CS 520
Theory and Practice of Software Engineering
Fall 2022

Data flow analysis

October 25, 2022
Ways to get your code right

• Verification & Validation
  – Purpose is to uncover problems and increase confidence
  – Combination of manual and automated reasoning (e.g., data flow analysis, model checkers) as well as testing

• Debugging
  – Purpose is finding out why a program is not functioning as intended
  – Pinpoint location + cause of problem

• Defensive programming
  – Programming with validation and debugging in mind
Dynamic versus static analysis

• **Analysis:** Check for desirable (or undesirable) behaviors of a given program
  – e.g., data races, deadlock

• **Dynamic:** Automate the check by executing that program on selected user inputs
  – e.g., run-time assertions, testing, model inference

• **Static:** Automate the check by essentially considering all possible executions of that program on all possible user inputs
  – e.g., **data flow analysis**, model checking
Data flow analysis (DFA)

- All possible program executions are conservatively modeled as paths through a CFG (Control Flow Graph)

- Facts about the program execution states (often variable values) are associated with the nodes
Data flow analysis (DFA)

• Propagates the facts through the nodes

• Since there are a finite number of nodes and facts, the propagation will terminate and the analysis results can be determined
Possible uses of DFA

• Inspections
• Refactoring
• Compiler optimizations
• Security taint analysis
• Model checking
• ...

6
Control Flow Graphs (CFGs)

Directed graph represents all possible paths of execution through a given program

- Each node represents a program statement (or block of program statements)
- Each edge represents control flow among the nodes
Representing variable values

• A **variable** corresponds to an array element, class field, or parameter

• A **definition** writes a value to a variable

• A **reference** reads a value from a variable

Example 1: \( x = \text{foo}(); \)

Example 2: \( x = x + y; \)
Representing variable values

• A **variable** corresponds to an array element, class field, or parameter
• A **definition** writes a value to a variable
• A **reference** reads a value from a variable

Example 1: \texttt{x = foo(); // def(n1) = \{ x \}, ref(n1) = \{ \}}

Example 2: \texttt{x = x + y; // def(n5) = \{ x \}, ref(n5) = \{x, y\}}
All possible paths

• Start at an initial node. Propagate from each current node to its next node.
  – e.g., [ 1, 2, 3, 4 ], [ 1, 2, 3, 5, 4 ]

• Since a conservative model, each actual execution will correspond to at least one path.

• Since an abstracted model (e.g., array elements, unknown values), each path may not correspond to an actual execution.
Example 1: Reaching definitions

**Goal:** A given definition, d, reaches a node n if there is a def-clear path from d to n

**Possible use:** Compiler error for uninitialized variable
Example 1: Initial facts

- For each node n, compute def(n):
  1: \{ x_1 \}
  2: \{ y_2 \}
  3: \{ \}
  4: \{ \}
  5: \{ x_5 \}
Example 1: Generating new facts

1. Start with the reaching definitions preceding node n
2. Remove any redefinitions at n
3. Add any new definitions at n

Example: Node 5
1. \{ x_1 \}
2. \{ \}
3. \{ x_5 \}
Example 1: Uninitialized variables

• Consider each variable v referenced by node n

• Check if the reaching definitions of n contains v

• If not, report a compiler error for an uninitialized variable
Example 1: Possible use

```java
String fullDocumentation;
try {
    Annotator annotator = FactoryFactory.createASTNodeAdapterFactory().wrapAsAnnotator(step.get
    Object ann = annotator.getAnnotations().get("documentation.fullDocumentation");
    if(ann instanceof Documentation) {
        fullDocumentation = ((Documentation) ann).getText();
        //System.out.println("fullDocSection: "+ fullDocumentation);
    }
} catch(NoSuchAdapterForNodeException e) {
    fullDocumentation = null;
} catch(NullPointerException e) {
    fullDocumentation = null;
} catch (Exception sce) {
    // The getAnnotations above wrapped the java.io.StreamCorruptedException
    // in a RuntimeException.
    // Visual-JIL may have corrupted the full documentation annotations.
    // This skips over such annotations.
    System.err.println("WARNING: Cannot retrieve documentation for step "+ step.getName());
    fullDocumentation = null;
}
if ((fullDocumentation != null) && (!fullDocumentation.trim().isEmpty())) {
    notes = fullDocumentation;
}
return
```
Example 2: Live variables

**Goal:** A variable, x, is live at node n if there exists a def-clear path wrt x from node n to a reference of x

**Possible use:** Register allocation
Example 2: Initial facts

• For each node n, compute def(n) and ref(n):
  1: { x } and { }
  2: { y } and { x }
  3: { } and { x }
  4: { } and { }
  5: { x } and { x, y }
Example 2: Generating new facts

1. Start with the live sets succeeding node n
2. Remove def(n)
3. Add ref(n)

Example: Node 5
4. { }
5. { }
6. { x, y }
DFA framework

• Initial facts: GEN and KILL sets

• Generating new facts: OUT sets

• Merging facts: IN sets
DFA framework (cont.)

• Compute facts for each node of a control-flow graph
  – The IN and OUT sets
  – Depends on the direction facts are propagated
Initial facts: GEN and KILL sets

• For each node $i$ associate sets
  – $\text{GEN}(i)$ - what is to be added (generated)
  – $\text{KILL}(i)$ - what is to be eliminated (killed)

• The definitions of GEN and KILL depend on the problem that is being solved.

• Often the GEN and KILL sets can be derived from the abstract syntax tree
  – E.g., variables defined in a node variables referenced in a node
Generating new facts: OUT sets

• For each node we have an equation of the form:
  – $\text{OUT}(n) := f(\text{IN}(n))$

• Transfer function $f$ usually depend on GEN and KILL information that is computed for each node
  – Usually: $\text{OUT} := (\text{IN} \setminus \text{KILL}) \cup \text{GEN}$
### Merging facts: IN sets

<table>
<thead>
<tr>
<th></th>
<th>Forward-flow for $i \neq \text{initial}$</th>
<th>Backward-flow for $i \neq \text{final}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any-path</td>
<td>$\text{IN}(i) = \bigcup_{j \in \text{preds}(i)} \text{OUT}(j)$</td>
<td>$\text{IN}(i) = \bigcup_{j \in \text{succs}(i)} \text{OUT}(j)$</td>
</tr>
<tr>
<td>All-path</td>
<td>$\text{IN}(i) = \bigcap_{j \in \text{preds}(i)} \text{OUT}(j)$</td>
<td>$\text{IN}(i) = \bigcap_{j \in \text{succs}(i)} \text{OUT}(j)$</td>
</tr>
</tbody>
</table>
Termination: Fixed point

• Compute the new IN and OUT values for each node, until all the values stabilize

• Reaches a fixed point if:
  – there are only a finite number of possible sets that can be associated with a node, and
  – the function that determines the sets that can be associated with a node is monotonic (defined by a meet semilattice)
Example Meet Semilattice

Moves values down the lattice, from top (T) to bottom (⊥)

\{n\} \cup \{s,x\} = \{n,s,x\}

values = \text{PowerSet}(\{i,n,s,x\})
T = \{\}\n⊥ = \{i,n,s,x\}
Ordering = \subset
Meet = \cup
Example 1: Reaching definitions

- **Initialization:** Each node \( n \) is associated with \( \text{def}(n) \)
- **Propagation:** Forward (\( \text{pred} \))
- **Merge op:** Union (\( U \))

\[
\text{In}(n) = U \ k \in \text{pred} \ \text{Out}(k)
\]

\[
\text{Out}(n) = \text{In}(n) - \text{def}(n) \cup \text{def}(n_n)
\]
Example 2: Live variables

- Initialization: Each node \( n \) is associated with \( \text{def}(n) \) and \( \text{ref}(n) \)
- Propagation: Backward (\( \text{succ} \))
- Merge op: Intersection (\( \cap \))

\[
\text{In}(n) = \cap k \in \text{succ}(n) \text{ Out}(k)
\]

\[
\text{Out}(n) = \text{In}(n) - \text{def}(n) \cup \text{ref}(n)
\]
Formulating DFA Problems

• Need to determine the facts that should be computed at a node
• Need to determine how those facts should flow from node to node
  – Backward or Forward
  – Union or Intersection
• Often there is more than one way to solve a problem
  – Can often be solved forward or backward, but usually one way is easier than the other
Security: Taint checking

Taint checking is a feature in some computer programming languages, such as Perl,\(^1\) Ruby\(^2\) or Ballerina\(^3\) designed to increase security by preventing malicious users from executing commands on a host computer. Taint checks highlight specific security risks primarily associated with web sites which are attacked using techniques such as SQL injection or buffer overflow attack approaches.
Basic Stats app (Version 1)

- **src/**
  - BasicStats
  - BasicStatsApp
- **test/**
  - BasicStatsTests (3)

**Issues:**
- Simple architecture
- Poor design
- Violates best programming practices

```
$ git clone https://github.com/LASER-UMASS/basic-stats -b v2.0.0
```
Apply the MVC (Model View Controller) architecture pattern

Separates data representation (Model), visualization (View), and client interaction (Controller)
Basic Stats app (Version 2)

- src/
  - BasicStatsApp
  - gui/ // Views and Controllers
    - BasicStats
    - BasicStatsGUI // Should be Composite
    - view/
      - View // Should be Component
  - model/
    - BasicStatsModel
- test/
  - BasicStatsTests (3 old + 9 new = 12)

Goals:
- Architecture: MVC
- OO design principles and patterns: Composite
- Satisfies more best programming practices:
  - Understandability with documentation
  - Modularity with architecture and design patterns
  - Extensibility
  - Testability

git clone https://github.com/LASER-UMASS/basic-stats -b v3.0.0