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?
  o For normal programs, requires dataflow
    o Each use can have multiple possible definitions
• Ideally, we would like if each use only has one possible definition
• How is this possible?

\[
\begin{align*}
  X &= 2 \\
  Y &= X + 2 \\
  X &= 3
\end{align*}
\]
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

\[
X_1 = 2 \quad \text{X}_2 = 3
\]

\[
X_3 = \Phi(X_1, X_2) \\
Y_1 = X_3 + 2
\]
What is SSA?

• Static Single Assignment form: type of intermediate representation
  o Each variable is assigned statically (in code) exactly once
  o Each definition is assigned a unique name

• Properties:
  o Makes def-use chains explicit
  o Definitions dominate uses (key property)
  o This makes certain optimizations simpler or more efficient

• Used in pretty much every modern optimizing compiler
  o Most notably, LLVM: Clang, rustc, swiftc, GHC, Julia
  o GCC (GNU Compiler Collection), Microsoft Visual C++ compiler
  o Java HotSpot JVM
  o 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:

  \[
  \begin{align*}
  X &= 1 \\
  X &= 2 \\
  \cdots &
  \end{align*}
  \]

  Each definition of \( X \) has \( n \) uses
  
  \[
  \begin{align*}
  Y_1 &= X + 1 \\
  Y_2 &= X + 2 \\
  \cdots \\
  Y_n &= X + n \\
  \end{align*}
  \]
  
  = \( n^2 \) 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 $x$
  o If $x \neq y$, then $x$ strictly dominates $y$ ($x$ sdom $y$)

o How to find dominance?
  • Dataflow (you did this for HW)
  • Faster algorithm: Lengauer-Tarjan ($O(E \alpha(E, N))$ time)
    o $\alpha$ is less than 5 for all practical inputs
Inserting $\Phi$ Functions

• Suppose basic block A has a definition of variable V
• Which blocks B need a $\Phi$ 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 $\Phi$ 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
  - $\text{DF}(A) = \text{set of blocks } B \text{ where } A \text{ does not strictly dominate } B,$
    and some predecessor of $B$ is dominated by $A$.
- $\text{DF}(1)\,?$
- $\text{DF}(5)\,?$
Where to Insert $\Phi$ Functions: Iterated Dominance Frontier

- For a given variable $v$, let $\text{defs}(v)$ be the set of blocks that define $v$.
  - Ex: $\text{defs}(X) = \{1, 3\}$.
- Insert $\Phi$ functions at $\bigcup_{A \in \text{defs}(v)} \text{DF}(A)$.
  - Defn: $\text{DF}(S) = \bigcup_{A \in S} \text{DF}(A)$ for a set of blocks $S$.
  - Ex: $\text{DF}(\text{defs}(X)) = \{2, 3, 7\}$.
- These are new definitions!
- Insert $\Phi$ functions at $\text{DF}(\text{DF}(\text{defs}(v))), \text{DF}(\text{DF}(\text{DF}(\text{defs}(v)))), \ldots$
  - Define $\text{DF}_1(S) = \text{DF}(S)$
    $\text{DF}_{i+1}(S) = \text{DF}(S \cup \text{DF}_i(S))$
  - Iterated dominance frontier $\text{DF}^+(S)$ is the limit $\text{DF}_\infty(S)$.
  - Ex: $\text{DF}^+(\text{defs}(X)) = \{2, 3, 7, 9\}$.

Note: we are only deciding where to insert $\Phi$ functions; we fill in the operands later.
SSA Form Overview

• Converting to SSA form:
  o Insert $\Phi$ functions
  o Assign unique names to definitions
  o Propagate definitions to uses (including operands in $\Phi$ functions)

• Perform optimizations...

• Converting out of SSA form:
  o For each $\Phi$ function, insert a copy in predecessors
  o Remove $\Phi$ 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

---

Sparse Conditional Constant Propagation

- Standard constant propagation cannot deduce that the statement 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 for 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. $g(a) = c \land f(g(a)) \neq f(c)$
  - 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 $F \land \neg A$ 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     int i = 0;
3     while (true) {
4         int next;
5             next = data[i];
6             if (!(i < next && next < N)) return;
7             i = i + 1;
8             for (; i < next; i = i + 1) {
9                 if (data[i] == cookie)
10                     i = i + 1;
11                 else
12                     Process(data[i]);
13             }
14         }
15     }
16 }

One execution path (SSA)

\[
F = \begin{cases} 
3 \ i_1 = 0; \\
7 \ next_1 = data_0 [i_1]; \\
8 \ i_1 < next_1 \&\& \ next_1 < N_0 \\
9 \ i_2 = i_1 + 1; \\
10 \ i_2 < next_1; \\
11 \ data_0 [i_2] = cookie_0; \\
12 \ i_3 = i_2 + 1; \\
13 \ i_4 = i_3 + 1; \\
14 \ ! (i_4 < next_1); \\
15 \ next_2 = data_0 [i_4];
\end{cases}
\]

Line 7: Array bound assertion \( A: \)

\[(0 \leq i_1 \land i_1 < N_0)\]
Checking for Out-of-Bound Array Access (Line 7, \textit{iteration 1})

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

```c
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)**

\[
F = \begin{cases}
3 & i_1 = 0; \\
7 & \text{next}_1 = \text{data}_0 [i_1]; \\
8 & i_1 < \text{next}_1 \land \text{next}_1 < N_0 \\
9 & i_2 = i_1 + 1; \\
10 & i_2 < \text{next}_1; \\
11 & \text{data}_0 [i_2] = \text{cookie}_0; \\
12 & i_3 = i_2 + 1; \\
13 & i_4 = i_3 + 1; \\
14 & !(i_4 < \text{next}_1); \\
15 & \text{next}_2 = \text{data}_0 [i_4]; \\
\end{cases}
\]

**Line 7: Array bound assertion \( A \):**

\[
(0 \leq i_1 \land i_1 < N_0)
\]

**1st execution of Line 7**

Check: Is \( F \land \neg A \) satisfiable?

\[
i_1 = 0 \land \neg (0 \leq i_1 \land i_1 < N_0)
\]
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 }

One execution path (SSA)

Line 7: Array bound assertion $A$:

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

⇒ maps to

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

1st execution of Line 7
Check: Is $F \land \neg A$ satisfiable?

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

Yes! $\{ i_1 \mapsto 0, \ N_0 \mapsto 0 \} \ \text{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 }

Line 7: Array bound assertion $A$:

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

One execution path (SSA)

$F = \langle$

3 $i_1 = 0;$

7 $next_1 = data_0 [i_1]$;

8 $i_1 < next_1 \land next_1 < N_0$

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 $next_2 = data_0 [i_4]$;

$2^{nd}$ execution of Line 7

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

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

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 }

Line 7: Array bound assertion $A$:

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

One execution path (SSA)

$F = \uparrow$

3 $i_1 = 0$;
7 $next_1 = data_0[i_1]$;
8 $i_1 < next_1 \land next_1 < N_0$
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 $i_4 = i_3 + 1$;
7 $next_2 = data_0[i_4]$;
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:** $\varphi$ functions in SSA

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

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

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

M. Lam
CS243: SMT
Applying the Resolution Rule to Example

- Is $F \land \neg A$ satisfiable?
  \[
  \varphi_1 = (i_0 > 0) \land (\varphi_1 \rightarrow c_3 = 5) \land (\neg \varphi_1 \rightarrow c_3 = 5) \land (c_3 \neq 5)
  \]

- Recall: $p \rightarrow q \equiv \neg p \lor q$
  \[
  \varphi_1 = (i_0 > 0) \land (\neg \varphi_1 \lor c_3 = 5) \land (\varphi_1 \lor c_3 = 5) \land (c_3 \neq 5)
  \]

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

Resolution rule in propositional logic:

\[
\text{Given } p \lor A \text{ and } \neg p \lor B, \text{ add the resolvent } A \lor B
\]

\[
\begin{array}{c}
\text{Resolve} \\
\hline
p \lor A & \neg p \lor B \\
\hline
A \lor B
\end{array}
\]
Loops

• Optimistically: Unroll two times

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

```c
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

1 if (i < next) {
2   if (data[i] == cookie)
3     i = i + 1;
4   else
5     Process(data[i]);
6
7   i = i + 1;
8
9   if (i < next) {
10      if (data[i] == cookie)
11         i = i + 1;
12      else
13         Process(data[i]);
14
15     i = i + 1;
16 }  
17 }

1 φ₁ = (i₀ < next₀);
2 φ₂ = (data₀ [i₀] == cookie₀);
3 i₁ = i₀ + 1;
4
5
6 i₂ = φ₂ ? i₁ : i₀;
7 i₃ = i₂ + 1;
8
9 φ₃ = (i₃ < next₀);
10 φ₄ = (data₀ [i₃] == cookie₀);
11 i₄ = i₃ + 1;
12
13
14 i₅ = φ₄ ? i₄ : i₃;
15 i₆ = i₅ + 1;
16 i₇ = φ₃ ? i₆ : i₃;
17 i₈ = φ₁ ? i₇ : i₀;

Unrolling the loop twice finds many (but not) all bugs
# Major Categories of Program Analysis Tools

<table>
<thead>
<tr>
<th>Complete (Small programs)</th>
<th>Static Property Based</th>
<th>Dynamic Execution Based</th>
</tr>
</thead>
</table>
| **Verification**          | *Prove a property in a program*  
  *Floyd-Hoare logic:*  
  \{pre-condition\} s \{post-condition\} | *(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.* |
| Applicable to small programs | Symbolic: many states all at once |

<table>
<thead>
<tr>
<th>Incomplete (Large programs)</th>
<th>Static Analysis (e.g. Data flow)</th>
<th>Test case generation (e.g. SMT)</th>
</tr>
</thead>
</table>
| 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* | *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?