Isolating Cause-Effect Chains

Andreas Zeller

---

double bug(double z[], int n) {
    int i, j;
    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}

---

What is the cause of this failure?

What do we do now?

What is the cause of this failure?
From Defect to Failure

1. The programmer creates a defect – an error in the code.
2. When executed, the defect creates an infection – an error in the state.
3. The infection propagates.
4. The infection causes a failure.

This infection chain must be traced back – and broken.

Tracing Infections

- For every infection, we must find the earlier infection that causes it.
- Program analysis tells us possible causes.
Real Code

- Opaque – e.g. third-party code
- Parallel – threads and processes
- Distributed – across multiple machines
- Dynamic – e.g. reflection in Java
- Multilingual – say, Python + C + SQL

Obscure Code

```c
struct foo {
    int tp, len;
    union {
        char        c[1];
        int         i[1];
        struct foo *p[1];
    }
}
```

And even if we know everything, there still is code which is almost impossible to analyze. In C, for instance, only the programmer knows how memory is structured; there is no general way for static analysis to find this out.

Isolating Input

In the last lecture, we have seen delta debugging on input.
Isolating States

Difference causes failure

Comparing States

- What is a program state, anyway?
- How can we compare states?
- How can we narrow down differences?

A Sample Program

$ sample 9 8 7
Output: 7 8 9

$ sample 11 14
Output: 0 11

Where is the defect which causes this failure?

Now let's take a deeper view. If a program is a succession of states, can't we treat each state as an input to the remainder of the run?

Let's look at a simpler example first.
```c
int main(int argc, char *argv[]) {
    int *a;

    // Input array
    a = (int *)malloc((argc - 1) * sizeof(int));
    for (int i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    // Sort array
    shell_sort(a, argc);

    // Output array
    printf("Output: ");
    for (int i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");
    free(a);
    return 0;
}
```

A sample state

- We can access the entire state via the debugger:
  1. List all base variables
  2. Expand all references...
  3. …until a fixpoint is found

Sample States

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc</td>
<td>4</td>
</tr>
<tr>
<td>argv[0]</td>
<td>&quot;./sample&quot;</td>
</tr>
<tr>
<td>argv[1]</td>
<td>&quot;9&quot;</td>
</tr>
<tr>
<td>argv[2]</td>
<td>&quot;8&quot;</td>
</tr>
<tr>
<td>argv[3]</td>
<td>&quot;7&quot;</td>
</tr>
<tr>
<td>i</td>
<td>1073834752</td>
</tr>
<tr>
<td>j</td>
<td>1074077312</td>
</tr>
<tr>
<td>h</td>
<td>1961</td>
</tr>
<tr>
<td>size</td>
<td>4</td>
</tr>
</tbody>
</table>

at shell_sort()
Narrowing State Diffs

■ = δ is applied, □ = δ is not applied

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7 8 9</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 11</td>
<td>✗</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0 11 14</td>
<td>✗</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7 11 14</td>
<td>?</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0 9 14</td>
<td>✗</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7 9 14</td>
<td>?</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0 8 9</td>
<td>✗</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 8 9</td>
<td>✗</td>
</tr>
</tbody>
</table>

Result

Since this worked so well, we built a debugging server.
Complex State

- Accessing the state as a table is not enough:
- References are not handled
- Aliases are not handled
- We need a richer representation

A Memory Graph
Structure

Construction

- Start with <root> node and base variables
- Base variables are on the stack and at fixed locations
- Expand all references, checking for aliases…
- …until all accessible variables are unfolded

Unfolding Memory

- Any variable: make new node
- Structures: unfold all members
- Arrays: unfold all elements
- Pointers: unfold object being pointed to
- Does p point to something? And how many?
Comparing States

• Basic idea: compute common subgraph
• Any node that is not part of the common subgraph becomes a difference
• Applying a difference means to create or delete nodes – and adjust references
• All this is done within GDB

Applying Diffs

$\delta_{15}$ creates a variable, $\delta_{20}$ deletes another
State of the GNU compiler (GCC)
42991 vertices
44290 edges - and 1 is wrong :-) 
An actual GCC execution has millions of these states.

Causes in State

Sane state
Infected state

The difference causes GCC to crash!

Search in Space

Infected state
Sane state

Test
Mixed state

?
Search in Space


Search in Space

Infected state

Sane state

Test

Mixed state
Search in Space

Infected state

Sane state

Test

Mixed state

Search in Time

Failing run

Passing run

link → fld[0].rtx → fld[0].rtx == link
Search in Time

Failing run
<PLUS node>
<PLUS node>
<Tree cycle>

Passing run

Capturing State for Python programs

```python
if __name__ == "__main__":
    sys.settrace(tracer)
...

def tracer(frame, event, arg):
    dump_stack(frame)
    return tracer
```
Capturing State for Python programs

```python
def dump_stack(frame):
    while frame is not None:
        dump_frame(frame)
        frame = frame.f_back

def dump_frame(frame):
    locals = frame.f_locals
    globals = frame.f_globals
    print locals, globals
```

Manipulating State for Python programs

```python
def dump_frame(frame):
    locals = frame.f_locals
    locals['a'] = 42
```

This is equivalent to assignment “a = 42” in frame.

Caveats

Python frame objects are translated back to internal frames only after tracer() has returned:

- Frames can be inspected at any time, but changed only in tracer()
- Output of variables during tracer() may inhibit their translation at return
Open Issues

• How do we capture an accurate state?
• How do we ensure the cause is valid?
• Where does a state end?
• What is the cost?
• When do we compare states? (next lecture)

Concepts

★ Delta Debugging on program states isolates a cause-effect chain through the run
★ Use memory graphs to extract and compare program states
★ Demanding, yet effective technique