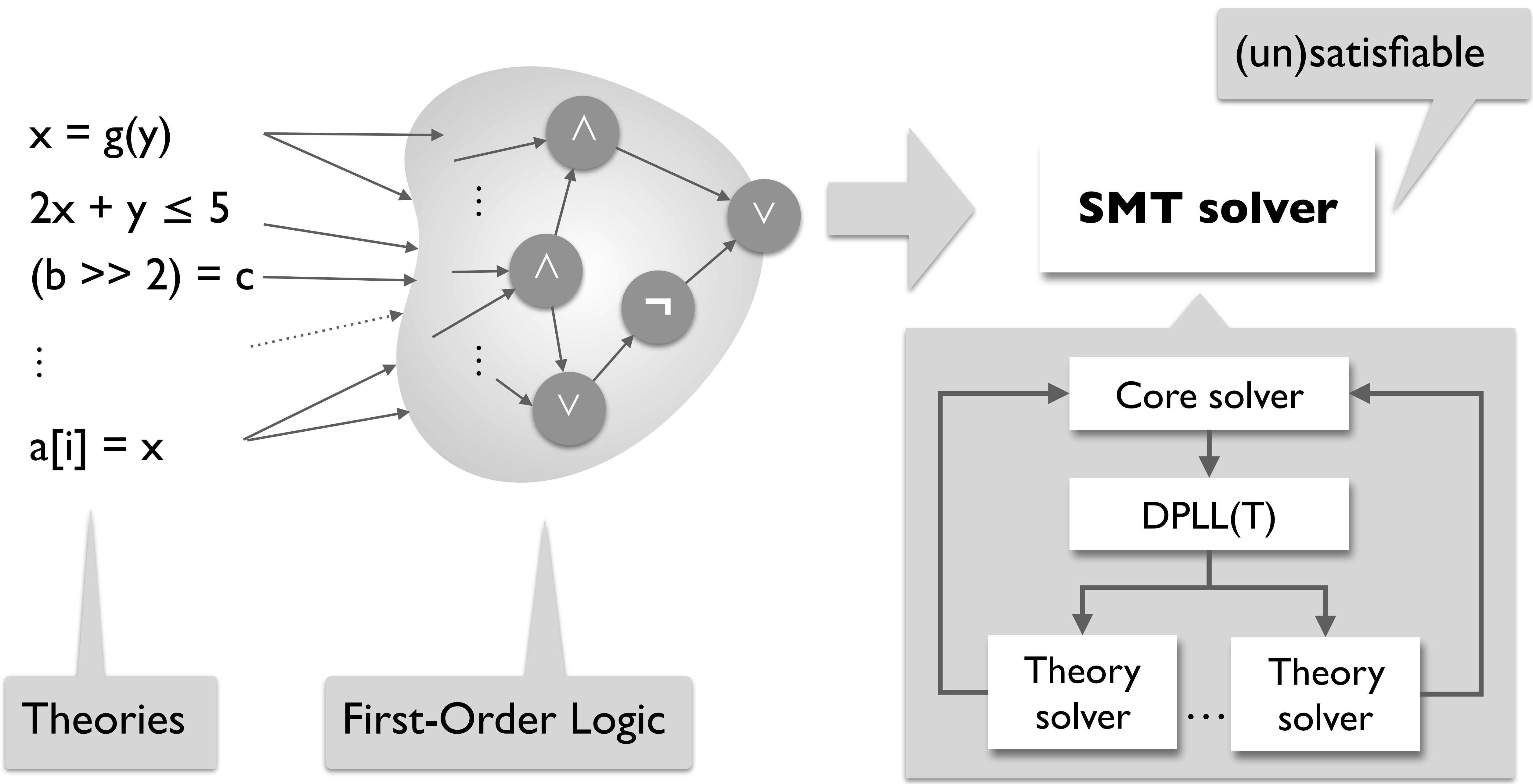
Satisfiability Modulo Theories (SMT)



Satisfiability Modulo Theories (SMT)



Satisfiability Modulo Theories (SMT)



Syntax of First-Order Logic (FOL)

Logical symbols

- Connectives: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$
- Parentheses: ()
- Quantifiers: $\forall$, $\exists$

Non-logical symbols

- Constants: x, y, z
- N-ary functions: f, g
- N-ary predicates: p, q
- Variables: u, v, w

Syntax of First-Order Logic (FOL)

Logical symbols

- Connectives: ¬, ∧, ∨, →, ↔
- Parentheses: ()
- ✗ Quantifiers: ∀, ∃

We will only consider the **quantifier-free** fragment of FOL.

Non-logical symbols

- Constants: x, y, z
- N-ary functions: f, g
- N-ary predicates: p, q
- Variables: u, v, w

Syntax of First-Order Logic (FOL)

Logical symbols

- Connectives: ¬, ∧, ∨, →, ↔
- Parentheses: ()
- ✗ Quantifiers: ∀, ∃

Non-logical symbols

- Constants: x, y, z
- N-ary functions: f, g
- N-ary predicates: p, q
- ✗ Variables: u, v, w

> We will only consider the **quantifier-free** fragment of FOL.

> In particular, we will consider quantifier-free **ground** formulas.

Syntax of quantifier-free ground FOL formulas

Logical symbols

- Connectives: ¬, ∧, ∨, →, ↔
- Parentheses: ()

Non-logical symbols

- Constants: x, y, z
- N-ary functions: f, g
- N-ary predicates: p, q

A **term** is a constant, or an n-ary function applied to n terms.

An **atom** is ⊤, ⊥, or an n-ary predicate applied to n terms.

A **literal** is an atom or its negation.

A (quantifier-free ground) **formula** is a literal or the application of logical connectives to formulas.

A quantifier-free ground FOL formula: example

Logical symbols

- Connectives: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
- Parentheses: ()

isPrime(x) $\rightarrow$ $\neg$ isInteger(sqrt(x))

Non-logical symbols

- Constants: x, y, z
- N-ary functions: f, g
- N-ary predicates: p, q

Semantics of FOL: first-order structures $\langle$U, I$\rangle$

Universe

Interpretation

Semantics of FOL: universe

Universe

- A non-empty set of values
- Finite or (un)countably infinite

Interpretation

Semantics of FOL: interpretation

Universe

- A non-empty set of values
- Finite or (un)countably infinite

Interpretation

- Maps a constant symbol c to an element of U: $I[c] \in U$
- Maps an n-ary function symbol f to a function $f_I : U^n \rightarrow U$
- Maps an n-ary predicate symbol p to an n-ary relation $p_I \subseteq U^n$

Semantics of FOL: inductive definition

Universe

- A non-empty set of values
- Finite or (un)countably infinite

Interpretation

- Maps a constant symbol c to an element of U: $I[c] \in U$
- Maps an n-ary function symbol f to a function $f_I : U^n \rightarrow U$
- Maps an n-ary predicate symbol p to an n-ary relation $p_I \subseteq U^n$

$I[f(t_1, \ldots, t_n)] = I[f](I[t_1], \ldots, I[t_n])$

$I[p(t_1, \ldots, t_n)] = (\langle I[t_1], \ldots, I[t_n] \rangle \in I[p])$

$\langle U, I \rangle \vDash \top$

$\langle U, I \rangle \nvDash \bot$

$\langle U, I \rangle \vDash p(t_1, \ldots, t_n)$ **iff** $I[p(t_1, \ldots, t_n)] = \text{true}$

$\langle U, I \rangle \vDash \neg F$ **iff** $\langle U, I \rangle \nvDash F$

…

Semantics of FOL: example

Universe

- A non-empty set of values
- Finite or (un)countably infinite

Interpretation

- Maps a constant symbol c to an element of U: $I[c] \in U$
- Maps an n-ary function symbol f to a function $f_I : U^n \to U$
- Maps an n-ary predicate symbol p to an n-ary relation $p_I \subseteq U^n$

U = {☀, ☁}

I[x] = ☀

I[y] = ☁

I[f] = {☀ ↦ ☁, ☁ ↦ ☀}

I[p] = {⟨☀,☀⟩, ⟨☀,☁⟩}

$\langle U, I \rangle \models p(f(y), f(f(x)))$?

Satisfiability and validity of FOL

F is **satisfiable** iff $M \vDash F$ for some structure $M = \langle U, I \rangle$.

F is **valid** iff $M \vDash F$ for all structures $M = \langle U, I \rangle$.

Duality of satisfiability and validity:

F is valid iff $\neg F$ is unsatisfiable.

First-order theories

Signature Σ_T

Set of T-models

First-order theories

Signature Σ_T

- Set of constant, predicate, and function symbols

Set of T-models

First-order theories

Signature Σ_T

- Set of constant, predicate, and function symbols

Set of T-models

- One or more (possibly infinitely many) models that fix the interpretation of the symbols in Σ_T
- Can also view a theory as a set of axioms over Σ_T (and T-models are the models of the theory axioms)

First-order theories

Signature Σ_T

- Set of constant, predicate, and function symbols

Set of T-models

- One or more (possibly infinitely many) models that fix the interpretation of the symbols in Σ_T
- Can also view a theory as a set of axioms over Σ_T (and T-models are the models of the theory axioms)

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

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

Common theories

Equality (and uninterpreted functions)

- $x = g(y)$

Fixed-width bitvectors

- (b >> 1) = c

Linear arithmetic (over R and Z)

- $2x + y \leq 5$

Arrays

- a[i] = x

Theory of equality with uninterpreted functions

Signature: a binary = predicate, plus all other symbols

- $\{=, x, y, z, \ldots, f, g, \ldots, p, q, \ldots\}$

Axioms

- $\forall x.\ x = x$
- $\forall x, y.\ x = y \rightarrow y = x$
- $\forall x, y, z.\ x = y \wedge y = z \rightarrow x = z$
- $\forall x_1, \ldots, x_n, y_1, \ldots, y_n.\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n) \rightarrow (f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$
- $\forall x_1, \ldots, x_n, y_1, \ldots, y_n.\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n) \rightarrow (p(x_1, \ldots, x_n) \leftrightarrow p(y_1, \ldots, y_n))$

Conjunctions of ground formulas modulo $T_=$ decidable in polynomial time

$T_=$ example: checking 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;
}
```

A formula that is unsatisfiable iff programs are equivalent:

$T_=$ example: checking 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;
}
```

A formula that is unsatisfiable iff programs are equivalent:

$(z_1 = y_0 \wedge y_1 = x_0 \wedge x_1 = z_1 \wedge r_1 = x_1 * x_1) \wedge$

$(r_2 = y_0 * y_0) \wedge$

$\neg(r_2 = r_1)$

Example from Sanjit Seshia

$T_=$ example: checking 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;
}
```

A formula that is unsatisfiable iff programs are equivalent:

$(z_1 = y_0 \land y_1 = x_0 \land x_1 = z_1 \land r_1 = x_1 * x_1) \land$

$(r_2 = y_0 * y_0) \land$

$\neg(r_2 = r_1)$

Using 32-bit integers, a SAT solver fails to return an answer in 5 min.

Example from Sanjit Seshia

$T_=$ example: checking 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;
}
```

A formula that is unsatisfiable iff programs are equivalent:

$(z_1 = y_0 \land y_1 = x_0 \land x_1 = z_1 \land r_1 = mul(x_1, x_1)) \land$

$(r_2 = mul(y_0, y_0)) \land$

$\neg(r_2 = r_1)$

Using $T_=$, an SMT solver proves unsatisfiability in a fraction of a second.

Example from Sanjit Seshia

$T_=$ example: checking program equivalence

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

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

Example from Sanjit Seshia

$T_=$ example: checking program equivalence

```
int fun1(int y) {
  int x;
  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?

$T_=$ example: checking program equivalence

```
int fun1(int y) {
  int x;
  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).

Example from Sanjit Seshia

Theory of fixed-width bitvectors

Signature

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

Satisfiability problem: NP-complete.

Theories of linear integer and real arithmetic

Signature

- {…, -1, 0, 1, …, -2, 2, …, +, -, =, ≤, x, y, z, …}
- Constants, integers (or reals), multiplication by an integer (or real) value, addition, subtraction, equality, greater-than.

Satisfiability problem:

- NP-complete for linear integer arithmetic (LIA).
- Polynomial time for linear real arithmetic (LRA).
- Polynomial time for difference logic (conjunctions of the form x - y ≤ c, where c is an integer constant).

LIA 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 LIA formula that is unsatisfiable iff this transformation is valid:

LIA 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 LIA formula that is unsatisfiable iff this transformation is valid:

$(i \geq 1) \wedge (i \leq 10) \wedge$

$(j + i = j)$

Polyhedral model

Theory of arrays

Signature

- {read, write, =, x, y, z, …}

Axioms

- $\forall i.\ read(write(a, i, v), i) = v$
- $\forall i, j.\ \neg(i = j) \rightarrow (read(write(a, i, v), j) = read(a, j))$
- $(\forall i.\ read(a, i) = read(b, i)) \rightarrow a = b$

Satisfiability problem: NP-complete.

Used in many software verification tools to model memory.

Summary

Today

- Introduction to SMT
- Quantifier-free FOL (syntax & semantics)
- Overview of common theories

Next lecture

- Survey of theory solvers