

## Simplifying Boolean Formulas

### CS357 Lecture 9

## The Problem

- Analysis and verification systems tend to generate gigantic formulas
- And this is bad for solvers

## 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;  
}
```

## 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;  
}  
(op = 0)
```

## 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;  
}  
(op = 0) ∨ (op ≠ 0 ∧ op = 1) ∨
```

## 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;  
}  
(op = 0) ∨ (op ≠ 0 ∧ op = 1) ∨ (op ≠ 0 ∧ op ≠ 1 ∧ op = 2)
```



## 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;
}
(op = 0) ∨ (op ≠ 0 ∧ op = 1) ∨ (op ≠ 0 ∧ op ≠ 1 ∧ op = 2) ∨
(op ≠ 0 ∧ op ≠ 1 ∧ op ≠ 2 ∧ op = 3 ∧ y ≠ 0)
```

Profs. Aiken & Dill CS357 Lecture 9

7

## 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;
}
(op = 0) ∨ (op ≠ 0 ∧ op = 1) ∨ (op ≠ 0 ∧ op ≠ 1 ∧ op = 2) ∨
(op ≠ 0 ∧ op ≠ 1 ∧ op ≠ 2 ∧ op = 3 ∧ y ≠ 0) ∨ (op ≠ 0 ∧ op ≠ 1 ∧ op ≠ 2 ∧ op ≠ 3)
```

Profs. Aiken & Dill CS357 Lecture 9

8

## 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;
}

```

$op \neq 3 \vee y \neq 0$

Profs. Aiken & Dill CS357 Lecture 9

9

## Simplification

- Given a (boolean) formula  $F$ , find a formula  $F'$ 
  - $F'$  is equivalent to  $F$
  - $F'$  is no larger than  $F$
  - $F'$  has no *redundant* subexpressions
- Then  $F'$  is in *simplified* form if it has no redundant subparts

Profs. Aiken & Dill CS357 Lecture 9

10

## Preliminaries

- Assume formulas are in *negation normal form*
  - *NNF*
- Push negations all the way in using DeMorgan's laws

Profs. Aiken & Dill CS357 Lecture 9

11

## Literals

- A *literal* is a variable or a negated variable
- Literals are the *leaves* of a tree of ands and ors
  - For a formula in NNF

Profs. Aiken & Dill CS357 Lecture 9

12



## Redundancy

---

- A leaf  $L$  is nonconstraining in formula  $F$  if  $F \equiv F[\text{true}/L]$
- A leaf  $L$  is nonrelaxing in  $F$  if  $F \equiv F[\text{false}/L]$
- $L$  is redundant if it is nonconstraining or nonrelaxing

## Examples

---

$$(a \vee b) \vee a$$

$$(a \wedge b) \wedge a$$

## Simplification

---

- Given a boolean formula  $F$ , find a formula  $F'$ 
  - $F'$  is equivalent to  $F$
  - $F'$  is no larger than  $F$
  - $F'$  has no **redundant leaves**
- Then  $F'$  is in *simplified form*

## Useful Lemma

---

- If a formula is in simplified form, we cannot obtain a smaller, equivalent formula by replacing *any* subset of the leaves by true or false.
- Proof is by induction
  - See the paper

## Corollary

---

- Assume  $F$  is in simplified form
- Then  $F$  is
  - Satisfiable if it is not syntactically *false*
  - Valid if it is syntactically *true*

## Another Fact

---

- Simplified forms are not canonical
- $(a \wedge b) \vee c$
- $(a \vee c) \wedge (b \vee c)$



## Compare with BDDs

---

- Size
- Canonical?
- Valid/satisfiable formulas

## Easy Algorithm

---

- Algorithm
  - Pick any leaf  $L$  of  $F$
  - Check  $F = F[\text{true}/L]$  and/or  $F = F[\text{false}/L]$
  - Repeat until no leaf can be replaced.
- Problem:
  - Requires testing validity of formulas twice as large as the original

## Critical Constraints

---

- For each leaf  $L$  compute a *critical constraint*  $C$  s.t.
- $L$  is non-constraining iff  $C \Rightarrow L$
- $L$  is non-relaxing iff  $C \Rightarrow \neg L$

## Computing Critical Constraints

---

- Work top down
  - Compute a critical constraint for each subexpression of formula  $F$
  - Initially, critical constraint for  $F$  is  $C(F) = \text{true}$

$\text{true} \Rightarrow F$  then  $F$  is valid  
 $\text{true} \Rightarrow \neg F$  then  $F$  is unsatisfiable

## Cases

---

- $A \wedge B$ 
  - $C(A) = C(A \wedge B) \wedge B$
  - $C(B) = C(A \wedge B) \wedge A$
- $A \vee B$ 
  - $C(A) = C(A \vee B) \wedge \neg B$
  - $C(B) = C(A \vee B) \wedge \neg A$

## Proof

---

- Assume  $C(B) \Rightarrow B$  but  $F[\text{true}/B] \wedge \neg F$  is SAT
- Proof sketch:
  - Consider any model of  $F[\text{true}/B] \wedge \neg F$
  - Must use the difference between the two formulas



## Simplify

```
Simplify(N, a □ □
if N is a leaf
  if a ⇒ N then return true
  if a ⇒ ¬N then return false
  return N
if N = A ∧ B
  A' = A, B' = B
  repeat
    A = A', B = B'
    A' = Simplify(A, N ∧ B)
    B' = Simplify(B, N ∧ A')
  until A == A', B == B'
  return A' ∧ B'
if N = A ∨ B
  ...
```

Profs. Aiken & Dill CS357 Lecture 9

25

## Why Must We Iterate?

- Consider

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

Profs. Aiken & Dill CS357 Lecture 9

26

## Complexity

- If there are N literals
  - may need to consider all N to eliminate 1
  - each step requires 2 validity queries
- $O(N^2)$  validity queries

Profs. Aiken & Dill CS357 Lecture 9

27

## An Interlude: SMT Solvers

- Satisfiability Modulo Theories
  - SAT + additional theories
- Example: integer arithmetic
  - $(x \neq 0) \wedge ((x > 1) \vee (x < -1) \vee (x = 0))$
- Boolean combination of theory-specific predicates

Profs. Aiken & Dill CS357 Lecture 9

28

## Why SMT Solvers?

- Option 1
  - Reduce everything to SAT
  - E.g., bit blasting
- Option 2
  - Use theory-specific decision procedures for the appropriate fragments
  - Often much more compact representations and more efficient solving than translation to SAT
- Popular approach in the '00's

Profs. Aiken & Dill CS357 Lecture 9

29

## SMT Solving Algorithm

- Separate SAT and theory-specific parts
- Replace theory-specific predicates by variables
  - If a predicate is repeated, use the same variable
- Example
  - $(x \neq 0) \wedge ((x > 1) \vee (x < -1) \vee (x = 0))$
  - $\hat{e} \wedge (\beta \vee \gamma \vee \delta)$

Profs. Aiken & Dill CS357 Lecture 9

30



## SMT Algorithm

- Solve the boolean formula
  - If boolean formula is UNSAT, so is original
  - But the converse does not hold!

## Example

$$(x \neq 0) \wedge ((x > 1) \vee (x < -1) \vee (x = 0))$$
$$a \wedge (\beta \vee \gamma \vee \delta)$$

- Sat solver returns
$$a \square \text{true}, \delta \square \text{true}$$
- Test  $(x \neq 0) \wedge (x = 0)$  is satisfiable in the theory
  - Theory decision procedure says no

## Example (Continued)

- Add a conflict clause to the boolean formula

$$(x \neq 0) \wedge ((x > 1) \vee (x < -1) \vee (x = 0))$$
$$a \wedge (\beta \vee \gamma \vee \delta) \wedge (\neg a \vee \neg \delta)$$

- Now the SAT solver picks a different solution
  - E.g.,  $a = \text{true}, \delta \square \text{true}$
  - Theory confirms that  $(x \neq 0) \wedge (x > 1)$  is satisfiable
  - Note theory only needs to handle conjunctions

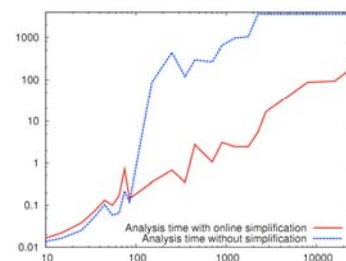
## Simplification for SMT

- Same algorithm for simplification
- With an optimization
  - Remember and reuse all conflict clauses learned from every query
- Why?
  - All the queries are over the same set of literals
  - Reduces number of calls to the solver for the theory very dramatically

## Simplification in Program Analysis

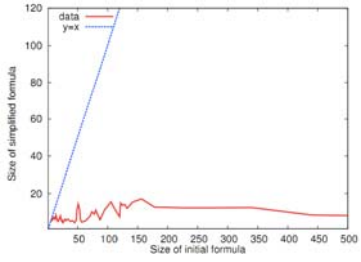
- Maintain formulas in simplified form
- Whenever a formula is constructed, immediately simplify it

## Results: Compass



### Before & After Formula Size (Compass)

---

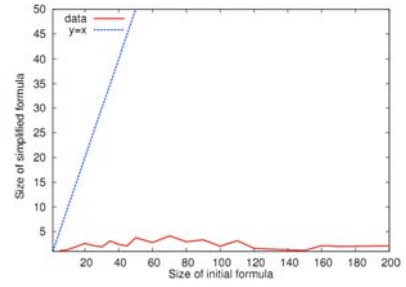


Profs. Aiken & Dill CS357 Lecture 9

37

### Before & After Formula Size (Saturn)

---



Profs. Aiken & Dill CS357 Lecture 9

38

### Relationship to Constrain

---

Profs. Aiken & Dill CS357 Lecture 9

39

