

Small Formulas for Large Programs: On-line Constraint Simplification In Scalable Static Analysis

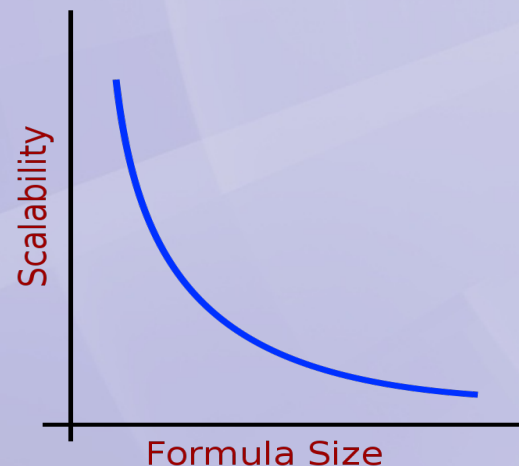
Isil Dillig, Thomas Dillig, Alex Aiken



Stanford University

Scalability and Formula Size

- Many program analysis techniques represent **program states** as **SAT or SMT formulas**.
 - Queries about program \Rightarrow Satisfiability and validity queries to the constraint solver
- **Scalability** of these techniques is often very sensitive to **formula size**.



Techniques to Limit Formula Size

- Many different techniques to control formula size:
 - **Basic Predicate abstraction**
 - Formulas are over a finite, fixed set of predicates.
 - **Predicate abstraction with CEGAR** SLAM, BLAST
 - Iteratively discover “relevant” predicates.
 - **Property simulation** ESP
 - Track only those path conditions where property differs along arms of the branch.
 - **and many others...**

Our Approach

- Afore-mentioned approaches control formula size by *restricting the set of facts* that are tracked by the analysis.
- We attack the problem from a different angle:

Instead of aggressively restricting which facts to track a-priori, our focus is to guarantee non-redundancy of formulas via **constraint simplification**.

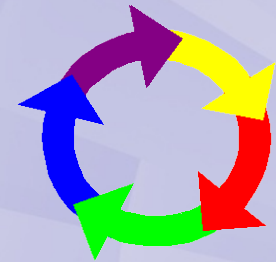
Goal #1: Non-redundancy

- Given formula F , we want to find formula F' such that:
 - F' is *equivalent* to F
 - F' has no redundant subparts
 - F' is no larger than F
- If F is a formula characterizing program property P , then predicates *irrelevant to* P are not mentioned in F' .
 - No need to guess in advance which facts/predicates may be needed later to prove P .

Such a formula is
in *simplified form*

Goal #2: On-line

- Simplification should be *on-line*:
 - Formulas are **continuously simplified** and **reused** throughout the analysis.
 - Important because program analyses construct new formulas from existing formulas.
 - Simplification prevents incremental build-up of massive, redundant formulas.
 - In our system, formulas are simplified at **every** satisfiability or validity query.



An 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;  
}
```

*Performs op
on x and y*

Suppose we are interested in the condition under which `perform_op` successfully returns, i.e., **does not abort**.

An 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;
}
```

Branch	Success Condition
$op = 0$	<i>true</i>
$op \neq 0 \wedge op = 1$	<i>true</i>
$op \neq 0 \wedge op \neq 1 \wedge op = 2$	<i>true</i>
$op \neq 0 \wedge op \neq 1 \wedge op \neq 2 \wedge op = 3$	$y \neq 0$
$op \neq 0 \wedge op \neq 1 \wedge op \neq 2 \wedge op \neq 3$	<i>true</i>

Program analysis tool examines every branch and computes condition under which **each branch** succeeds.

An 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;
}
```

$$\begin{aligned} op = 0 \vee (op \neq 0 \wedge op = 1) \vee (op \neq 0 \wedge op \neq 1 \wedge op = 2) \vee \\ (op \neq 0 \wedge op \neq 1 \wedge op \neq 2 \wedge op = 3 \wedge y \neq 0) \vee \\ (op \neq 0 \wedge op \neq 1 \wedge op \neq 2 \wedge op \neq 3) \end{aligned}$$

An 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;
}
```

In simplified form:

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

*No irrelevant predicates,
much more concise*

Now that this example has convinced you simplification is a good idea, **how do we actually do it?**



Leaves of a Formula

- We consider quantifier-free formulas using the boolean connectives **AND**, **OR**, and **NOT** over *any decidable theory*.
- We assume formulas are in **NNF**.
- A formula that does not contain conjunction or disjunction is an **atomic formula**.
- Each *syntactic occurrence* of an atomic formula is a **leaf**.
- Example: $\neg f(x) = 1 \vee (\neg f(x) = 1 \wedge x + y \leq 1)$

3 distinct leaves

Redundant Leaves

- A leaf L is *non-constraining* in formula F if replacing L with **true** in F yields an equivalent formula.
- L is *non-relaxing* in F if replacing L with **false** is equivalent to F .
- L is *redundant* if it is non-constraining **or** non-relaxing.

$$\underbrace{x = y}_{L_0} \wedge \left(\underbrace{f(x) = 1}_{L_1} \vee \underbrace{f(y) = 1}_{L_2} \wedge \underbrace{x + y \leq 1}_{L_3} \right)$$

Non-relaxing because formula is equivalent when it is replaced by false.

Both non-constraining and non-relaxing.

Simplified Form

- A formula F is in *simplified form* if no leaf in F is redundant.

Important Fact:

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.

This means that we only need to check one leaf at a time for redundancy, not subsets of leaves.

Properties of Simplified Forms

- A formula in simplified form is **satisfiable** if and only if it is *not syntactically false*, and it is **valid** iff it is *syntactically true*.
- Simplified forms are **preserved under negation**.
- Simplified forms are **not unique**.
 - Consider formula $x = 1 \vee x = 2 \vee (1 \leq x \wedge x \leq 2)$ in **linear integer arithmetic**. Both $x = 1 \vee x = 2$ and $1 \leq x \wedge x \leq 2$ are simplified forms.

*Equivalence of simplified forms **cannot** be determined syntactically.*

Algorithm

- Definition of simplified form suggests **trivial algorithm**:
 - Pick any leaf, **replace** it by true/false.
 - **Check** if formula is equivalent.
 - **Repeat** until no leaf can be replaced.
- Requires repeatedly checking satisfiability of formulas ***twice as large*** as the original formula.
- But we **can do better** than this naive algorithm!



Critical Constraint



Idea:

Compute a constraint C , called **critical constraint**, for each leaf L such that:

- (i) L is non-constraining iff $C \Rightarrow L$
- (ii) L is non-relaxing iff $C \Rightarrow \neg L$

Intuitively, C describes the condition under which L determines whether an assignment satisfies the formula.

*C is **no larger than** original formula F , so redundancy is checked using formulas at most as large as F .*

Constructing Critical Constraint

- Assume we represent formula as a **tree**.
- The critical constraint for **root** is **true**.
- Let **N** be any non-root node with parent **P** and i'th sibling **S(i)**.

- If **P** is an **AND** connective:

$$C(N) = C(P) \wedge \bigwedge_i S(i)$$

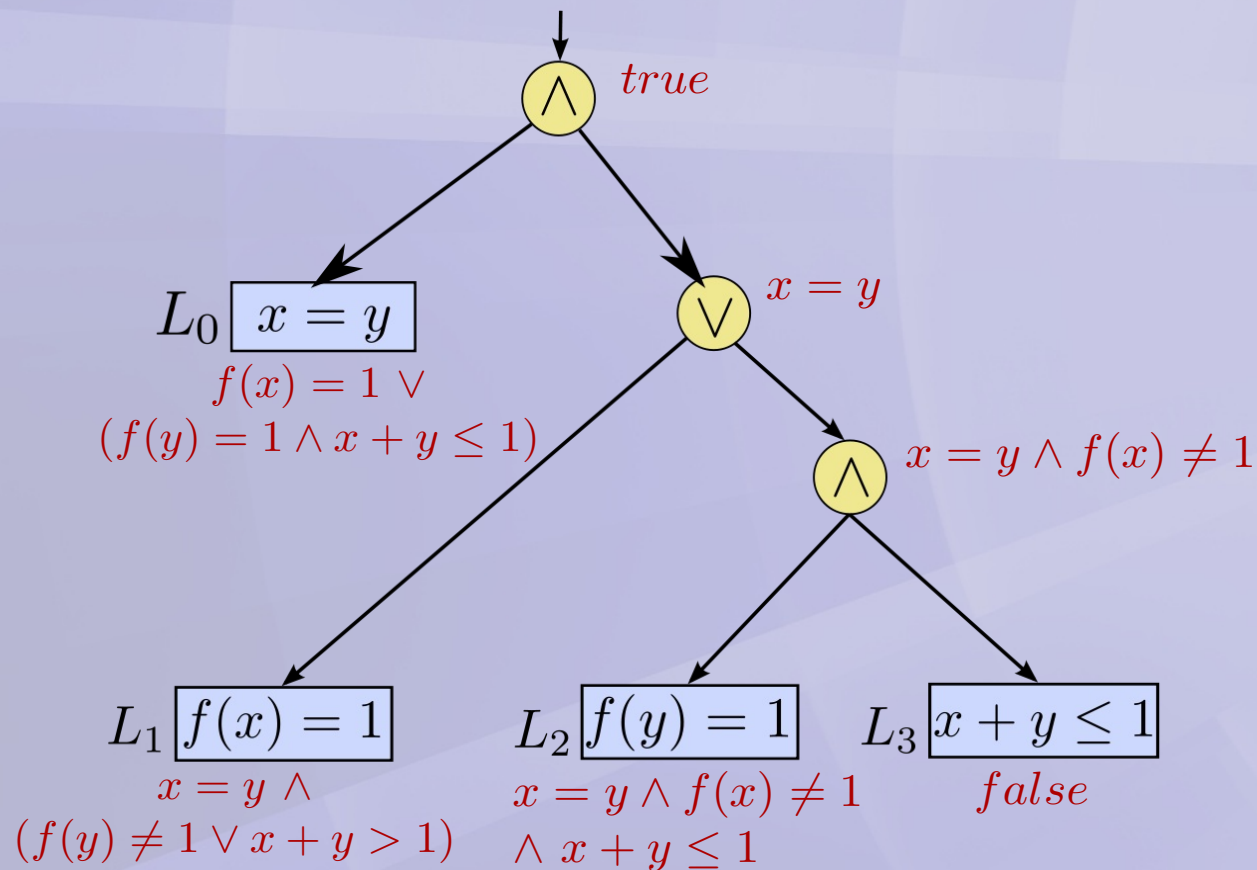
- If **P** is an **OR** connective:

$$C(N) = C(P) \wedge \bigwedge_i \neg S(i)$$

Example

- Consider again the formula:

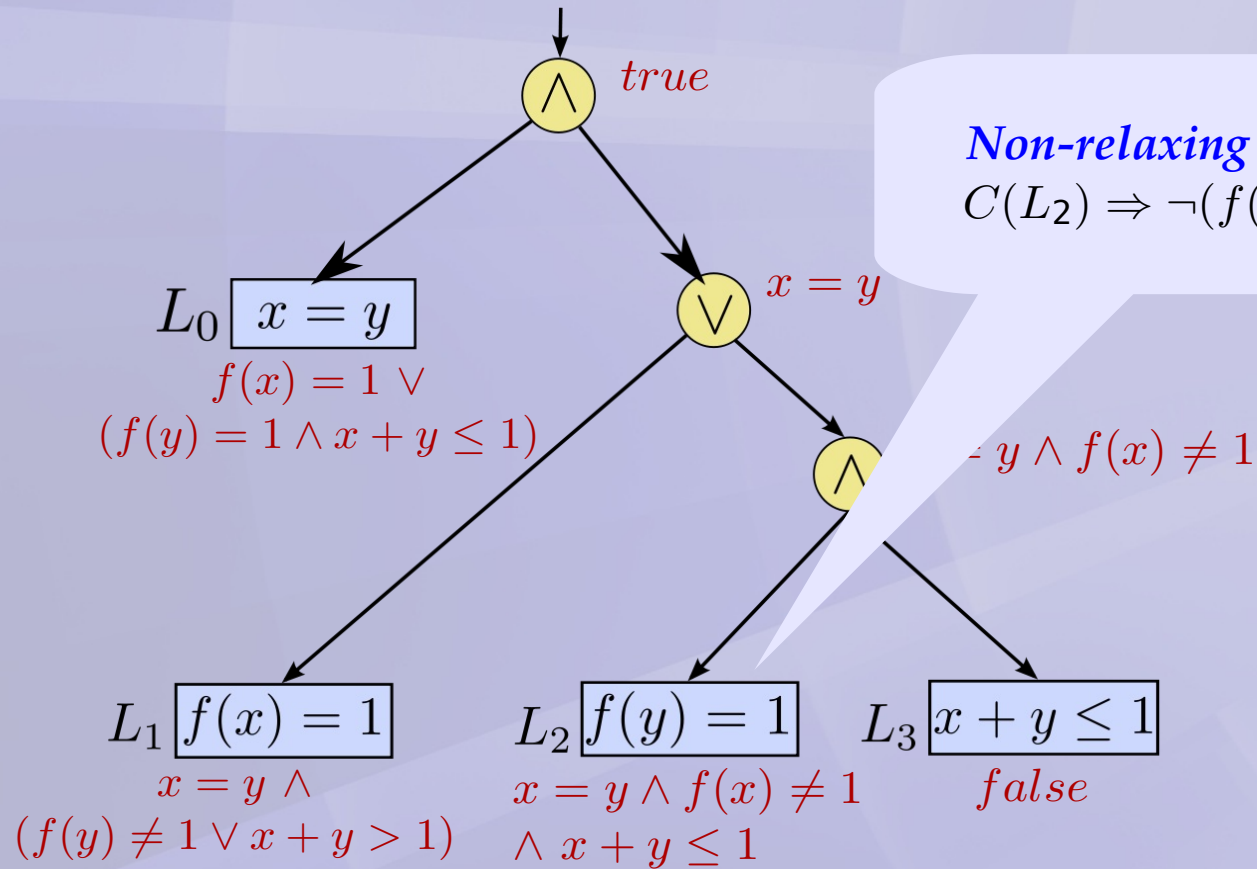
$$x = y \wedge (f(x) = 1 \vee (f(y) = 1 \wedge x + y \leq 1))$$



Example

- Consider again the formula:

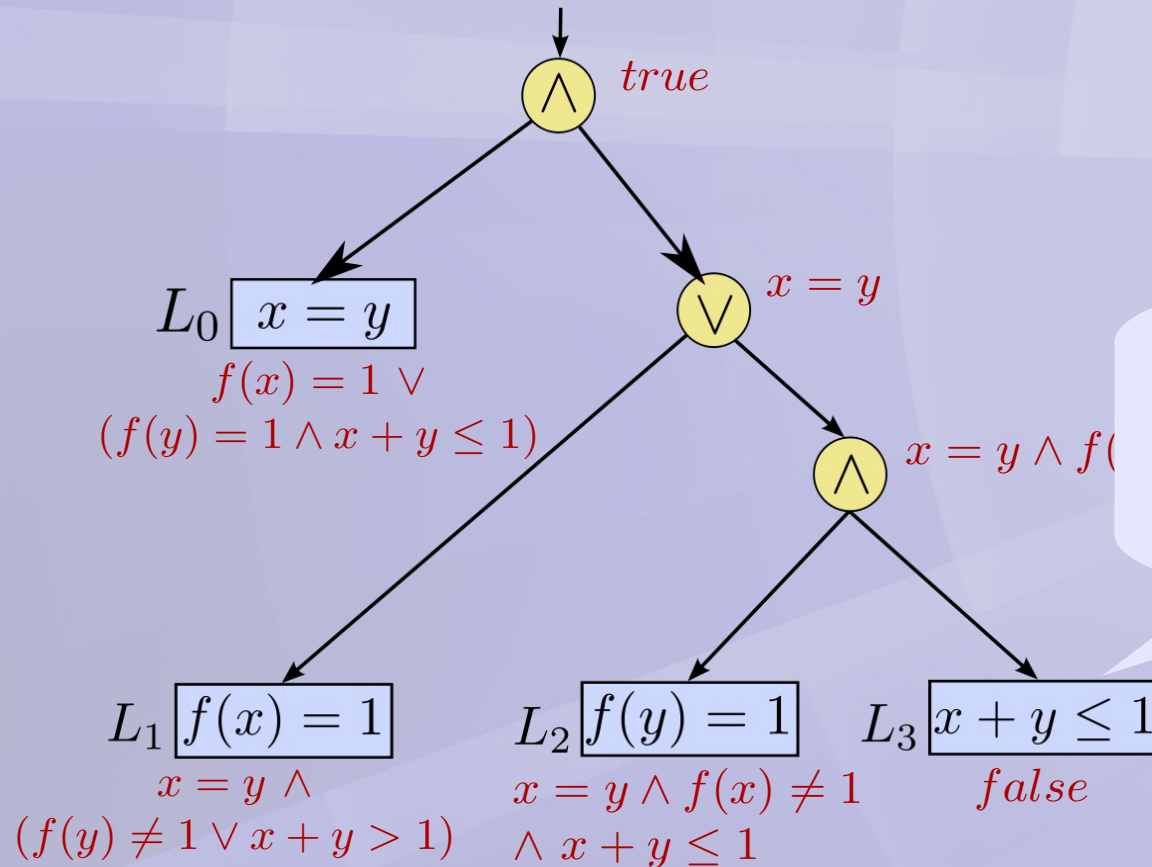
$$x = y \wedge (f(x) = 1 \vee (f(y) = 1 \wedge x + y \leq 1))$$



Example

- Consider again the formula:

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




*Both non-constraining
and non-relaxing because
false implies leaf and its
negation.*

The Full Algorithm

```
/*  
 * Recursive algorithm to compute simplified form.  
 * N: current subformula, C: critical constraint of N.  
 */  
simplify(N, C)  
{  
  - 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 /* Neither */  
  
  - If N is a connective, for each child X of N:  
  
    - Compute critical constraint C(X)  
    -  $X = \text{simplify}(X, C(X))$   
    - Repeat until no child of N can be further simplified.  
  
}
```

Critical constraint is recomputed because siblings may change.

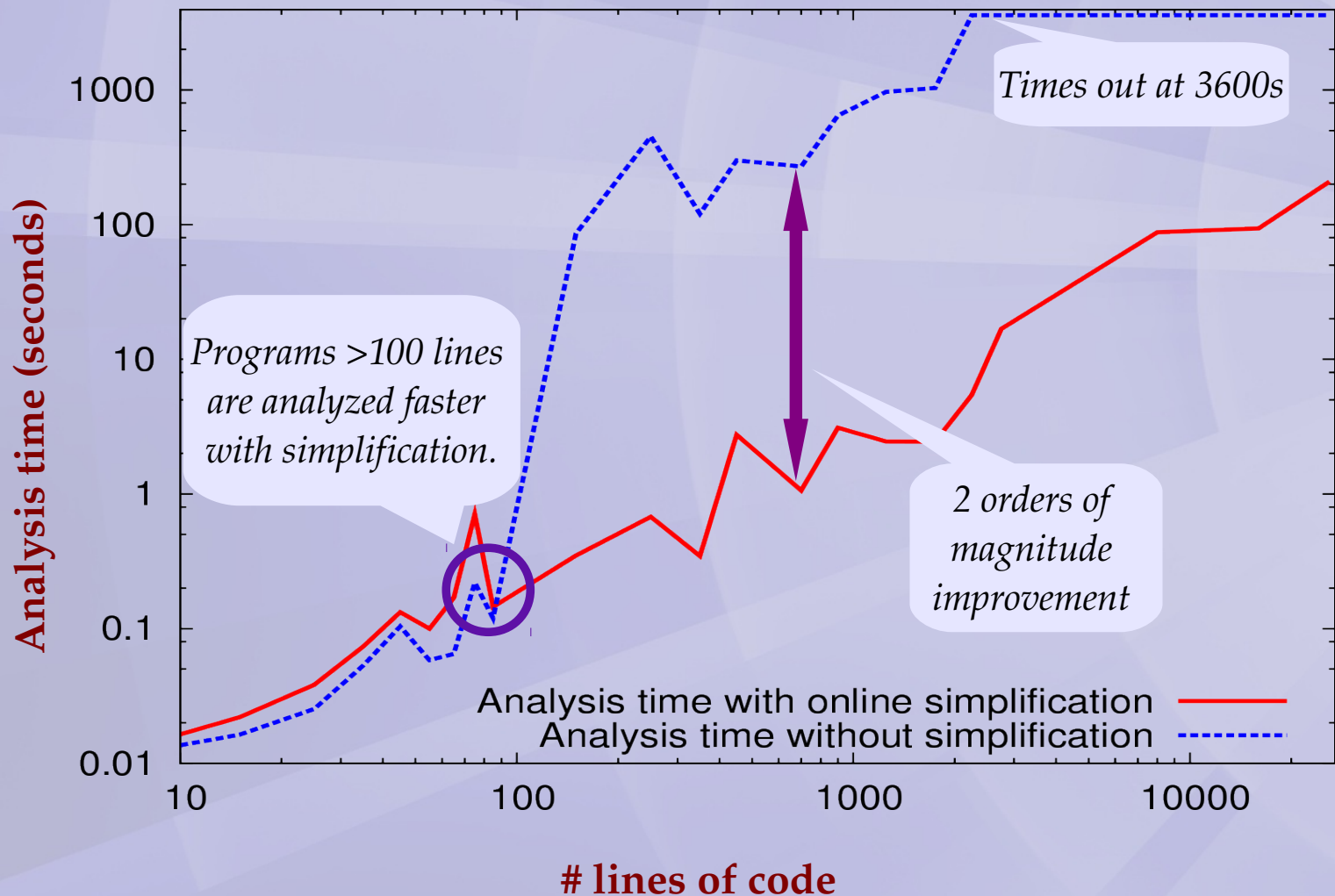
Making it Practical

- **Worst case:** Requires $2n^2$ validity checks. ($n = \#$ leaves)
- **Important Optimization:**
 -  **Insight:** The leaves of the formulas whose validity is checked are always the same.
 - For simplifying SMT formulas, we can gainfully reuse the *same conflict clauses* throughout simplification
- **Empirical Result:** Overhead of simplification over solving *sub-linear (logarithmic)* in practice for constraints generated by our program analysis system.

Impact on Analysis Scalability

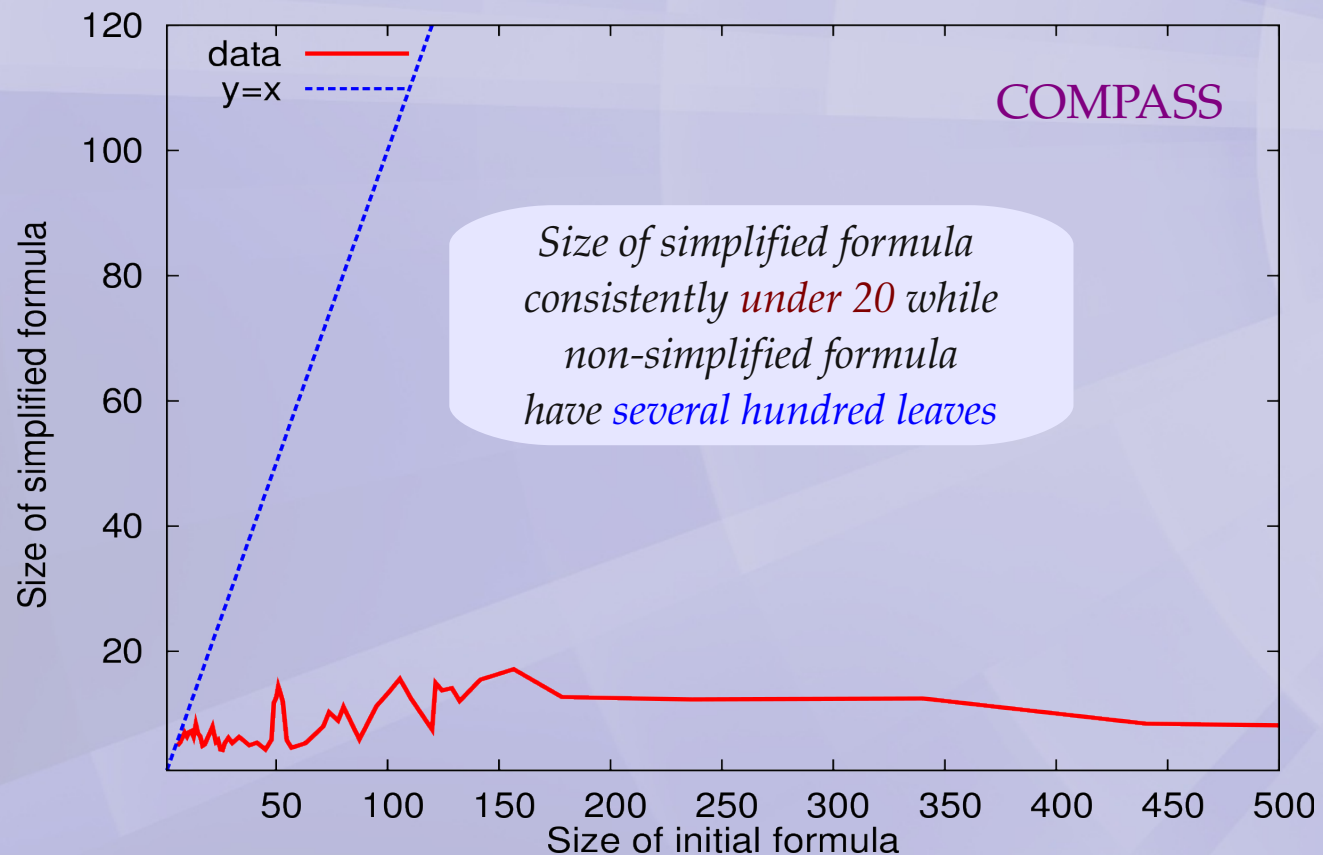
- To evaluate impact of on-line simplification on *analysis scalability*, we ran our program analysis system, *Compass*, on 811 benchmarks.
 - 173,000 LOC
 - Programs ranging from 20 to 30,000 lines
 - Checked for assertions and various memory safety properties.
- Compared running time of runs that *use on-line simplification* with runs that *do not*.

Impact on Analysis Scalability



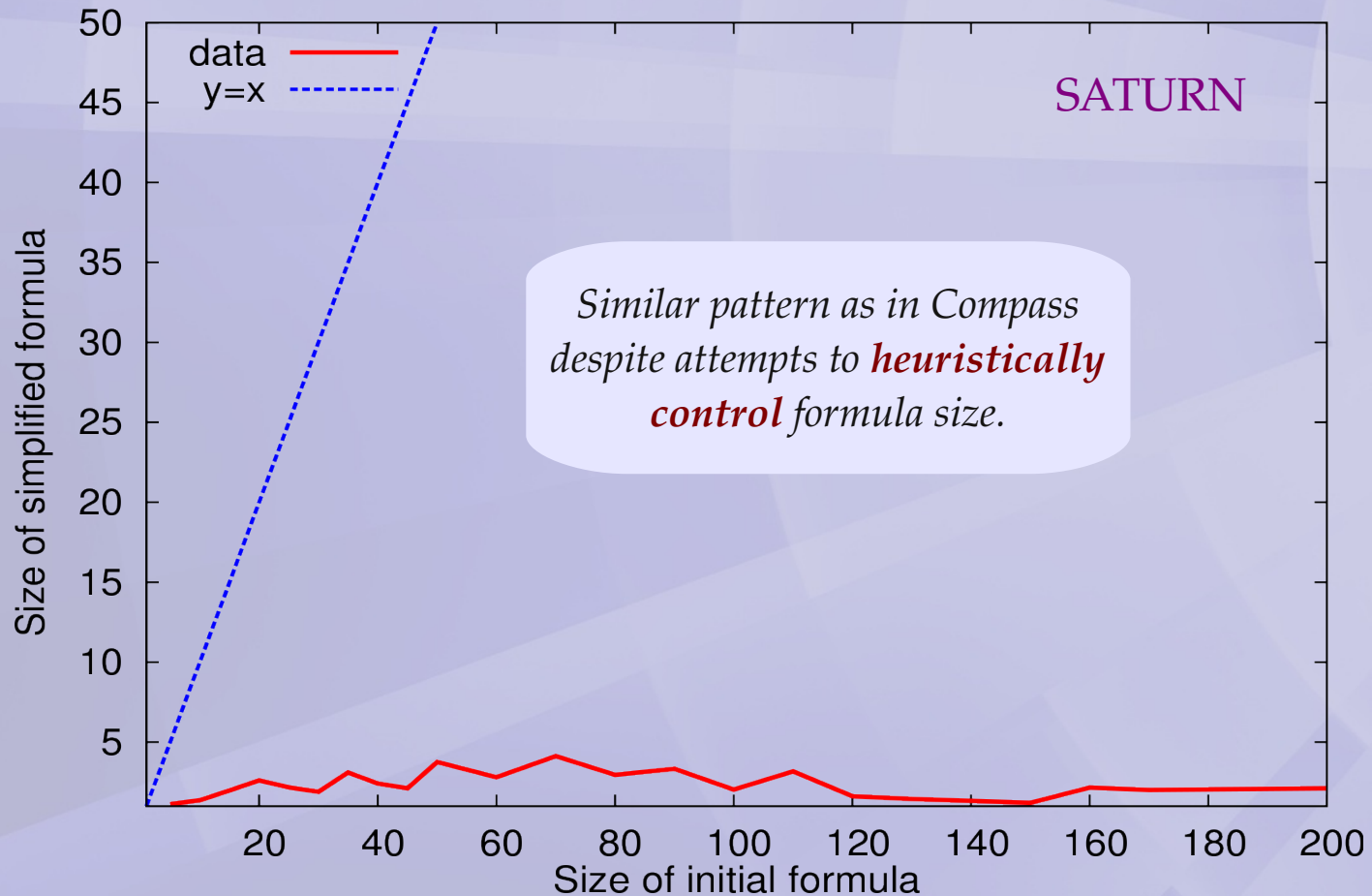
Why Such a Difference?

- Because program analysis systems typically generate *highly redundant constraints!*



It's not just Compass

- Measured redundancy of constraints in a different analysis system, SATURN.



Related Work

- **Contextual Rewriting**
 - Lucas, S. Fundamentals of Context-Sensitive Rewriting. LNCS 1995
 - Armando, A., Ranise, S. Constraint contextual rewriting. Journal of Symbolic Computation 2003
- **Logic Synthesis and ATPG**
 - Mishchenko, A., Chatterjee, S., Brayton, R. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. DAC 2006
 - Mishchenko, A., Brayton, R., Jiang, J., Jang, S. SAT-based logic optimization and resynthesis IWLS 2007
- **And many others:**
 - BDDs and BMDs, vacuity detection in CTL, term rewrite systems, optimizing CLP compilers ...



Any questions?

Thank you!