Program Synthesis “is” Program Reachability
One-Slide Summary

• The template-based program synthesis problem asks if values can be found for template parameters such that the instantiated program passes all tests.

• The program reachability problem asks if values can be found for a set of program variables such that program execution reaches a given label.

• There is a constructive, polytime reduction between synthesis and reachability.
Program Repair via Synthesis

• Suppose we have a **buggy** program
  • It passes some tests and fails others
• Suppose we have **localized** the bug
  • We know which line is buggy
• Suppose we have a repair **template**
  • Fix is of the form “\( x = \square + \square(y, \square); \)”
• Can we fill in the template so that the program passes all of the tests?
Templated Program Syntax

\[
\text{cmd ::= skip} \\
| \text{cmd}_1 ; \text{cmd}_2 \\
| v := \text{aexp} \\
| ... \\
\text{aexp ::= aexp}_1 + \text{aexp}_2 \\
| \text{aexp}_1 - \text{aexp}_2 \\
| \boxed{\text{c}_i} \quad \text{Called a template parameter} \\
| ... \
\]
Template Instantation

• Given a templated program with template parameters $c_1 \ldots c_n$, and given template values $\overline{\bar{v}} = v_1 \ldots v_n$ (expressions or constants), we can instantiate, yielding a non-templated program.

• $\text{inst}(\text{skip}, \overline{\bar{v}}) \rightarrow \text{skip}$

• $\text{inst}(\text{cmd}_1; \text{cmd}_2, \overline{\bar{v}}) \rightarrow \text{inst}(\text{cmd}_1, \overline{\bar{v}}) ; \text{inst}(\text{cmd}_2, \overline{\bar{v}})$

• $\text{inst}(x = \text{aexp}, \overline{\bar{v}}) \rightarrow x = \text{inst}(\text{aexp}, \overline{\bar{v}})$

• $\text{inst}(\overline{c_i}, \overline{\bar{v}}) \rightarrow v_i$
Template-Based Program Synthesis

- **Given** a templated program $P$ with template parameters $c_1 \ldots c_n$, and a set $T$ of input-output pairs (tests), do there exist template values $\bar{v} = v_1 \ldots v_n$ such that for all $<\text{input, output}>$ pairs in $T$, $(\text{inst}(P, \bar{v}))(\text{input}) = \text{output}$?
Analysis

• How hard is it to solve program synthesis in general?
  • “Can you find values for these template variables such that this program passes all of its tests?”
1.1 Hello World

To illustrate the process of sketching, we begin with the simplest sketch one can possibly write: the "hello world" of sketching.

```c
harness void doubleSketch(int x){
    int t = x * ??;
    assert t == x + x;
}
```

The syntax of the code fragment above should be familiar to anyone who has programmed in C or Java. The only new feature is the symbol `??`, which is Sketch syntax to represent an unknown constant. The synthesizer will replace this symbol with a suitable constant to satisfy the programmer’s requirements. In the case of this example, the programmer’s requirements are stated in the form of an assertion. The keyword `harness` indicates to the synthesizer that it should find a value for `??` that satisfies the assertion for all possible inputs `x`.

**Flag `--bnd-inbits`** In practice, the solver only searches a bounded space of inputs ranging from zero to $2^{bnd-inbits} - 1$. The default for this flag is 5; attempting numbers much bigger than this is not recommended.

1.2 Running the synthesizer

To try this sketch out on your own, place it in a file, say `test1.sk`. Then, run the synthesizer with the following command line:

```
> sketch test1.sk
```

When you run the synthesizer in this way, the synthesized program is simply written to the console. If
1.1 Hello World

To illustrate the process of sketching, we begin with the simplest sketch one can possibly write: the "hello world" of sketching.

```plaintext
harness void doubleSketch(int x){
    int t = x; ??;
    assert t == x + x;
}
```

The syntax of the code fragment above should be familiar to anyone who has programmed in C or Java. The only new feature is the symbol `??`, which is Sketch syntax to represent an unknown constant. The synthesizer will replace this symbol with a suitable constant to satisfy the programmer’s requirements. In the case of this example, the programmer’s requirements are stated in the form of an assertion. The keyword `harness` indicates to the synthesizer that it should find a value for `??` that satisfies the assertion for all possible inputs `x`.

Flag `--bnd-inbits` In practice, the solver only searches a bounded space of inputs ranging from zero to $2^{bnd-inbits}-1$. The default for this flag is 5; attempting numbers much bigger than this is not recommended.

1.2 Running the synthesizer

To try this sketch out on your own, place it in a file, say `test1.sk`. Then, run the synthesizer with the following command line:

```
> sketch test1.sk
```

When you run the synthesizer in this way, the synthesized program is simply written to the console. If
Program Synthesis as Repair

- A program synthesis algorithm can be used to solve program repair
- Conceptually: replace the buggy line with □
- If you can synthesize XYZ to fill in that hole, the patch is “delete that line and replace it with XYZ”
- In practice, template: □ = □ + □*a + □*b + □*c;
  - where a, b, c are all in-scope variables
Program Repair Example

```c
int is_upward(int in, int up, int down) {
    int bias, r;
    if (in)
        bias = down;  // fix: bias = up + 100
    else
        bias = up;
    if (bias > down)
        r = 1;
    else
        r = 0;
    return r;
}
```

<table>
<thead>
<tr>
<th>Test</th>
<th>in</th>
<th>up</th>
<th>down</th>
<th>expected</th>
<th>observed</th>
<th>Passed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
<td>110</td>
<td>1</td>
<td>0</td>
<td>✗</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>100</td>
<td>50</td>
<td>1</td>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>-20</td>
<td>60</td>
<td>1</td>
<td>0</td>
<td>✗</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>-10</td>
<td>1</td>
<td>1</td>
<td>✓</td>
</tr>
</tbody>
</table>
Program Repair Example

```c
int is_upward(int in, int up, int down){
    int bias, r;
    if (in)
        bias = $c_0 + c_1 \times \text{bias} + c_2 \times \text{in} + c_3 \times \text{up} + c_4 \times \text{down}$;
    else
        bias = up;
    if (bias > down)
        r = 1;
    else
        r = 0;
    return r;
}
```

<table>
<thead>
<tr>
<th>Test</th>
<th>in</th>
<th>up</th>
<th>down</th>
<th>expected</th>
<th>observed</th>
<th>Passed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>✔️</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
<td>110</td>
<td>1</td>
<td>0</td>
<td>✗</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>100</td>
<td>50</td>
<td>1</td>
<td>1</td>
<td>✔️</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>-20</td>
<td>60</td>
<td>1</td>
<td>0</td>
<td>✗</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>✔️</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>-10</td>
<td>1</td>
<td>1</td>
<td>✔️</td>
</tr>
</tbody>
</table>
Program Repair Example

```c
int is_upward(int in, int up, int down) {
    int bias, r;
    if (in)
        bias = c0 + c1 * bias + c2 * in + c3 * up + c4 * down;
    else
        bias = up;
    if (bias > down)
        r = 1;
    else
        r = 0;
    return r;
}
```

<table>
<thead>
<tr>
<th>Test</th>
<th>in</th>
<th>up</th>
<th>down</th>
<th>expected</th>
<th>observed</th>
<th>Passed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
<td>110</td>
<td>1</td>
<td>0</td>
<td>×</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>100</td>
<td>50</td>
<td>1</td>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>-20</td>
<td>60</td>
<td>1</td>
<td>0</td>
<td>×</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>-10</td>
<td>1</td>
<td>1</td>
<td>✓</td>
</tr>
</tbody>
</table>
Program Reachability

- **Given** a program $P$ and a set of program variables $x_1 \ldots x_n$ and a program label $L$, do there exist values $c_1 \ldots c_n$ such that $P$ with $x_i$ set to $c_i$ reaches label $L$ in finite time?

- This is what SLAM and BLAST do (repeatedly).
  - L is the error label, $c_i$ is the counterexample.

- This is what HW #5 does (repeatedly).
  - L is the end of a path, $c_i$ is the test input.
Reachability Example

```c
int x, y; /* global input */

int P() {
    if (2 * x == y)
        if (x > y + 10)
            [L]

    return 0;
}
```
Reachability Example

```c
int x, y; /* global input */

int P() {
    if (2 * x == y)
        if (x > y + 10)
            [L]

    return 0;
}
```

\[ x = -20 \]
\[ y = -40 \]
Reachability Analysis

- How hard is it to solve reachability in general?
  - “Can you find values for these variables such that this program reaches this label?”

- Many tools exist, including some that are quite mature:
  - DART, KLEE, SLAM, BLAST, PEX, CREST, CUTE, AUSTIN, “tigen”
Comparative Analysis

- Program synthesis and program reachability are both undecidable in general.
- The “heart” of reachability is solving all *path constraints*.
  - Each “if” makes it harder to find a single consistent set of values.
- The “heart” of synthesis is handling all *tests*.
  - Each new test makes it harder to find a single consistent set of values.
Reductions

• Problem A is **reducible** to Problem B if an efficient algorithm for B could be used as a subroutine to solve A efficiently.

• A **gadget** is a subset of a problem instance that simulates the behavior of one of the fundamental units of a different problem.
  
  • Gadgets are hard to come up with the first time (e.g., when you are doing your Algo homework)
  
  • Gadgets often look simple once presented
Reduction Recipe

- Given an instance I of problem X
- Assume an oracle that can solve Y
- Transform I into f(I), verify f is polytime
- Let J = Y(f(I))
- Transform J into g(J), verify g is polytime
- Verify g(J) = X(I)
- Return g(J)
Gadget Example

- Use Graph 3-Colorability to solve 3-SAT
- Instance shown:
  
  \[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z)\]
Gadget Example

- Use Graph 3-Colorability to solve 3-SAT
- Instance shown:

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z)\]
Gadget Example

- Use Graph 3-Colorability to solve 3-SAT
- Instance shown:
  \[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z)\]
Gadget Example

- Use Graph 3-Colorability to solve 3-SAT
- Instance shown:
  \[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z)\]
Gadget Example

• Use Graph 3-Colorability to solve 3-SAT

• Instance shown:
  \((x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z)\)
Gadget Example

- Use Graph 3-Colorability to solve 3-SAT
- Instance shown:
  \[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z)\]
Gadget Example

- Use Graph 3-Colorability to solve 3-SAT
- Instance shown:
  \[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z)\]
Gadget Example

- Use Graph 3-Colorability to solve 3-SAT
- Instance shown:
  \[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z)\]
Gadget Example

- Use Graph 3-Colorability to solve 3-SAT
- Instance shown: 
  \((x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z)\)
Gadget Example

- Use Graph 3-Colorability to solve 3-SAT
- Instance shown:
\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z)\]
Gadget Example

- Use Graph 3-Colorability to solve 3-SAT

- Instance shown:
  \[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z)\]
Gadget Example

- Use Graph 3-Colorability to solve 3-SAT
- Instance shown:
  \[(x \lor y \lor \neg z) \land \neg x \lor \neg y \lor z\]
Gadget Example

- Use Graph 3-Colorability to solve 3-SAT
- Instance shown:
  
  \[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z)\]

- \(X = \text{true}\)
- \(Y = \text{false}\)
- \(Z = \text{true}\)
Trivia

- The *this*-Howard Isomorphism establishes a direct relationship between computer program and proofs. It shows a correspondence between proof calculi and type systems for models of computation.

<table>
<thead>
<tr>
<th>Logic side</th>
<th>Programming side</th>
</tr>
</thead>
<tbody>
<tr>
<td>axiom</td>
<td>variable</td>
</tr>
<tr>
<td>introduction rule</td>
<td>constructor</td>
</tr>
<tr>
<td>elimination rule</td>
<td>destructor</td>
</tr>
<tr>
<td>normal deduction</td>
<td>normal form</td>
</tr>
<tr>
<td>normalisation of deductions</td>
<td>weak normalisation</td>
</tr>
<tr>
<td>provability</td>
<td>type inhabitation problem</td>
</tr>
<tr>
<td>intuitionistic tautology</td>
<td>inhabited type</td>
</tr>
</tbody>
</table>
Physics

- This 1909 experiment involved tiny charged droplets of a fluid falling between two horizontal electrodes. With the electrodes uncharged, the drops reach terminal velocity while falling. By varying the voltage in the electrode plates and inducing an electromagnetic field, the drops could be perfectly suspended (electric force = gravitational force). Using the mass of the drops and the voltage, they solved for the electric charge, finding it to be always a small integer multiple of a basic constant (1.6 * 10^-19 C): the charge of a single electron. Name the experimenter or the fluid.
Reducing Synthesis To Reachability

• Given an instance of a synthesis (repair) problem, and assuming we have an oracle that can solve reachability, let us convert the synthesis instance into a reachability instance.

• If we can do this efficiently, any existing reachability tool (e.g., DART, KLEE, SLAM) could be used to repair programs.
```c
int is_upward(int in, int up, int down) {
    int bias, r;
    if (in)
        bias = \[c_0 + c_1 \ast bias + c_2 \ast in + c_3 \ast up + c_4 \ast down;\]
    else
        bias = up;
    if (bias > down)
        r = 1;
    else
        r = 0;
    return r;
}
```

<table>
<thead>
<tr>
<th>Test</th>
<th>Inputs</th>
<th>Output</th>
<th>Passed?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in</td>
<td>up</td>
<td>down</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>-20</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>-10</td>
</tr>
</tbody>
</table>
```c
int is_upward(int in, int up, int down) {
    int bias, r;
    if (in)
        bias = \[c_0 \] + \[c_1\] * bias + \[c_2\] * in + \[c_3\] * up + \[c_4\] * down;
    else
        bias = up;
    if (bias > down)
        r = 1;
    else
        r = 0;
    return r;
}
```

<table>
<thead>
<tr>
<th>Test</th>
<th>in</th>
<th>up</th>
<th>down</th>
<th>expected</th>
<th>observed</th>
<th>Passed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
<td>110</td>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>100</td>
<td>50</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>-20</td>
<td>60</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>-10</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
int x, y; /* global input */

int P() {
    if (2 * x == y)
        if (x > y + 10)
            [L]
        return 0;
    }
```
```
1 int is_upward(int in, int up, int down) {
2   int bias, r;
3   if (in)
4       bias = \(c_0 + c_1 \cdot \text{bias} + c_2 \cdot \text{in} + c_3 \cdot \text{up} + c_4 \cdot \text{down}\);
5   else
6       bias = up;
7   if (bias > down)
8       r = 1;
9   else
10      r = 0;
11  return r;
12 }
```

"Heart" insights:

Multiple tests make Synthesis difficult.

Multiple path conditions make Reachability difficult.

<table>
<thead>
<tr>
<th>Test</th>
<th>in</th>
<th>up</th>
<th>down</th>
<th>expected</th>
<th>observed</th>
<th>Passed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
<td>110</td>
<td>1</td>
<td>0</td>
<td>✗</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>100</td>
<td>50</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>-20</td>
<td>60</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>-10</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

```
int x, y; /* global input */

int P() {
  if (2 * x == y)
    if (x > y + 10) [L]
      return 0;
}
```
int is_upward(int in, int up, int down) {
    int bias, r;
    if (in)
        bias = $c_0 + c_1 \cdot \text{bias} + c_2 \cdot \text{in} + c_3 \cdot \text{up} + c_4 \cdot \text{down};$
    else
        bias = up;
    if (bias > down)
        r = 1;
    else
        r = 0;
    return r;
}

int c_0, c_1, c_2, c_3, c_4; /* global input */

int pis_upward(int in, int up, int down) {
    int bias, r;
    if (in)
        bias = $c_0 + c_1 \cdot \text{bias} + c_2 \cdot \text{in} + c_3 \cdot \text{up} + c_4 \cdot \text{down};$
    else
        bias = up;
    if (bias > down)
        r = 1;
    else
        r = 0;
    return r;
}

int main() {
    if (pis_upward(1, 0, 100) == 0 &&
        pis_upward(1, 11, 110) == 1 &&
        pis_upward(0, 100, 50) == 1 &&
        pis_upward(1, -20, 60) == 1 &&
        pis_upward(0, 0, 10) == 0 &&
        pis_upward(0, 0, -10) == 1) {
        [L]
    }
    return 0;
}
Proving Correctness

- We must show that the constructed reachability instance is solvable (with values c₁ ... cₙ) iff the original synthesis instance is solvable (with values c₁ ... cₙ).

- The reachability instance is solved if those values cause execution to reach L.

- The synthesis instance is solved if those values cause every test to pass.
High-Level Proof Structure

- Lemma 1. The reachability instance method and the synthesis instance method agree on all (non-template) variables.
- Lemma 2. If the reachability instance reaches L from a state S (with values c1 ... cn), then that state and values model the weakest precondition of the synthesis instance method passing each test.
- Theorem 1. The synthesis instance is solvable iff the reachability instance is solvable (with the same values).
int is_upward(int in, int up, int down) {
    int bias, r;
    if (in)
        bias = $c_0 + c_1 \cdot \text{bias} + c_2 \cdot \text{in} + c_3 \cdot \text{up} + c_4 \cdot \text{down}$;
    else
        bias = up;
    if (bias > down)
        r = 1;
    else
        r = 0;
    return r;
}

int main() {
    if (is_upward(1, 0, 100) == 0 &&
        is_upward(1, 11, 110) == 1 &&
        is_upward(0, 100, 50) == 1 &&
        is_upward(1, -20, 60) == 1 &&
        is_upward(0, 0, 10) == 0 &&
        is_upward(0, 0, -10) == 1) {
        [L]
    } return 0;
}
Lemma 1 (Agree on Vars)

- Let Q be the input synthesis instance method with template variables $v_1 \ldots v_n$.
- Let $P = \text{Gadget}(Q)$ be the reachability instance corresponding to method P.
- For all states $\sigma_1, \sigma_2, \sigma_3$, all values $c_1 \ldots c_n$, all inputs values $x$, it holds that
  - If $\sigma_1(v_i) = c_i$, then $<P(x), \sigma_1> \downarrow S_2$ iff $<\text{inst}(Q, \bar{c}), \sigma_1> \downarrow \sigma_3$
  - and for all $y \neq v_i$, $\sigma_2(y) = \sigma_3(y)$. 
Lemma 1 Proof

- If $\sigma_1(v_i) = c_i$, then $<P(x), \sigma_1> \downarrow S_2$ iff
  $<\text{inst}(Q,c), \sigma_1> \downarrow \sigma_3$
  and for all $y \neq v_i$, $\sigma_2(y) = \sigma_3(y)$.

- How shall we prove it? What proof technique should we use?
Lemma 1 Proof

- If $\sigma_1(v_i) = c_i$, then $D_1 :: <P(x), \sigma_1> \downarrow S_2$ iff $D_2 :: <\text{inst}(Q,\overline{c}), \sigma_1> \downarrow \sigma_3$ and for all $y \neq v_i$, $\sigma_2(y) = \sigma_3(y)$.

- The proof proceeds by induction on the structure of the operational semantics derivation $D_1$. By inversion, the structure of $D_1$ corresponds exactly to the structure of $D_2$ except for template variables.
Lemma 1 Case: Template Variable

- **Case.** Suppose $D_1$ (reachability instance) is:

$$
\sigma_2 = \sigma_1 [ a \rightarrow \sigma_1(v_i) ]
$$

$$< a := v_i, \sigma_1 > \downarrow \sigma_2
$$

- By **inversion** and the construction of $P$, $D_2$ is:

$$
\sigma_3 = \sigma_1 [ a \rightarrow c_i ]
$$

$$< a := \text{exp}, \sigma_1 > \downarrow \sigma_3
$$

- where $\text{exp} = \text{inst}(\overline{c_i}, \overline{c}) = c_i$
Lemma 1 Case: Template Variable

- Have: $\sigma_2 = \sigma_1 [ a \rightarrow \sigma_1(v_i) ]$

- Have: $\sigma_3 = \sigma_1 [ a \rightarrow c_i ]$

- To show: “for all $y \neq v_i$, $\sigma_2(y) = \sigma_3(y)$”

- Sub-Case 1. $y \neq a$. Then $\sigma_2(y) = \sigma_3(y)$.

- Sub-Case 2. $y = a$. To show: $\sigma_1(v_i) = c_i$. This was actually one of the assumptions in the statement of the lemma. (Intuitively, it means the reachability analysis assigned $c_i$ to each variable $v_i$ to reach the label L.)
Lemma 1 (Agree on Vars)

- Let Q be the input synthesis instance method with template variables \( v_1 \ldots v_n \).
- Let \( P = \text{Gadget}(Q) \) be the reachability instance corresponding to method P.
- For all states \( \sigma_1, \sigma_2, \sigma_3 \), all values \( c_1 \ldots c_n \), all inputs values \( x \), it holds that
  - If \( \sigma_1(v_i) = c_i \), then \( <P(x), \sigma_1> \downarrow S_2 \) iff
    - \( <\text{inst}(Q, \overline{c}), \sigma_1> \downarrow \sigma_3 \)
    - and for all \( y \neq v_i \), \( \sigma_2(y) = \sigma_3(y) \).
Lemma 2 (Reach L = Pass Tests)

- Let Q be the input synthesis instance method with template variables $v_1 \ldots v_n$ and tests $<input_1, output_n>$.
- Let $P = \text{Gadget}(Q)$ be the reachability instance method $\text{main}$.
- The execution of $P$ reaches $L$ starting from state $\sigma_1$ iff $\sigma_1 | = wp(\text{result} = \text{inst}(Q, c)(input_1), \text{result} = output_1)$
  && $wp(\text{result} = \text{inst}(Q, c)(input_n), \text{result} = output_n)$
  where $\sigma_1(v_i) = c_i$. 
Lemma 2 Proof

• By gadget construction there is only one label \( L \) in \( P \), “if \( e \) then \([L]\)” where \( e \) is of the form 
\[
\text{if } f(\text{input}_1) = \text{output}_1 \land \ldots \land f(\text{input}_n) = \text{output}_n.
\]

• By standard **weakest precondition** definitions for if, conjunction, equality and function calls, we have that \( L \) is reachable iff 
\[
\sigma_1 \models wp(\text{result} = f(\text{input}_1), \text{result} = \text{output}_1) \land \ldots \land wp(\text{result} = f(\text{input}_n), \text{result} = \text{output}_n).
\]
Lemma 2 Proof

- **Have:** $L$ is reachable iff $\sigma_1 \models wp(result = f(input_1), result = output_1) \land \ldots \land wp(result = f(input_n), result = output_n)$.

- **Want:** $L$ is reachable iff $\sigma_1 \models wp(result = inst(Q, \overline{c})(input_1), result = output_1) \land \ldots \land wp(result = inst(Q, \overline{c})(input_n), result = output_n)$

- **To show:** $\sigma_1 \models wp(result = f(input_i), result = output_i)$ iff $\sigma_1 \models wp(result = inst(Q, \overline{c})(input_i), result = output_i)$
Lemma 2 Proof

- To show: $\sigma_1 \models wp(\text{result} = f(\text{input}_i), \text{result} = \text{output}_i)$
  iff $\sigma_1 \models wp(\text{result} = \text{inst}(Q, c)(\text{input}_i), \text{result} = \text{output}_i)$

  ... where $f$ is the method from Gadget($Q$)

- By the **soundness and completeness** of weakest preconditions with respect to operational semantics, we have $< \text{result} = f(\text{input}_i), \sigma_1 > \downarrow \sigma_2$ iff $\sigma_2 \models \text{result} = \text{output}_i$. 
Lemma 2 Proof

- Have: \(< \text{result} = f(\text{input}_i), \sigma_1 > \downarrow \sigma_2 \iff \sigma_2 |\models \text{result} = \text{output}_i.\)

- By Lemma 1, we have \(< \text{result} = \text{inst}(Q, \bar{c})(\text{input}_i), \sigma_1 > \downarrow \sigma_3 \iff \sigma_1(y) = \sigma_3(y) \text{ for all } y \neq v_i.\)

- Since “result” \(\neq v_i, \sigma_1(\text{result}) = \sigma_1(\text{result}) = \text{output}_i.\)

- So running the template program Q instantiated with \(c_i = v_i\) on a test input produces the required output.
Correctness Theorem

- Let \( Q \) be the input synthesis instance method with template variables \( v_1 \ldots v_n \) and tests \( <\text{input}_1, \text{output}_n> \).
- Let \( P = \text{Gadget}(Q) \) be the reachability instance method \textit{main}.
- There exist parameter values \( c_i \) such that for all \( <\text{input}, \text{output}> \), \( \text{inst}(Q, c_i)(\text{input}) = \text{output} \) iff there exist input values \( t_i \) such that the execution of \( P \) with \( v_i \to t_i \) reaches \( L \).
- Proof: From Lemma 2 with \( t_i = c_i \).
Reducing Reachability To Synthesis

- We can also carry out a constructive reduction going the other direction.
- Suppose we are given an instance of program reachability. Can we convert it into a program synthesis instance to solve it?
Reachability to Synthesis Example

```c
int x, y; /* global input */

int P() {
    if (2 * x == y)
        if (x > y + 10)
            [L]
        return 0;
}
```

```c
1 int is_upward(int in, int up, int down){
2     int bias, r;
3     if (in)
4         bias = \text{bias} + \text{bias} + \text{bias} + \text{bias} + \text{bias} + \text{bias};
5     else
6         bias = up;
7     if (bias > down)
8         r = 1;
9     else
10    r = 0;
11 } return r;
```

<table>
<thead>
<tr>
<th>Test</th>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in</td>
<td>up</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>-20</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Reachability to Synthesis Example

```c
int x, y; /* global input */

int P() {
    if (2 * x == y)
        if (x > y + 10)
            [L]
    return 0;
}

int qP() {
    if (2*x == y)
        if (x > y + 10)
           /* location of [L]
            in P */
            raise REACHED;
    return 0;
}

Test suite: Q() = 1

int qmain() {
    /* Find x and y.
        Equivalently, synthesize:
        x = cx
        y = cy */
    try {
        qP();
    } catch (REACHED) {
        return 1;
    }
    return 0;
}
```
Implications

- Program reachability tools are much more mature than program repair tools.
- CETI Program Repair Algorithm
  - For each buggy line, in ranked order
    - For every repair template, in ranked order
      - Convert repair instance to reachability instance
      - Call off-the-shelf reachability tool (e.g., SMT solver / KLEE)
      - If reachable, return parameters as patch
## Prototype CETI Evaluation

<table>
<thead>
<tr>
<th>Bug Type</th>
<th>R-Progs</th>
<th>Time (secs)</th>
<th>Repair?</th>
<th>Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1 incorrect op</td>
<td>6143</td>
<td>21</td>
<td>✓</td>
<td>T&lt;sub&gt;op&lt;/sub&gt;</td>
</tr>
<tr>
<td>v2 missing code</td>
<td>6993</td>
<td>27</td>
<td>✓</td>
<td>T&lt;sub&gt;lincomb&lt;/sub&gt;</td>
</tr>
<tr>
<td>v3 incorrect op</td>
<td>8006</td>
<td>18</td>
<td>✓</td>
<td>T&lt;sub&gt;op&lt;/sub&gt;</td>
</tr>
<tr>
<td>v4 incorrect op</td>
<td>5900</td>
<td>27</td>
<td>✓</td>
<td>T&lt;sub&gt;const&lt;/sub&gt;</td>
</tr>
<tr>
<td>v5 missing code</td>
<td>8440</td>
<td>394</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v6 incorrect op</td>
<td>5872</td>
<td>19</td>
<td>✓</td>
<td>T&lt;sub&gt;op&lt;/sub&gt;</td>
</tr>
<tr>
<td>v7 incorrect const</td>
<td>7302</td>
<td>18</td>
<td>✓</td>
<td>T&lt;sub&gt;const&lt;/sub&gt;</td>
</tr>
<tr>
<td>v8 incorrect const</td>
<td>6013</td>
<td>19</td>
<td>✓</td>
<td>T&lt;sub&gt;const&lt;/sub&gt;</td>
</tr>
<tr>
<td>v9 incorrect op</td>
<td>5938</td>
<td>24</td>
<td>✓</td>
<td>T&lt;sub&gt;op&lt;/sub&gt;</td>
</tr>
<tr>
<td>v10 incorrect op</td>
<td>7154</td>
<td>18</td>
<td>✓</td>
<td>T&lt;sub&gt;op&lt;/sub&gt;</td>
</tr>
<tr>
<td>v11 multiple</td>
<td>6308</td>
<td>123</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v12 incorrect op</td>
<td>8442</td>
<td>25</td>
<td>✓</td>
<td>T&lt;sub&gt;op&lt;/sub&gt;</td>
</tr>
<tr>
<td>v13 incorrect const</td>
<td>7845</td>
<td>21</td>
<td>✓</td>
<td>T&lt;sub&gt;const&lt;/sub&gt;</td>
</tr>
<tr>
<td>v14 incorrect const</td>
<td>1252</td>
<td>22</td>
<td>✓</td>
<td>T&lt;sub&gt;const&lt;/sub&gt;</td>
</tr>
<tr>
<td>v15 multiple</td>
<td>7760</td>
<td>258</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v16 incorrect const</td>
<td>5470</td>
<td>19</td>
<td>✓</td>
<td>T&lt;sub&gt;const&lt;/sub&gt;</td>
</tr>
<tr>
<td>v17 incorrect const</td>
<td>7302</td>
<td>12</td>
<td>✓</td>
<td>T&lt;sub&gt;const&lt;/sub&gt;</td>
</tr>
<tr>
<td>v18 incorrect const</td>
<td>7383</td>
<td>18</td>
<td>✓</td>
<td>T&lt;sub&gt;const&lt;/sub&gt;</td>
</tr>
<tr>
<td>v19 incorrect const</td>
<td>6920</td>
<td>19</td>
<td>✓</td>
<td>T&lt;sub&gt;const&lt;/sub&gt;</td>
</tr>
<tr>
<td>v20 incorrect op</td>
<td>5938</td>
<td>19</td>
<td>✓</td>
<td>T&lt;sub&gt;op&lt;/sub&gt;</td>
</tr>
<tr>
<td>v21 missing code</td>
<td>5939</td>
<td>31</td>
<td>✓</td>
<td>T&lt;sub&gt;lincomb&lt;/sub&gt;</td>
</tr>
<tr>
<td>v22 missing code</td>
<td>5553</td>
<td>175</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v23 missing code</td>
<td>5824</td>
<td>164</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v24 missing code</td>
<td>6050</td>
<td>231</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v25 incorrect op</td>
<td>5983</td>
<td>19</td>
<td>✓</td>
<td>T&lt;sub&gt;op&lt;/sub&gt;</td>
</tr>
<tr>
<td>v26 missing code</td>
<td>8004</td>
<td>195</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v27 missing code</td>
<td>8440</td>
<td>270</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v28 incorrect op</td>
<td>9072</td>
<td>11</td>
<td>✓</td>
<td>T&lt;sub&gt;op&lt;/sub&gt;</td>
</tr>
<tr>
<td>v29 missing code</td>
<td>6914</td>
<td>195</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v30 missing code</td>
<td>6533</td>
<td>170</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v31 multiple</td>
<td>4302</td>
<td>16</td>
<td>✓</td>
<td>T&lt;sub&gt;lincomb&lt;/sub&gt;</td>
</tr>
<tr>
<td>v32 multiple</td>
<td>4493</td>
<td>17</td>
<td>✓</td>
<td>T&lt;sub&gt;lincomb&lt;/sub&gt;</td>
</tr>
<tr>
<td>v33 multiple</td>
<td>9070</td>
<td>224</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v34 incorrect op</td>
<td>8442</td>
<td>75</td>
<td>✓</td>
<td>T&lt;sub&gt;lincomb&lt;/sub&gt;</td>
</tr>
<tr>
<td>v35 multiple</td>
<td>9070</td>
<td>184</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v36 incorrect const</td>
<td>6334</td>
<td>10</td>
<td>✓</td>
<td>T&lt;sub&gt;const&lt;/sub&gt;</td>
</tr>
<tr>
<td>v37 missing code</td>
<td>7523</td>
<td>174</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v38 missing code</td>
<td>7685</td>
<td>209</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v39 incorrect op</td>
<td>5983</td>
<td>20</td>
<td>✓</td>
<td>T&lt;sub&gt;op&lt;/sub&gt;</td>
</tr>
<tr>
<td>v40 missing code</td>
<td>7364</td>
<td>136</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v41 missing code</td>
<td>5899</td>
<td>29</td>
<td>✓</td>
<td>T&lt;sub&gt;lincomb&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

- Considered 41 bugs and simple one-line templates
- Fixed 100% of bugs admitting one-line fixes
- 22 seconds each, average
- Debroy & Wong (random mutation): 9 repairs
- GenProg: 11 repairs
- Forensic (concolic execution): 23 repairs
- CETI: 26 repairs
Concluding Thoughts

- PL Theory almost always translates into useful PL Practice (just with an X year lag time)
- There is plenty of scope for insight and creativity (e.g., whence these gadgets?)
- Techniques like structural induction, SMT solving, fault localization, substitution, axiomatic semantics, etc., remain relevant!
- HW0 (BLAST), HW5 (tigen), FlashFill (Gulwani Excel) and GenProg (last lecture) are all “secretly the same thing”
  = “statically reason about dynamic execution”