Obtaining a Hypothesis

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

Experimentation
n controlled runs

Induction
n runs

Observation
1 run

Deduction
0 runs
Reasoning about Runs

Deduction
0 runs
What’s relevant?

10 INPUT X
20 Y = 0
30 X = Y
40 PRINT “X = “, X
Fibonacci Numbers

\[ \text{fib}(n) = \begin{cases} 
1, & \text{for } n = 0 \lor n = 1 \\
\text{fib}(n - 1) + \text{fib}(n - 2), & \text{otherwise} 
\end{cases} \]
```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><code>fib(n)</code></td>
<td></td>
<td><code>n</code></td>
<td>1-10</td>
</tr>
<tr>
<td><code>int f</code></td>
<td></td>
<td><code>f</code></td>
<td></td>
</tr>
<tr>
<td><code>f0 = 1</code></td>
<td></td>
<td><code>f0</code></td>
<td></td>
</tr>
<tr>
<td><code>f1 = 1</code></td>
<td></td>
<td><code>f1</code></td>
<td></td>
</tr>
<tr>
<td><code>while (n &gt; 1)</code></td>
<td><code>n</code></td>
<td></td>
<td>5-8</td>
</tr>
<tr>
<td><code>n = n - 1</code></td>
<td><code>n</code></td>
<td><code>n</code></td>
<td></td>
</tr>
<tr>
<td><code>f = f0 + f1</code></td>
<td><code>f0</code>, <code>f1</code></td>
<td><code>f</code></td>
<td></td>
</tr>
<tr>
<td><code>f0 = f1</code></td>
<td><code>f1</code></td>
<td><code>f0</code></td>
<td></td>
</tr>
<tr>
<td><code>f1 = f</code></td>
<td><code>f</code></td>
<td><code>f1</code></td>
<td></td>
</tr>
<tr>
<td><code>return f</code></td>
<td><code>f</code></td>
<td><code>&lt;ret&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>
Control Flow

```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;
}
```
Control Flow Patterns

while (COND)
  BODY

if (COND)
  THEN-BLOCK
  ELSE-BLOCK

for
  INIT;
  COND;
  INCR
  BODY;

while (COND)
  BODY
  do {
    BODY
  } while (COND);

for (INIT; COND; INCR)
  BODY;
A → B
Data dependency:
A's data is used in B;
B is data dependent on A

A → B
Control dependency:
A controls B's execution;
B is control dependent on A

Dependences

Entry: fib(n)

int f

int f0 = 1

int f1 = 1

while (n > 1)

n = n - 1

f = f0 + f1

f0 = f1

f1 = f

return f

Exit
Dependences

Following the dependences, we can answer questions like

- Where does this value go to?
- Where does this value come from?
Navigating along 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:
  \[ S^F(A) = \{ B \mid A \rightarrow^* B \} \]
Backward Slice

- Given a statement $B$, the backward slice contains all statements that could influence the read variables or execution of $B$.

- Formally:
  
  \[ S^B(B) = \{ A \mid A \rightarrow^* B \} \]
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

Backward slice of sum
Backward slice of mul
Backbone

- Contains only those statement that occur in both slices
- Useful for focusing on common behavior

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

Backward slice of sum
Backward slice of mul
Dice

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

```c
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);
}
```

(Note: This slice is executable!)
Deducing Code Smells

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

$ gcc -Wall -O -o fibo fibo.c
fibo.c: In function `fib':
fibo.c:7: warning: `f' might be used uninitialized in this function
int go;
switch (color) {
    case RED:  
    case AMBER:  
    go = 0;
    break;
    case GREEN:  
    go = 1;
    break;
}
if (go) { ... }
Unreachable Code

```c
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

```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)
            return 0;  // end-of-file
    }
    return p;
}
```

memory leak
void readfile()
{
    int fp = open(file);
    int size = readint(file);
    if (size <= 0)
        return;
    ...  
    close(fp);
}
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;
}
Uninitialized read of field in constructor

This constructor reads a field which has not yet been assigned a value. This is often caused when the programmer mistakenly uses the field instead of one of the constructor's parameters.
• Class implements Cloneable but does not define or use clone method

**Defect Patterns**

• Method might ignore exception

• Null pointer dereference in method

• Class defines `equals();` 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

• Unwritten 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)
- Problem is undediced yet
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!
To reason about programs, use

- deduction (0 runs)
- observation (1 run)
- induction (multiple runs)
- experimentation (controlled runs)
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
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.