**Overview**

- **Last lecture**: Learned about DPLL($T$) 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 property holds is achieved by checking satisfiability/validity of SMT formulas
- SMT formulas generated by static analysis are very large but typically extremely redundant ⇒ scalability problems

**Motivating Example**

```c
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**

$$\begin{align*}
    op &= 0 \lor (op \neq 0 \land op = 1) \lor (op \neq 0 \land op \neq 1 \land op \neq 2) \lor \\
    & (op \neq 0 \land op \neq 1 \land op \neq 2 \land op = 3 \land y \neq 0) \lor \\
    & (op \neq 0 \land op \neq 1 \land op \neq 2 \land op \neq 3)
\end{align*}$$

**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., $F' \iff F$)
  2. $F'$ not redundant
  3. size of $F'$ ≤ 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 occurrence 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:
  \[\neg f(x) = 1 \lor (\neg f(x) = 1 \land x + y \leq 1)\]
- Size of formula $\phi$ is number of leaves $\phi$ contains

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

- What is the relationship between the size of $\phi^+(L)$ and $\phi$?
  \[\text{size}(\phi^+(L)) < \text{size}(\phi)\]
- Similarly, $\text{size}(\phi^-(L)) < \text{size}(\phi)$
- $\phi^+(L)$ 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 $\phi$:
  \[\phi : \neg f(x) = 1 \lor (\neg f(x) = 1 \land x + y \leq 1)\]
- Is $L_1$ redundant? no
- Is $L_2$ redundant? yes because non-relaxing
- Is $L_3$ redundant? yes, both non-constraining and non-relaxing
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 simplified 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 actually 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:
  
  $$x = y \land (f(x) = 1 \lor (f(y) = 1 \lor x + y \leq 1))$$

  - Represent formula as tree:
    
    ![Tree Diagram](https://via.placeholder.com/150)

  - $L_1$ represents $x = y$
  - $L_2$ represents $(f(x) = 1 \lor (f(y) = 1 \lor x + y \leq 1))$
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

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 $\land$ node, then:
     \[ C(N) = C(P) \land F_S \]
  2. If $P$ is an $\lor$ node, then:
     \[ C(N) = C(P) \land \neg F_S \]

Critical Constraint Example

Consider again the formula $x = y \land (f(x) = 1 \lor (f(y) = 1 \land x + y \leq 1))$

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 $\bot$

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 \( \text{simplify}(R, \text{true}) \)
- Suppose this yields new tree rooted at \( R' \).
- Simplified form of \( \phi \) is simply \( R' \) represented as formula
- Thus, if we have auxiliary algorithm \( \text{simplify}(\text{N}, \text{C}) \), this immediately gives way to simplify any formula \( \phi \) in NNF

Example

- Simplify children of topmost \( \land \)
- \( L_0 \) leaf, but stays the same
- To simplify \( \lor \) node, need to simplify children
- \( L_1 \) leaf, but stays unchanged
- To simplify bottom \( \land \) 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 \( \lor \) node changed, re-simplify \( L_1 \)
- New critical constraint: \( x = y \)
- Does \( L_1 \) change? no
- Result of simplify \( \lor \) node: \( f(x) = 1 \)

Full Algorithm

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

\[ \text{simplify}(N, C) \]

- Inductive case 1: If \( N \) is an \( \land \) node with children \( N_1, N_2 \):
  - \( N'_1 = \text{simplify}(N_1, C \land N_2) \)
  - \( N'_2 = \text{simplify}(N_2, C \land N'_1) \)
  - \( N_1 := N'_1; N_2 := N'_2; \) repeat until \( N'_1 = N_1 \) and \( N'_2 = N_2 \)
  - If \( N'_1 \) or \( N'_2 \) is false, return false
  - Else if \( N'_1 \) and \( N'_2 \) is true, return true
  - Else return new subtree with root \( \land \) and children \( N'_1 \) and \( N'_2 \)

- Inductive case 2: If \( N \) is an \( \lor \) node with children \( N_1, N_2 \):
  - \( N'_1 = \text{simplify}(N_1, C \land \neg N_2) \)
  - \( N'_2 = \text{simplify}(N_2, C \land \neg N'_1) \)
  - \( N_1 := N'_1; N_2 := N'_2; \) repeat until \( N'_1 = N_1 \) and \( N'_2 = N_2 \)
  - If \( N'_1 \) or \( N'_2 \) is true, return true
  - Else if \( N'_1 \) and \( N'_2 \) is false, return false
  - Else return new subtree with root \( \lor \) and children \( N'_1 \) and \( N'_2 \)

- Otherwise, return \( N \)
Discussion of Algorithm, cont.

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

- 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 $L_1$

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($T$) framework, we learn theory conflict clauses

- Theory conflict clauses are valid modulo $T$ 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!

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 manageable 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

![Impact on Running Time of Static Analysis Tool](image)

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.