Static Analysis: Overview, Data-Flow
Static analysis

• Purpose
  ▪ Gather information about run-time behavior of programs without executing them

• Information
  ▪ Does the variable x have a constant value?
  ▪ Is the value of the variable x always positive?
  ▪ May the pointer p be null at a code location?
  ▪ What are possible values of the variable y?
Static analysis: characteristics

• Target model of program behavior
  ▪ some kind of *Control Flow Graph (CFG)*

• Provides *approximate* answers
  ▪ Decision problems: yes / no / don’t know
  ▪ Collecting some values: superset / subset

• Information valid for all possible runs

• Summarizing different execution paths
  ▪ branches of the *if-else* statement, loop iterations

• Does not know run-time values (inputs)
## Comparison

<table>
<thead>
<tr>
<th>Static analysis</th>
<th>Model checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>control-flow graph</td>
<td>program state space</td>
</tr>
<tr>
<td>summarizes information from different paths</td>
<td>reasons about execution paths independently</td>
</tr>
<tr>
<td>approximation</td>
<td>path-sensitivity</td>
</tr>
<tr>
<td>scalability</td>
<td>precision</td>
</tr>
</tbody>
</table>
Static analysis in practice

- Optimizing compilers
  - Detect superfluous evaluations of the same expression
  - Detect unused local variables or dead code fragments

- Program verification
  - Search for possible runtime errors
    - Example: null pointer dereference, unsynchronized access
  - Constructing abstraction for model checking
    - Slicing: identify statements irrelevant for a given property
Q: What important restrictions there are?
Restrictions

• Approximation must be safe
  ▪ That precisely means “imprecise on the safe side”

• Target domain: optimizing compilers
  ▪ Under-approximation
    • Optimization performed on the basis of analysis results must not violate semantics of a given program
  ▪ Example: constant propagation
    • Sound analysis identifies a program variable as a constant only when it is really certain (100%)
Restrictions

- Approximation must be safe
  - That precisely means “imprecise on the safe side”

- Target domain: search for errors
  - Over-approximation
    - Safe analysis reports all real errors and also some spurious errors (false positives)
  - Example: possible null dereferences
    - We want to know about all of them so we can add runtime checks (if (v != null) ...)
Basic concepts (theory and examples)
**Running example**

- **Program**
  ```c
  int factorial(int n) {
    int r;
    if (n == 0) r = 0;
    int f = 1;
    while (n > 0) {
      f = f * n;
      n = n - 1;
      if (n == 0) r = f;
    }
    return r;
  }
  ```

- **Static analysis**: possibly uninitialized variables
Control flow graph (CFG)

- Directed graph with labels
- Nodes: program points (statements)
- Edges: possible flow of control
  - \( \text{pred}(n) \) and \( \text{succ}(n) \) for each node \( n \) in a CFG
- Single point of entry
- Single point of exit
sequence
$S_1;S_2$

if (E) \{ S \}

if (E) \{ S_1 \}
else \{ S_2 \}

while (E) \{ S \}
• Set of possible values (facts)

• Finite lattice over the set
Partial order

- Mathematical structure $L = (S, \subseteq)$
  - $S$ is a set of values (e.g., analysis facts)
  - $\subseteq$ is a binary relation (e.g., is-subset)
    - Reflexivity: $\forall x \in S : x \subseteq x$
    - Transitivity: $\forall x, y, z \in S : x \subseteq y \land y \subseteq z \Rightarrow x \subseteq z$
    - Anti-symmetry: $\forall x, y \in S : x \subseteq y \land y \subseteq x \Rightarrow x = y$

- Examples
Bounds

Let's have a partial order \( L = (S, \subseteq) \) and \( X \subseteq S \)

- **Upper bound**
  - \( y \in S \) is an upper bound for \( X \), i.e. \( X \subseteq y \), if \( \forall x \in X : x \subseteq y \)

- **Lower bound**
  - \( y \in S \) is a lower bound for \( X \), i.e. \( y \subseteq X \), if \( \forall x \in X : y \subseteq x \)

- **Least upper bound of** \( X \), denoted as \( \sqcup X \)
  - \( X \subseteq \sqcup X \land \forall y \in S : X \subseteq y \Rightarrow \sqcup X \subseteq y \)

- **Greatest lower bound of** \( X \), denoted as \( \sqcap X \)
  - \( \sqcap X \subseteq X \land \forall y \in S : y \subseteq X \Rightarrow y \subseteq \sqcap X \)
Bounds: example 1

Let's have a partial order \( L = (S, \sqsubseteq) \) and the set \( S = \{a, b, c, d, e\} \)

The upper bounds of \( X = \{a, b\} \) are the elements \( \{c, e\} \)
Bounds: example 2

Let's have a partial order $L = (S, \sqsubseteq)$ and the set $S = \{a, b, c, d, e\}$

The greatest lower bound of $X = \{b, e\}$ is the element $b$
Lattice

- Partial order $L = (S, \sqsubseteq)$ such that
  - $\sqcup X$ and $\sqcap X$ exist for $\forall X \subseteq S$
  - Unique greatest element $\top = \sqcup S = \sqcap \emptyset$
  - Unique least element $\bot = \sqcap S = \sqcup \emptyset$

- Height of a lattice
  - Length of the longest path from $\bot$ to $\top$
Finite lattice

- Partial order \( L = (S, \sqsubseteq) \) such that
  - \( \forall x, y \in S \) there is
    - Least upper bound \( x \sqcup y \)
    - Greatest lower bound \( x \sqcap y \)
Lattice: examples
Lattice $L = (S, \sqsubseteq)$
- Set $S$ of analysis facts (units of information)
- Relation $\sqsubseteq$ defines an ordering with respect to precision of the abstraction
  - $x \sqsubseteq y \Rightarrow x$ is more precise than $y$
  - $x \sqsubseteq y \Rightarrow y$ approximates $x$

Example
- Sign abstraction: $x = \{ \text{POS} \}$, $y = \{ \text{POS, ZERO} \}$
How to construct lattices

- Finite set $R$ induces a lattice $(2^R, \sqsubseteq)$
  - $\perp = \cup \emptyset$
    - No information available
  - $\top = R$
    - Any possible value
  - $x \sqcup y = x \cup y$
  - $x \sqcap y = x \cap y$
  - Height $|R|$

- Example
  - Set $R = \{0, 1, 2\}$
  - Height = 3

\[ T = \{0,1,2\} \]
\[ \perp = \{\} \]
\[ \top = \{0,1,2\} \]
\[ \{0,1\} \quad \{0,2\} \quad \{1,2\} \]
\[ \{0\} \quad \{1\} \quad \{2\} \]
\[ \perp = \{\} \]
Running example

- Program
  ```c
  int factorial(int n) {
    int r;
    if (n == 0) r = 0;
    int f = 1;
    while (n > 0) {
      f = f * n;
      n = n - 1;
      if (n == 0) r = f;
    }
    return r;
  }
  ```

- Static analysis: possibly uninitialized variables
Encoding program statements

- Data for each node in the CFG
  - IN: valid before the program statement
  - OUT: valid after the program statement

- Merge operator $\sqcup$
  - CFG nodes with multiple predecessors
  - Typical approach: union or intersection

- Transfer functions
Transfer functions

- For each node in CFG (statement), we must define a transfer function

\[ \text{OUT} = (\text{IN} \setminus \text{kill}) \cup \text{gen} \]

- Examples
  - Statement \( \text{int } r; \)
    
    \[ \text{kill} = \{\}, \text{gen} = \{ r \} \]
  - Statement \( r = f; \)
    
    \[ \text{kill} = \{ r \}, \text{gen} = \{\} \]
Monotone functions

- Function $f : S \rightarrow S$ is monotone if
  - $\forall x, y \in S : x \subseteq y \Rightarrow f(x) \subseteq f(y)$

- Examples
  - Constant functions
  - Operators $\sqcap$ and $\sqcup$
  - Their compositions
Computing static analysis

- **Input**
  - Control flow graph of the given program
  - Initial value for each CFG node (⊥ or ∅)
    - Value is the set of known analysis facts (information)
  - Merge operator defined as the set union
  - Transfer functions $F_i$ for each node in CFG

- **Approach:** *compute fixed points*
  - Information associated with the CFG nodes
Duality

\((S, \sqsubseteq)\) is a lattice ⇔ \((S, \sqsupseteq)\) is a lattice

\[
\begin{align*}
\bigsqcup_{(S, \sqsubseteq)} &= \bigsqcap_{(S, \sqsupseteq)} \\
\bigsqcap_{(S, \sqsubseteq)} &= \bigsqcup_{(S, \sqsupseteq)}
\end{align*}
\]

\(\top_{(S, \sqsubseteq)} = \bot_{(S, \sqsupseteq)}\)

\(\bot_{(S, \sqsubseteq)} = \top_{(S, \sqsupseteq)}\)

• We focus just on \(\sqsubseteq\) and initial values \(\bot\)
Computing fixed points

• Motto: “walk up the lattice starting at \( \perp \), until you reach a fixed point”
  ▪ In the worst case, \( \top \) is the fixed point (if exists)

• Three algorithms
  ▪ Naive (brute force)
  ▪ Chaotic iteration
  ▪ Worklist algorithm
Worklist algorithm

\[ u_1 = \perp; \ldots, u_n = \perp; \]
\[ q = [1, \ldots, n]; \]
\[ \text{while} \ (q \neq []) \{ \]
\[ \quad i = \text{head}(q); \]
\[ \quad v_{\text{IN}} = \text{merge}(\text{pred}(i)); \]
\[ \quad v_{\text{OUT}} = F_i(v_{\text{IN}}); \]
\[ \quad q = \text{tail}(q); \]
\[ \quad \text{if} \ (v_{\text{OUT}} \neq u_i) \{ \]
\[ \quad \quad \text{append}(q, \text{succ}(i)); \]
\[ \quad \quad u_i = v_{\text{OUT}}; \]
\[ \quad \} \]
\[ \} \]
Classification
Static analysis categories

- Data-flow analysis
- Call graph construction
- Pointer analysis (aliasing)
- Escape analysis (threads)
- Side effect analysis
Data-flow analysis

- Available expressions
- Reaching definitions
- Live variables (values)
Available expressions

```javascript
var x, y, a, b;
y = a - b;
while (y < a + b) {
    a = a - 1;
    x = a + b;
}
```

```javascript
var x, y, a, b, t;
y = a - b;
t = a + b;
while (y < t) {
    a = a - 1;
    t = a + b;
    x = t;
}
```
Direction

- **Forward analysis**
  - Computes information about the past behavior
  - Starts at the entry node (CFG) and goes forward

- **Backward analysis**
  - Computes information about the future behavior
  - Starts at the exit CFG node and moves backwards
**May analysis**
- Computes information that **may be true** (over-approximation)
  - Information for P that is true at least for one path coming into P
- Merge operator: set union

**Must analysis**
- Computes information that **must be true** (under-approximation)
  - Information for P that is true for all execution paths coming into P
- Merge operator: set intersection
Flow sensitivity

- Flow-sensitive analysis
  - Considers the program’s control flow (CFG) and the order of individual statements
  - Example: available expressions

- Flow-insensitive analysis
  - Program seen as an unordered collection of statements
  - Results are valid for any order of program statements
    - $S1 ; S2$ versus $S2 ; S1$
  - Example: type analysis (inference)
Scope

• Intra-procedural
  - Every single procedure analyzed separately
  - Maximally pessimistic assumptions about side effects of procedure calls

• Inter-procedural
  - Whole program analyzed together
  - Sometimes without libraries (huge)
Context sensitivity

- Context-sensitive analysis
  - Call site: source code location for the call
  - Call stack: procedure calls and returns
  - Receiver objects for method calls ("this")
  - Analysis results for the method M depend on the specific caller of M

- Context-insensitive analysis
  - Same analysis results for every call site of M
Tools

- WALA
  - Java, JavaScript, JVM (bytecode)
  - [https://wala.github.io/](https://wala.github.io/)

- Soot
  - Java, JVM-based languages (bytecode)

- CIL
  - Only for programs written in C
  - [http://www.cs.berkeley.edu/~necula/cil/](http://www.cs.berkeley.edu/~necula/cil/)
  - [https://github.com/cil-project/cil](https://github.com/cil-project/cil)

- LLVM
  - C, C++, Objective-C
  - Clang static analyzer
  - [http://llvm.org/](http://llvm.org/)
Further reading

- M. Schwartzbach. *Lecture Notes on Static Analysis*. Department of CS, Aarhus University