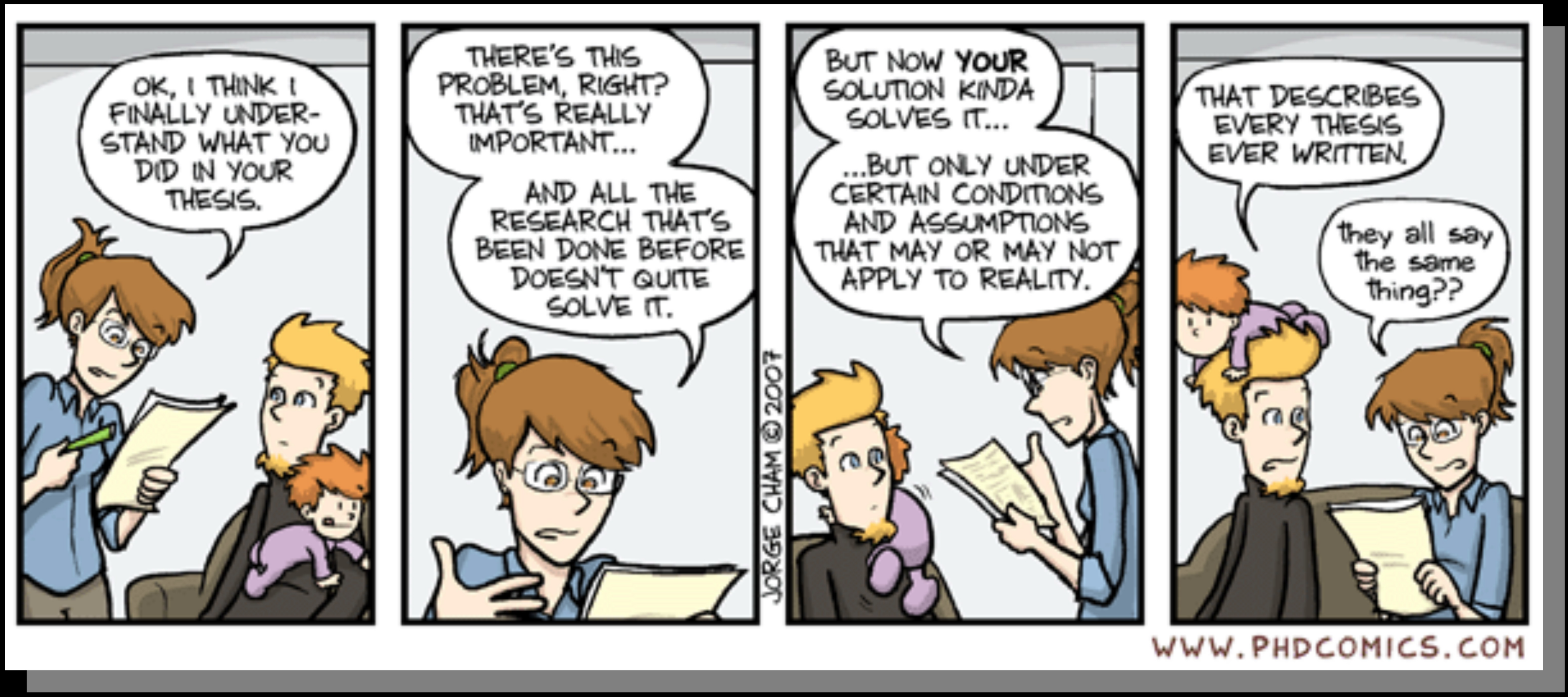
Program Synthesis *"is"* Program Reachability



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 = □ + □(y, □);”
- Can we fill in the template holes so that the program passes all of the tests?

Templated Program Syntax

$$
\begin{aligned}
cmd ::= \quad & skip \\
| \quad & cmd_1 \; ; \; cmd_2 \\
| \quad & v := aexp \\
| \quad & \ldots \\
aexp ::= \quad & aexp_1 + aexp_2 \\
| \quad & aexp_1 - aexp_2 \\
| \quad & \boxed{c_i} \\
| \quad & \ldots
\end{aligned}
$$

Called a template parameter

Template Instantation

- Given a templated program with template parameters $c_1 \ldots c_n$, and given template values $\overline{v} = v_1 \ldots v_n$ (expressions or constants), we can instantiate, yielding a non-templated program.
- $\text{inst}(\text{skip}, \overline{v}) \rightarrow \text{skip}$
- $\text{inst}(\text{cmd}_1; \text{cmd}_2, \overline{v}) \rightarrow \text{inst}(\text{cmd}_1, \overline{v}) ; \text{inst}(\text{cmd}_2, \overline{v})$
- $\text{inst}(x = \text{aexp}, \overline{v}) \rightarrow x = \text{inst}(\text{aexp}, \overline{v})$
- $\text{inst}(\boxed{c_i}, \overline{v}) \rightarrow v_i$

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 <input, output> pairs in T, $(inst(P, \bar{v}))(input) = output$?

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?”

Tools Exist: **sketch**

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.

```
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* $2^{\mathit{bnd\text{-}inbits}}-1$. *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

Armando Solar-Lezama: *The Sketching Approach to Program Synthesis*. APLAS 2009: 4-13.

Tools Exist: `sketch`

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.

```
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* $2^{bnd\text{-}inbits}-1$. *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

Armando Solar-Lezama: *The Sketching Approach to Program Synthesis*. APLAS 2009: 4-13.

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
  - cf. Linear Regression. cf. Daikon.

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;
}
```

Test	Inputs			Output		Passed?
	in	up	down	expected	observed	
1	1	0	100	0	0	✓
2	1	11	110	1	0	✗
3	0	100	50	1	1	✓
4	1	-20	60	1	0	✗
5	0	0	10	0	0	✓
6	0	0	-10	1	1	✓

Program Repair Example

```
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;
}
```

	Inputs			Output		
Test	in	up	down	expected	observed	**Passed?**
1	1	0	100	0	0	✓
2	1	11	110	1	0	✗
3	0	100	50	1	1	✓
4	1	-20	60	1	0	✗
5	0	0	10	0	0	✓
6	0	0	-10	1	1	✓

Program Repair Example

```
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;
}
```

c0 = 100
c1 = 0
c2 = 0
c3 = 1
c4 = 0
"bias = up + 100;"

Test	Inputs: in	Inputs: up	Inputs: down	Output: expected	Output: observed	Passed?
1	1	0	100	0	0	✓
2	1	11	110	1	0	✗
3	0	100	50	1	1	✓
4	1	-20	60	1	0	✗
5	0	0	10	0	0	✓
6	0	0	-10	1	1	✓

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 #6 does (repeatedly).
  - L is the end of a path, c_i is the test input.

Reachability Example

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

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

  return 0;
}
```

Reachability Example

```
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

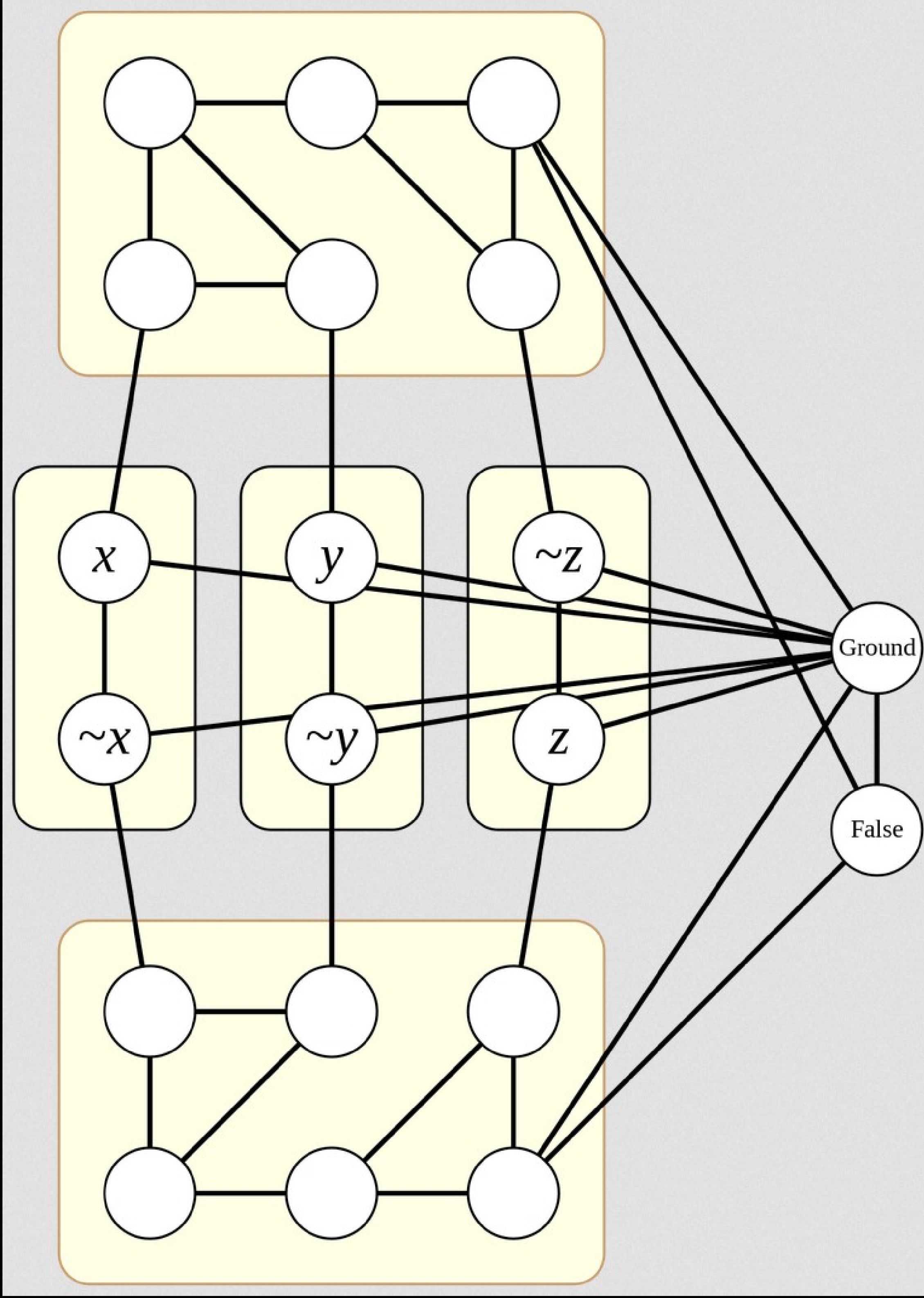
- Given an instance I of problem X
- Assume an oracle that can solve Y
- Transform I into f(I), verify f is polytime
- Let J = Y(f(I))
- Transform J into g(J), verify g is polytime
- Verify g(J) = X(I)
- Return g(J)

Gadget Example

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

  (x || y || !z) &&

  (!x || !y || z)

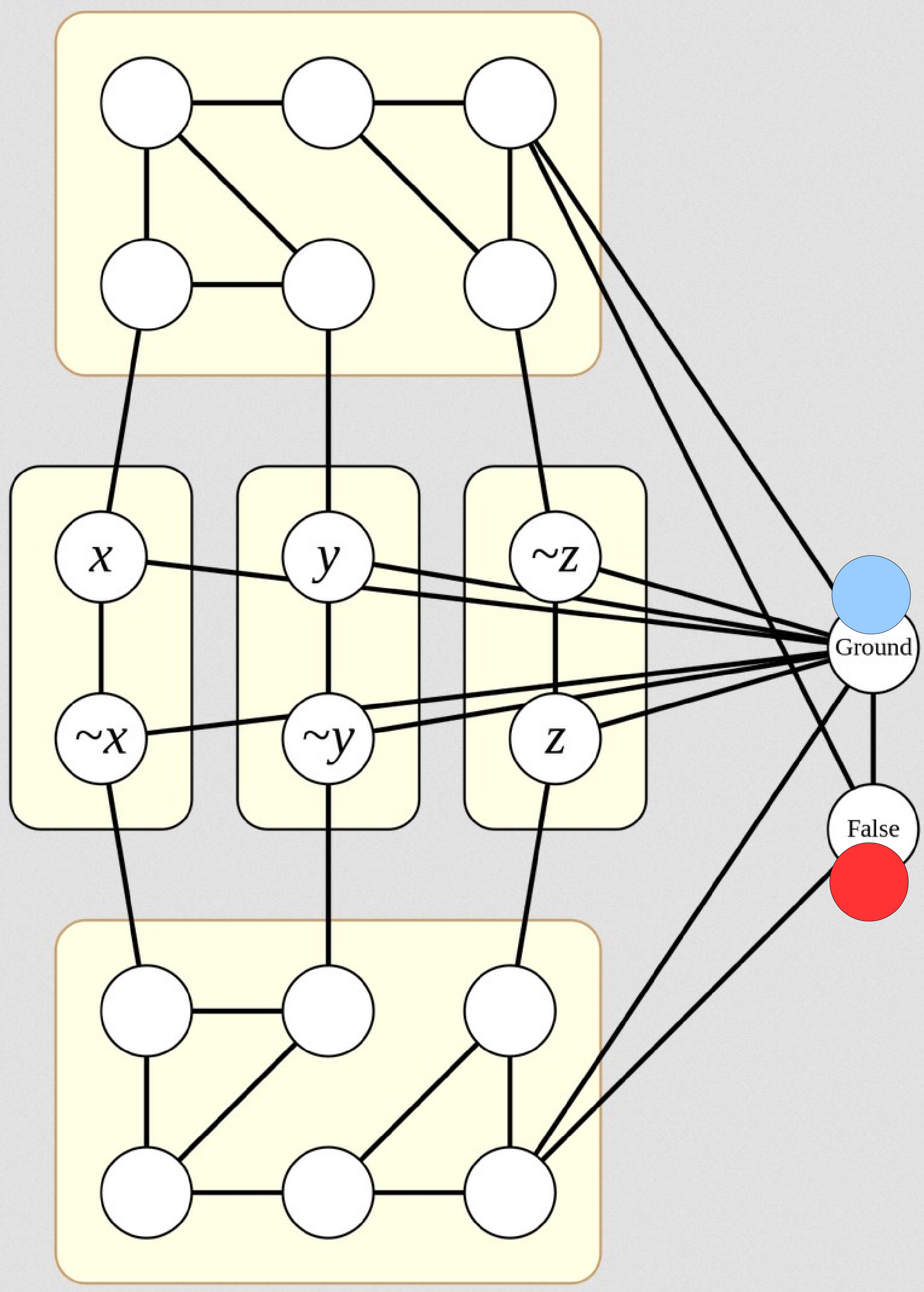


Gadget Example

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

  (x || y || !z) &&

  (!x || !y || z)

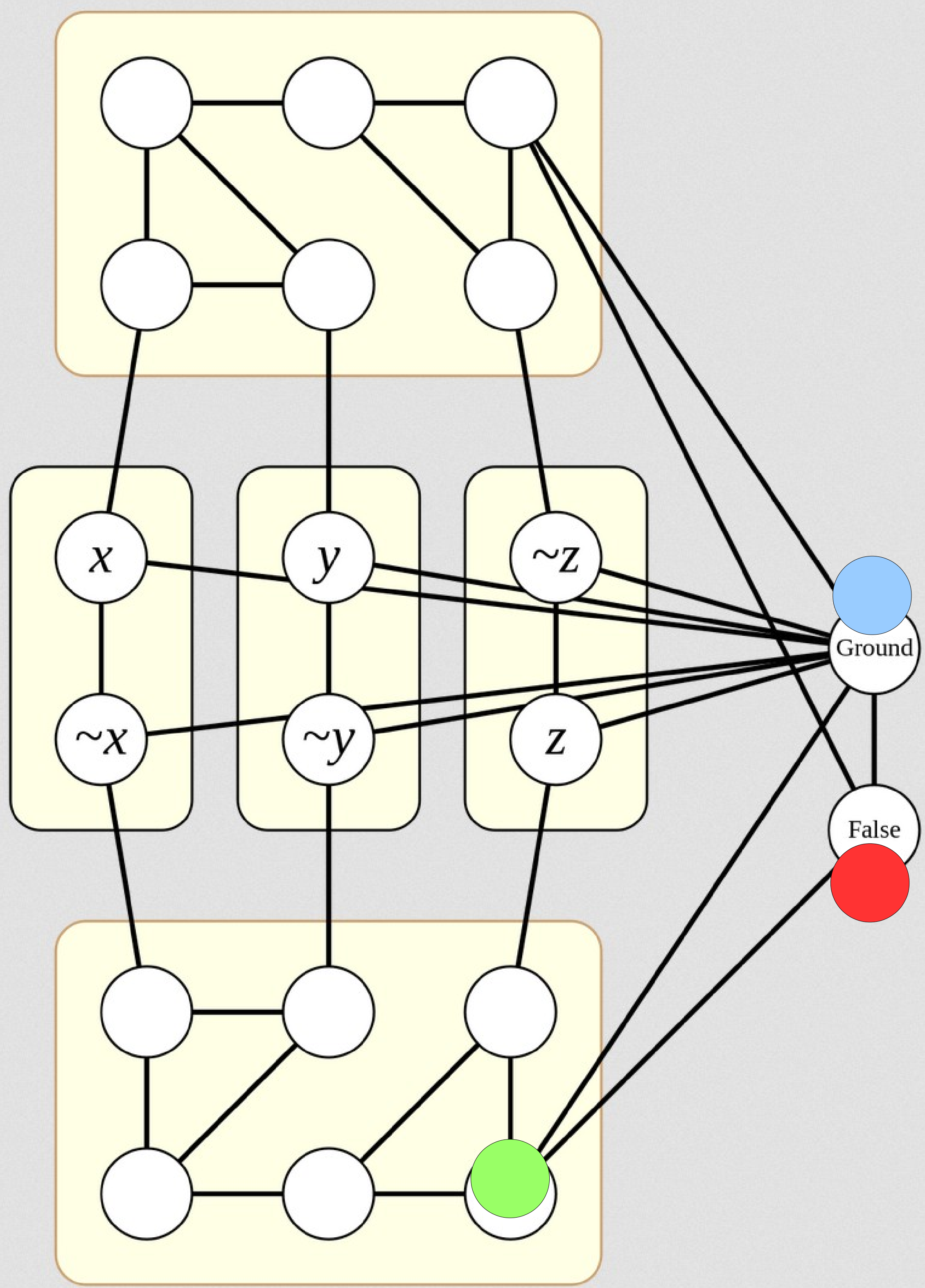


Gadget Example

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

  (x || y || !z) &&

  (!x || !y || z)

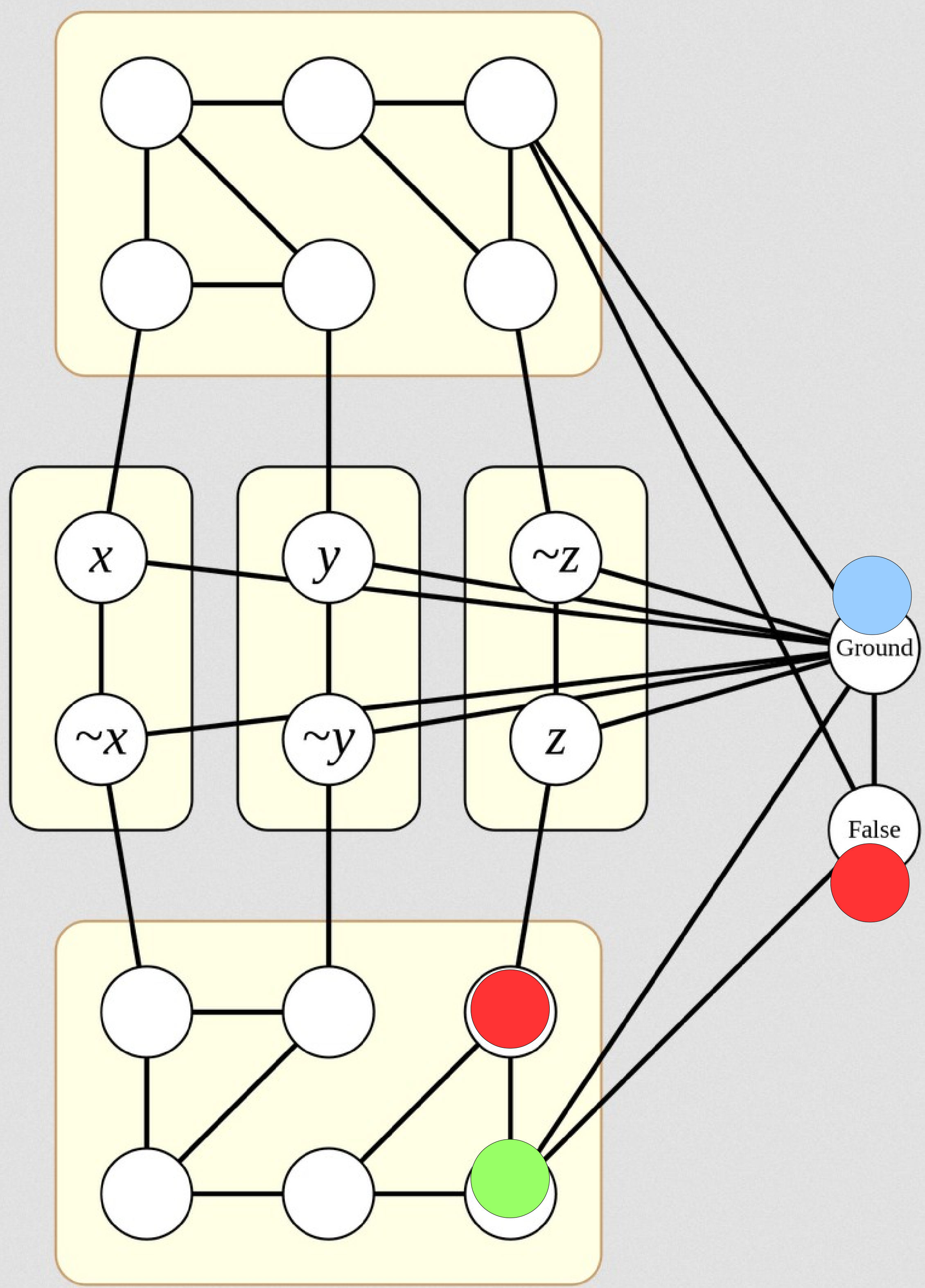


Gadget Example

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

  (x || y || !z) &&

  (!x || !y || z)

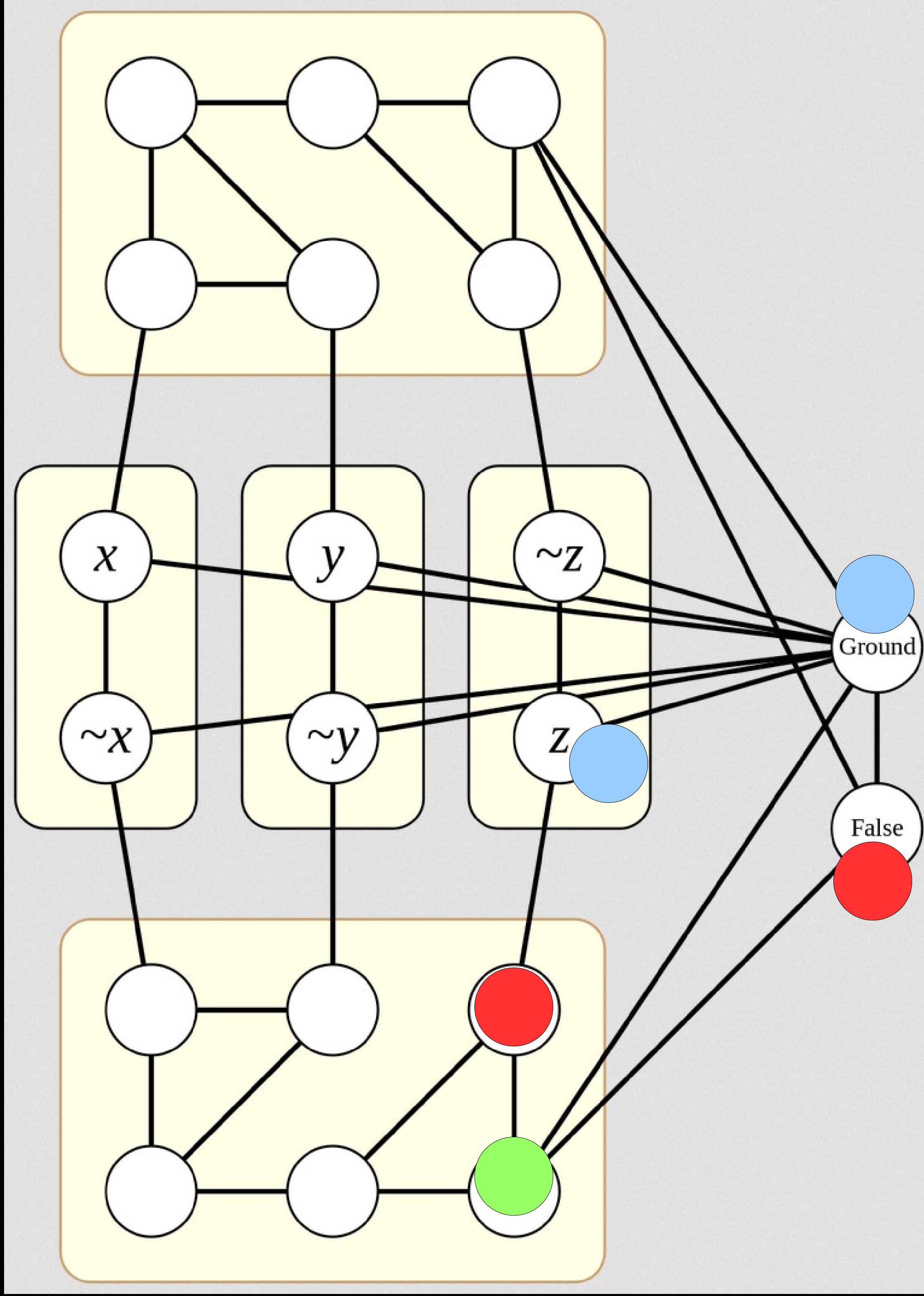


Gadget Example

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

  (x || y || !z) &&

  (!x || !y || z)

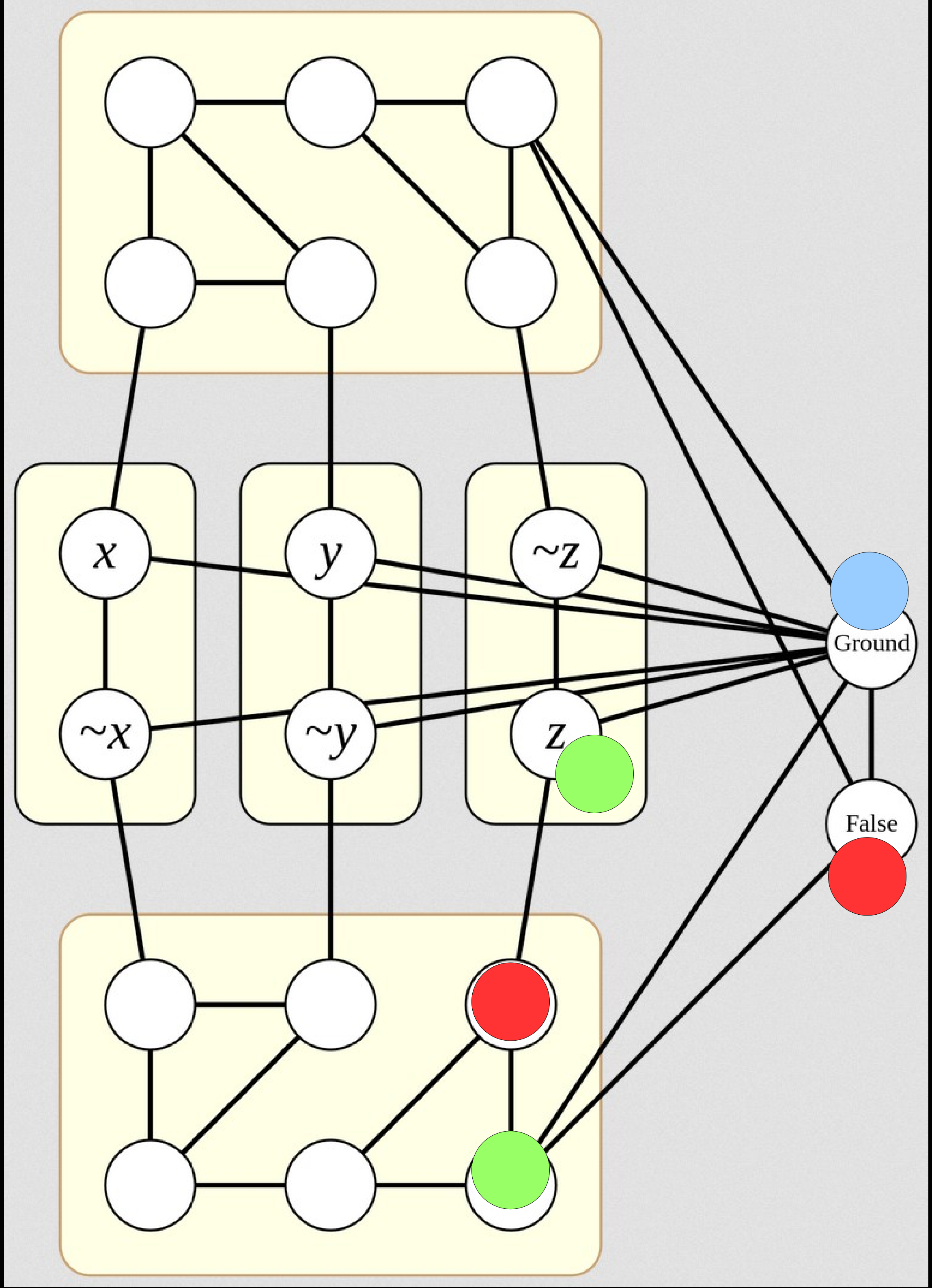


Gadget Example

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

  (x || y || !z) &&

  (!x || !y || z)

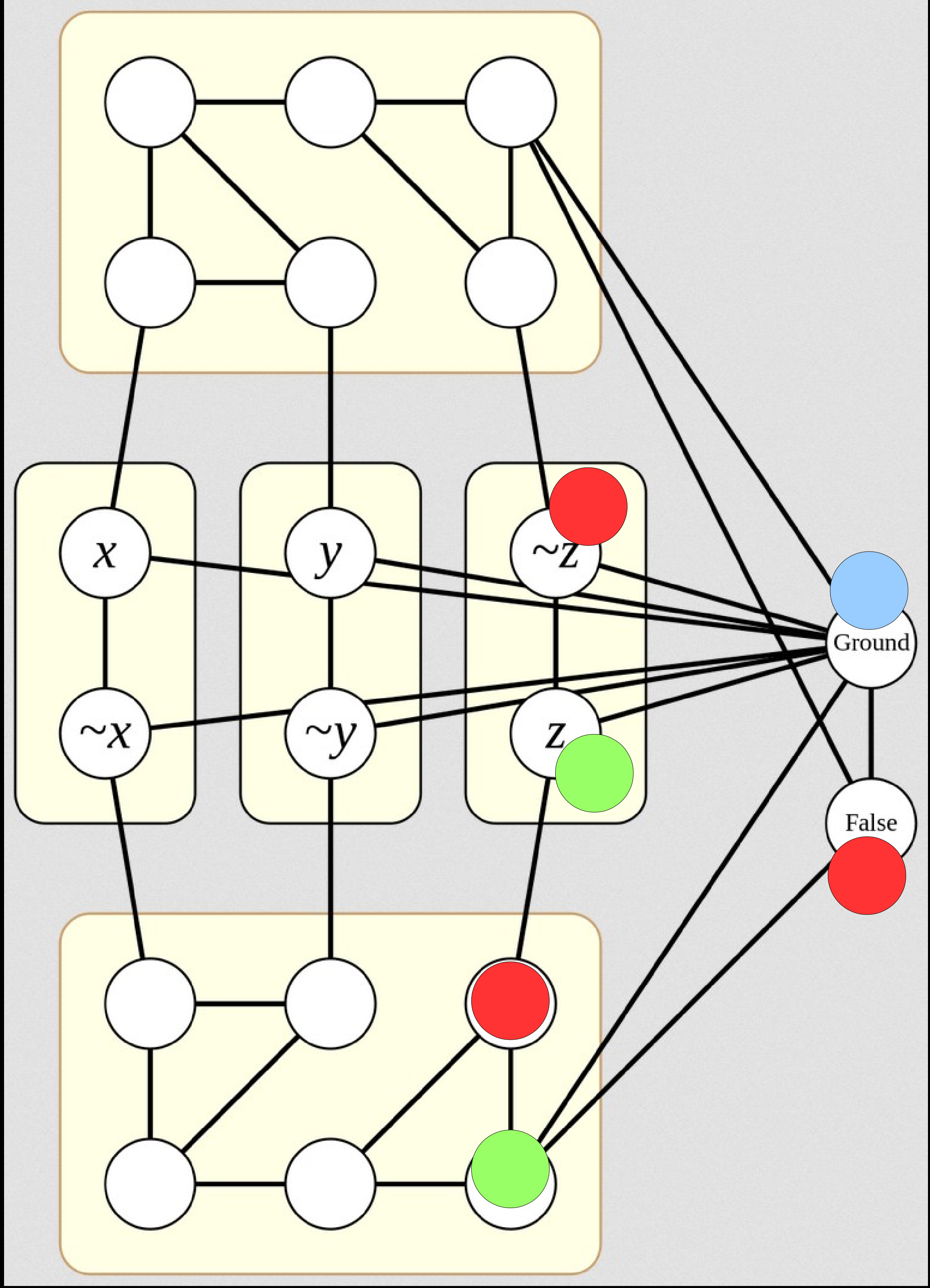


Gadget Example

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

  (x || y || !z) &&

  (!x || !y || z)

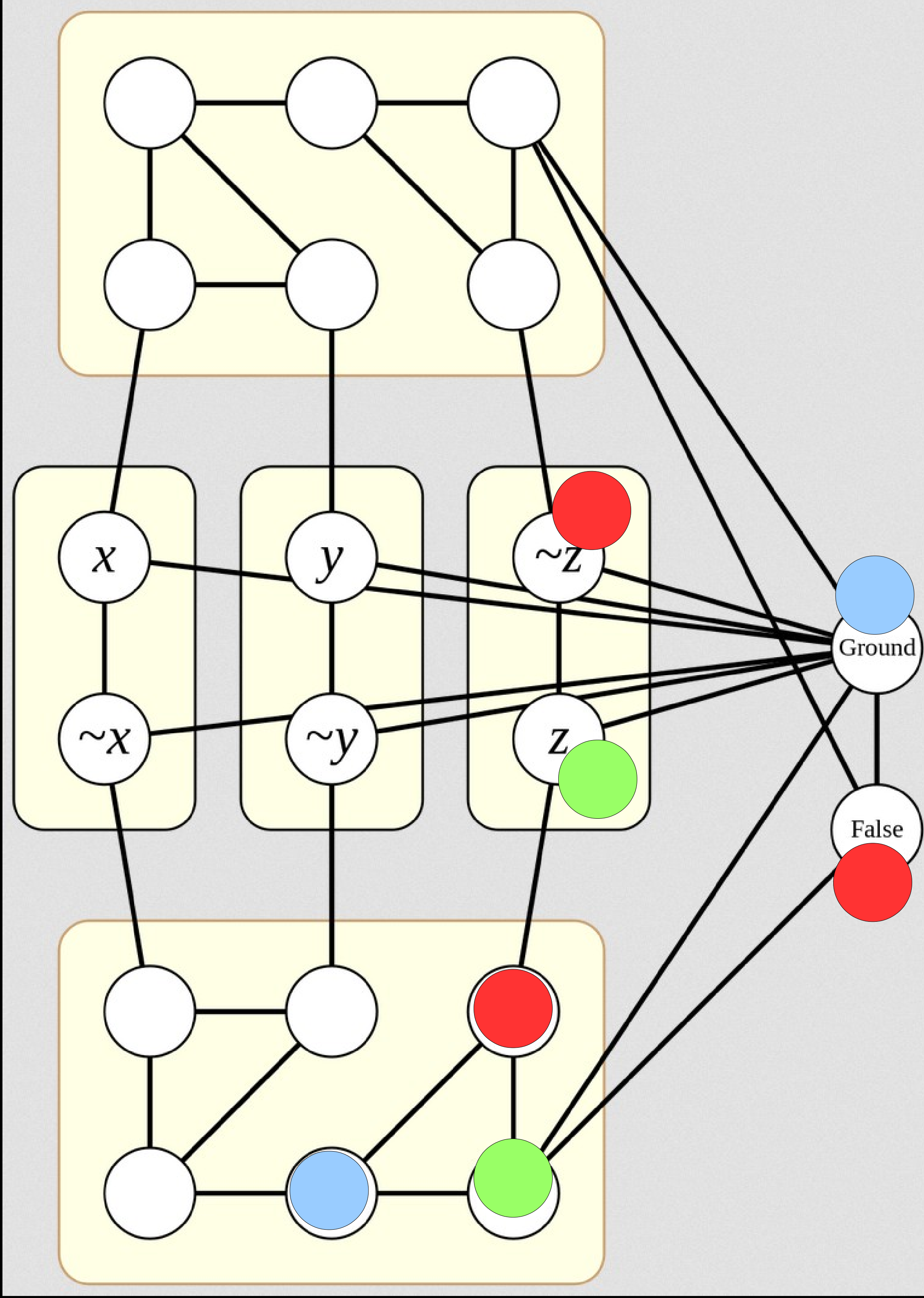


Gadget Example

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

  (x || y || !z) &&

  (!x || !y || z)

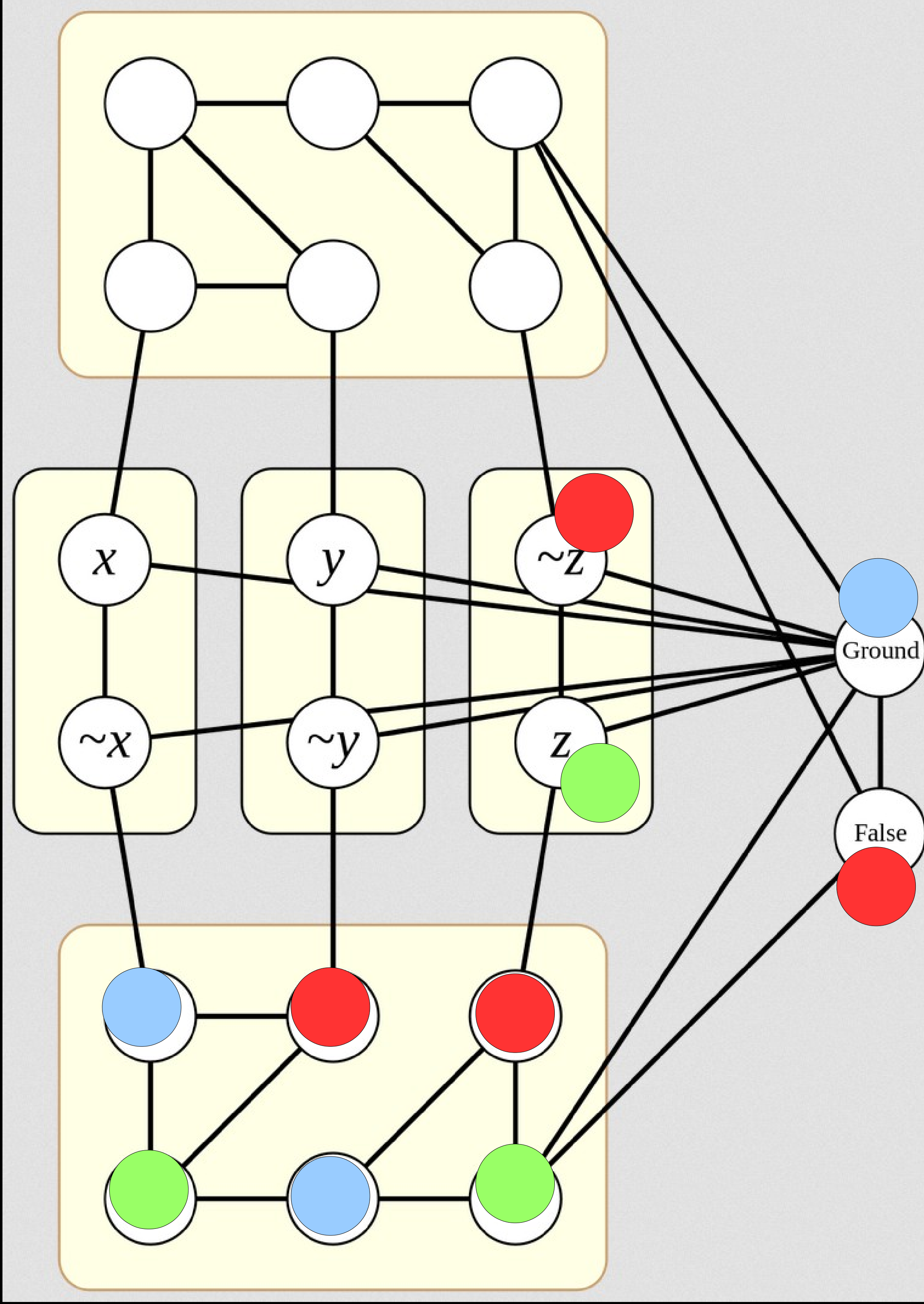


Gadget Example

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

  (x || y || !z) &&

  (!x || !y || z)

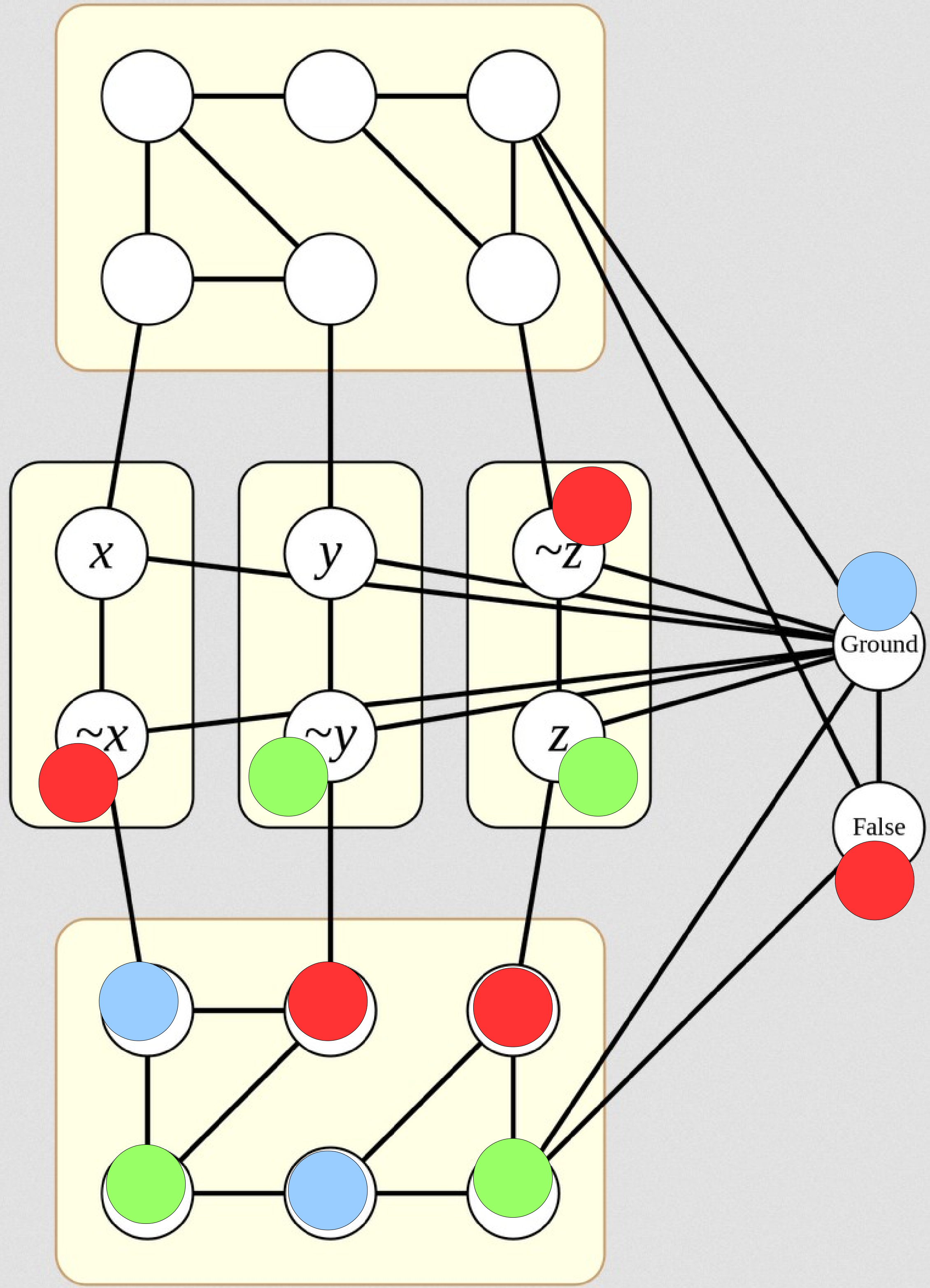


Gadget Example

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

  (x || y || !z) &&

  (!x || !y || z)

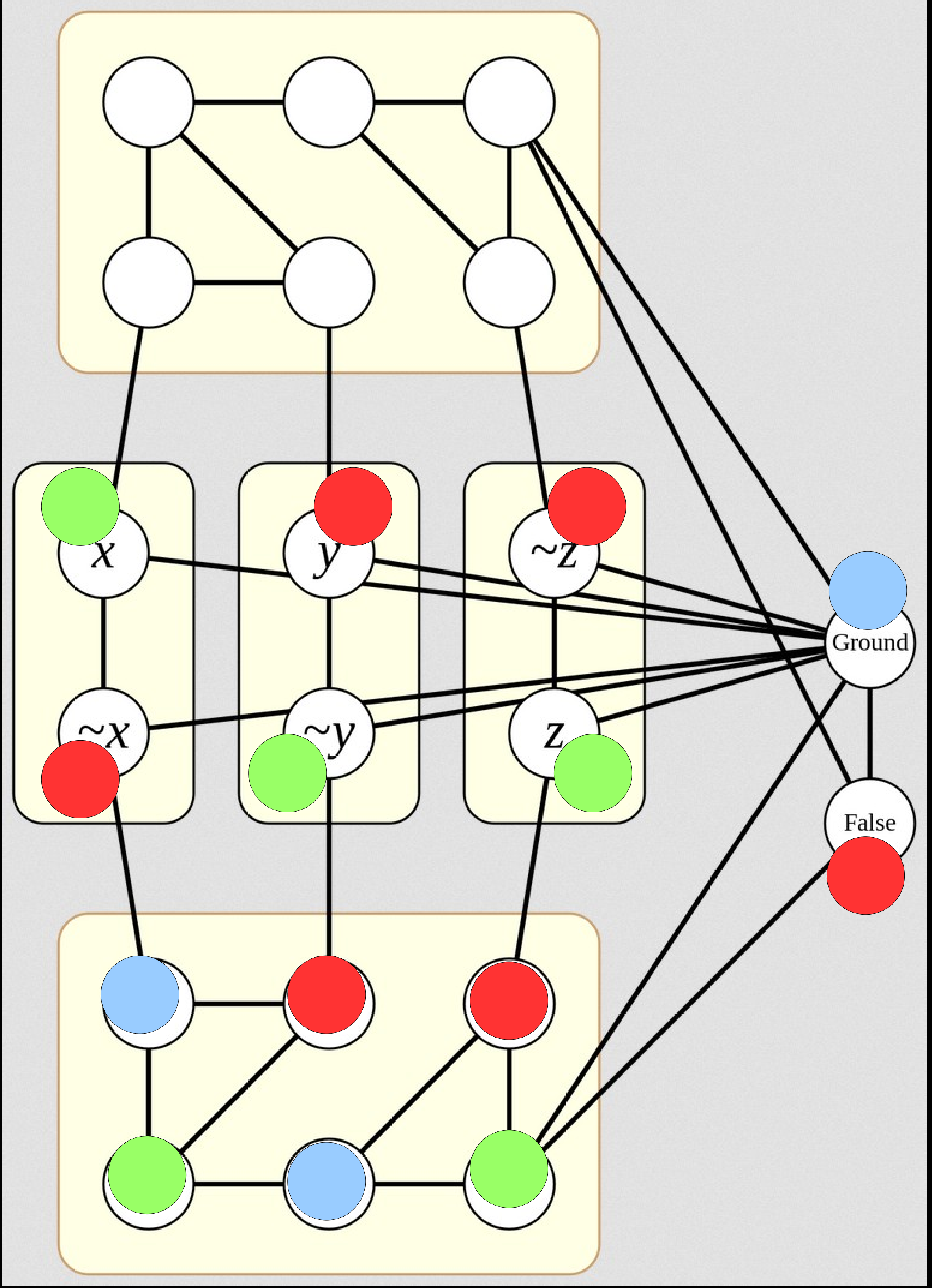


Gadget Example

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

  (x || y || !z) &&

  (!x || !y || z)

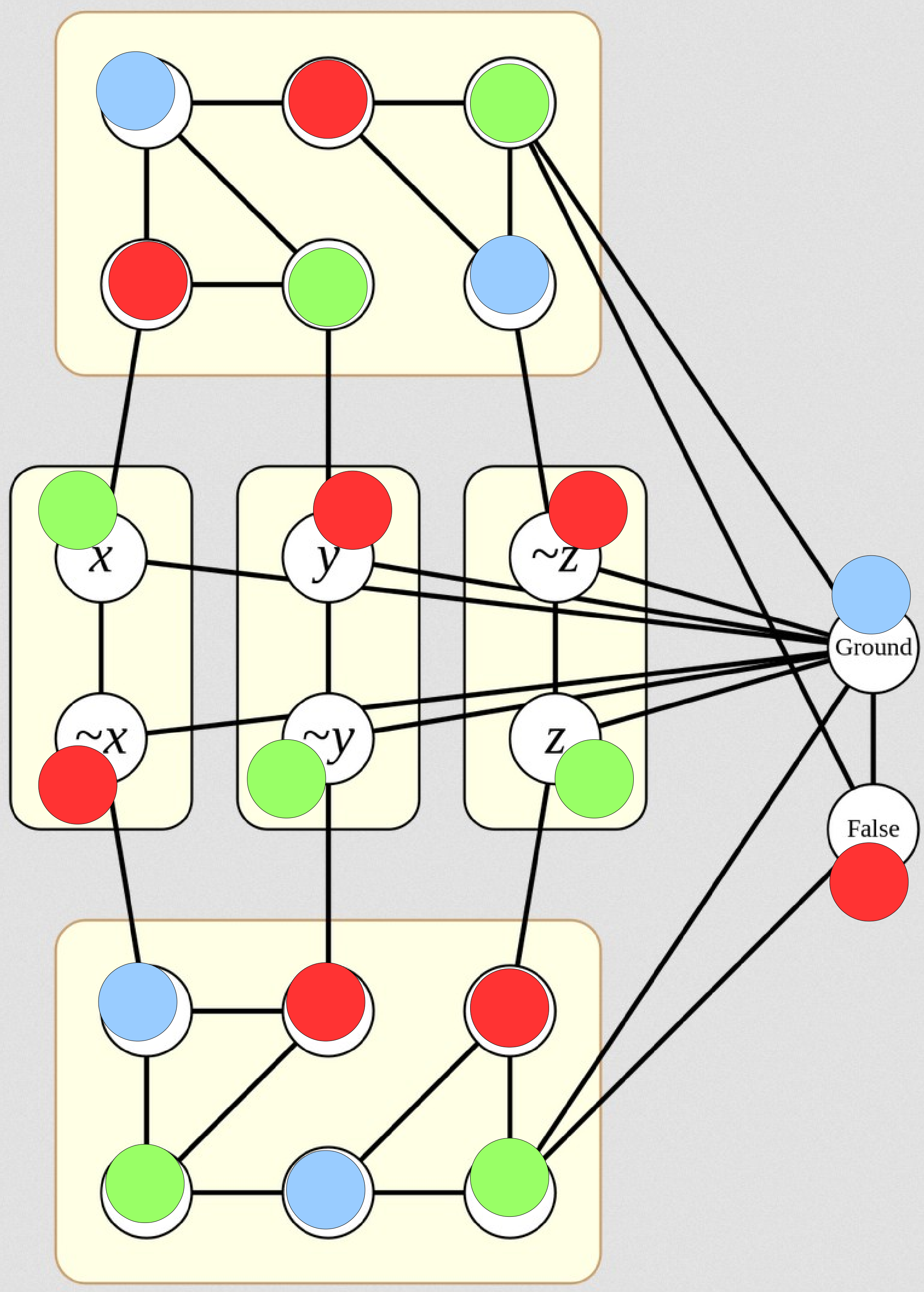


Gadget Example

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

  (x || y || !z) &&

  (!x || !y || z)

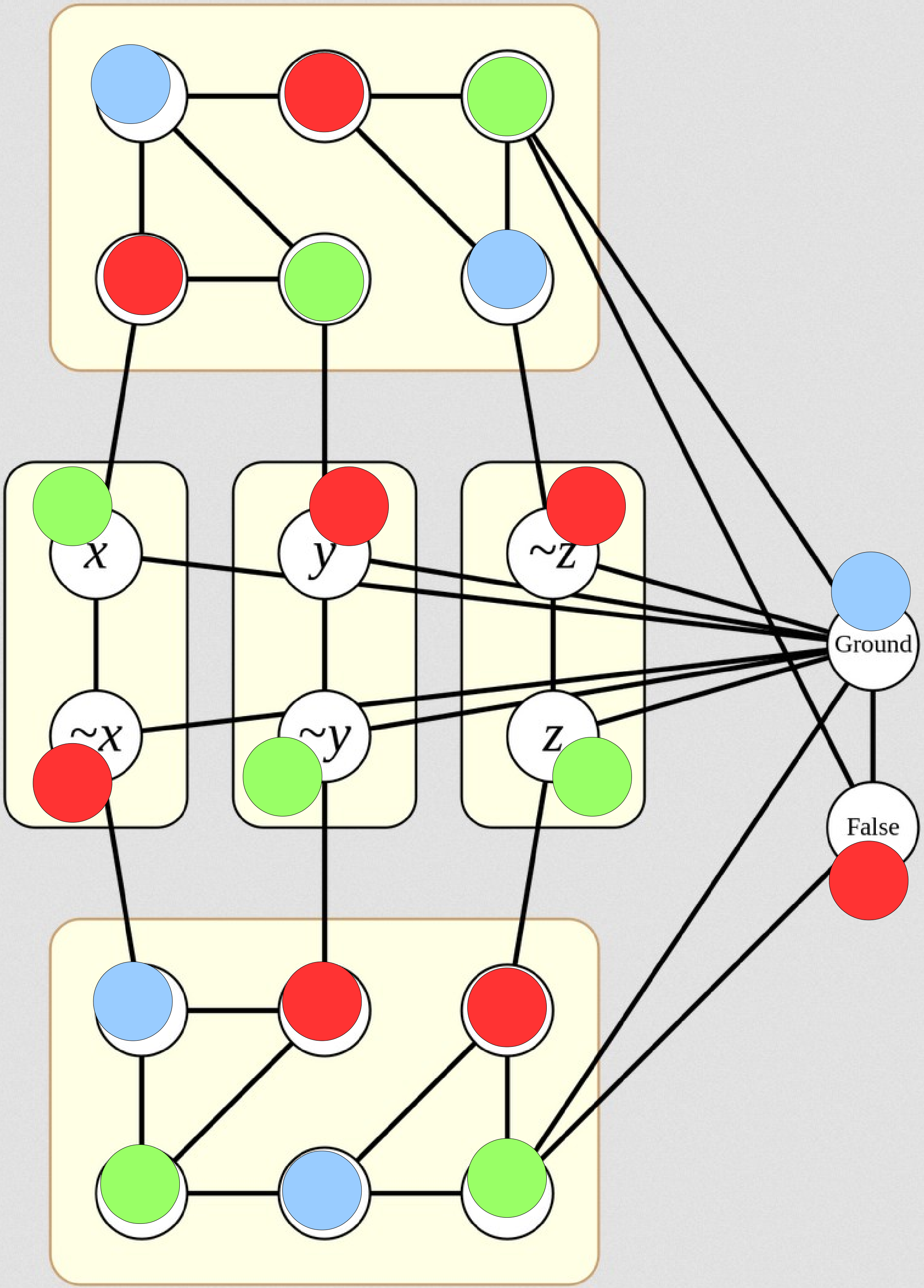


Gadget Example

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

  (x || y || !z) &&

  (!x || !y || z)
- X = true
- Y = false
- Z = 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.

Logic side	Programming side
axiom	variable
introduction rule	constructor
elimination rule	destructor
normal deduction	normal form
normalisation of deductions	weak normalisation
provability	type inhabitation problem
intuitionistic tautology	inhabited type

Authors

- This American author, screenwriter and filmmaker is associated with Jurassic Park, The Lost World, Westworld, ER, and (allegedly) The Pitt. In 1983, he also wrote Electronic Life, a book that introduced BASIC programming to readers and predicted the rise of computer networks (including sharing pictures online). A genus of dinosaur was named in his honor.

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

Test	Inputs			Output		Passed?
	in	up	down	expected	observed	
1	1	0	100	0	0	✓
2	1	11	110	1	0	✗
3	0	100	50	1	1	✓
4	1	-20	60	1	0	✗
5	0	0	10	0	0	✓
6	0	0	-10	1	1	✓

```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;
}
```

Test	in	up	down	expected	observed	Passed?
	Inputs			Output		
1	1	0	100	0	0	✓
2	1	11	110	1	0	✗
3	0	100	50	1		
4	1	-20	60	1		
5	0	0	10	0		
6	0	0	-10	1		

???

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

Test	Inputs: in	up	down	Output: expected	observed	Passed?
1	1	0	100	0	0	✓
2	1	11	110	1	0	✗
3	0	100	50	1		
4	1	-20	60	1		
5	0	0	10	0		
6	0	0	-10	1		

???

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

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

  return 0;
}
```

"Heart" insights:

Multiple tests make Synthesis difficult.

Multiple path conditions make Reachability difficult.

Your thoughts?

```
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;
}
```

Test	Inputs			Out
	in	up	down	expected
1	1	0	100	0
2	1	11	110	1
3	0	100	50	1
4	1	-20	60	1
5	0	0	10	0
6	0	0	-10	1

Convert

```
int c0, c1, c2, c3, c4; /* global input */

int p_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;
}

int main() {
    if(p_is_upward(1,0,100) == 0 &&
       p_is_upward(1,11,110) == 1 &&
       p_is_upward(0,100,50) == 1 &&
       p_is_upward(1,-20,60) == 1 &&
       p_is_upward(0,0,10) == 0 &&
       p_is_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).

```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;
}
```

Test	Inputs in	up	down	Output expected
1	1	0	100	0
2	1	11	110	1
3	0	100	50	1
4	1	-20	60	1
5	0	0	[illegible]	0
6	0	0	10	1

```c
int c0, c1, c2, c3, c4; /* global input */

int p_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;
}

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

Lemma 1: "Method Executions Agree On Variables"

Lemma 2: "Reaching L Corresponds To Passing All Tests"

Lemma 1 (Agree on Vars)

- Let Q be the input synthesis instance method with template variables $v_1 \ldots v_n$.
- Let P = Gadget(Q) be the reachability instance corresponding to method P.
- For all states σ_1, σ_2, σ_3, all values $c_1 \ldots c_n$, all inputs values x, it holds that
- If $\sigma_1(v_i) = c_i$, then $\langle P(x), \sigma_1 \rangle \downarrow \sigma_2$ iff

  $\langle inst(Q,\overline{c}), \sigma_1 \rangle \downarrow \sigma_3$

  and for all $y \neq v_i$, $\sigma_2(y) = \sigma_3(y)$.

Lemma 1 Proof

- If $\sigma_1(v_i) = c_i$, then $<P(x), \sigma_1> \downarrow \sigma_2$ iff
  $<inst(Q,\bar{c}), \sigma_1> \downarrow \sigma_3$
  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 $D_1 :: \langle P(x), \sigma_1\rangle \downarrow \sigma_2$ iff
  $D_2 :: \langle inst(Q,\bar{c}), \sigma_1\rangle \downarrow \sigma_3$
  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:

$$\frac{\sigma_2 = \sigma_1 [\, a \rightarrow \sigma_1(v_i) \,]}{<\, a := v_i,\ \sigma_1 \,> \downarrow \sigma_2}$$

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

$$\frac{\sigma_3 = \sigma_1 [\, a \rightarrow c_i \,]}{<\, a := \mathit{exp},\ \sigma_1 \,> \downarrow \sigma_3}$$

- where *exp* = $\mathrm{inst}(\boxed{c_i}, \bar{c}) = c_i$

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 = Gadget(Q) be the reachability instance corresponding to method P.
- For all states σ_1, σ_2, σ_3, all values $c_1 \ldots c_n$, all inputs values x, it holds that
- If $\sigma_1(v_i) = c_i$, then $\langle P(x), \sigma_1 \rangle \downarrow \sigma_2$ iff

  $\langle inst(Q,\overline{c}), \sigma_1 \rangle \downarrow \sigma_3$

  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 = Gadget(Q) be the reachability instance method *main*.
- The execution of P reaches L starting from state σ_1 iff $\sigma_1 \models$ *wp*(result = inst(Q,$\bar{c}$)($input_1$), result = $output_1$) && … *wp*(result = inst(Q,$\bar{c}$)($input_n$), 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 $f(input_1) = output_1$ && … $f(intput_n) = output_n$.
- By standard weakest precondition definitions for if, conjunction, equality and function calls, we have that L is reachable iff $\sigma_1 \models wp(result = f(input_1), result = output_1)$ && … $wp(result = f(input_n), result = output_n)$.

Lemma 2 Proof

- Have: L is reachable iff $\sigma_1 \models wp(\text{result} = f(\text{input}_1), \text{result} = \text{output}_1)$ && … $wp(\text{result} = f(\text{input}_n), \text{result} = \text{output}_n)$.
- Want: L is reachable iff $\sigma_1 \models wp(\text{result} = \text{inst}(Q, \bar{c})(\text{input}_1), \text{result} = \text{output}_1)$ && … $wp(\text{result} = \text{inst}(Q, \bar{c})(\text{input}_n), \text{result} = \text{output}_n)$
- To show: $\sigma_1 \models wp(\text{result} = f(\text{input}_i), \text{result} = \text{output}_i)$ iff $\sigma_1 \models wp(\text{result} = \text{inst}(Q, \bar{c})(\text{input}_i), \text{result} = \text{output}_i)$

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: $<$ result = f(input$_i$) , σ_1 $> \downarrow \sigma_2$ iff $\sigma_2 \models$ result = output$_i$.
- By Lemma 1, we have $<$ result = inst(Q, $\bar{c}$)(input$_i$) , σ_1 $> \downarrow \sigma_3$ iff $\sigma_1(y) = \sigma_3(y)$ for all $y \neq v_i$.
- Since "result" $\neq v_i$, σ_1(result) = σ_3(result) ("Lemma1") and σ_3(result) = output$_i$ ("Have"). Transitively …
- 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 <$input_1$, $output_n$>.
- Let P = Gadget(Q) be the reachability instance method *main*.
- There exist parameter values c_i such that for all <input,output>, $inst(Q,\bar{c})(input) = output$ iff there exist input values t_i such that the execution of P with $v_i \rightarrow t_i$ 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
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;
}
```

	Inputs			Output		
Test	in	up	down	expected	observed	Passed?
1	1	0	100	0	0	✓
2	1	11	110	1	0	✗
3	0	100	50	1	1	✓
4	1	-20	60	1	0	✗
5	0	0	10	0	0	✓
6	0	0	-10	1	1	✓

Reachability to Synthesis Example

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

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

  return 0;
}
```

Convert

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

    return 0;
}
```

Test suite: $Q() = 1$

```
int q_main() {
  /* Find [x] and [y].
     Equivalently,
       synthesize:
    x = c_x
    y = c_y */
  try {
    q_P();
  } 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

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

- 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), HW6 (tigen), Axiomatic Semantics and GenProg (last lecture) are all “secretly the same thing”

    = “statically reason about dynamic execution”