Contents

1 ASTs  4
Program Analysis Tools

2 Data Flow Analysis

3 Static Analysis Tools
   3.1 Style and Anomaly Checking
   3.2 Reverse Compilers & Obfuscators

4 Dynamic Analysis Tools
   4.1 Pointer/Memory Errors
   4.2 Profilers
Program Analysis Tools

Analysis Tools

• Static Analysis
  – style checkers
  – data flow analysis

• Dynamic Analysis
  – Memory use monitors
  – Profilers

..........................
Program Analysis Tools

Analysis Tools and Compilers
Analysis tools, particularly static, share a great deal with compilers

• Need to parse code & perform limited static analysis
  – Generally working from ASTs
  – Some exceptions (working from object code or byte code)

• Data flow techniques originated in compiler optimization

1 ASTs

Abstract Syntax Trees
Program Analysis Tools

- Output of a language parser
  - Simpler than parse trees
- Generally viewed as a generalization of operator-applied-to-operands

Abstract Syntax Trees (cont.)
Program Analysis Tools

- ASTs can be applied to larger constructions than just expressions
- In fact, generally reduce entire program or compilation unit to one AST

Abstract Syntax Trees (cont.)
Program Analysis Tools

Abstract Syntax Graphs

CS795
• Semantic analysis pairs uses of variables with declarations
  – Transforming the AST into an ASG
2 Data Flow Analysis

Data Flow Analysis

• All data-flow information is obtained by propagating data flow markers through the program.

• The usual markers are
  – $d(x)$: a definition of variable $x$ (any location where $x$ is assigned a value)
  – $r(x)$: a reference to $x$ (any location where the value of $x$ is used)
  – $u(x)$: an undefinition of $x$ (any location where $x$ becomes undefined/illegal)
Program Analysis Tools

Propagation of Markers

For each node (basic block) in the control flow graph, we define

- \( \text{gen}(n) \) = set of data-flow markers generated within node \( n \).
- \( \text{kill}(n) \) = set of data-flow markers killed within node \( n \).
- \( \text{in}(n) \) = set of data-flow markers entering node \( n \) from elsewhere.
- \( \text{out}(n) \) = set of data-flow markers leaving node \( n \) to go elsewhere.

The basic data flow problem is to find \( \text{in}() \) and \( \text{out}() \) for each node given the control flow graph and the \( \text{gen}() \) and \( \text{kill}() \) sets for each node.
Sample CFG
procedure SQRT (Q, A, B: in float; n0
  X: out float);
  // Compute X = square root of Q,
  // given that A <= X <= B
  X1, F1, F2, H: float;
begin
  X1 := A;
  X2 := B;
  F1 := Q - X1**2
  H := X2 - X1;
  while (ABS(H) >= 0.001) loop
    F2 := Q - X2**2;
    H := - F2 * ((X2-X1)/(F2-F1));
    X1 := X2;
    X2 := X2 + H;
    F1 := F2
  end loop;
  X := (X1 + X2) / 2.;
end SQRT;
Reaching Definitions

A definition $d_i(x)$ reached a node $n_j$ iff there exists a path from $n_i$ to $n_j$ on which $x$ is neither defined nor undefined.

The Reaching DF Problem

$gen(n) =$ set of definitions occurring in $n$ and reaching the end of $n$.

$kill(n) =$ set of all definitions $d_i(x)$ in the CFG such that $x$ is defined or undefined within $n$. 

CS795 13
\[ in(n) = \bigcup_{m \in pred(n)} out(m) \]

\[ out(n) = (in(n) - kill(n)) \cup gen(n) \]

Sample Nodes
Sample Nodes (kill)

CS795
Program Analysis Tools

\[
\text{kill}(n_0) = \{d_0(Q), d_0(A), d_0(B), d_1(X_1), d_1(X_2), \\
d_1(F1), d_1(H), d_3(F2), d_3(H), d_3(X_1), \\
d_3(X_2), d_3(F1), d_4(X)\}
\]

\[
\text{kill}(n_1) = \{d_1(X_1), d_1(X_2), d_1(F1), d_1(H), d_3(H), \\
d_3(X_1),\}
\]

\[
\text{kill}(n_2) = \{
\}
\]

\[
\text{kill}(n_3) = \{d_1(X_1), d_1(X_2), d_1(F1), d_1(H), d_3(F2), \\
d_3(H), d_3(X_1), d_3(X_2), d_3(F1)\}
\]

\[
\text{kill}(n_4) = \{d_4(X)\}
\]

\[
\text{kill}(n_5) = \{d_0(Q), d_0(A), d_0(B), d_1(X_1), d_1(X_2), \\
d_1(F1), d_1(H), d_3(F2), d_3(H), d_3(X_1), \\
d_3(X_2), d_3(F1)\}
\]
Solving for Reaching Defs
Solving iteratively, we start with $in(n) = out(n) = \{}$, and propagate definitions.

First Iteration:

\[
\begin{align*}
\text{in}(0) &= \{} \\
\text{out}(0) &= \text{gen}(0) \\
\text{in}(1) &= \text{gen}(0) \\
\text{out}(1) &= \text{gen}(0) \cup \text{gen}(1)
\end{align*}
\]

.........................

Iteration 1 (cont.)
Program Analysis Tools

\[\text{in}(2) = \text{gen}(0) \cup \text{gen}(1)\]
\[\text{out}(2) = \text{gen}(0) \cup \text{gen}(1)\]

\[\text{in}(3) = \text{gen}(0) \cup \text{gen}(1)\]
\[\text{out}(3) = \{d_0(Q), d_0(A), d_0(B), d_3(F2), d_3(H), d_3(X1), d_3(X2), d_3(F1)\}\]

\[\text{in}(4) = \text{gen}(0) \cup \text{gen}(1)\]
\[\text{out}(4) = \text{gen}(0) \cup \text{gen}(1) \cup \{d_4(X)\}\]

\[\text{in}(5) = \text{gen}(0) \cup \text{gen}(1) \cup \{d_4(X)\}\]
\[\text{out}(5) = \{d_4(X)\}\]
Iteration 2
Program Analysis Tools

\[
\begin{align*}
\text{in}(0) & = \text{unchanged} \\
\text{out}(0) & = \text{unchanged} \\
\text{in}(1) & = \text{unchanged} \\
\text{out}(1) & = \text{unchanged} \\
\text{in}(2) & = \text{gen}(0) \cup \text{gen}(1) \cup \{d_3(F2), d_3(H), d_3(X1), d_3(X2), d_3(F1)\} \\
\text{out}(2) & = \text{gen}(0) \cup \text{gen}(1) \cup \{d_3(F2), d_3(H), d_3(X1), d_3(X2), d_3(F1)\}
\end{align*}
\]
Iteration 2 (cont.)
\( in(3) = \text{gen}(0) \cup \text{gen}(1) \cup \{d_3(F2), d_3(H), d_3(X1), d_3(X2), d_3(F1), \} \)

\( out(3) = \text{unchanged} \)

\( in(4) = \text{gen}(0) \cup \text{gen1} \cup \{d_3(F2), d_3(H), d_3(X1), d_3(X2), d_3(F1), \} \)

\( out(4) = \text{gen}(0) \cup \text{gen1} \cup \{d_3(F2), d_3(H), d_3(X1), d_3(X2), d_3(F1), d_4(X)\} \)

\( in(5) = \text{gen}(0) \cup \text{gen1} \cup \{d_3(F2), d_3(H), d_3(X1), d_3(X2), d_3(F1), d_4(X)\} \)

\( out(5) = \text{unchanged} \)
Data Flow Anomalies

The reaching definitions problem can be used to detect anomalous patterns that may reflect errors.

- **ur anomalies**: if an undefinition of a variable reaches a reference of the same variable
- **dd anomalies**: if a definition of a variable reaches a definition of the same variable
- **du anomalies**: if a definition of a variable reaches an undefinition of the same variable
Available Expressions
An expression $e$ is *available* at a node $n$ iff every path from the start of the program to $n$ evaluates $e$, and iff, after the last evaluation of $e$ on each such path, there are no subsequent definitions or undefinitions to the variables in $e$.

The Available DF Problem
$\text{gen}(n) = \text{set of expressions evaluated in } n \text{ containing no variables subsequently defined or undefined within } n$.

$\text{kill}(n) = \text{set of all expressions in the program containing variables that are defined or undefined within } n$. 
Program Analysis Tools

\[ in(n) = \bigcap_{m \in \text{pred}(n)} \text{out}(m) \]

\[ \text{out}(n) = (in(n) - \text{kill}(n)) \cup \text{gen}(n) \]

.................................

Live Variables

A variable \( x \) is \textit{live} at node \( n \) iff there exists a path starting at \( n \) along which \( x \) is used without prior redefinition.

.................................

The Live Variable DF Problem

\( \text{gen}(n) = \text{set of variables used in } n \) without prior definition.
Program Analysis Tools

$kill(n) = \text{set of variables defined within } n.$

$$in(n) = gen(n) \cup (out(n) - kill(n))$$

$$out(n) = \bigcup_{m \in succ(n)} in(m)$$

Data Flow and Optimization
### Program Analysis Tools

<table>
<thead>
<tr>
<th>Optimization Technique</th>
<th>Data-Flow Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant Propagation</td>
<td>reach</td>
</tr>
<tr>
<td>Copy Propagation</td>
<td>reach</td>
</tr>
<tr>
<td>Elimination of Common Subexpressions</td>
<td>available</td>
</tr>
<tr>
<td>Dead Code Elimination</td>
<td>live, reach</td>
</tr>
<tr>
<td>Register Allocation</td>
<td>live</td>
</tr>
<tr>
<td>Anomaly Detection</td>
<td>reach</td>
</tr>
<tr>
<td>Code Motion</td>
<td>reach</td>
</tr>
</tbody>
</table>

#### 3 Static Analysis Tools
3.1 Style and Anomaly Checking

Lint

Perhaps the first such tool to be widely used, lint (1979) became a staple tool for C programmers.

Combines static analysis with style recommendations, e.g.,

- data flow anomalies
- potential arithmetic overflow
  - e.g., storing an int calculation in a char
- conditional statements with constant values
- potential = versus == confusion
Is there room for lint-like tools?

- **lint** was a response, in part, to the weak capabilities of early C compilers
- Much of what **lint** does is now handled by optimizing compilers
  - However compilers seldom do cross-module or even cross-function analysis

FindBugs

- Open source project from U.Md.
Program Analysis Tools

- Works on compiled Java bytecode
- Sample report
- Can be run via
  - GUI
  - `ant`
  - Eclipse
  - `maven`

What Bugs does FindBugs Find?
Program Analysis Tools

- “Bugs” categorized as
  - Correctness bug: an apparent coding mistake
  - Bad Practice: violations of recommended coding practices.
  - Dodgy: code that is “confusing, anomalous, or written in a way that leads itself to errors”

- Bugs are also given “priorities” (p1, p2, p3 from high to low)

- Bug list

..............................

PMD

CS795 32
Program Analysis Tools

- PMD, source analysis for Java, JavaScript, XSL
  - CPD, “copy-paste-detector” for many programming languages
- Works on source code
- Sample reports (PMD & CPD)
- Can be run via bii ant
  - maven
  - eclipse
PMD Reports

- Configured by selecting “rule set” modules
  - Otherwise, appears to lack categories & priorities
- Cross reference to source location

3.2 Reverse Compilers & Obfuscators

Reverse Compilers
a.k.a. “uncompilers”

- Generate source code from object code
Program Analysis Tools

• Originally clunky & more of a curiosity than usable tools
  – Improvements based on
    * “deep” knowledge of compilers (aided by increasingly limited field of available compilers)
    * Information-rich object codes (e.g., Java bytecode formats)

• Legitimate uses include
  – reverse-engineering
  – generating input for source-based analysis tools

• But also great tools for plagiarism

..........................................

CS795  35
Java and Decompilation

• Java is a particularly friendly field for decompilers
  – Rich object code format
  – Nearly monopolistic compiler suite

• Options for “protecting” programs compiled in Java:
  – gjc: compile into native code with a far less popular compiler
  – obfuscators

.........................
Java Obfuscators

Work by a combination of

- Renaming variables, functions, and classes to meaningless, innocuous, and very similar name sets
  - Challenge is to preserve those names of entry points needed to execute a program or applet or make calls upon a library’s public API
  - Stripping away debugging information (e.g., source code file names and line numbers associated with blocks of code)
  - Applying optimization techniques to reduce code size while also confusing the object-to-source mapping

Example, yguard
4 Dynamic Analysis Tools

Dynamic Analysis Tools
Not all useful analysis can be done statically

- Profiling
- Memory leaks, corruption, etc.
- Data structure abuse

Abusing Data Structures
Program Analysis Tools

• Traditionally, the C++ standard library does not check for common abuses such as over-filling and array or accessing non-existent elements
  – Various authors have filled in with “checking” implementations of the library for use during testing and debugging

• In a sense, the `assert` command of C++ and Java is the language’s own extension mechanism for such checks.

4.1 Pointer/Memory Errors

Memory Abuse

• Pointer errors in C++ are both common and frustrating
Program Analysis Tools

- Traditionally unchecked by standard run-time systems
  
  • Monitors can be added to help catch these
    
    - In C++, link in a replacement for malloc & free

How to Catch Pointer Errors

- Use *fenceposts* around allocated blocks of memory
  
  - check for unchanged fenceposts to detect over-writes
  
  - Check for fenceposts before a delete to detect attempts to delete addresses other than the start of an allocated block
Program Analysis Tools

- Add tracking info to allocated blocks indicating location of the allocation call
  - Scan heap at end of program for unrecovered blocks of memory
  - Report on locations from which those were allocated

- Add a “freed” bit to allocated blocks that is cleared when first allocated and set when the block is freed
  - Detect when a block is freed twice

Memory Analysis Tools

- Purify is a well-known commercial (pricey) tool
• At the other end of the spectrum, LeakTracer is a small, simple, but capable open source package that I’ve used for many years

  - Works with gcc/g++/gdb compiler suite

```bash
~/p/arc# ea/LeakTracer/leak-analyze ./arc
Gathered 8 (8 unique) points of data.
(gdb)
Allocations: 1 / Size: 36
0x80608e6 is in NullArcableInstance::NullArcableInstance(void) (Machine.cc:40).
39 public:
40         NullArcableInstance() : ArcableInstance(new NullArcable)
Allocations: 1 / Size: 8
0x8055b02 is in init_types(void) (Type.cc:119).
```
118 void init_types() {
119  Type::Integer = new IntegerType;

Allocations: 1 / Size: 132 (new[])
0x805f4ab is in Hashtable<NativeCallable, String, false, true>::Hashtable(unsigned int) (ea/h/Hashtable.h:15).
14 Hashtable (uint _size = 32) : size(_size), count(0) {
15  table = new List<E, own> [size];

4.2 Profilers

Profilers provide info on where a program is spending most of its execution time
Program Analysis Tools

- May express measurements in
  - Elapsed time
  - Number of executions

- Granularity may be at level of
  - functions
  - individual lines of code

- Measurement may be via
  - Probes inserted into code
  - Statistical sampling of CPU program counter register
Program Analysis Tools

Profiling Tools

• **gprof** for C/C++, part of the GNU compiler suite
  
  – Refer back to earlier lesson on statement and branch coverage
  
  – **gprof** is, essentially, the generalization of **gcov**

• **jvisualm** for Java, part of the Java SDK

• Provides multiple monitoring tools, including both CPU and memory **profiling**

.................................