Automated Reasoning 2018

Lecture 23: Quantifiers in SMT solvers

Instructor: Ashutosh Gupta

IITB, India

Compile date: 2018-11-05

Topic 23.1

Quantified normal form

Normalization steps

A modern FOL refutation prover first applies the following transformations

- Rename apart : rename variables for each quantifier
- Prenex : bringing quantifiers to front
- Skolemization: remove existential quantifiers (only sat preserving)
- CNF transformation: turn the internal quantifier free part of the formula into CNF
- Syntactical removal of universal quantifiers: a CNF with free variables.

Rename apart

Definition 23.1

A formula F is renamed apart if no quantifier in F use a variable that is used by another quantifier or occurs as free variable in F.

Due to the theorems like the following, we can safely assume that every quantifier has different variable. If that is not the case then we can rename quantified variables apart.

Theorem 23.1

Let F is a $\mathbf{S}$-formulas and y does not occur in F.

$$\models \forall x.F \Leftrightarrow \forall y.F\{x \mapsto y\}$$

Exercise 23.1

Rename apart the following formulas

- $\neg(\exists x.\forall y R(x,y) \Rightarrow \forall y.\exists x R(x,y))$

Prenex form

Definition 23.2

A formula F is in prenex form if all the quantifiers of the formula occur as prefix of F. The quantifier-free suffix of F is called matrix of F.

Due to the following equivalences, we can always move quantifiers to the front

- $\neg(\exists x.F) \equiv \forall x.\neg F$
- $\neg(\forall x.F) \equiv \exists x.\neg F$
- $\forall x.F \wedge G \equiv \forall x.(F \wedge G)$
- $\exists x.F \wedge G \equiv \exists x.(F \wedge G)$
- $F \wedge \forall x.G \equiv \forall x.(F \wedge G)$
- $F \wedge \exists x.G \equiv \exists x.(F \wedge G)$
- $\forall x.F \vee G \equiv \forall x.(F \vee G)$
- $\exists x.F \vee G \equiv \exists x.(F \vee G)$
- $F \vee \forall x.G \equiv \forall x.(F \vee G)$
- $F \vee \exists x.G \equiv \exists x.(F \vee G)$
- $\forall x.F \Rightarrow G \equiv \exists x.(F \Rightarrow G)$
- $\exists x.F \Rightarrow G \equiv \forall x.(F \Rightarrow G)$
- $F \Rightarrow \forall x.G \equiv \forall x.(F \Rightarrow G)$
- $F \Rightarrow \exists x.G \equiv \exists x.(F \Rightarrow G)$

Exercise 23.2

Convert $\neg(\exists x.\forall y R(x,y) \Rightarrow \forall y.\exists x R(x,y))$ into prenex form

Skolemization

Theorem 23.2

Let F be a **S***-formula with $FV(F) = \{x, y_1, \ldots, y_n\}$. Let $G(A)$ be a* **S***-formula in which atom A occurs positively and $G(\exists x.F(x))$ is a sentence. Let $f/n \in$* **F** *such that $f \notin vars(F(x), G(A))$.*

$$\models G(\exists x.F(x)) \qquad \textit{iff} \qquad \models G(F(f(y_1, \ldots, y_n)))$$

Since all the quantifiers occur positively in prenex form, all $\exists$s can be removed using skolem functions.

Skolemization is applied from out to inside, i.e., remove outermost $\exists$ first.

Example 23.1

After skolemization on $\forall x, y.\ \exists z.\ x + y \leq z$, we obtain $\forall x, y.\ x + y \leq f(x, y)$.

Exercise 23.3

Skolemize $\exists x. \forall y \exists z \forall w.\ \neg(R(x, y) \Rightarrow R(w, z))$

FOL CNF

Consider the following skolemized prenex formula,

$$\forall x_1, \ldots, x_n.\ F.$$

Since F is quantifier free, we may convert F into CNF, preferably using Tseitin encoding(what is the quantifier over fresh booleans?) and obtain

$$\forall x_1, \ldots, x_n.\ C_1 \wedge \cdots \wedge C_k.$$

Since $\forall$ distributes over $\wedge$, we may obtain

$$(\forall x_1, \ldots, x_n.\ C_1) \wedge \cdots \wedge (\forall x_1, \ldots, x_n.\ C_k).$$

We may rename apart variables in each of the above clauses and obtain

$$(\forall x_{11}, \ldots, x_{1n}.\ C'_1) \wedge \cdots \wedge (\forall x_{k1}, \ldots, x_{kn}.\ C'_k).$$

Commentary: The last renaming step is not necessary. In a tool, the variables in each clause is considered different from the other clauses.

De Bruijn indices

Renaming apart is usually not a real operation, since tools may not have explicit names for quantified variables.

Each occurrence of quantified variable is represented by a natural number, which is the number of quantifiers that are in scope between the occurrence and its corresponding quantifier.

De Bruijn indices uniquely identifies its quantifier.

Example 23.2

- $\forall x.\ \exists z.\ x + y \leq z$ *is represented by* $\forall.\ \exists.\ ?2 + y \leq ?1$
- $\forall x.\ x \leq 3 \Rightarrow \exists z.\ x + y \leq z$ *is represented by* $\forall.\ ?1 \leq 1 \Rightarrow \exists.\ ?2 + y \leq ?1$

Exercise 23.4

Write the following formulas using De Bruijn indices

- $R(x, y)$
- $\forall x, y.\ \exists z.\ x + y \leq z$
- $\exists y \forall z.\ \exists x.\ x + y \leq z + 2$
- $\forall x.\ x \leq 3 \Rightarrow \exists x.\ x - y \leq 6$

Solving quantified formulas

We may have two kinds of clauses in input formula

- Quantifier-free or ground clauses
- Quantified clauses

Consider clause $\forall x.\ f(x) \leq g(3, x)$.

Let S be the set of substitutions, i.e., mappings from the quantified variables in the clause to the ground terms in the theory.

The clause represents infinite conjunctions $\bigwedge_{\sigma \in S} \underbrace{(f(x) \leq g(3, x))\sigma}_{\text{instantiations}}$.

Example 23.3

Let $\sigma = \{x \mapsto f(a)\}$.

$(f(x) \leq g(3, x))\sigma := f(f(a)) \leq g(3, f(a))$

The generic solving strategy for quantified formulas

1. Ground clauses are solved using an SMT solver.
2. Quantified clauses are instantiated with some strategy.
3. The generated ground clauses are added to the SMT solver
4. If SMT solver says unsatisfiable, return unsatisfiable.
5. If instantiations are reached the limit, return satisfiable.
6. goto 2.

Depending on the underlying theory/logic, a stopping criteria is used to limit instantiations

Example: quantified reasoning

Example 23.4

$$\underbrace{\forall x.f(g(x,c)) = a}_{\textit{Quantified clauses}} \wedge \underbrace{b = c \wedge g(c,b) = c \wedge f(b) \neq a}_{\textit{Ground clauses}}$$

We can instantiate the quantified clause by replacing any ground term at x.

Let us choose $x = b$, which introduces ground clause $f(g(b,c)) = a$.

The formula becomes unsatisfiable.

How can we find the right instantiation?

Substitutions

We will denote the instantiations by substitutions that are mappings from quantified variables to ground terms

Example 23.5

Consider quantified clause $\forall x, y.\ f(g(x, c)) = a \vee x \neq f(y)$.

Substitution $\sigma = \{x \mapsto f(a), y \mapsto a\}$ *denotes the following instantiation.*

$$f(g(f(a), c)) = a \vee f(a) \neq f(a)$$

E-matching

The set of relevant terms is maintained by the congruence classes of QF_EUF.

We need to match terms in the quantified formula to the ground terms such that we can produce relevant instantiations.

E-matching takes

- a pattern(non-ground term) p and
- a ground term t

as input and finds if there is σ such that $p\sigma = t$.

We use the learned σs to instantiate the quantified clauses.

Efficient E-matching for SMT Solvers Leonardo de Moura and Nikolaj Bjorner

E-matching as an equation

Concretely, *ematch* takes three parameters

- pattern p to match
- term t to match and
- the set of valid substitutions constructed so far

and returns the set of valid substitutions.

Initial call, $ematch(p, t, \{\emptyset\})$

Matching with term pattern:

$$ematch(f(p_1, ..., p_n), t, S) := \bigcup_{f(t_1,...,t_n) \in class(t)} ematch(p_n, t_n, .., ematch(p_1, t_1, S))$$

Commentary: Matched with every ground f-term that is congruent with t. Subterms are matched one after another and substitutions are extended.

E-matching as an equation II

Matching with a variable:

$$\begin{aligned} ematch(x, t, S) :=& \{\sigma[x \mapsto t] \mid \sigma \in S \wedge x \notin dom(\sigma)\} \cup \\ & \{\sigma \mid \sigma \in S \wedge x \in dom(\sigma) \wedge x\sigma \in class(t)\} \end{aligned}$$

Matching with a constant:

$$ematch(c, t, S) := \begin{cases} S & \text{if } c \in class(t) \\ \emptyset & \text{otherwise.} \end{cases}$$

Example: ematch

Example 23.6

Let us match pattern $p = f(x, g(x, c))$ *with ground term* $t = f(b, g(a, c))$.

Let $\{\{a, b\}, \{c\}, ...\}$ *be the current congruence classes.*

$ematch(p, t, \{\emptyset\})$:
Does $f(b, g(a, c))$ *have an congruent term with top symbol* f *? yes, itself!.*

$ematch(g(x, c), g(a, c), ematch(x, b, \{\emptyset\}))$:
Match subterms and aggregate the results!

$ematch(x, b, \{\emptyset\}) := \{\{x \mapsto b\}\}$

$ematch(g(x, c), g(a, c), \{\{x \mapsto b\}\}) := \{\{x \mapsto b\}\}$:
needed a check if map of x *is congruent to* a.

Efficient pattern matching

We need to apply same sequence of operations to match with a pattern.

We translate matching with a given pattern into code.

We can use the code to repeatedly match with many ground terms.

Example 23.7

Again consider pattern $f(x, g(x, c))$. We need the following actions to match.

pc_0 :	$f(x, g(x, c))$	$\rightsquigarrow$	Is top f? If yes, check subterms.
pc_1 :	$g(x, c)$	$\rightsquigarrow$	Is top g? If yes, check subterms.
pc_2 :	x	$\rightsquigarrow$	Map x
pc_3 :	c	$\rightsquigarrow$	Is congruent to c?
pc_4 :	x	$\rightsquigarrow$	Is congruent with existing map for x?

Matching code with two stacks

We need two stacks for the matching.

- *tstack*: stack for storing the matched subterms so far
- *bstack*: Stack for alternate choices; popped each time when a match fails and backtracks.

Matching as code

We can translate matching as a sequence of the following instructions.

- `init` : initiating the term matching by copying the ground term to *tstack*
- `bind` : match head and populate *tstack* with subterms
  - Due to congruence there can be multiple ground term matches
  - the choices are pushed in the *bstack*
- `compare`: if a variable is repeated check congruence with the past matching
- `check` : check if congruent with a constant
- `yield` : report the σ.

Matching as code : handling multiple choices

- `backtrack:` executed upon failure of the conditions in the instructions
  - pop *bstack* and execute according to available choices
  - There can be two kinds of choices

- `choose-app`: called by backtrack, if we need to replay `bind` with another congruent ground term (first kind of choice)

- `choose`: choose a sub-pattern to match (introduces the second kind of choice in the stack)
  not yet explained!!

Example: matching with code

Example 23.8

Again consider pattern $f(x, g(x, c))$. We need the following actions to match.

positions in *tstack*

```
pc0 :  init(0,pc1)
pc1 :  bind(f,0,[1-2],pc2)
pc2 :  bind(g,2,[3-4],pc3)
pc3 :  check(c,4,pc4)
pc4 :  compare(3,1,pc5)
pc5 :  yield([1],backtrack)
```

position in *tstack* where map of x is stored

Exercise 23.5

Write code to match with pattern $f(x, g(x, a), h(y), b)$.

Example: running the matching code

Example 23.9

Let us match ground term $f(h(a), g(b, c))$ *with* $f(x, g(x, c))$.

$pc0$: `init(0,pc1)`	$tstack[0] := f(h(a), g(b, c))$
$pc1$: `bind(f,0,[1-2],pc2)`	$tstack[1] := h(a)$ $tstack[2] := g(b, c)$
$pc2$: `bind(g,2,[3-4],pc3)`	$tstack[3] := b$ $tstack[4] := c$
$pc3$: `check(c,4,pc4)`	Since $tstack[4] == c$, ✓
$pc4$: `compare(3,1,pc5)`	Since $tstack[1] \neq tstack[3]$, ✗. `backtrack`
$pc5$: `yield([1],backtrack)`	

Since nothing was pushed in *bstack*, `backtrack` terminates with no match.

Example: running the matching code

Example 23.10

Let us match ground term $f(h(a), g(h(a), c))$ *with* $f(x, g(x, c))$.

$pc0$: `init(0,pc1)`	$tstack[0] := f(h(a), g(h(a), c))$
$pc1$: `bind(f,0,[1-2],pc2)`	$tstack[1] := h(a)$ $tstack[2] := g(h(a), c)$
$pc2$: `bind(g,2,[3-4],pc3)`	$tstack[3] := h(a)$ $tstack[4] := c$
$pc3$: `check(c,4,pc4)`	Since $tstack[4] == c$, ✓
$pc4$: `compare(3,1,pc5)`	Since $tstack[1] == tstack[3]$, ✓
$pc5$: `yield([1],backtrack)`	return $\sigma = \{x \mapsto h(a)\}$

Code tree: matching with multiple patterns via `choose`

- We can match with multiple patterns (not simultaneous) that have common prefix of code.
- At the point of divergence, we have instruction `choose` that pushes the available choices on *bstack* and calls `backtrack`.

Example: matching code for multiple patterns

Example 23.11

Again consider patterns $\{f(x, g(x, c)), f(x, g(y, c))\}$. *We need the following actions to match to any one of the patterns.*

$pc0:$ `init(0,pc1)`
$pc1:$ `bind(f,0,[1-2],pc2)`
$pc2:$ `bind(g,2,[3-4],pc3)`
$pc3:$ `check(c,4,pc4)`
$pc4:$ `choose([pc5,pc7])`
$pc5:$ `compare(3,1,pc6)`
$pc6:$ `yield([1],backtrack)`
$pc7:$ `yield([1,3],backtrack)`

Exercise 23.6

Write code to match the following patterns

- $\{g(f(x), f(x)), g(f(g(x, c)), y)\}$
- $\{f(x, g(a, y)), f(x, g(x, y)), f(h(x, y), b), f(h(x, g(x, y)), b)\}$

Incremental e-matching

Incomplete and non-terminating method

- E-matching is incomplete

Example 23.12

Consider $\forall x.\ f(x) > 0 \wedge \forall x.\ f(x) < 0$.
Since there are no ground terms, no instantiations.

- E-matching may lead to non-terminating sequence of instantiations.

Example 23.13

Consider $\forall x.\ f(x) = g(f(x)) \wedge \forall x.\ g(x) = f(g(x))$
Since no ground terms, no instantiations.

The complete version of E-matching is called superposition calculus. There are solver based on those, not covered in this course.

Eager vs. lazy instantiations

- A solver may eagerly introduce instantiations using E-matching as we learn more equivalences over ground terms.
  - Sometimes assisted by additional patterns given in the input
- The solvers may also choose lazy instantiations, i.g., on demand.
  - Next we will see an example of such strategy

Model-based quantifier instantiations

Algorithm 23.1: MBQI(F: quantified clauses, G: ground clauses)

while $m \models G$ **do**
  if *there is* $\forall x.\ C \in F$ *such that* $m' \models \neg C^m(x)$ **then**
    find ground term t such that $t^{m'} = x^{m'}$;
    $G := G \wedge C\{x \mapsto t\}$
  else
    return *sat*
return *unsat*

- There can be many t that match the requirement
  - Finding t appears to be an art?
- If the number of potential ts is finite, the above is very effective
  - If no function symbols in the theory (effectively propositional (EPR))
  - Array property fragment (seen in theory of arrays!)
  - Some classes of synthesis problem

Topic 23.2

Problems

Translating to prenex form

Exercise 23.7

Let us suppose a formula is given in De Bruijn indicies. Give an algorithm to convert the formula into prenex form.

End of Lecture 23