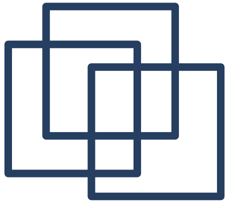
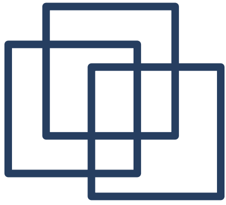


CS157: Computational Logic
Fall 2007

Static Program Analysis

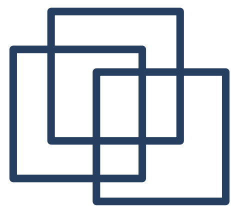


- Everything I will present is joint work with Isil Dillig.

- Unfortunately, she is sick and cannot present as planned today.

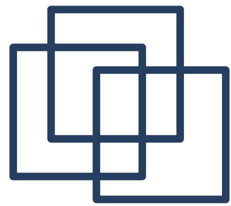
Motivation

- Software is increasingly becoming an inherent part of our every day life.
- Software failures can have very significant consequences.
- Example 1: The Ariane Rocket
  - Cost: 7.5 billion dollars
- Example 2: Therac-25
  - At least 5 patients died.



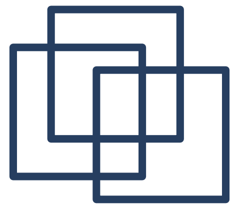
How do we prevent software failures?

- Traditional Testing
  - Cannot achieve good 'coverage'
  - Does not guarantee lack of errors
- Dynamic program analysis
  - Instrument program execution to identify errors
  - Achieves better coverage than testing
  - But still cannot guarantee lack of errors.



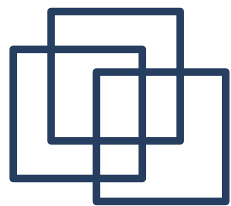
Static Program Analysis

- Find errors in the program without actually executing it.
- Static analysis tool
  - A program that takes another program as input and automatically identifies errors in the input program.
    - 'Symbolic execution' => 100% coverage
    - Verify lack of errors: 'soundness'
      - Partial verification: guarantees lack of a certain class of errors, e.g null dereferences, integer overflow etc.



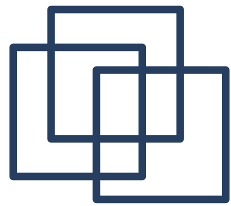
But what about Rice's Theorem?

- Rice's Theorem:
  - Any non-trivial property of programs is undecidable!
    - So, what do we mean by 'verification'?
- We deal with over-approximations of programs
  - such that these over-approximations are over a decidable theory (e.g. boolean SAT)
  - but static analysis tools will report some errors that are not actually errors in the original program
    - false positives



How is computational logic relevant to all this?

- Computational logic comes up all over the place, but especially in constraint-based static analysis.
- Constraint-based analysis
  - Step 1: Generate constraints
  - Step 2: Solve these constraints
    - This is the where the techniques from CS157 are useful!



Constraint Generation

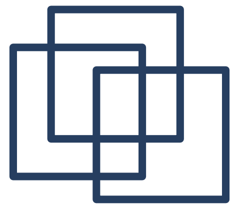
- Example:

```
void foo(bool flag, int b)
{
    int a;
    if(flag)
        a = 3;
    else if(b != 0)
        a = b;
    else a=-1;
    assert(a>0);

}
```

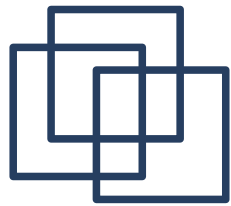
- The constraint under which the assertion holds:

```
flag || (b!=0 & b>0)
```



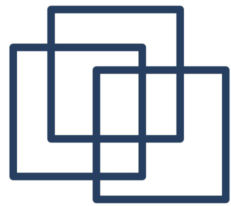
Why are constraints useful?

- Given a constraint for a program property P, we are often interested in knowing either of the following two:
  - Is this constraint satisfiable?
    - If so, then property P MAY hold.
  - Is this constraint valid?
    - Then, property P MUST hold.



Example: Safety Properties

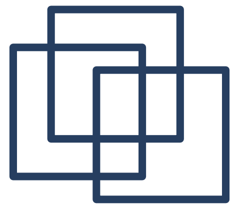
- A safety property is a property stating that 'something bad' never happens:
    - A null pointer may not be dereferenced.
    - An unitialized variable may not be used.
    - A tainted value may not be referenced etc.
- Requires satisfiability query
    - If the constraint under which 'something bad' happens is SAT, then the program may violate the safety property.



Example: Liveness Properties

- A liveness property is a property that specifies that 'something good' eventually happens:
  - Allocated memory must be freed.
  - An opened file must be closed etc.

- Requires querying validity
  - If the constraint is not valid, then liveness property may be violated.

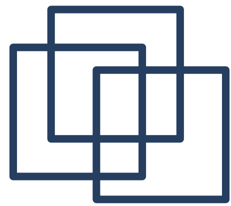


Constraint Resolution

- To solve the kinds of constraints that come up in static analysis, we typically make use of two kinds of theorem provers:
  - SAT solvers
  - SMT solvers

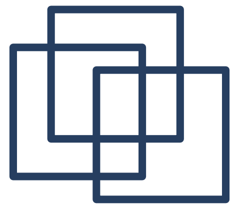
SAT solvers

- Boolean Satisfiability
  - Since every variable is represented as n # of bits in hardware (e.g. n=32), we can model every program variable as n boolean variables.
  - Constraints expressed in propositional logic!
  - Many program analysis tools use this trick
    - E.g. SATURN: http://saturn.stanford.edu/



Example

- Suppose type `supershort` has only 4 bits.
- Assume:
  - The variable `i` has type `supershort`
  - The first bit of `i` is modeled by variable a, second bit by b, third bit by c, and last bit by d.



Example Continued

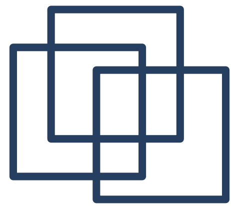
- Then the constraint

```
i=1 & i>1
```

results in the following propositional formula:

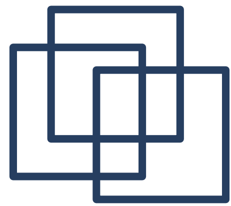
```
a & !b & !c & !d & (b | c | d)
```

- By inspection, this is UNSAT!



Advantages of SAT

- Very efficient SAT solvers already exist!
  - No need to implement ad-hoc constraint solvers for a specific application.
  - Even though boolean SAT is NP-hard, modern SAT solvers can often solve 100,000 clauses in under a second!



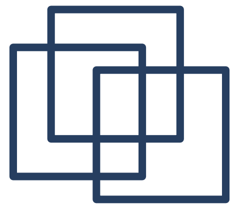
Disadvantages of SAT

- Translating to boolean formulas involves loss of high level information.
  - Ex: To prove that the constraint `i<1 & i>2` unsatisfiable, we need to represent `i` as 32 (or 64) boolean variables!

SMT Solvers

- SMT stands for *satisfiability modulo theories.*
  - Generalization of boolean satisfiability that also incorprates other (decidable) background theories.

- Some example background theories:
  - Theory of integers (Presburger arithmetic)
  - Theory of uninterpreted functions
  - Theory of recursive data structures

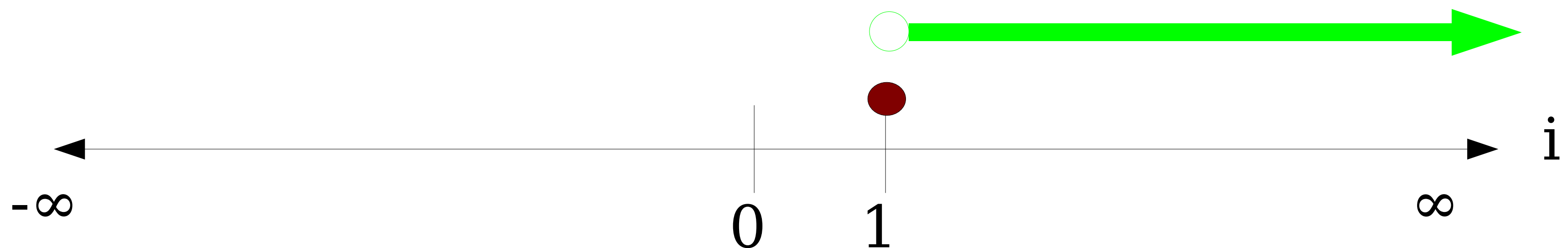


SMT Solvers, continued

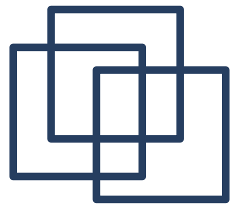
- Advantage:
  - often much faster (can exploit high-level structure of contraints)
- Disadvantage:
  - tailored for a specific constraint representation
  - Perfomance is only good if a problem instance has exploitable higher level structures anticipated by SMT developer.
- Most SMT solvers use SAT solvers as a back-up...

SMT Example

- Assume our SMT solver uses linear intervals and conceptually represents each distinct variable in a constraint as an interval on a line.
- Then the constraint `i=1` & `i>1` results in the following



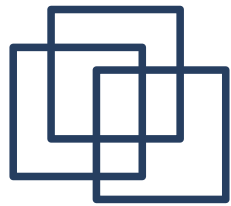
- However, what would happen with `i=1`&`i*2>1`?



Current Research

Disclaimer:

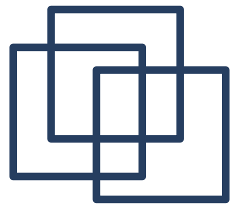
- From now on, I will talk specifically about my current research
- „Sound, Complete and Scalable Path-sensitive Analysis“
  - I. Dillig, T. Dillig, A. Aiken



Recursive Constraints

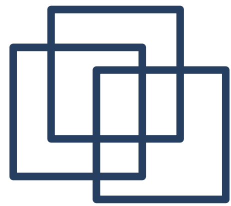
- Unfortunately, so far we only solved constraints from very simple straight-line programs.
- But how can we write the constraint describing the condition under which queryUser returns true/false?

```
bool queryUser(bool featureEnabled){
  if(!featureEnabled) return false;
  char userInput = getUserInput();
  if(userInput == 'y') return true;
  if(userInput=='n') return false;
  printf("Please try again");
  return queryUser(featureEnabled);
}
```



Loops and Recursion

- Since many concepts are easier to explain on recursive functions, the rest of this talk will only use recursion and ignore loops.
- However, since loops can easily be expressed as tail-recursive functions, all the results immediately apply to loops as well.

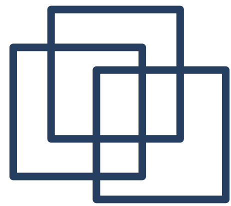


Back to the recursive function

- Let's try to write a constraint the way we did earlier:

```
bool queryUser(bool featureEnabled) {
    if(!featureEnabled) return false;
    char userInput = getUserInput();
    if(userInput == 'y') return true;
    if(userInput=='n') return false;
    printf("Input must be y or n! Please try
                again");
    return queryUser(featureEnabled);
}
```

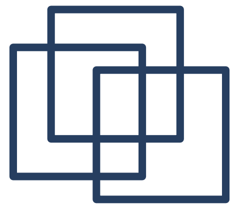
$$\Pi_{\alpha,\mathbf{true}} = \exists\beta.(\alpha = \mathbf{true}) \wedge (\beta = '\mathbf{y}' \vee (\neg(\beta = '\mathbf{n}') \wedge \Pi_{\alpha,\mathbf{true}}[\mathbf{true}/\alpha]))$$



Example Cont.

$$\Pi_{\alpha,\mathtt{true}} = \exists \beta.(\alpha = \mathtt{true}) \wedge (\beta = '\mathbf{y}' \vee (\neg(\beta = '\mathbf{n}') \wedge \Pi_{\alpha,\mathtt{true}}[\mathtt{true}/\alpha]))$$

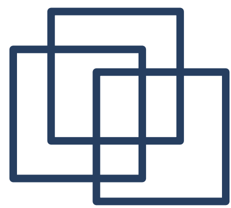
- The existentially quantified β's model *unobservable* program behavior.
  - User input
  - System state (e.g. did malloc() return NULL?)
  - Imprecision in the static analysis
    - e.g. If analysis cannot reason about arrays, treat them as unobservable
- *Important restriction:* We allow only a finite set of values for all variables for the technique to be complete!



Example Cont.

$$\Pi_{\alpha,\mathtt{true}} = \exists \beta.(\alpha = \mathtt{true}) \wedge (\beta = '\mathbf{y}' \vee (\neg(\beta = '\mathbf{n}') \wedge \Pi_{\alpha,\mathtt{true}}[\mathtt{true}/\alpha]))$$

- This contraint is *recursive.*
- What have we achieved so far?
  - We can write recursive constraints, one for each value a function can return.
  - *Good:* Our contraints are modular (can reason about one function at a time)
  - *Bad:* We still have to „solve“ these recursive constraints, i.e. find a closed-form solution…



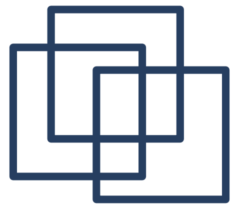
What's so difficult about solving these constraints?

$$\Pi_{\alpha,\mathtt{true}} = \exists\beta.(\alpha = \mathtt{true}) \wedge (\beta = '\mathtt{y}' \vee (\neg(\beta = '\mathtt{n}') \wedge \Pi_{\alpha,\mathtt{true}}[\mathtt{true}/\alpha]))$$

- Repeated substitution does not work!

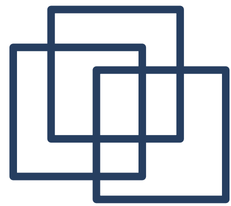
$$\begin{aligned}\Pi_{\alpha,\mathtt{true}} = \quad & \exists\beta.(\alpha = \mathtt{true}) \wedge (\beta = '\mathtt{y}' \vee \neg(\beta = '\mathtt{n}') \wedge \\ & \exists\beta'.(\mathtt{true} = \mathtt{true}) \wedge (\beta' = '\mathtt{y}' \vee \neg(\beta' = '\mathtt{n}') \wedge \\ & \exists\beta''.(\mathtt{true} = \mathtt{true}) \wedge (\beta'' = '\mathtt{y}' \vee \neg(\beta'' = '\mathtt{n}') \wedge \\ & \ldots\end{aligned}$$

- Problem: Different recursive calls can return different values even with the same input
  - Every substitution intruduces a fresh existentially quantified β



What do we really want?

- Recall from earlier that we are really interested in *safety (may)* and *liveness (must)* properties.
  - Is this constraint satisfiable?
  - Is this constraint valid?
- So, all we really need are closed form solutions that preserve:
  - (1) the satisfiability of the solution to original system
  - (2) the validity of the solution to original system



Nessesary and Sufficient Conditions

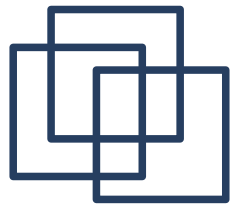
- More precisely, we are interested in the *strongest neccessary* and *weakest sufficient* conditions containing only observable variables (no β's or recursive instantiations).

- Strongest nessary condition:

$$\begin{array}{ll}(1) & \phi \Rightarrow \lceil \phi \rceil \\ (2) & \forall \phi'.((\phi \Rightarrow \phi') \Rightarrow (\lceil \phi \rceil \Rightarrow \phi')) \\ & \textit{where } \mathcal{V}^-(\phi') = \emptyset \ \wedge \ \mathcal{V}^+(\phi') = \mathcal{V}^+(\phi)\end{array}$$

- Weakest sufficient condition:

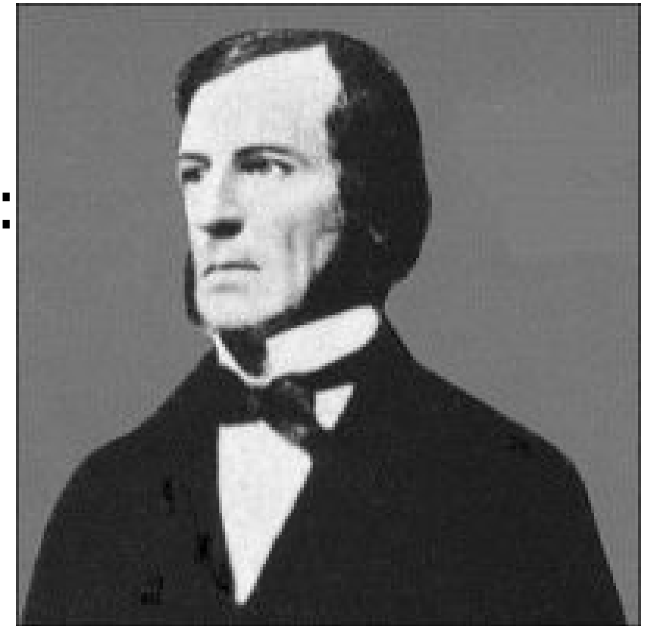
$$\begin{array}{ll}(1) & \lfloor \phi \rfloor \Rightarrow \phi \\ (2) & \forall \phi'.((\phi' \Rightarrow \phi) \Rightarrow (\phi' \Rightarrow \lfloor \phi \rfloor)) \\ & \textit{where } \mathcal{V}^-(\phi') = \emptyset \ \wedge \ \mathcal{V}^+(\phi') = \mathcal{V}^+(\phi)\end{array}$$



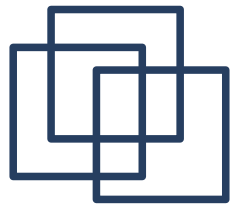
Eliminating the Middle Term

- Let's focus for now on *satisfiability* (strongest necessary conditon).
- The following result is well-known:

$$SNC(\phi, v) \equiv \phi[true/v] \vee \phi[false/v]$$



- George Boole, 1858.
- But rediscovered and published ~once a decade since then!!!



Eliminating β's

- We can apply this to our constraints:

$$\Pi_{\alpha,\mathtt{true}} = \exists\beta.(\alpha = \mathtt{true}) \wedge (\beta = '\mathbf{y}' \vee (\neg(\beta = '\mathbf{n}') \wedge \Pi_{\alpha,\mathtt{true}}[\mathtt{true}/\alpha]))$$

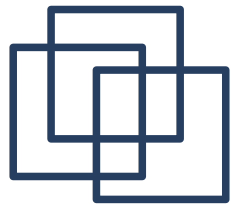
- „Translate“ to boolean constraint:

$$\Pi_{\alpha,\mathtt{true}} = \exists\beta_y\,\exists\beta_n.(\alpha = \mathtt{true}) \wedge (\beta_y \vee (\neg(\beta_n) \wedge \Pi_{\alpha,\mathtt{true}}[\mathtt{true}/\alpha])) \wedge \neg(\beta_y \wedge \beta_n)$$

- Eliminating the β's then yields:

$$\lceil \Pi_{\alpha,\mathtt{true}} \rceil = (\alpha = \mathtt{true} \wedge \Pi_{\alpha,\mathtt{true}}[\mathtt{true}/\alpha])$$

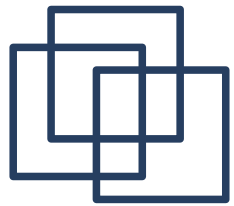
Uniqueness constraint



Where are we?

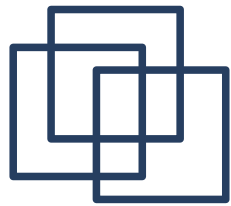
- We now have over and under-approximations of our original constraint system only containing observable variables.

- However, we can only solve this system if it is *monotonic.*
  - Monotonicity required to guarantee that fixed point computation converges (terminates).



Causes of non-monotonicity

- Two causes on non-monotonicity:
  - (1) Recursive instantiation appears negated.
  - (2) Uniqueness and existence constraints also appear anti-monotonically!

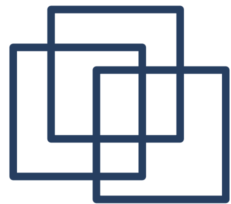


Solution

(1) Recursive instantiation appears negated

- Here we use our inital *finite constant assumtion*.
- Recall that we assumed that there are a finite number of values each function can return. Therefore:

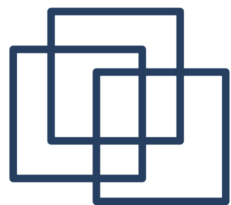
$$\neg\Pi_{f,\alpha,C_i} = \bigvee_{j \neq i} \Pi_{f,\alpha,C_j}$$



(2) Uniqueness constraint

- Here, we cannnot apply the same trick!
  - this trick only preserves satisfiability if we explicitly have *uniqueness and existence constraints.*

- Goal: We want to get rid of uniqueness and existence constraints, but we have to preserve satisfiability (resp. validity).

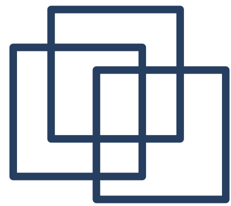


(2) Uniqueness constraint, cont'd

- Solution: Convert to disjunctive normal form and drop all clauses where v is assigned two distinct values, i.e. a violation of the uniqueness constraint.

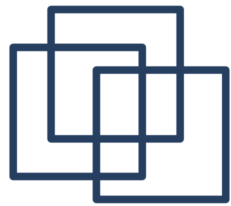
$$\phi' = DNF(\mathcal{M}(\phi)) = \begin{array}{l}(p_{11}... \wedge ...p_{1n}) \vee ...(p_{i1} \wedge ...p_{ij}... \wedge p_{in}) \\ \vee ...(p_{m1}... \wedge ...p_{mn})\end{array}$$

- Intuitively, this means we implicitly enforce all uniqueness constraints.
- Result:
  - Formulas are monotonic in recursive instantiations
  - And preserve satisfiability of original system



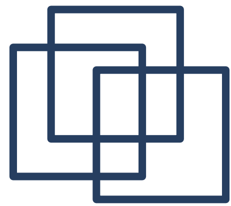
What did we achieve?

- Claim:
  - The modified (monotonic) formulas preserve strongest necessary conditions under *syntactic substitution*
    - Proof omitted
  - Fixed point computation can progressively „plug in“ better approximations to the actual strongest necessary condition



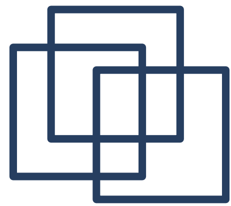
Solving the recursive system

- Since our system is now monotonic the fixed-point computation is guaranteed to converge.
- Main result:
  - This approach entrails no loss in precision for answering may or must queries, assuming a finite domain of tracked values.
  - Sound and complete



Advantages

- The main advantages of this approch are:
  - Works in the presence of recursion (or loops)
  - Is both sound and complete for a finite domain, and still sound for an infinite domain
  - Deals with one function at a time, therefore can scale to very large programs.
    - e.g. the entire Linux kernel > 6 MLOC



Applications

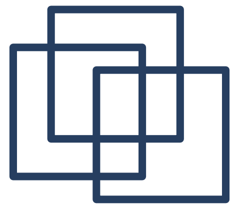
- We used this technique to verify the absence of null dereference errors in three large open-source applications (excluding unbounded data strutures).
- Our implementation builds on the SATURN infrastructure.
- While these experiments are computationally expensive (we use a 50 machine cluster), dealing with only one function at a time means we can scale ~linearly in the size of the input program.

Results

Uses the technique discussed

	Interprocedurally Path-sensitive			Intraprocedurally Path-sensitive		
	OpenSSH 4.3p2	Samba 3.0.23b	Linux 2.6.17.1	OpenSSH 4.3p2	Samba 3.0.23b	Linux 2.6.17.1
Total Reports	2	28	171	26	379	1495
Bugs	1	10	134	1	10	134
False Positives	1	14	37	25	365	1344
Undecided	0	4	17	0	4	17
Report to Bug Ratio	2	2.8	1.3	26	37.9	11.2

- First practical Null-pointer verification (with acceptable false positive rate) on large applications.
- Finds ~10x more bugs then previous unsound analysis.
  - „Static Error Detection Using Semantic Inconsistency Inference“ -- Dillig, Dillig, Aiken, PLDI 2007



Thanks a lot!

- Any Questions?