Deducing Errors
Andreas Zeller

Next Week

- Tuesday: Q&A session 09:00 – 10:00
- Thursday: Andrzej on Mining Anomalies

Deducing Errors
Andreas Zeller
Obtaining a Hypothesis

- Problem Report
- Deducing from Code
- Earlier Hypotheses + Observations
- Observing a Run
- Learning from More Runs

Reasoning about Runs

- Experimentation
- Induction
- Observation
- Deduction

Reasoning about Runs
What's relevant?

10 INPUT X
20 Y = 0
30 X = Y
40 PRINT "X = ", X

Fibonacci Numbers

\[
fib(n) = \begin{cases} 
1, & \text{for } n = 0 \lor n = 1 \\
\text{fib}(n-1) + \text{fib}(n-2), & \text{otherwise}.
\end{cases}
\]

| 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 |

fibo.c

```c
int fib(int n)
{
    int f, f0 = 1, f1 = 1;
    while (n > 1) {
        n = n - 1;
        f = f0 + f1;
        f0 = f1;
        f1 = f;
    }
    return f;
}

int main()
{
    int n = 9;
    while (n > 0) {
        printf("fib(%d)=%d\n", n, fib(n));
        n = n - 1;
    }
    return 0;
}
```
Fibo in Action

$ gcc -o fibo fibo.c
$ ./fibo
fib(9)=55
fib(8)=34
...
fib(2)=2
fib(1)=134513905

Where does fib(1) come from?

Effects of Statements

- **Write.** A statement can change the program state (i.e. write to a variable)
- **Control.** A statement may determine which statement is executed next (other than unconditional transfer)

Affected Statements

- **Read.** A statement can read the program state (i.e. from a variable)
- **Execution.** To have any effect, a statement must be executed.
Effects in fibo.c

<table>
<thead>
<tr>
<th>Statement</th>
<th>Reads</th>
<th>Writes</th>
<th>Controls</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 fib(n)</td>
<td>n</td>
<td></td>
<td>1-10</td>
</tr>
<tr>
<td>1 int f</td>
<td>f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 f0 = 1</td>
<td>f0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 f1 = 1</td>
<td>f1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 while (n &gt; 1)</td>
<td>n</td>
<td>5-8</td>
<td></td>
</tr>
<tr>
<td>5 n = n - 1</td>
<td>n</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>6 f = f0 + f1</td>
<td>f0, f1</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>7 f0 = f1</td>
<td>f1</td>
<td>f0</td>
<td></td>
</tr>
<tr>
<td>8 f1 = f</td>
<td>f</td>
<td>f1</td>
<td></td>
</tr>
<tr>
<td>9 return f</td>
<td>f</td>
<td>&lt;ret&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Control Flow

int fib(int n) {
    int f, f0 = 1, f1 = 1;
    while (n > 1) {
        n = n - 1;
        f = f0 + f1;
        f0 = f1;
        f1 = f;
    }
    return f;
}

Control Flow Patterns

The CFG is best developed incrementally on an extra board.
Again, this is best developed interactively on the board (possibly by having the students call further dependences)

Again, this is best developed interactively on the board (possibly by having the students call further dependences)

Navigating along Dependences

Again, this is best developed interactively on the board (possibly by having the students call further dependences)
Program Slicing

- A slice is a subset of the program
- Allows programmers to focus on what’s relevant with respect to some statement S:
  - All statements influenced by S
  - All statements that influence S

Forward Slice

- Given a statement A, the forward slice contains all statements whose read variables or execution could be influenced by A
- Formally:
  \[ S^F(A) = \{ B | A \rightarrow^* B \} \]

Backward Slice

- Given a statement B, the backward slice contains all statements that could influence the read variables or execution of B
- Formally:
  \[ S^B(B) = \{ A | A \rightarrow^* B \} \]

Again, this is best developed interactively on the board (possibly by having the students call further dependences)
int main() {
    int a, b, sum, mul;
    sum = 0;
    mul = 1;
    a = read();
    b = read();
    while (a <= b) {
        sum = sum + a;
        mul = mul * a;
        a = a + 1;
    }
    write(sum);
    write(mul);
}

Slice Operations:
• Backbones
• Dices
• Chops

a = read();
b = read();
while (a <= b) {
    a = a + 1;
}
Dice

```c
sum = 0;
sum = sum + a;
write(sum);
```

- Contains only the difference between two slices
- Useful for focusing on differing behavior

Chop

- Intersection between a forward and a backward slice
- Useful for determining influence paths within the program

Leveraging Slices

(Complete code for the program is not shown, but it is mentioned that the slice is executable.)

Again, this is best developed interactively on the board (possibly by having the students call further dependences)
Deducing Code Smells

- Use of uninitialized variables
- Unused values
- Unreachable code
- Memory leaks
- Interface misuse
- Null pointers

Uninitialized Variables

$ gcc -Wall -o fibo fibo.c
fibo.c: In function `fib':
fibo.c:7: warning: `f' might be used uninitialized in this function

False Positives

```c
int go;
switch (color) {
    case RED:
    case AMBER:
        go = 0;
        break;
    case GREEN:
        go = 1;
        break;
}
if (go) { ... }
```
Unreachable Code

if (w >= 0)
    printf("w is non-negative\n");
else if (w > 0)
    printf("w is positive\n");

warning: will never be executed

Memory Leaks

int *readbuf(int size)
{
    int *p = malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        p[i] = readint();
        if (p[i] == 0)
            return 0;  // end-of-file
    }
    return p;
}

memory leak

Interface Misuse

void readfile()
{
    int fp = open(file);
    int size = readint(file);
    if (size <= 0)
        return;
    ...  
    close(fp);
}

stream not closed
Null Pointers

```c
int *readbuf(int size) {
    int *p = malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        p[i] = readint();
        if (p[i] == 0) // end-of-file
            return 0;
    }
    return p;
}
```

Findbugs

- Class implements Cloneable but does not define or use clone method
- Method might ignore exception
- Null pointer dereference in method
- Class defines equal(); should it be equals()?
- Method may fail to close database resource
- Method may fail to close stream
- Method ignores return value
- Unread field
- Unused field
Limits of Analysis

```c
int x;
for(i=j=k=1;--j||k;k=j?i%j?k:k-j:(j=i+=2));
write(x);
```

- Is x being used uninitialized or not?
- Loop halts only if there is an odd perfect number (= a number that's the sum of its proper positive divisors)

---

static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j > h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}

Conservative approximation:
any a[] depends on all a[]

---

Causes of Imprecision

- Indirect access, as in a[i]
- Pointers
- Functions
- Dynamic dispatch
- Concurrency
Risks of Deduction

- Code mismatch. Is the run created from this very source code?
- Abstracting away. Failures may be caused by a defect in the environment.
- Imprecision. A slice typically encompasses 90% of the source code.

Increasing Precision

- Verification. If we know that certain properties hold, we can leverage them in our inference process.
- Observation. Facts from concrete runs can be combined with deduction.

...in the weeks to come!

Concepts

- To reason about programs, use
  - deduction (0 runs)
  - observation (1 run)
  - induction (multiple runs)
  - experimentation (controlled runs)
Concepts (2)

★ To isolate value origins, follow back the dependences
★ Dependences can uncover code smells such as
  • uninitialized variables
  • unused values
  • unreachable code
★ Get rid of smells before debugging

Concepts (3)

★ To slice a program, follow dependences from a statement S to find all statements that
  • could be influenced by S (forward slice)
  • could influence S (backward slice)

Concepts (4)

★ Using deduction alone includes a number of risks, including code mismatch, abstracting away, and imprecision.
★ Any deduction is limited by the halting problem and must thus resort to conservative approximation.
★ For debugging, deduction is best combined with actual observation.
Next Week

• Tuesday: Q&A session 09:00 – 10:00
• Thursday: Andrzej on Mining Anomalies