Lecture 13

Static Single Assignment &

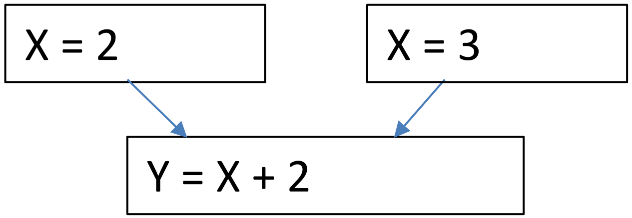
Intro to Satisfiability Modulo Theories

1. Static single assignment (SSA)
2. Optimizations with SSA
3. Introduction to Satisfiability Modulo Theories (SMT)
4. SMT Application: Path Sensitive Program Analysis

Thanks to Clark Barrett, Nikolaj Bjørner Leonardo de Moura, Bruno Dutertre, Albert Oliveras, and Cesare Tinelli for contributing material used in this lecture.

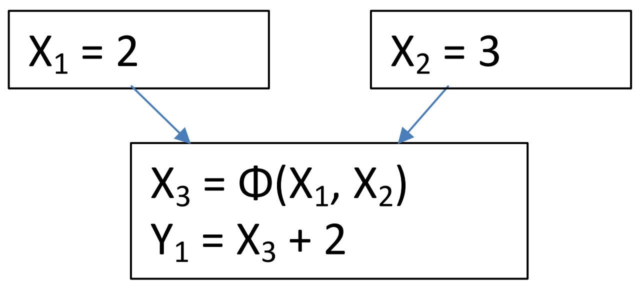
SSA Motivation

- Simple question: for a given use of a variable, where was this variable defined?
  - For normal programs, requires dataflow
  - Each use can have multiple possible definitions
- Ideally, we would like if each use only has one possible definition
- How is this possible?



Φ Functions

- Introduce Φ functions: for a block with multiple possible definitions, represent all the definitions that can reach that block
    - One operand for each predecessor
- Now we can assign each definition a unique name
- This type of representation is known as SSA (static single assignment)
    - Each variable has exactly one definition

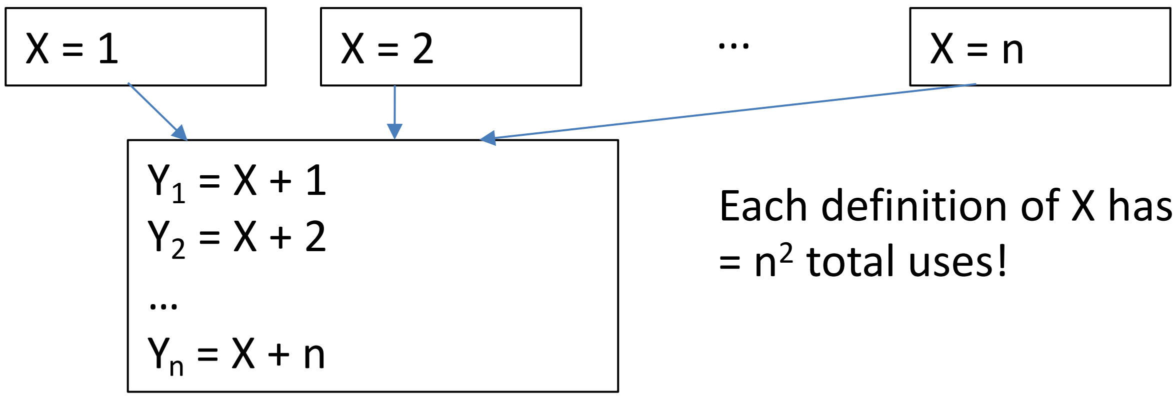


What is SSA?

- Static Single Assignment form: type of intermediate representation
  - Each variable is assigned statically (in code) exactly once
  - Each definition is assigned a unique name
- Properties:
  - Makes def-use chains explicit
  - Definitions dominate uses (key property)
  - This makes certain optimizations simpler or more efficient
- Used in pretty much every modern optimizing compiler
  - Most notably, LLVM: Clang, rustc, swiftc, GHC, Julia
  - GCC (GNU Compiler Collection), Microsoft Visual C++ compiler
  - Java HotSpot JVM
  - V8 (Google Chrome), SpiderMonkey (Firefox), JavaScriptCore (Safari/WebKit)

Def-use chains

- Connects a definition (def) to a use
- In non-SSA programs, total size of def-use chains can be quadratic
- Example:



Each definition of X has n uses
= n^2 total uses!

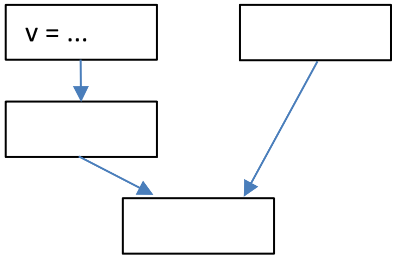
- In SSA, exactly one definition for each variable → guaranteed linear
  - Definitions must *dominate* uses
  - A basic block dominates another (B1 dom B2) iff all paths from entry to B2 must pass through B1

Dominance

- Recall that a basic block x **dominates** another basic block y iff all control paths from entry to y must pass through x
  - If x ≠ y, then x strictly dominates y (x sdom y)
- How to find dominance?
  - Dataflow (you did this for HW)
  - Faster algorithm: Lengauer-Tarjan ($O(E\ \alpha(E, N))$ time)
    - α is less than 5 for all practical inputs

Inserting Φ Functions

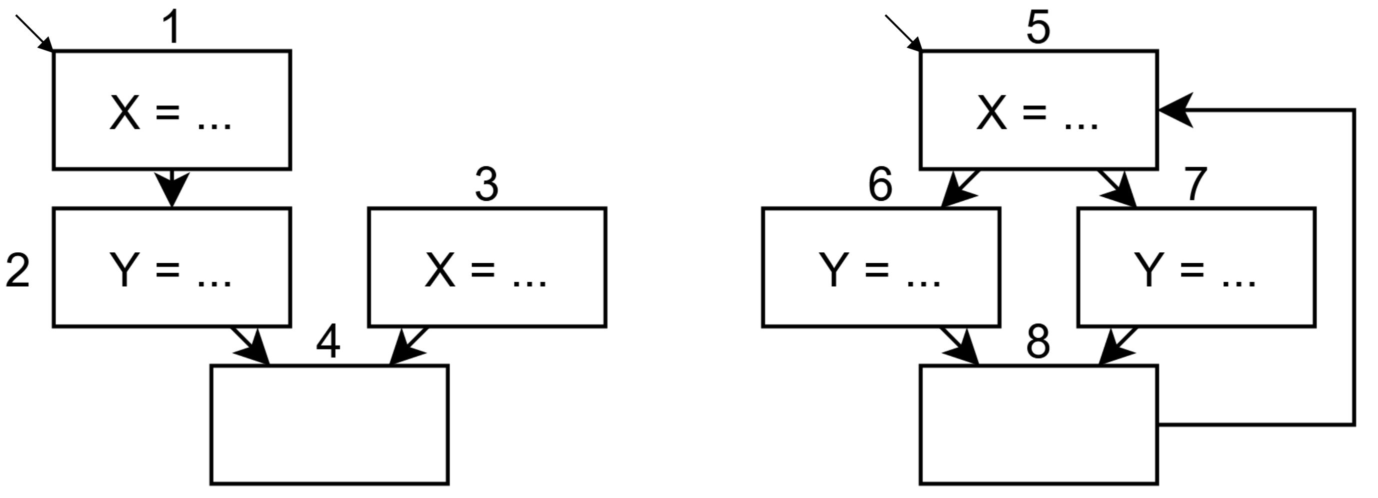
- Suppose basic block A has a definition of variable V
- Which blocks B need a Φ function for variable V?
  (will worry about the operands later)
  - If all predecessors of block x are dominated by A
    (equiv: A sdom x),
    a definition to V (at A, or in a block that strictly dominates x)
    must reach the entry to b.
  - To avoid redundant Φ functions,
    only insert at earliest possible block for each path
    where there can be multiple definitions.
    - B is not strictly dominated by A
    - One of B's predecessors must be dominated by A:
      otherwise, we could insert at predecessor



i.e. B is in the *dominance frontier* of A

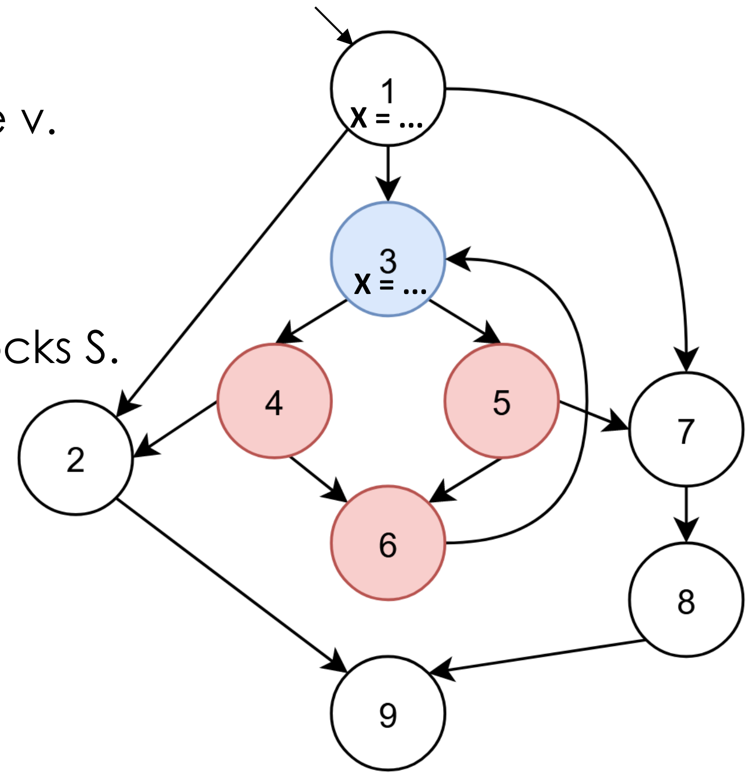
Dominance Frontier

- Dominance frontier of a basic block A
  - DF(A) = set of blocks B where A does not strictly dominate B, and some predecessor of B is dominated by A.
- DF(1)?
- DF(5)?



Where to Insert Φ Functions: Iterated Dominance Frontier

- For a given variable v,
  let defs(v) be the set of blocks that define v.
  - Ex: defs(X) = {1, 3}.
- Insert Φ functions at $\cup_{A \in defs(v)}$ DF(A).
  - Defn: DF(S) = $\cup_{A \in S}$ DF(A) for a set of blocks S.
  - Ex: DF(defs(X)) = {2, 3, 7}.
- These are new definitions!
- Insert Φ functions at
  DF(DF(defs(v)), DF(DF(DF(defs(v)))), …
  - Define $DF_1(S) = DF(S)$
    $DF_{i+1}(S) = DF(S \cup DF_i(S))$
  - Iterated dominance frontier
    $DF^+(S)$ is the limit $DF_\infty(S)$.
  - Ex: DF^+(defs(X)) = {2, 3, 7, 9}.



Note: we are only deciding where to insert Φ functions; we fill in the operands later.

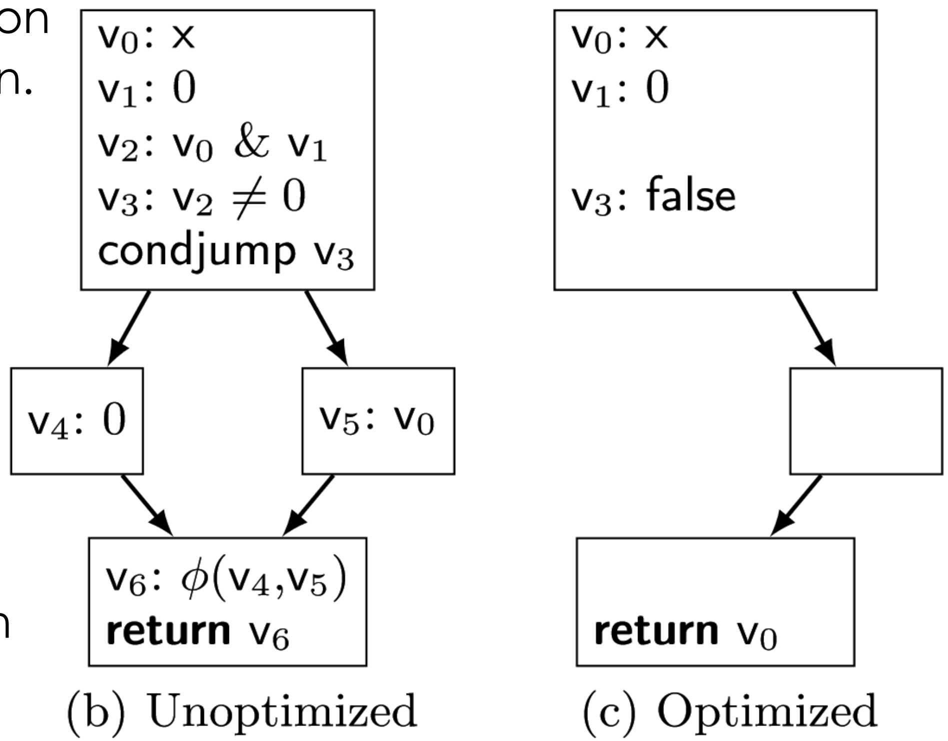
SSA Form Overview

- Converting to SSA form:
  - Insert Φ functions
  - Assign unique names to definitions
  - Propagate definitions to uses (including operands in Φ functions)
- Perform optimizations...
- Converting out of SSA form:
  - For each Φ function, insert a copy in predecessors
  - Remove Φ functions

2. On-the-Fly Optimizations

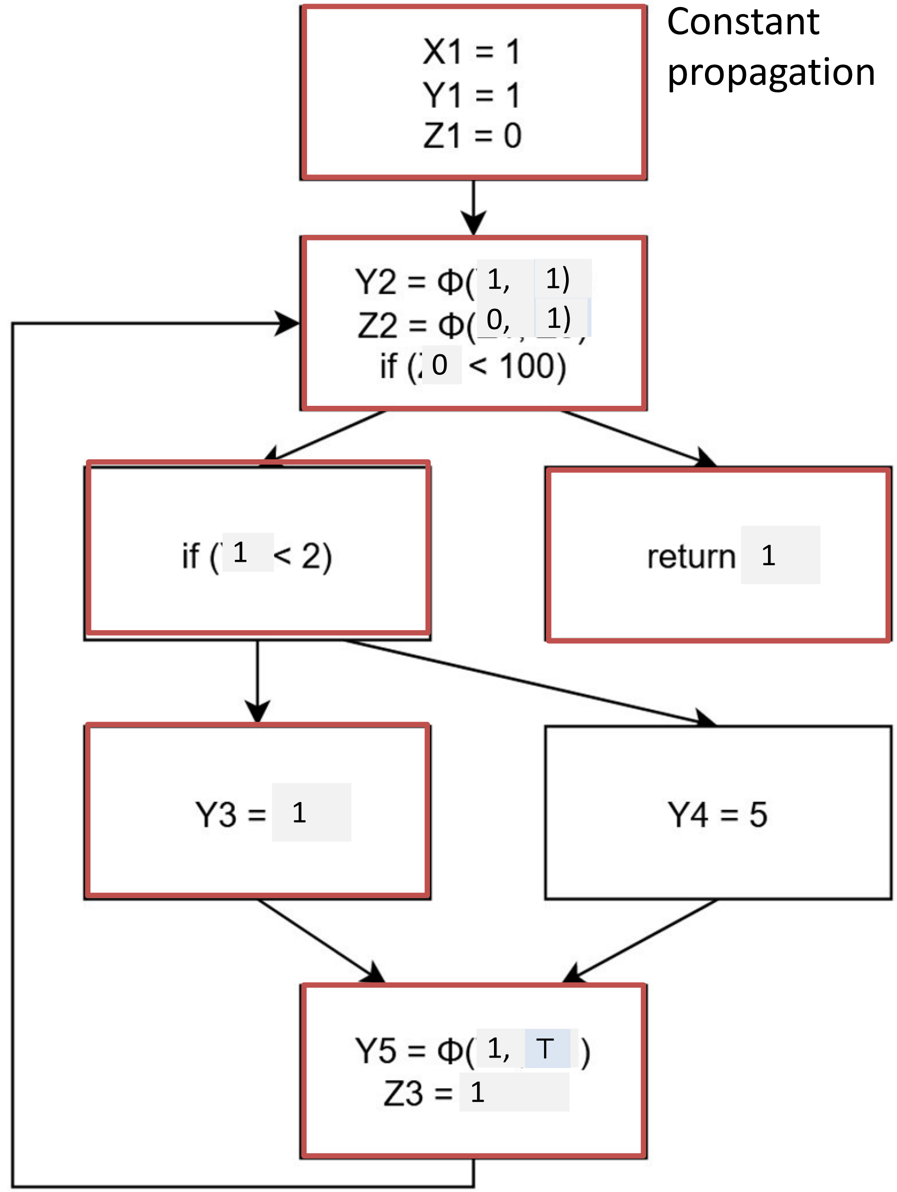
- We can optimize "on the fly" while constructing the SSA form.
- Each variable only has one definition → cheap to keep some information per variable during construction.
- Simple optimizations:
  - Arithmetic simplification
  - Constant folding
  - Copy propagation
  - Common subexpression elimination
    - Needs to maintain some information per expression
- Without doing dataflow, can miss optimization opportunities in loops

Braun, M. et al (2013). Simple and Efficient Construction of Static Single Assignment Form.



Sparse Conditional Constant Propagatio

- Standard constant propagation cannot deduce that the statemen Y4 = 5 is never executed
- Solution: assume each block is not executed until shown otherwise
  - Perform dataflow on
    - Variables: standard constant prop lattice
    - Blocks: top (not executed) or bottom (executed)
- Use use-def information to quickly propagate information from definitions to uses
- A similar approach can be used fc aggressive dead code elimination



3. What is Satisfiability Modulo Theories (SMT)?

- Satisfiability
  - the problem of determining whether a formula has a model (an assignment that makes the formula true)

- SAT: Satisfiability of **propositional formulas**
  - A model is a truth assignment to Boolean variables
  - SAT solvers: check satisfiability of propositional formulas
    - Decidable, NP-complete

- SMT: Satisfiability modulo theories
  - Satisfiability of first-order formulas containing operations from background theories such as arithmetic, arrays, uninterpreted functions, etc.
    E.g. $g(a) = c \wedge f(g(a)) \neq f(c)$
  - SMT Solvers:
    - check satisfiability of SMT formulas with respect to a theory

Use of SMT for Program Correctness & Test Generation

- Precision: Path sensitivity
- Is assertion A in a program?
  If it is not true, find an input that triggers the error
- SMT formulation:
  Given an assertion A,
  can we generate an input that triggers an error on a given path p?
  - Let F be the formula representing the execution of p
  - Is the formula $F \wedge \neg A$ satisfiable?
    - Not satisfiable? No error on that path
    - Satisfiable? Find 1 assignment that satisfies the formula (1 set of test inputs)

Each Statement is a Logical Clause

Program Assume data array bound is [0, N-1]

```
1 void ReadBlocks(int data[], int cookie)
2 {
3   int i = 0;
4   while (true)
5   {
6      int next;
7      next = data[i];
8      if (!(i < next && next < N)) return;
9      i = i + 1;
10     for (; i < next; i = i + 1){
11        if (data[i] == cookie)
12          i = i + 1;
13        else
14          Process(data[i]);
15     }
16   }
17 }
```

One execution path

Static Single Assignment (SSA)

3 $i_1 = 0;$

7 $next_1 = data_0 [i_1];$

8 $i_1 < next_1 \ \&\& \ next_1 < N_0$

9 $i_2 = i_1 + 1;$

10 $i_2 < next_1;$

11 $data_0 [i_2] = cookie_0;$

12 $i_3 = i_2 + 1;$

10 $i_4 = i_3 + 1;$

10 $!(i_4 < next_1);$

7 $next_2 = data_0 [i_4];$

An Execution Path as a Logic Formula

Program Assume data array bound is [0, N-1]

```
1 void ReadBlocks(int data[], int cookie)
2 {
3   int i = 0;
4   while (true)
5   {
6      int next;
7      next = data[i];
8      if (!(i < next && next < N)) return;
9      i = i + 1;
10     for (; i < next; i = i + 1){
11        if (data[i] == cookie)
12          i = i + 1;
13        else
14          Process(data[i]);
15     }
16   }
17 }
```

One execution path (SSA)

$F = \{$

3 $i_1 = 0;$

7 $next_1 = data_0[i_1];$

8 $i_1 < next_1 \ \&\& \ next_1 < N_0$

9 $i_2 = i_1 + 1;$

10 $i_2 < next_1;$

11 $data_0[i_2] = cookie_0;$

12 $i_3 = i_2 + 1;$

10 $i_4 = i_3 + 1;$

10 $!(i_4 < next_1);$

7 $next_2 = data_0[i_4];$

$\}$

Line 7: Array bound assertion A:

$(0 \leq i_1 \wedge i_1 < N_0)$

Checking for Out-of-Bound Array Access (Line 7, iteration 1)

Program Assume data array bound is [0, N-1]

```
1 void ReadBlocks(int data[], int cookie)
2 {
3   int i = 0;
4   while (true)
5   {
6      int next;
7      next = data[i];
8      if (!(i < next && next < N)) return;
9      i = i + 1;
10     for (; i < next; i = i + 1){
11        if (data[i] == cookie)
12           i = i + 1;
13        else
14           Process(data[i]);
15     }
16   }
17 }
```

One execution path (SSA)

$F = \{$

```
 3 i1 = 0;
 7 next1 = data0 [i1];
 8 i1 < next1 && next1 < N0
 9 i2 = i1 + 1;
10 i2 < next1;
11 data0 [i2] = cookie0;
12 i3 = i2 + 1;
10 i4 = i3 + 1;
10 !(i4 < next1);
 7 next2 = data0 [i4];
```

$\}$

Line 7: Array bound assertion A:

$(0 \leq i_1 \wedge i_1 < N_0)$

1st execution of Line 7

Check: Is $F \wedge \neg A$ satisfiable?

$i_1 = 0 \wedge \neg(0 \leq i_1 \wedge i_1 < N_0)$

Answer for Out-of-Bound Array Access (Line 7, iteration 1)

Program Assume data array bound is [0, N-1]

```
1 void ReadBlocks(int data[], int cookie)
2 {
3   int i = 0;
4   while (true)
5   {
6      int next;
7      next = data[i];
8      if (!(i < next && next < N)) return;
9      i = i + 1;
10     for (; i < next; i = i + 1){
11        if (data[i] == cookie)
12          i = i + 1;
13        else
14          Process(data[i]);
15     }
16   }
17 }
```

Line 7: Array bound assertion A:

$(0 \le i_1 \wedge i_1 < N_0)$

$\mapsto$ maps to

One execution path (SSA)

$F = \{$ 3 $i_1 = 0;$ $\}$

7 $next_1 = data_0[i_1];$
8 $i_1 < next_1 \ \&\& \ next_1 < N_0$
9 $i_2 = i_1 + 1;$
10 $i_2 < next_1;$
11 $data_0[i_2] = cookie_0;$
12 $i_3 = i_2 + 1;$

10 $i_4 = i_3 + 1;$
10 $!(i_4 < next_1);$
7 $next_2 = data_0[i_4];$

1st execution of Line 7

Check: Is $F \wedge \neg A$ satisfiable?

$i_1 = 0 \wedge \neg(0 \le i_1 \wedge i_1 < N_0)$

Yes! $\{i_1 \mapsto 0, \ N_0 \mapsto 0\}$ *BUG!!*

Checking for Out-of-Bound Array Access (Line 7, iteration 2)

Program Assume data array bound is [0, N-1]

```
1 void ReadBlocks(int data[], int cookie)
2 {
3   int i = 0;
4   while (true)
5   {
6      int next;
7      next = data[i];
8      if (!(i < next && next < N)) return;
9      i = i + 1;
10     for (; i < next; i = i + 1){
11        if (data[i] == cookie)
12          i = i + 1;
13        else
14          Process(data[i]);
15     }
16   }
17 }
```

One execution path (SSA)

$F = \bigwedge$ {

```
 3 i_1 = 0;
 7 next_1 = data_0 [i_1];
 8 i_1 < next_1 && next_1 < N_0
 9 i_2 = i_1 + 1;
10 i_2 < next_1;
11 data_0 [i_2] = cookie_0;
12 i_3 = i_2 + 1;
10 i_4 = i_3 + 1;
10 !(i_4 < next_1);
 7 next_2 = data_0 [i_4];
```

}

2nd execution of Line 7

Line 7: Array bound assertion A:

$(0 \leq i_4 \wedge i_4 < N_0)$

Check: Is $F \wedge \neg A$ satisfiable?

$F \wedge \neg(0 \leq i_4 \wedge i_4 < N_0)$

Yes! SMT solver finds an assignment that satisfies the proposition

Answer for Out-of-Bound Array Access (Line 7, iteration 2)

Program Assume data array bound is [0, N-1]

```
1 void ReadBlocks(int data[], int cookie)
2 {
3    int i = 0;
4    while (true)
5    {
6       int next;
7       next = data[i];
8       if (!(i < next && next < N)) return;
9       i = i + 1;
10      for (; i < next; i = i + 1){
11         if (data[i] == cookie)
12           i = i + 1;
13         else
14           Process(data[i]);
15      }
16    }
17 }
```

One execution path (SSA)

$F = \bigwedge$

```
 3 i1 = 0;
 7 next1 = data0 [i1];
 8 i1 < next1 && next1 < N0
 9 i2 = i1 + 1;
10 i2 < next1;
11 data0 [i2] = cookie0;
12 i3 = i2 + 1;
10 i4 = i3 + 1;
10 !(i4 < next1);
 7 next2 = data0 [i4];
```

Line 7: Array bound assertion A:

$(0 \le i_4 \wedge i_4 < N_0)$

Var	$\mapsto$
N_0	3
i_1	0
i_2	1
i_3	2
i_4	3
$next_1$	2
$data_0$	<2,0,0>
$cookie_0$	0

BUG!!

Checking the Whole Program All at Once

- A program has many execution paths
- Conditional statements
  - Represent alternative paths symbolically with one formula using SSA
- Loops
  - Optimistically: Unroll a few times
  - Catches many errors, but not all errors

Conditional Statements

- Conditional statements: φ functions in SSA

```
1 if (i > 0) {
2       a = 2;
3       b = 3;
4} else
5{
6       a = 3;
7       b = 2;
8}

9       c = a+b;
```

$F = \bigwedge$ {

```
1   φ1 = (i0 > 0)
2   a1 = 2
3   b1 = 3


6   a2 = 3
7   b2 = 2
8   a3 = φ1?a1:a2;  //(φ1→a3=a1)∨(¬φ1→a3=a2)
8   b3 = φ1?b1:b2;  //(φ1→b3=b1)∨(¬φ1→b3=b2)
9   c3 = a3+b3
```

}

- Assert A: $c_3 = 5$
- Substituting with constants,

  F: $\varphi_1 = (i_0 > 0) \wedge (\varphi_1 \rightarrow c_3 = 5) \wedge (\neg\varphi_1 \rightarrow c_3 = 5)$
- Is $F \wedge \neg A$ satisfiable? (Substituting a_1 b_1 a_2 b_2 with constants)

  $\varphi_1 = (i_0 > 0) \wedge (\varphi_1 \rightarrow c_3 = 5) \wedge (\neg\varphi_1 \rightarrow c_3 = 5) \wedge (c_3 \neq 5)$

Applying the Resolution Rule to Example

- Is $F \wedge \neg A$ satisfiable?

  $\varphi_1 = (i_0 > 0) \wedge (\varphi_1 \rightarrow c_3 = 5) \wedge (\neg\varphi_1 \rightarrow c_3 = 5) \wedge (c_3 \neq 5)$

- Recall: $p \rightarrow q \equiv \neg p \vee q$

  $\varphi_1 = (i_0 > 0) \wedge (\neg\varphi_1 \vee c_3 = 5) \wedge (\varphi_1 \vee c_3 = 5) \wedge (c_3 \neq 5)$

  $\varphi_1 = (i_0 > 0) \wedge (c_3 = 5) \wedge (c_3 \neq 5)$

- $F \wedge \neg A$ is not satisfiable
- The assertion A is true.

Resolution rule in propositional logic:

Given $p \vee A$ and $\neg p \vee B$, add the resolvent $A \vee B$

Resolve $$\frac{p \vee A \qquad \neg p \vee B}{A \vee B}$$

Loops

- Optimistically: Unroll two times

```
for (; i<next; i = i + 1){
   if (data[i] == cookie)
      i = i + 1;
   else
      Process(data[i]);
}
```

```
if (i < next) {
  if (data[i] == cookie)
     i = i + 1;
  else
     Process(data[i]);

  i = i + 1;

  if (i < next) {
    if (data[i] == cookie)
       i = i + 1;
    else
       Process(data[i]);

    i= i + 1;
  }
}
```

Loops: Apply SSA

```
if (i < next) {
  if (data[i] == cookie)
     i = i + 1;
  else
     Process(data[i]);
  i = i + 1;
  if (i < next) {
    if (data[i] == cookie)
       i = i + 1;
    else
       Process(data[i]);
    i= i + 1;
  }
}
```

```
φ1 = (i0 < next0);
φ2 = (data0 [i0] == cookie0);
i1 = i0 + 1;
i2 = φ2 ? i1 : i0;
i3 = i2 + 1;
φ3 = (i3 < next0);
φ4 = (data0 [i3] == cookie0);
i4 = i3 + 1;
i5 = φ4 ? i4 : i3;
i6 = i5 + 1;
i7 = φ3 ? i6 : i3;
i8 = φ1 ? i7 : i0;
```

Unrolling the loop twice finds many (but not) all bugs

Major Categories of Program Analysis Tools

	Static Property Based	Dynamic Execution Based
Complete **(Small programs)**	**Verification** Prove a property in a program Floyd-Hoare logic: {pre-condition} s {post-condition} Applicable to small programs	**(Symbolic) Model Checking (e.g. SMT)** Given a system model (sw/hw), simulate the execution to check if a property is true for all possible inputs. Symbolic: many states all at once
Incomplete **(Large programs)**	**Static Analysis (e.g. Data flow)** Abstract the program conservatively Find fixed-point for all possible executions Sound: no false-negatives--find all bugs False-positives: false warnings If the analysis is too imprecise → useless	**Test case generation (e,g. SMT)** Check a property opportunistically (e.g. unroll loops twice) Use analysis to generate test inputs No false-positives: generate a test False-negatives: cannot find all bugs No correctness/security guarantees

Summary

- SSA: Static single assignment
  - A useful program representation
  - Facilitate optimizations, path-sensitive analysis
  - Used in many optimizations

- SMT: Satisfiability Modulo Theories
  - Application: Finds errors with path-sensitive analysis
  - Next class:
    - How to create an SMT solver?
    - What are other applications?