# CS395T: Automated Logical Reasoning

## Simplification of SMT Formulas

Işıl Dillig



## Overview

- Last lecture: Learned about DPLL($\mathcal{T}$) framework to solve SMT formulas
- However, in some applications, solving formula is not enough; also need to find compact representation
- Already saw one example of this idea in propositional logic: binary decision diagrams (BDDs)
- This lecture: How to simplify SMT formulas



## Static Program Analysis and SMT

- Heavy user of SMT formulas: static analysis and verification systems
- Static analysis proves properties about programs by analyzing source code (i.e., does not execute program)
- Many static analysis techniques use SMT formulas to symbolically represent program states
- Deciding whether a program proeerty holds is achieved by checking satisfiability/validity of SMT formulas
- SMT formulas generated by static analysis are very large but typically extremely redundant $\Rightarrow$ scalability problems



## Motivating Example

```
enum op_type ADD=0, SUBTRACT=1, MULTIPLY=2, DIV=3;

int perform_op(op_type op, int x, int y) {
    int res;
    if(op == ADD) res = x+y;
    else if(op == SUBTRACT) res = x-y;
    else if(op == MULTIPLY) res = x*y;
     else if(op == DIV)  assert(y!=0); res = x/y;
    else res = UNDEFINED;
    return res;
 }
```

Condition for Success

$$op = 0 \lor (op \neq 0 \land op = 1) \lor (op \neq 0 \land op \neq 1 \land op \neq 2) \lor$$
$$(op \neq 0 \land op \neq 1 \land op \neq 2 \land op = 3 \land y \neq 0) \lor$$
$$(op \neq 0 \land op \neq 1 \land op \neq 2 \land op \neq 3)$$



## Take-Away Message

- If we automatically generate constraints from source code, resulting formulas are huge, but very redundant
- To allow analyses to run on large software, necessary to keep size of formulas under control
- Thus, want to find compact representation of SMT formulas called simplified form
- Simplified form of formula $F$ should be equivalent to $F$ and should not contain redundancies
- Furthermore, unlike BDDs, simplified form should not cause blow-up in formula size (in fact, should never be larger!)



## Simplified Form

- Goal: To give algorithm to convert each SMT formula $F$ to a simplified form $F'$ with following guarantees:
  1. $F'$ logically equivalent to $F$ (i.e., $F' \Leftrightarrow F$)
  2. $F'$ not redundant
  3. size of $F' \leq$ size of $F$
- Our algorithm will deal with qff formulas over any decidable first-order theory and that are in NNF
- First, need to be precise about what we mean by redundancy and size of formula

## Atomic Formula vs. Leaf

- An atomic formula is a formula without $\land, \lor$
- Each syntactic occurence of an atomic formula is called leaf
- How many leaves does this formula have? 3

$$\neg f(x) = 1 \lor (\neg f(x) = 1 \land x + y \leq 1)$$

- Leaves:

$$\underbrace{\neg f(x) = 1}_{L_1} \lor (\underbrace{\neg f(x) = 1}_{L_2} \land \underbrace{x + y \leq 1}_{L_3}))$$

- Size of formula $\phi$ is number of leaves $\phi$ contains



## $\phi^+(L)$ and $\phi^-(L)$

- Define over and underapproximations of formula $\phi$ w.r.t leaf $L$
- $\phi^+(L)$ obtained by replacing $L$ with $\top$ in $\phi$ and constant folding resulting formula (i.e., removing $\top, \bot$)
- Consider again formula $\phi$:

$$\phi: \underbrace{\neg f(x) = 1}_{L_1} \lor (\underbrace{\neg f(x) = 1}_{L_2} \land \underbrace{x + y \leq 1}_{L_3})$$

- What is $\phi^+(L_1)$? $\top$
- $\phi^-(L)$ obtained by replacing $L$ with $\bot$ in $\phi$ and constant folding resulting formula (i.e., removing $\top, \bot$)
- What is $\phi^-(L_2)$? $\neg f(x) = 1$



## Properties of $\phi^+$ and $\phi^-$

- What is the relationsip between the size of $\phi^+(L)$ and $\phi$? $size(\phi^+(L)) < size(\phi)$
- Similarly, $size(\phi^-(L)) < size(\phi)$
- $\phi^+(L)$ overapproximates $\phi$ because $\phi \Rightarrow \phi^+(L)$
- Similarly, $\phi^-(L)$ underapproximates $\phi$ because $\phi^-(L) \Rightarrow \phi$



## Redundancy of Leaves

- If $\phi^+(L) \Rightarrow \phi$, then leaf $L$ is called non-constraining
- Thus, if $L$ is non-constraining, we have $\phi^+(L) \Leftrightarrow \phi$
- In this case, $L$ does not "constrain" the formula – can replace it with $\top$ to get equivalent formula
- If $\phi \Rightarrow \phi^-(L)$, then leaf $L$ is called non-relaxing
- Hence, if $L$ is non-relaxing, we have $\phi^-(L) \Leftrightarrow \phi$
- A leaf $L$ is redundant if it is either non-constraining or non-relaxing
- Such a leaf is redundant because can be replaced with $\top$ or $\bot$ to obtain smaller, but equivalent formula



## Example

- Consider again formula $\phi$:

$$\phi: \underbrace{\neg f(x) = 1}_{L_1} \lor (\underbrace{\neg f(x) = 1}_{L_2} \land \underbrace{x + y \leq 1}_{L_3})$$

- Is $L_1$ redundant? no
- Is $L_2$ redundant? yes because non-relaxing
- Is $L_3$ redundant? yes, both non-constraining and non-relaxing



## Simplified Form

- Fact: If no leaf in $\phi$ is redundant, can't obtain smaller equivalent formula by replacing any subset of leaves by $\top, \bot$
- Thus, a formula with no redundant leaves is said to be in simplified form
- Goal: Given formula $\phi$, we want to compute simplified form of $\phi$ by removing all redundant leaves in $\phi$

- Resulting simplified form equivalent to original formula but smaller and has no redundancy

## Properties of Simplified Form

- A formula in simplified form is satisfiable iff it is not syntactically false.
- Proof: Suppose this is not true (i.e., formula unsat, but simplified form not false).
- Now consider replacing every leaf by $\bot$. Resulting formula: $\bot$
- Since formula is unsat, resulting formula $\bot$ equivalent to original formula
- Thus, formula could not have been in simplied form.



## Properties of Simplified Form, cont

- A formula in simplified form is valid iff it is syntactically true.
- Thus, if formulas are kept in simplified form, deciding satisfiability and validity just a syntactic check
- Recall: A representation is called canonical if two equivalent formulas have same representation
- Is simplified form a canonical representation? No
- Formulas $a \land (b \lor c)$ and $(a \land b) \lor (a \land c)$ are equivalent and both in simplified form, but not syntactically identical
- Thus, if we keep formulas in simplified form, checking equivalence is not a syntactic test



## Simple Algorithm to Compute Simplified Forms

- Definition of simplified form suggests very simple algorithm:
  1. Pick any leaf $L$ in formula $\phi$
  2. Compute $\phi^+(L)$ by replacing $L$ with $\top$
  3. Test if $\phi^+(L) \Rightarrow \phi$ If so, $\phi := \phi[\top/L]$
  4. Otherwise, compute $\phi^-(L)$
  5. Test if $\phi \Rightarrow \phi^-(L)$ If so, $\phi := \phi[\bot/L]$
  6. Repeat until no leaf can be replaced



## Discussion of Simple Algorithm

- Algorithm requires checking $\phi^+(L) \Rightarrow \phi$ and $\phi^-(L) \Rightarrow \phi$
- What is the size of formula $\phi^+(L) \Rightarrow \phi$? twice as large as $\phi$
- Thus, algorithm requires repeatedly checking validity of formulas twice as large as original formula
- But actuallly we can do much better!
- Idea: Can determine if leaf is redundant by querying validity of formula no larger than $\phi$
- Key concept: critical constraint



## Critical Constraint

- For each leaf $L$, compute critical constraint $C(L)$
- Critical constraint has following properties:
  1. $C(L)$ is never larger than original formula
  2. $L$ is non-constraining iff $C(L) \Rightarrow L$
  3. $L$ is non-relaxing iff $C(L) \Rightarrow \neg L$

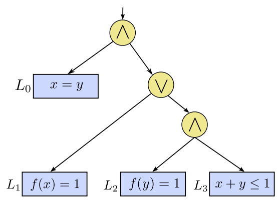


## Computing Critical Constraint

- To compute critical constraint for each leaf, (conceptually) represent formula as tree
- For instance, consider formula:

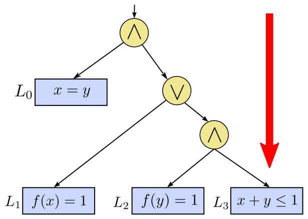
$$x = y \land (f(x) = 1 \lor (f(y) = 1 \land x + y \leq 1))$$

- Represent formula as tree:



## Computing Critical Constraint, cont

- Compute critical constraint for each tree node
- Do this top-down
- Start with root node
- Recursively compute critical constraint for each node using critical constraint for parent
- Base case: Initialize critical constraint of root to *true*






## Computing Critical Constraint, cont

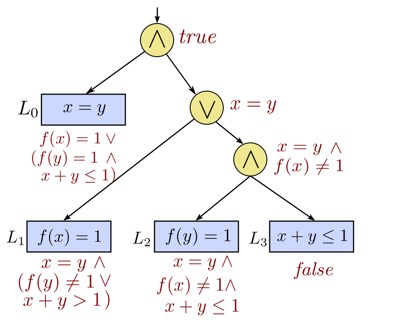
- Inductive case: Let $N$ be any non-root node.
- $N$ has parent $P$ with critical constraint $C(P)$
- $N$ has sibling $S$ with formula rooted at $S$ being $F_S$
- There are two cases to consider:
  1. If $P$ is an $\wedge$ node, then:
  $$C(N) = C(P) \wedge F_S$$
  2. If $P$ is an $\vee$ node, then:
  $$C(N) = C(P) \wedge \neg F_S$$



## Critical Constraint Example

- Consider again the formula

$$x = y \wedge (f(x) = 1 \vee (f(y) = 1 \wedge x + y \leq 1))$$



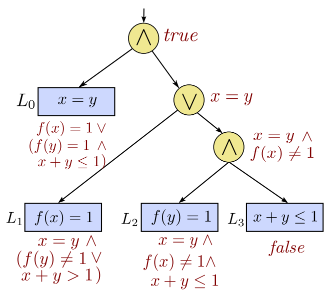



## Using Critical Constraint to Check Redundancy

- Recall: Can use critical constraint to check redundancy of leaf
- Leaf $L$ is non-constraining iff $C(L) \Rightarrow L$
- Leaf $L$ is non-relaxing iff $C(L) \Rightarrow \neg L$
- Thus, if $C(L) \Rightarrow L$, we get smaller, equivalent formula when we replace $L$ with boolean constant $\top$
- If $C(L) \Rightarrow \neg L$, we get smaller, equivalent formula when we replace $L$ with boolean constant $\perp$



## Example




- Does $C(L_0)$ imply $L_0$ or $\neg L_0$? no, so $L_0$ not redundant
- Does $C(L_1)$ imply $L_1$ or $\neg L_1$? no, so $L_1$ not redundant
- Does $C(L_2)$ imply $L_2$ or $\neg L_2$? implies $\neg L_2$, so $L_2$ non-relaxing
- Does $C(L_3)$ imply $L_3$ or $\neg L_3$? implies both, so $L_3$ non-constraining and non-relaxing



## Putting it All Together

- We want an algorithm to convert any formula $\phi$ in NNF to simplified form
- To do this, represent $\phi$ as tree and formulate auxiliary algorithm simplify(N, C)
- First arg. of simplify is subformula represented by tree node $N$
- Second argument $C$ is critical constraint of $N$
- The output of simplify(N, C) is a new tree representing simplified form of subformula rooted at $N$

## Putting It All Together, cont.

- If we have such an auxiliary algorithm simplify(N, C), how do we compute simplified form of $\phi$?
- Represent $\phi$ as tree with root $R$ and call simplify(R, true)
- Suppose this yields new tree rooted at $R'$.
- Simplified form of $\phi$ is simply $R'$ represented as formula
- Thus, if we have auxiliary algorithm simplify(N,C), this immediately gives way to simplify any formula $\phi$ in NNF



## Full Algorithm

```
/*
 * Recursive algorithm to compute simplified form.
 * N: current subformula, C: critical constraint of N
 */
```

simplify(N, C)

- Base case: If N is a leaf:
  - If $C \Rightarrow N$ return true /* Non-constraining */
  - If $C \Rightarrow \neg N$ return false /* Non-relaxing */
  - Otherwise, return N



## Full Algorithm, cont

simplify(N, C)

- Inductive case 1: If N is an $\wedge$ node with children $N_1, N_2$:
  - $N_1' = \text{simplify}(N_1, C \wedge N_2)$
  - $N_2' = \text{simplify}(N_2, C \wedge N_1')$
  - $N_1 := N_1', N_2 := N_2'$; repeat until $N_1' = N_1$ and $N_2' = N_2$
  - If $N_1'$ or $N_2'$ is false, return false
  - If $N_1'$ and $N_2'$ is true, return true
  - Else if $N_1'$ (resp. $N_2'$) is true, return $N_2'$ (resp. $N_1'$)
  - Else return new subtree with root $\wedge$ and children $N_1'$ and $N_2'$



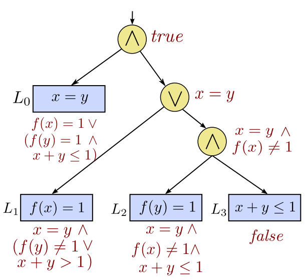
## Full Algorithm, cont

simplify(N, C)

- Inductive case 2: If N is an $\vee$ node with children $N_1, N_2$:
  - $N_1' = \text{simplify}(N_1, C \wedge \neg N_2)$
  - $N_2' = \text{simplify}(N_2, C \wedge \neg N_1')$
  - $N_1 := N_1', N_2 := N_2'$; repeat until $N_1' = N_1$ and $N_2' = N_2$
  - If $N_1'$ or $N_2'$ is true, return true
  - If $N_1'$ and $N_2'$ is false, return false
  - Else if $N_1'$ (resp. $N_2'$) is false, return $N_2'$ (resp. $N_1'$)
  - Else return new subtree with root $\vee$ and children $N_1'$ and $N_2'$



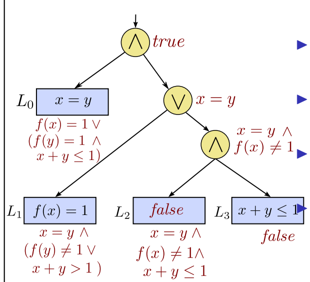
## Example




- Simplify children of topmost $\wedge$
- $L_0$ leaf, but stays the same
- To simplify $\vee$ node, need to simplify children
- $L_1$ leaf, but stays unchanged
- To simplify bottom $\wedge$ node, need to simplify $L_2, L_3$
- For $L_2$, $C(L_2) \Rightarrow \neg L_2$, thus replace with: false



## Example, cont




- Now, since child of $\vee$ node changed, re-simplify $L_1$
- New critical constraint: $x = y$
- Does $L_1$ change? no
- Result of simplify $\vee$ node: $f(x) = 1$
- Now simplify $L_0$ again, new critical constraint: $f(x) = 1$
- Does $L_0$ change? no
- Simplified form: $x = y \wedge f(x) = 1$

## Discussion of Algorithm

- In simplify algorithm, we resimplify children of connectives if any of the siblings change. Why is this necessary?
- Because critical constraint changes, so it might expose new simplification opportunities
- Example: $\underbrace{x \neq 1}_{L_1} \land \underbrace{(\underbrace{x \leq 0}_{L_2} \lor \underbrace{x > 2}_{L_3} \lor \underbrace{x = 1}_{L_4})}_{N}$
- Critical constraint for $L_1$: $x \leq 0 \lor x > 2 \lor x = 1$
- Does it imply $L_1$ or $\neg L_1$? no
- So, initially can't eliminate $L_1$



## Discussion of Algorithm, cont.

$$\underbrace{x \neq 1}_{L_1} \land \underbrace{(x \leq 0 \lor x > 2 \lor x = 1)}_{N}$$

- Critical constraint for $N$: $x \neq 1$
- Result of simplifying $N$ is $x \leq 0 \lor x > 2$
- So, if we wouldn't resimplify $L_1$, result would be

$$x \neq 1 \land (x \leq 0 \lor x > 2)$$

- Is this in simplified form? no
- If we resimplify $L_1$, $C(L_1) \Rightarrow L_1$, thus replace with true
- Actual simplified form: $x \leq 0 \lor x > 2$



## Discussion of Algorithm, cont.

- As example illustrates, need to resimplify subformulas as long as siblings change
- Otherwise, resulting formula might not be in simplified form
- To compute simplified form, algorithm makes at most $2n^2$ validity queries
- However, these validity queries are not independent of each other, so we can optimize



## Optimization

- Specifically, all formulas whose validity are queried have same set of leaves
- How can we use this to our advantage?
- Recall: When solving SMT formulas in DPLL($\mathcal{T}$) framework, we learn theory conflict clauses
- Theory conflict clauses are valid modulo $\mathcal{T}$ and prevent wrong assignments to boolean structure
- Since our formulas have same set of leaves, a theory conflict clause we learned during previous validity query will be useful for next query!



## Optimization, cont

- Thus, if we reuse theory conflict clauses, solving SMT formula should become as fast as solving SAT formula
- In practice, this optimization makes a huge difference!
- Using this optimization, overhead of simplification over solving observed to be sub-linear (logarithmic)
- Simplifying is more expensive than just solving, but in practice, not quadratically worse



## Benefit of Simplification

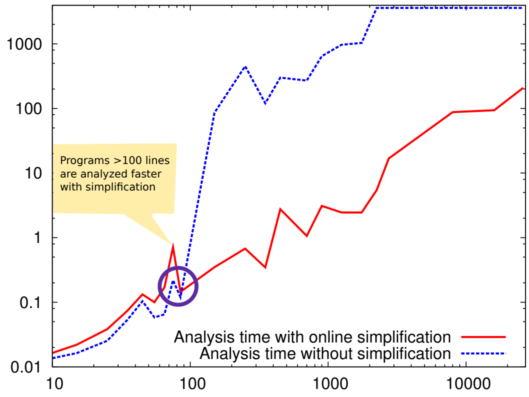
- If simplifying formula more expensive than solving, why bother simplifying?
- Recall: Motivation for simplification is applications that incrementally build formulas from existing formulas, such as program analysis
- In these kinds of applications, redundancies accumulate as formula is built from existing formulas
- Goal of simplification: Prevent accumulation of redundancies so that formulas at every step are managable in size
- Thus, to evaluate benefit of simplification, need to compare running times of applications that only solve vs. simplify

## Benefit of Simplification in Static Analysis

- We evaluated benefit of simplification in the context of static analysis
- Used a static analysis tool, Compass, that incrementally builds formulas from existing formulas
- Ran Compass on 811 benchmarks, totaling 173,000 LOC to verify memory safety
- Compared running time of analysis runs that use simplification with runs that do not
- In former case, every time analysis queries satisfiability of formula, we simplify formula and give back this simplified form
- In latter case, just give yes/no answer



## Impact on Running Time of Static Analysis Tool






## Summary

- In applications that incrementally build formulas, simplification might be very beneficial
- Looked at one application of simplified forms: static analysis
- Haven't applied this idea to other domains, but could have other interesting applications
- How does simplified form compare with BDDs?
- Guaranteed not to cause increase in formula size (often desirable)
- But it's not a canonical representation, so equivalence checking is not syntactic