

CS320: Programming Languages

Introduction to Automatic Software Verification

Hongseok Yang
KAIST

Guest lecture



Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you. (0% complete)

If you'd like to know more, you can search online later for this error: HAL_INITIALIZATION_FAILED

You need to restart your computer. Hold down the Power button for several seconds or press the Restart button.

Veuillez redémarrer votre ordinateur. Maintenez la touche de démarrage enfoncée pendant plusieurs secondes ou bien appuyez sur le bouton de réinitialisation.

Sie müssen Ihren Computer neu starten. Halten Sie dazu die Einschalttaste einige Sekunden gedrückt oder drücken Sie die Neustart-Taste.

コンピュータを再起動する必要があります。パワーボタンを数秒間押し続けるか、リセットボタンを押してください。

Software verification

- Active research area in computer science.
- Aims at verifying “no blue screen”, i.e., programs do not crash due to errors.
- Uses many ideas from the PL research.

[Quiz] Who wrote the earliest paper on software verification?



Hoare



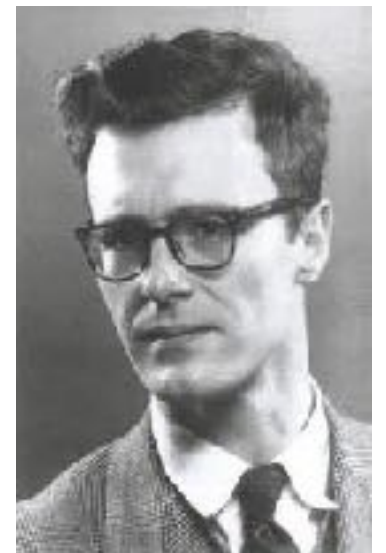
Floyd



Turing



Ryu



Dijkstra

[Quiz] Who wrote the earliest paper on software verification?



Hoare



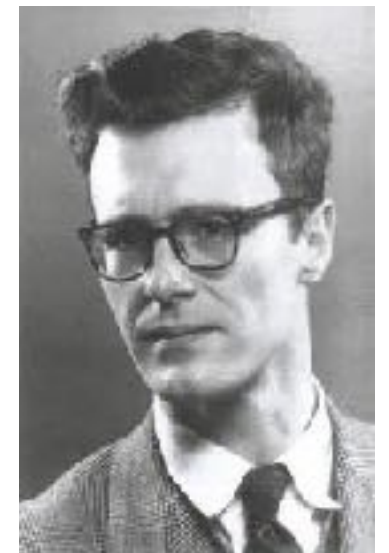
Floyd



Turing



Ryu



Dijkstra

Turing in June 1949

Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

EDSAC
First storable computer
First time to have "subroutine"

Turing's idea

Use intermediate assertions.

Turing's idea

Use intermediate assertions.

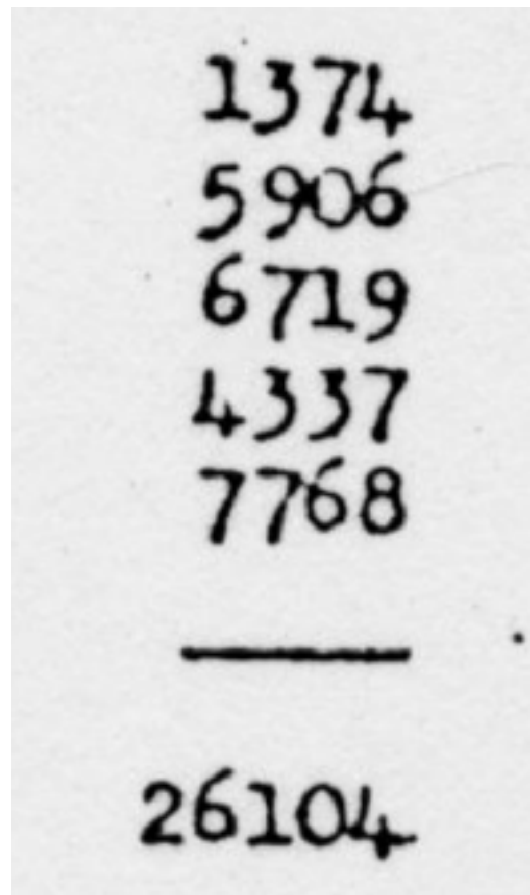
x = 60
and i = 2

x = 30
and i = 1

x = 2970
and i = 29

Verify that this program computes “ $100 * 30$ ”.

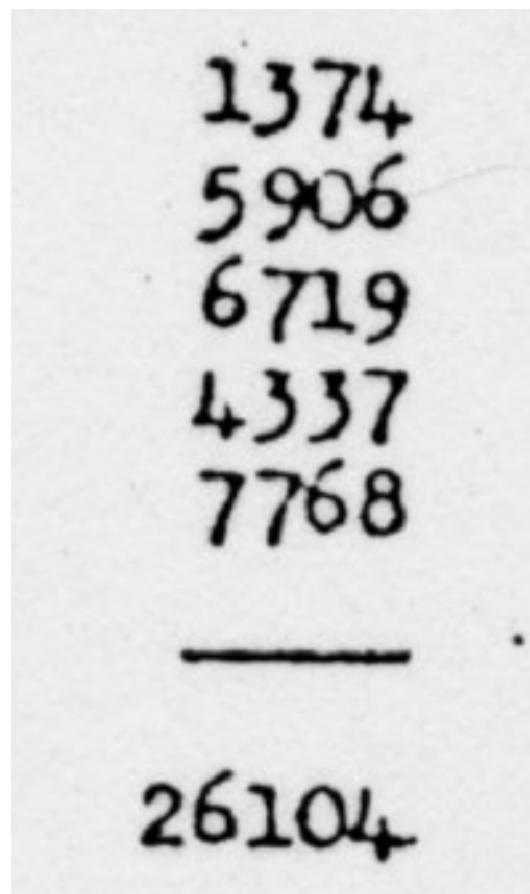
Turing's example



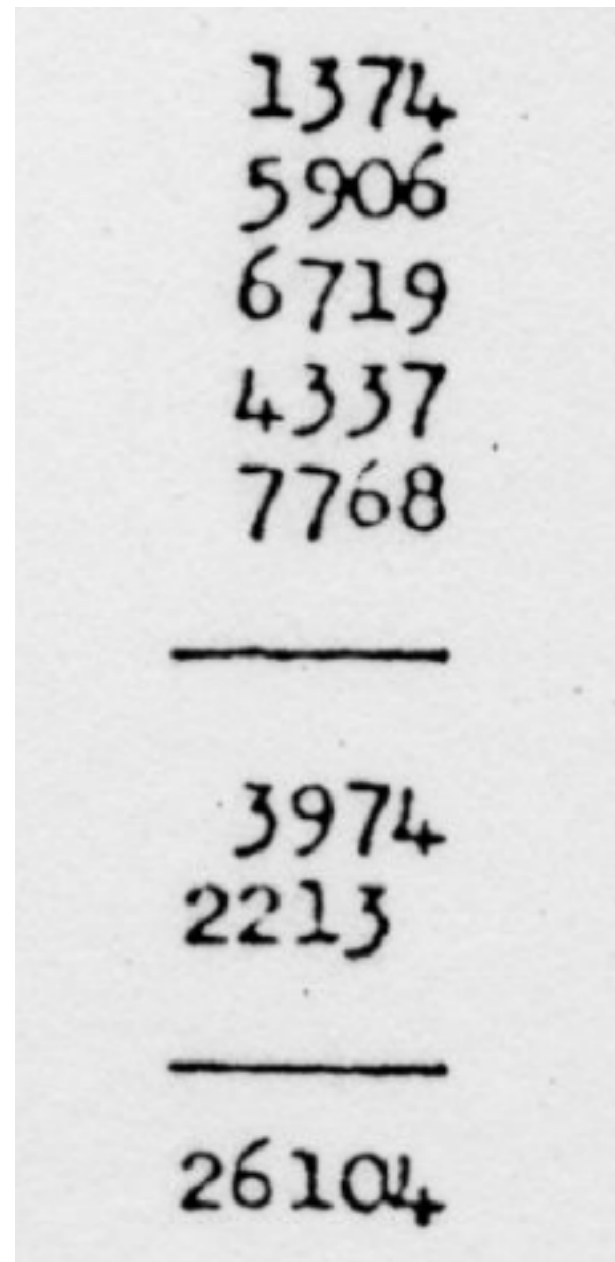
1374
5906
6719
4337
7768
—
26104

A photograph of a piece of paper with handwritten numbers in black ink. The numbers are arranged in a vertical column. The first five numbers are 1374, 5906, 6719, 4337, and 7768. A horizontal line is drawn below the fifth number. Below the line is the number 26104. The handwriting is somewhat cursive and the paper appears slightly aged or off-white.

Turing's example



1374
5906
6719
4337
7768
—
26104



1374
5906
6719
4337
7768
—
3974
2213
—
26104

Turing's example

1374
5906
6719
4337
7768
—
26104

1374
5906
6719
4337
7768
—
3974
2213
—
26104

$$4+6+9+7+8 = 34$$

Turing's example

1374
5906
6719
4337
7768
—
26104

1374
5906
6719
4337
7768
—
3974
2213
—
26104

$$4+6+9+7+8 = 34$$

$$7+0+1+3+6 = 17$$

Turing's example

1374
5906
6719
4337
7768
—
26104

1374
5906
6719
4337
7768
—
3974
2213
—
26104

$$4+6+9+7+8 = 34$$

$$7+0+1+3+6 = 17$$

$$3+9+7+3+7 = 29$$

Turing's example

1374
5906
6719
4337
7768
—
26104

1374
5906
6719
4337
7768
—
3974
~~2213~~
—
26104

$$4+6+9+7+8 = 34$$

$$7+0+1+3+6 = 17$$

$$3+9+7+3+7 = 29$$

$$1+5+6+4+7 = 23$$

Turing's example

1374
5906
6719
4337
7768
—
26104

1374
5906
6719
4337
7768
—
3974
2213
—
26104

$$4+6+9+7+8 = 34$$

$$7+0+1+3+6 = 17$$

$$3+9+7+3+7 = 29$$

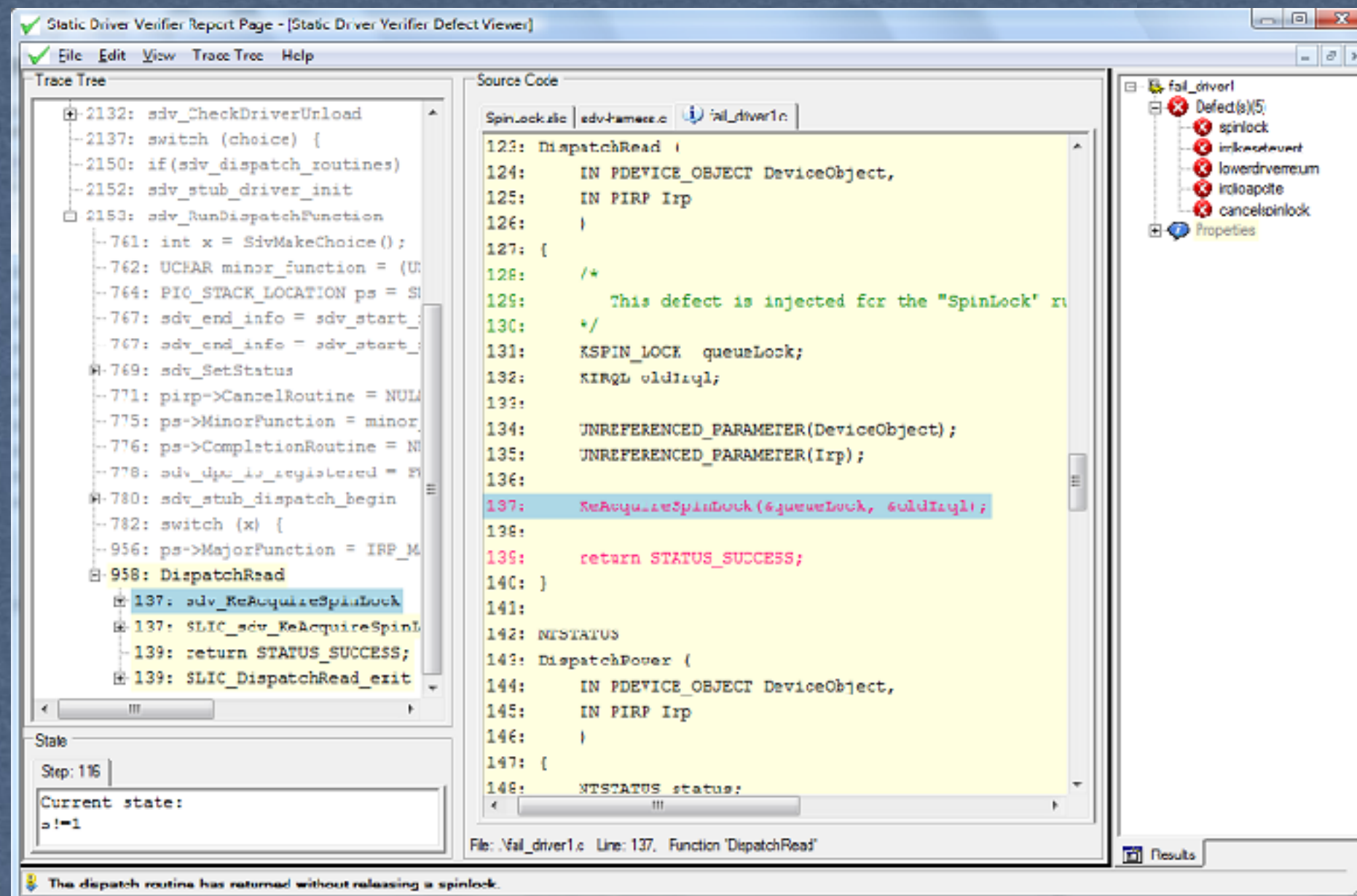
$$1+5+6+4+7 = 23$$

Intermediate assertions

- Form the basis of modern verification methods.
- Inferred automatically by commercial tools nowadays.

Commercial tools in 2017

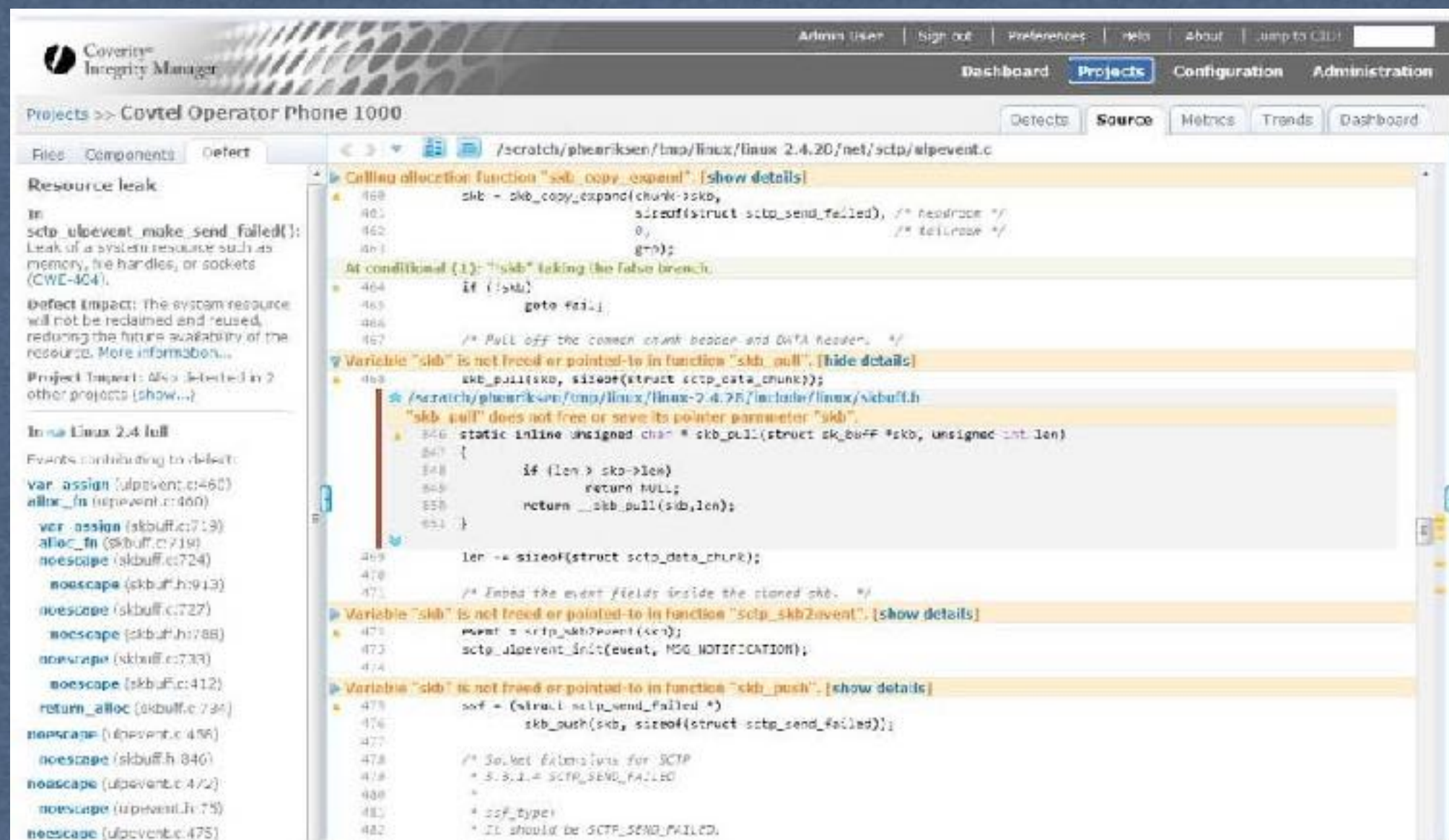
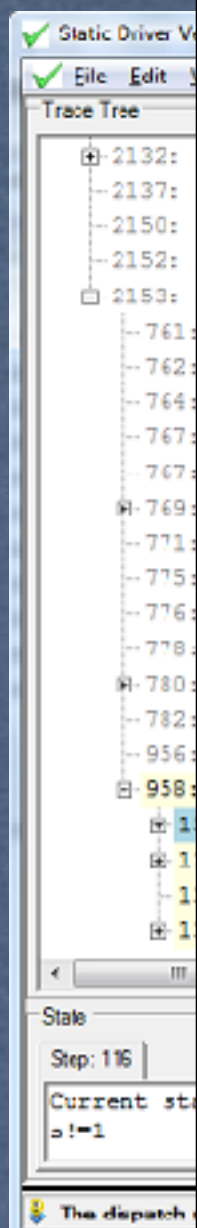
Microsoft SDV



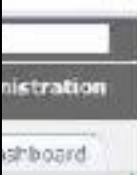
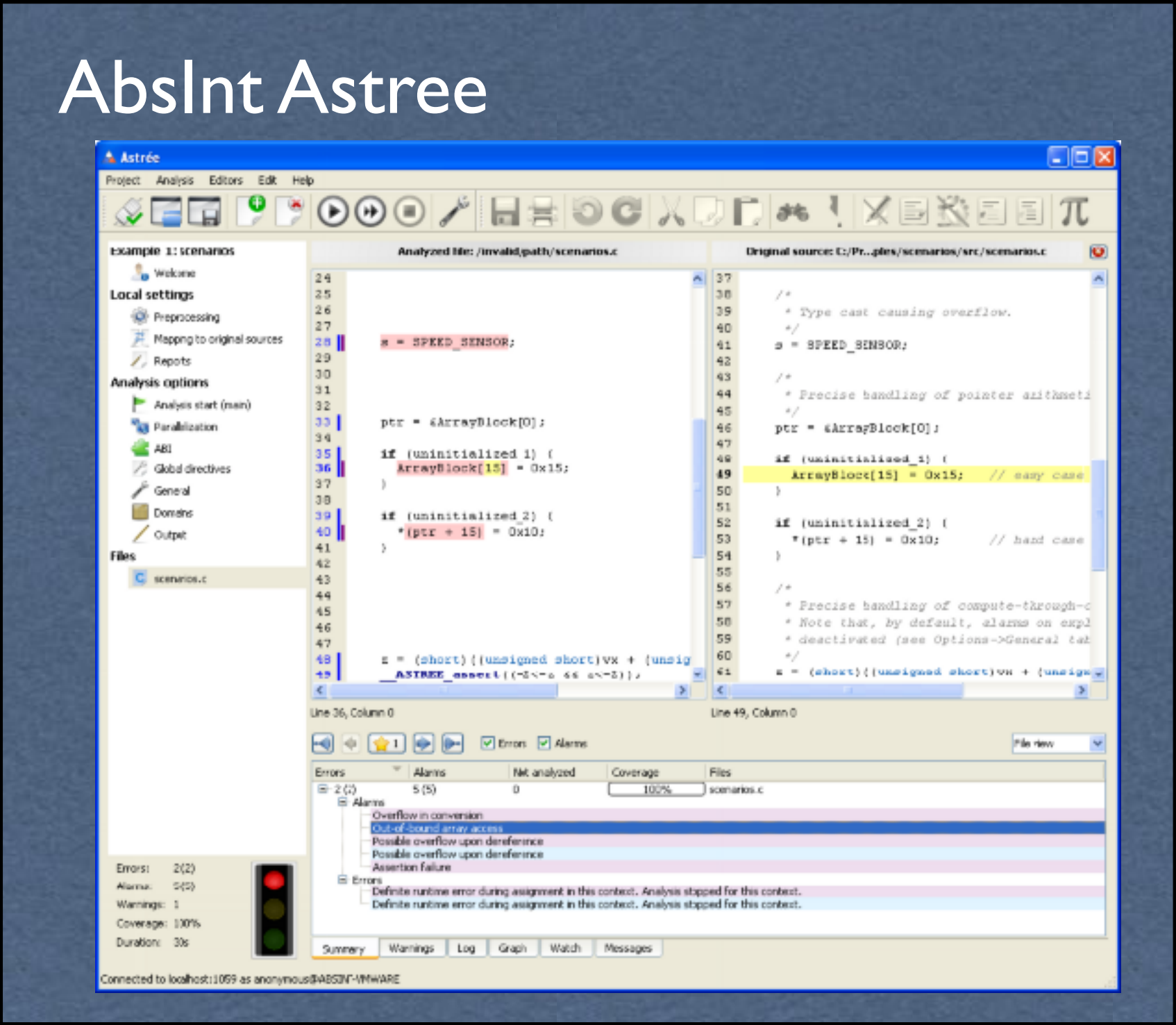
Commercial tools in 2017

Micro

Synopsis Coverity



Micro

[illegible]

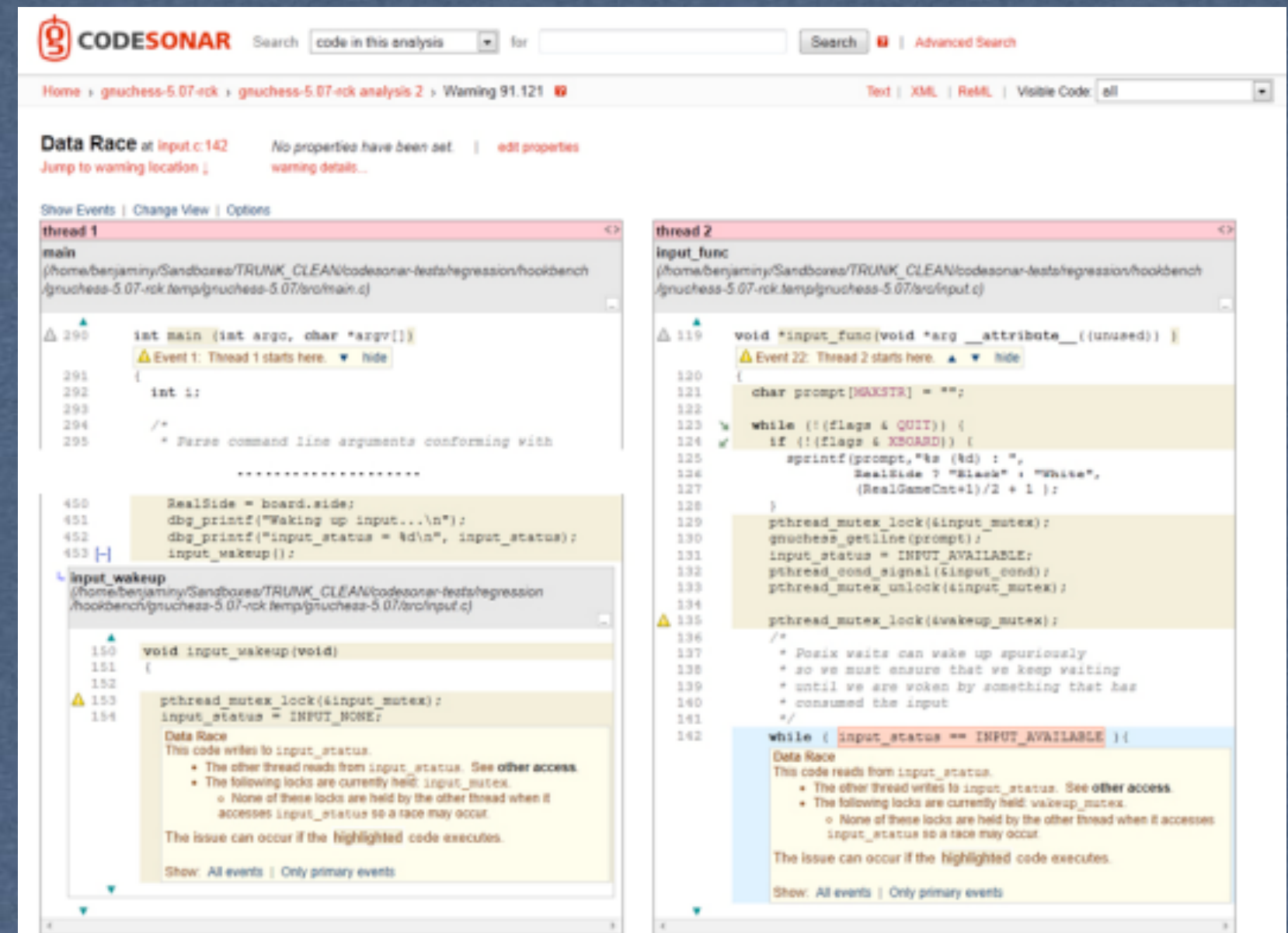
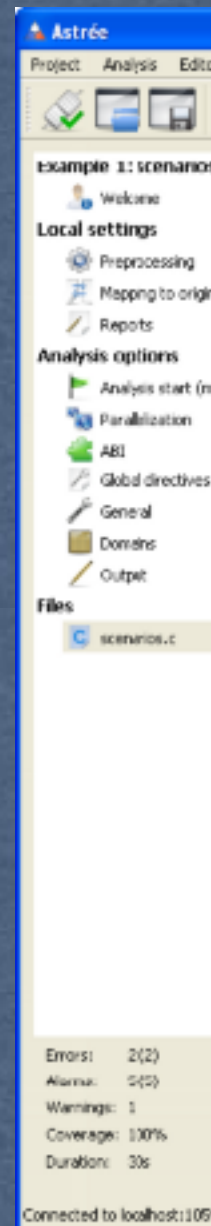
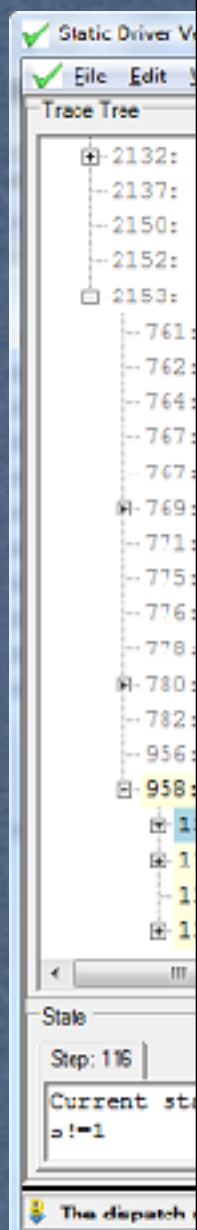
Commercial tools in 2017

Micro

Syn

AbsIn

Grammtech Codesonar



Commercial tools in 2017

Micro

Syn

AbsIn

Grammatech Codesonar

Facebook Infer



Infer

Abstraction

- Key idea behind automation.
- Keep only important properties of programs. Forget all the rest.

Abstraction for arithmetic calculation

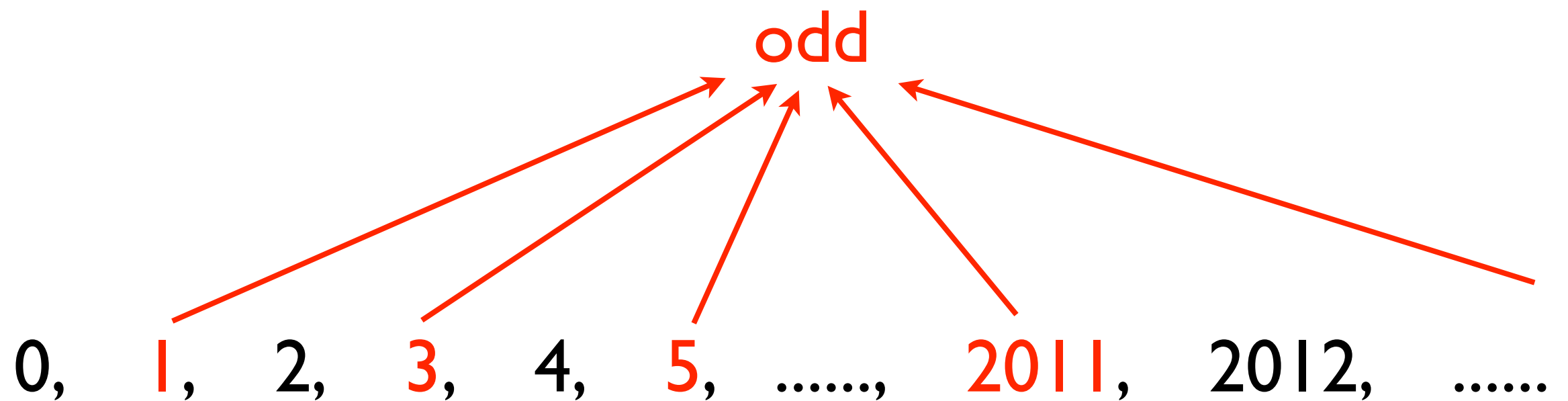
Question

- $48729405 + 38572988 = 87302392$?

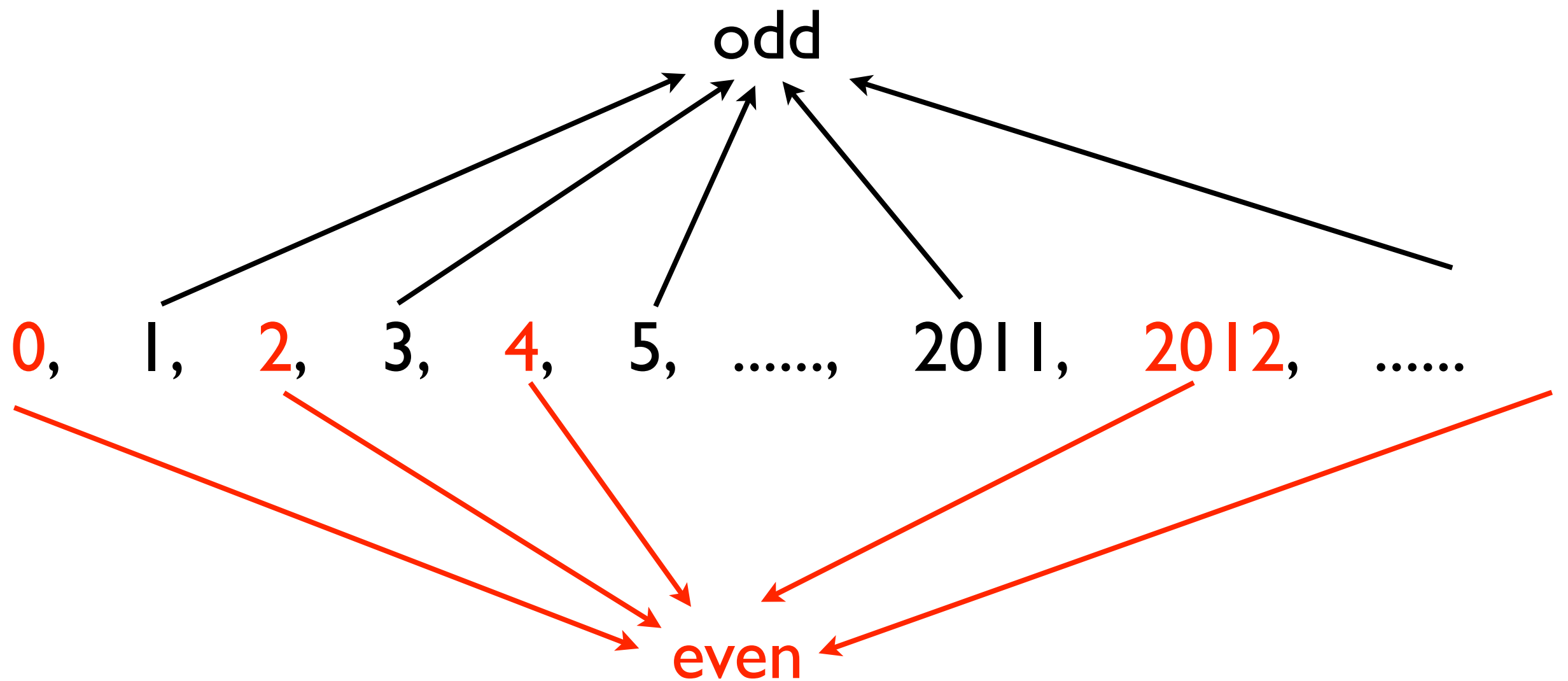
Even/odd abstraction

0, 1, 2, 3, 4, 5,, 2011, 2012,

Even/odd abstraction



Even/odd abstraction



Even/odd abstraction

- $\text{even} + \text{even} =$
- $\text{even} + \text{odd} =$
- $\text{odd} + \text{even} =$
- $\text{odd} + \text{odd} =$

Even/odd abstraction

- $\text{even} + \text{even} = \text{even}$
- $\text{even} + \text{odd} =$
- $\text{odd} + \text{even} =$
- $\text{odd} + \text{odd} =$

Even/odd abstraction

- $\text{even} + \text{even} = \text{even}$
- $\text{even} + \text{odd} = \text{odd}$
- $\text{odd} + \text{even} =$
- $\text{odd} + \text{odd} =$

Even/odd abstraction

- $\text{even} + \text{even} = \text{even}$
- $\text{even} + \text{odd} = \text{odd}$
- $\text{odd} + \text{even} = \text{odd}$
- $\text{odd} + \text{odd} =$

Even/odd abstraction

- $\text{even} + \text{even} = \text{even}$
- $\text{even} + \text{odd} = \text{odd}$
- $\text{odd} + \text{even} = \text{odd}$
- $\text{odd} + \text{odd} = \text{even}$

Question

- $48729405 + 38572988 = 87302392$?

Question

odd

- $48729405 + 38572988 = 87302392 ?$

Question

odd

even

- $48729405 + 38572988 = 87302392 ?$

Question

odd

even

even

- $48729405 + 38572988 = 87302392 ?$

odd + even = odd. So, the answer is no.

Exercise

- $48729405 * 38572988 = 1879638754312141 ?$

Exercise

- $48729405 * 38572988 = 1879638754312141 ?$
- No! odd * even = even.

Exercise

- $48729405 * 38572988 = 1879638754312142 ?$

Exercise

- $48729405 * 38572988 = 1879638754312142 ?$
- No! 5 * 8 = 0

Abstraction for software verification

Fibonacci number

$$F_0 = F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$$

```
1:  assert(n >= 1);
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
```

Fibonacci number


$$F_0 = F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$$

```
1:  assert(n >= 1);    [n:3,x:0,y:0]
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
```

Fibonacci number

$$F_0 = F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$$

```
1:  assert(n >= 1);    [n:3,x:0,y:0]
2:  x = 1;
3:  y = 1;              [n:3,x:1,y:1]
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
```



Fibonacci number

$$F_0 = F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$$

```
1:  assert(n >= 1);    [n:3,x:0,y:0]
2:  x = 1;              ↓
3:  y = 1;              [n:3,x:1,y:1]
4:  while (n > 1) {     ↓
5:      (x,y) = (y,x+y);
6:      n = n-1;        [n:2,x:1,y:2]
7:  }
```

Fibonacci number

$$F_0 = F_1 = 1 \quad F_n = F_{n-1} + F_{n-2}$$

```
1:  assert(n >= 1);    [n:3,x:0,y:0]
2:  x = 1;              ↓
3:  y = 1;              [n:3,x:1,y:1]  [n:2,x:1,y:2]
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;        [n:2,x:1,y:2]
7:  }
```

The diagram illustrates the state transitions of a Fibonacci algorithm. It shows a vertical sequence of states connected by downward arrows: $[n:3, x:0, y:0]$ to $[n:3, x:1, y:1]$ to $[n:2, x:1, y:2]$. A red arrow points from the $[n:2, x:1, y:2]$ state to a red $[n:2, x:1, y:2]$ state to its right.

Fibonacci number

$$F_0 = F_1 = 1 \quad F_n = F_{n-1} + F_{n-2}$$

```
1:  assert(n >= 1);    [n:3,x:0,y:0]
2:  x = 1;
3:  y = 1;              [n:3,x:1,y:1]  [n:2,x:1,y:2]
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
```

The diagram illustrates the state transitions of the Fibonacci algorithm. It starts with the initial state $[n:3, x:0, y:0]$ at line 1. Line 2 updates x to 1, leading to $[n:3, x:1, y:1]$. Line 3 updates y to 1, leading to $[n:2, x:1, y:2]$. Line 4 (while loop) leads to $[n:2, x:1, y:2]$ at line 5. Line 5 updates (x, y) to $(y, x+y)$, leading to $[n:1, x:2, y:3]$ at line 6. Line 6 updates n to $n-1$, leading back to $[n:2, x:1, y:2]$ at line 5. Line 7 ends the loop.

Fibonacci number

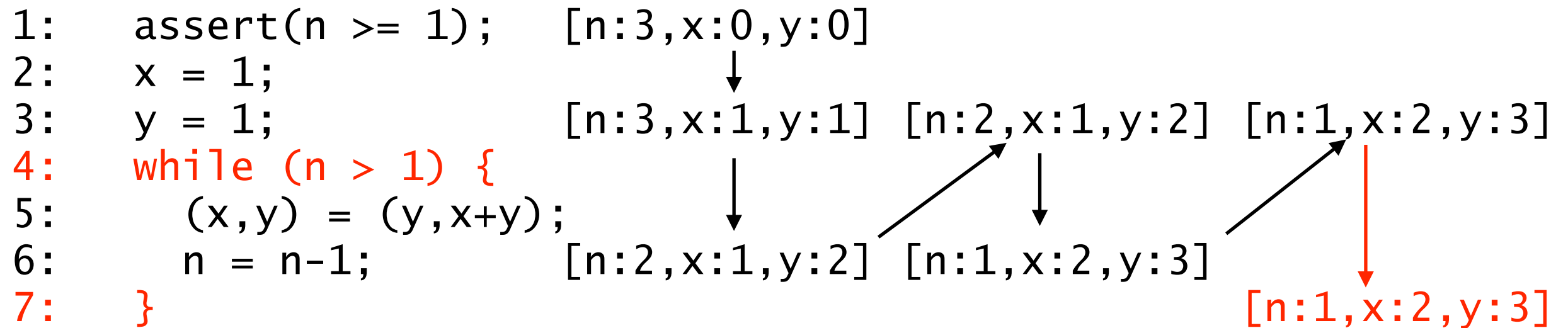
$$F_0 = F_1 = 1 \quad F_n = F_{n-1} + F_{n-2}$$

```
1:  assert(n >= 1);    [n:3,x:0,y:0]
2:  x = 1;
3:  y = 1;              [n:3,x:1,y:1] [n:2,x:1,y:2] [n:1,x:2,y:3]
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;        [n:2,x:1,y:2] [n:1,x:2,y:3]
7:  }
```

The diagram illustrates the state transitions of a Fibonacci algorithm. The initial state is $[n:3, x:0, y:0]$. After line 2, it becomes $[n:3, x:1, y:1]$. After line 3, it becomes $[n:2, x:1, y:2]$. After line 4, it becomes $[n:1, x:2, y:3]$. The state $[n:1, x:2, y:3]$ is highlighted in red. Arrows indicate the flow of execution from one state to the next.

Fibonacci number

$$F_0 = F_1 = 1 \quad F_n = F_{n-1} + F_{n-2}$$



Fibonacci number

$$F_0 = F_1 = 1 \quad F_n = F_{n-1} + F_{n-2}$$

```
1:  assert(n >= 1);    [n:3, x:0, y:0]
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
```

Diagram illustrating the state transitions of the Fibonacci algorithm:

- Initial state: $[n:3, x:0, y:0]$
- After `x = 1`: $[n:3, x:1, y:1]$
- After `y = 1`: $[n:3, x:1, y:1]$
- After `while (n > 1) {`: $[n:2, x:1, y:2]$
- After `(x,y) = (y,x+y);`: $[n:2, x:1, y:2]$
- After `n = n-1;`: $[n:1, x:2, y:3]$
- After `}`: $[n:1, x:2, y:3]$

Fibonacci number

$$F_0 = F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$$

```
1:  assert(n >= 1);
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  assert(y >= 0);
```

Fibonacci number

$$F_0 = F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$$

```
1:  assert(n >= 1);
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  assert(y >= 0);
```

Because it computes fib. number.

Fibonacci number

$$F_0 = F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$$

```
1:  assert(n >= 1);
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  assert(y >= 0);
```

Because it computes fib. number.

Irrelevant n. No negative numbers nor minus.

Simple sign abstraction

- Abstract values:

\top
|
+

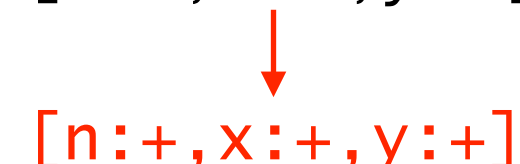
- An abstract state is a map from variables to abstract values. E.g. $[n:\top, x:+, y:+]$.

Analysing Fibonacci with simple sign abstraction

```
1:  assert(n >= 1);    [n:+,x:τ,y:τ]
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  assert(y >= 0);
```

Analysing Fibonacci with simple sign abstraction

```
1:  assert(n >= 1);    [n:+,x:⊤,y:⊤]
2:  x = 1;
3:  y = 1;              [n:+,x:⊤,y:⊤]
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  assert(y >= 0);
```



Analysing Fibonacci with simple sign abstraction

1:	assert($n \geq 1$);	$[n:+, x:\top, y:\top]$
2:	$x = 1$;	\downarrow
3:	$y = 1$;	$[n:+, x:+, y:+]$
4:	while ($n > 1$) {	\downarrow
5:	$(x, y) = (y, x+y)$;	\downarrow
6:	$n = n-1$;	$[n:\top, x:+, y:+]$
7:	}	
8:	assert($y \geq 0$);	

Analysing Fibonacci with simple sign abstraction

1:	assert($n \geq 1$);	$[n:+, x:\top, y:\top]$
2:	$x = 1$;	\downarrow
3:	$y = 1$;	$[n:+, x:+, y:+] \rightarrow [n:\top, x:+, y:+]$
4:	while ($n > 1$) {	\downarrow
5:	$(x, y) = (y, x+y)$;	$[n:\top, x:+, y:+] \rightarrow [n:\top, x:+, y:+] \text{ (via red arrow)}$
6:	$n = n-1$;	
7:	}	
8:	assert($y \geq 0$);	

Analysing Fibonacci with simple sign abstraction

```
1:  assert(n >= 1);
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  assert(y >= 0);
```

Control flow graph illustrating the analysis of the Fibonacci code snippet:

- Initial state: $[n:+, x:\tau, y:\tau]$
- After line 2: $[n:+, x:+, y:+] \leftarrow [n:+, x:\tau, y:\tau]$
- After line 3: $[n:+, x:+, y:+] \leftarrow [n:+, x:+, y:+] \leftarrow [n:+, x:\tau, y:\tau]$
- After line 4 (loop header): $[n:+, x:+, y:+] \leftarrow [n:+, x:+, y:+] \leftarrow [n:+, x:\tau, y:\tau]$
- After line 5 (loop body): $[n:+, x:+, y:+] \leftarrow [n:+, x:+, y:+] \leftarrow [n:+, x:\tau, y:\tau]$
- After line 6 (loop body): $[n:+, x:+, y:+] \leftarrow [n:+, x:+, y:+] \leftarrow [n:+, x:\tau, y:\tau]$
- After line 7 (loop body): $[n:+, x:+, y:+] \leftarrow [n:+, x:+, y:+] \leftarrow [n:+, x:\tau, y:\tau]$
- After line 8 (loop body): $[n:+, x:+, y:+] \leftarrow [n:+, x:+, y:+] \leftarrow [n:+, x:\tau, y:\tau]$

Analysing Fibonacci with simple sign abstraction

```
1:  assert(n >= 1);  
2:  x = 1;  
3:  y = 1;  
4:  while (n > 1) {  
5:      (x,y) = (y,x+y);  
6:      n = n-1;  
7:  }  
8:  assert(y >= 0);
```

Control flow graph illustrating the analysis of the Fibonacci code snippet:

- Initial state: $[n:+, x:\tau, y:\tau]$
- After line 3: $[n:+, x:+, y:+]$
- After line 5: $[n:\tau, x:+, y:+]$
- After line 7: $[n:\tau, x:+, y:+]$
- Final state (red): $[n:\tau, x:+, y:+]$

Analysing Fibonacci with simple sign abstraction

```
1:  assert(n >= 1);  
2:  x = 1;  
3:  y = 1;  
4:  while (n > 1) {  
5:      (x,y) = (y,x+y);  
6:      n = n-1;  
7:  }  
8:  assert(y >= 0);
```

Control flow graph illustrating the analysis of the Fibonacci code:

- Initial state: $[n:+, x:\tau, y:\tau]$
- After line 3: $[n:+, x:+, y:+]$
- After line 5: $[n:\tau, x:+, y:+]$
- From $[n:\tau, x:+, y:+] (line 5)$, the graph branches to $[n:\tau, x:+, y:+] (line 6)$ and $[n:\tau, x:+, y:+] (line 8)$.
- From $[n:\tau, x:+, y:+] (line 6)$, the graph goes to $[n:\tau, x:+, y:+] (line 8)$.
- From $[n:\tau, x:+, y:+] (line 8)$, the graph goes to $[n:\tau, x:+, y:+] (line 8)$.

Finding a good abstraction

```
1:  assert(n >= 1);
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  assert(y >= 0);
```

- Typically done by hand.

Finding a good abstraction

```
1:  assert(n >= 1);
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  y = y-1;
9:  assert(y >= 0);
```

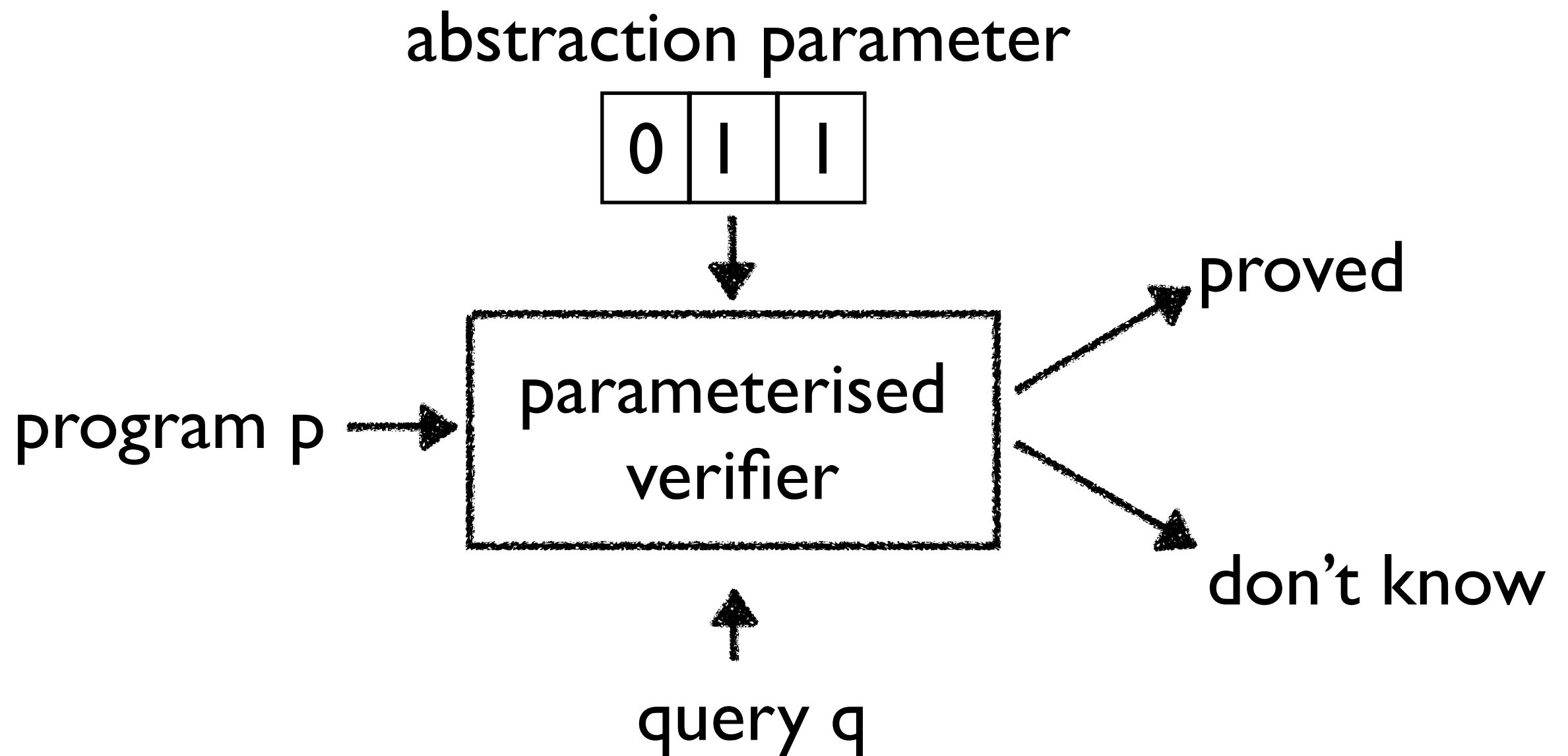
- Typically done by hand.
- Tricky.
- Active research area:
how to automate this?

How to find a good
program abstraction
automatically?

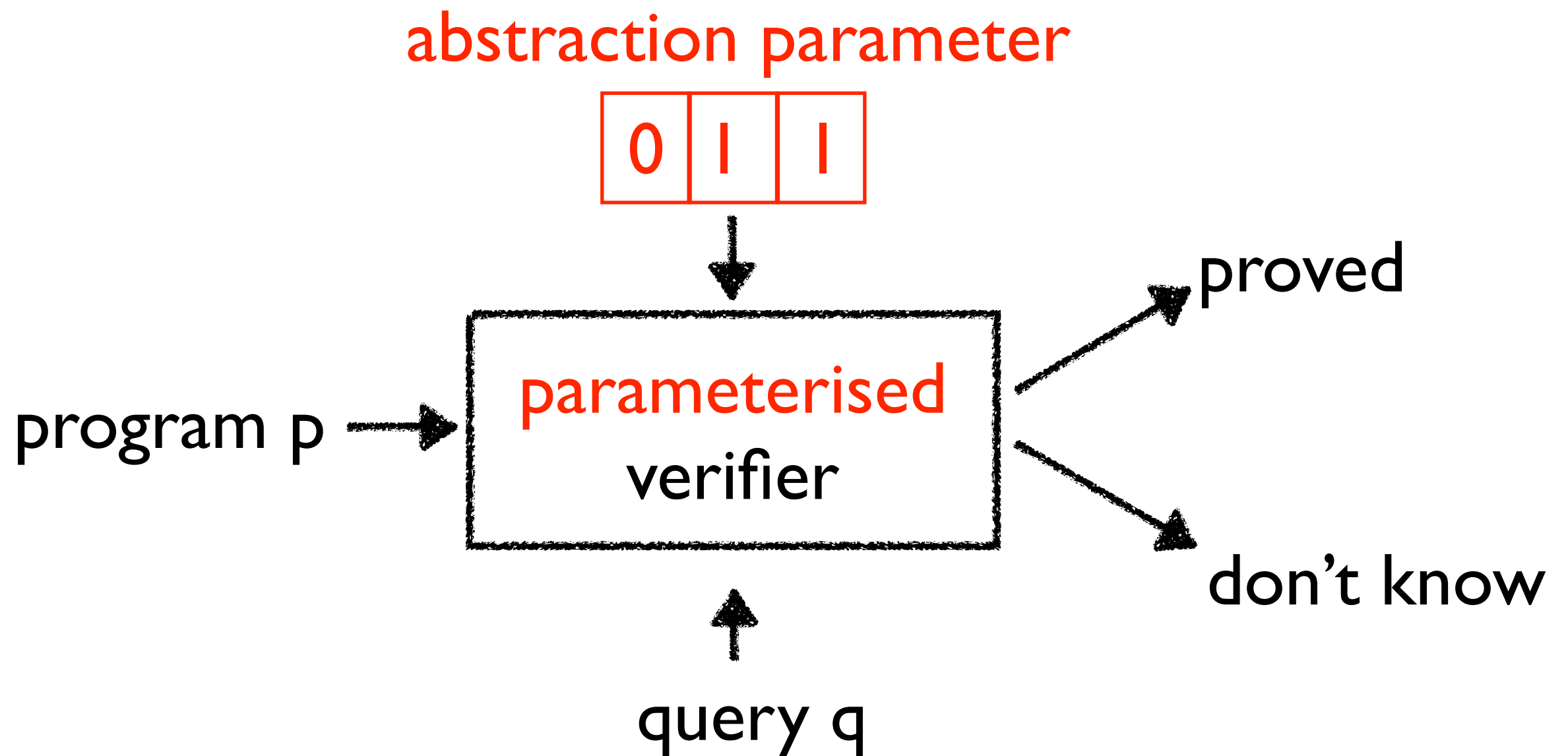
Abstraction finding as search problem

- Formulate abstraction finding as a search problem.
- Choose search space carefully.
- Develop an efficient search algorithm.
- Direction pursued by me and my friends.

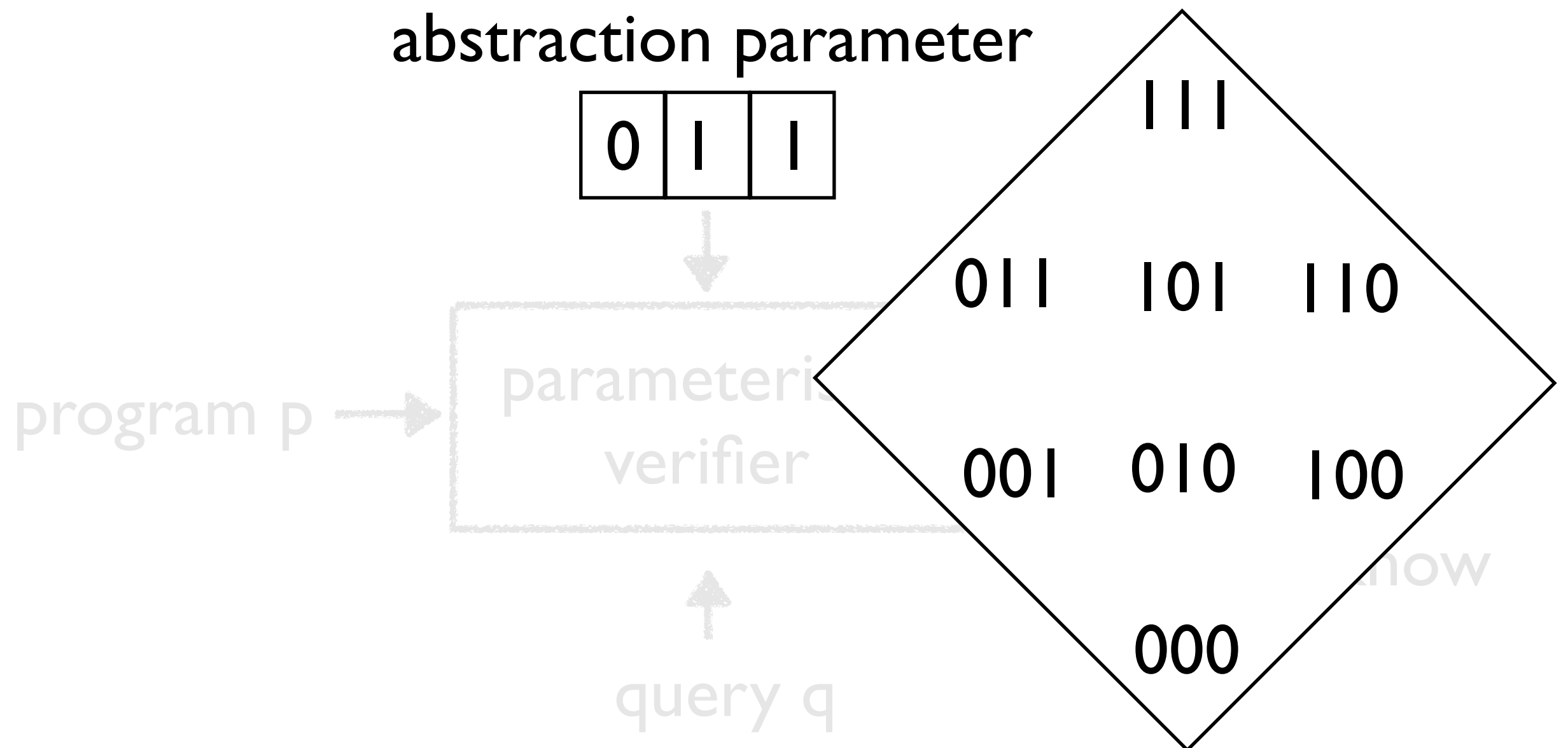
Verifier with explicit abstraction parameters



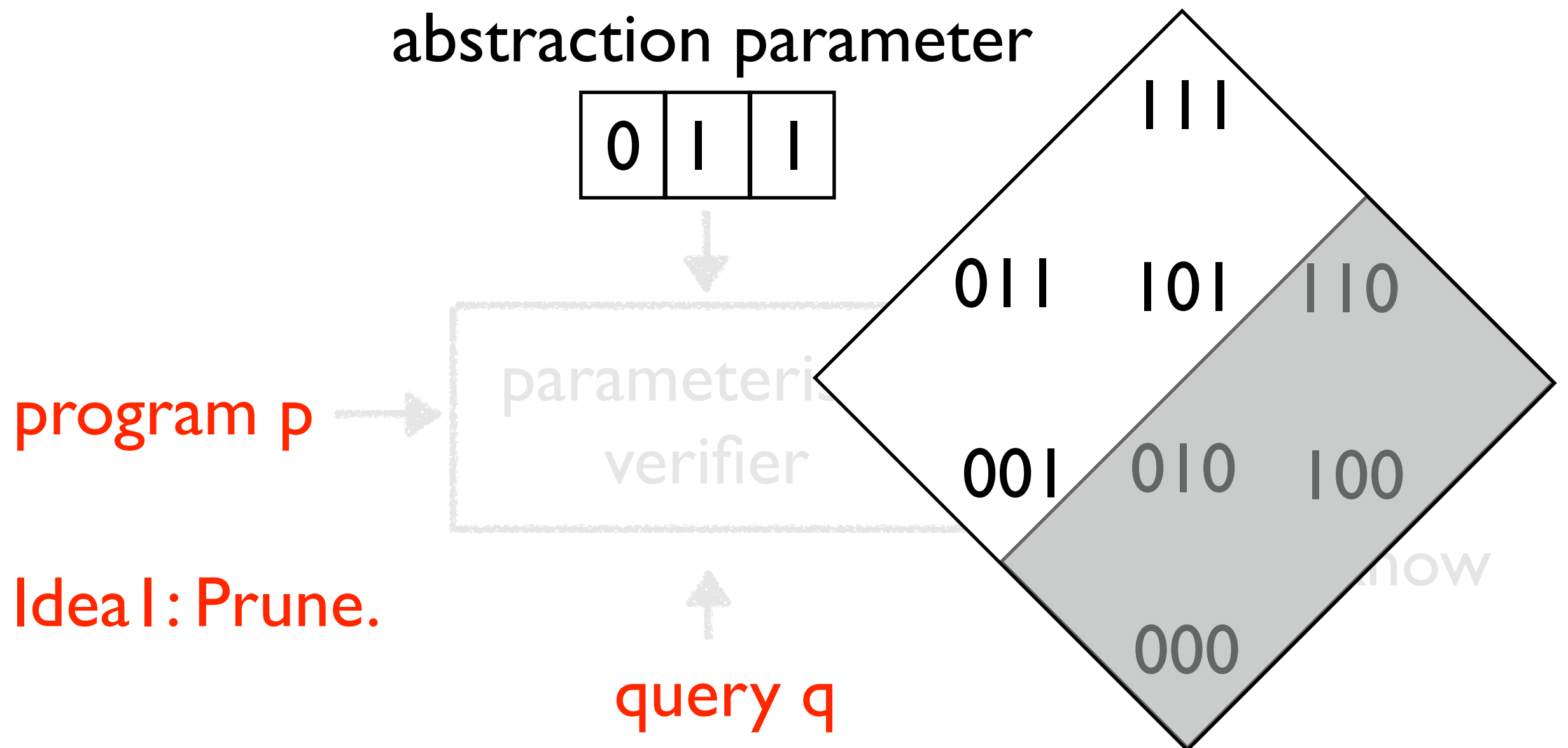
Verifier with explicit abstraction parameters



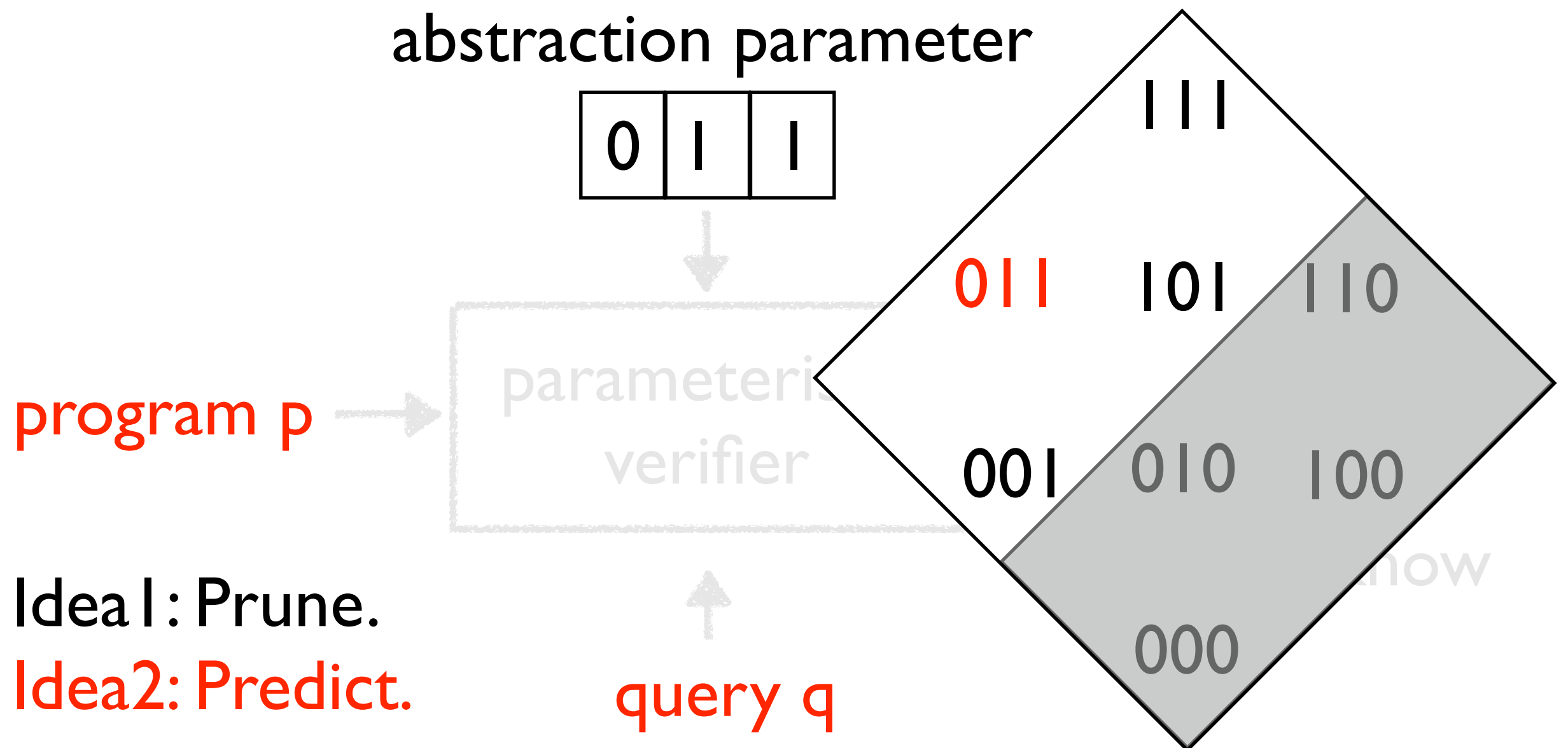
Verifier with explicit abstraction parameters



Verifier with explicit abstraction parameters



Verifier with explicit abstraction parameters



Pruning based on testing
results [POPL'12]

Two sign abstractions

$$S_0 = \left\{ \begin{array}{c} \top \\ | \\ + \end{array} \right\}$$

$$S_1 = \left\{ \begin{array}{ccccc} & & \top & & \\ & \swarrow & | & \searrow & \\ -0 & & -+ & & 0+ \\ & \swarrow & | & \searrow & \\ | & \times & & \times & | \\ - & & 0 & & + \\ & \swarrow & | & \searrow & \\ & & \perp & & \end{array} \right\}$$

Abstraction parameters

$$S_0 = \left\{ \begin{array}{c} \top \\ | \\ + \end{array} \right\}$$

$$S_1 = \left\{ \begin{array}{ccccc} & & \top & & \\ & \swarrow & | & \searrow & \\ -0 & & -+ & & 0+ \\ | & \swarrow & & \searrow & | \\ - & & 0 & & + \\ & \swarrow & | & \searrow & \\ & & \perp & & \end{array} \right\}$$

$$\text{Abs} = \{ n, x, y \} \rightarrow \{0, 1\}$$

$$\text{abs}_0 = \langle\langle n:0, x:0, y:0 \rangle\rangle$$

$$\text{abs}_1 = \langle\langle n:1, x:1, y:1 \rangle\rangle$$

$$\text{abs}_2 = \langle\langle n:0, x:0, y:1 \rangle\rangle$$

Abstraction parameters

$$\text{Abs} = \{ n, x, y \} \rightarrow \{0, 1\}$$

```
1:  assert(n >= 1);
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  y = y-1;
9:  assert(y >= 0);
```

Abstraction parameters

$$\text{Abs} = \{ n, x, y \} \rightarrow \{0, 1\}$$

《n:1, x:1, y:0》

```
1:  assert(n >= 1);
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  y = y-1;
9:  assert(y >= 0);
```

[n:+, x:τ, y:τ]

⋮

4 iter

[n:τ, x:+, y:+]

[n:τ, x:+, y:τ]

Abstraction parameters

$$\text{Abs} = \{ n, x, y \} \rightarrow \{0, 1\}$$

```
1:  assert(n >= 1);
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  y = y-1;
9:  assert(y >= 0);
```

《n:1, x:1, y:0》

[n:+, x:τ, y:τ]

⋮

4 iter

[n:τ, x:+, y:+]

[n:τ, x:+, y:τ]

《n:0, x:0, y:1》

[n:+, x:τ, y:τ]

⋮

2 iter

[n:τ, x:+, y:+]

[n:τ, x:+, y:0+]

Testing and pruning


- Test a program.
- If a bug is found, report an error.
- Otherwise, identify bad abstractions and prune the search space.

Testing and pruning

```
1:  assert(n >= 1);
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  y = y-1;
9:  assert(y >= 0);
```

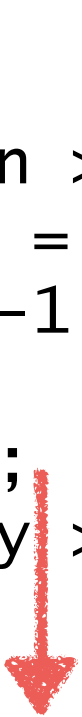

Testing and pruning

```
1:  assert(n >= 1);    [n:1,x:0,y:0]
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  y = y-1;           [n:1,x:1,y:0]
9:  assert(y >= 0);
```



Testing and pruning

```
1:  assert(n >= 1);    [n:1,x:0,y:0]
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  y = y-1;           [n:1,x:1,y:0]
9:  assert(y >= 0);
```



**[n:_, x:_, y:⊤] if abs(y)=0.
Because $S_0 = \{+, \top\}$.**

Testing and pruning

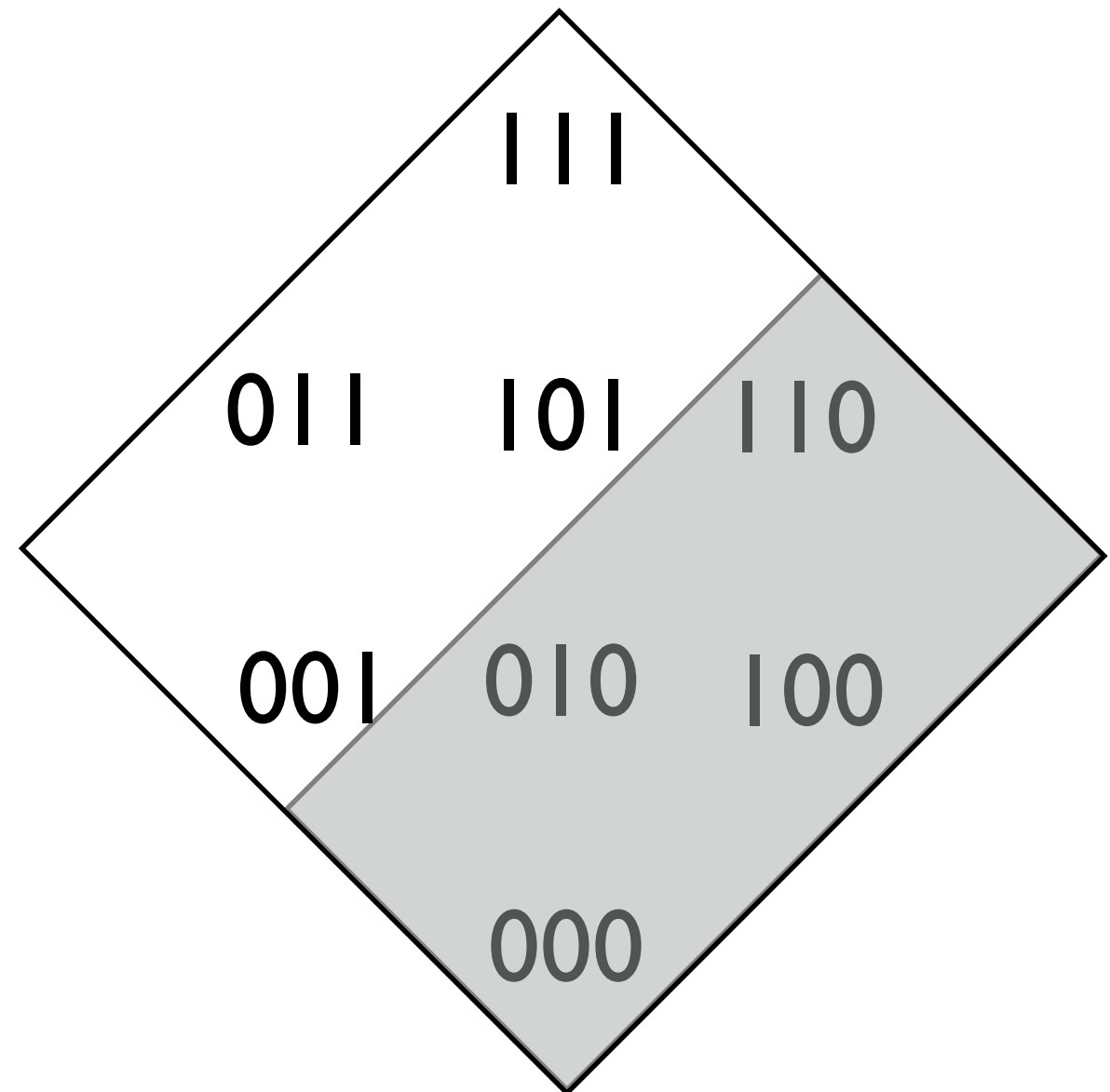
```
1:  assert(n >= 1);    [n:1,x:0,y:0]
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  y = y-1;
9:  assert(y >= 0);
```

↓

[n:1,x:1,y:0]

↓

$[n:_, x:_, y:\top]$ if $\text{abs}(y)=0$.
Because $S_0 = \{+, \top\}$.



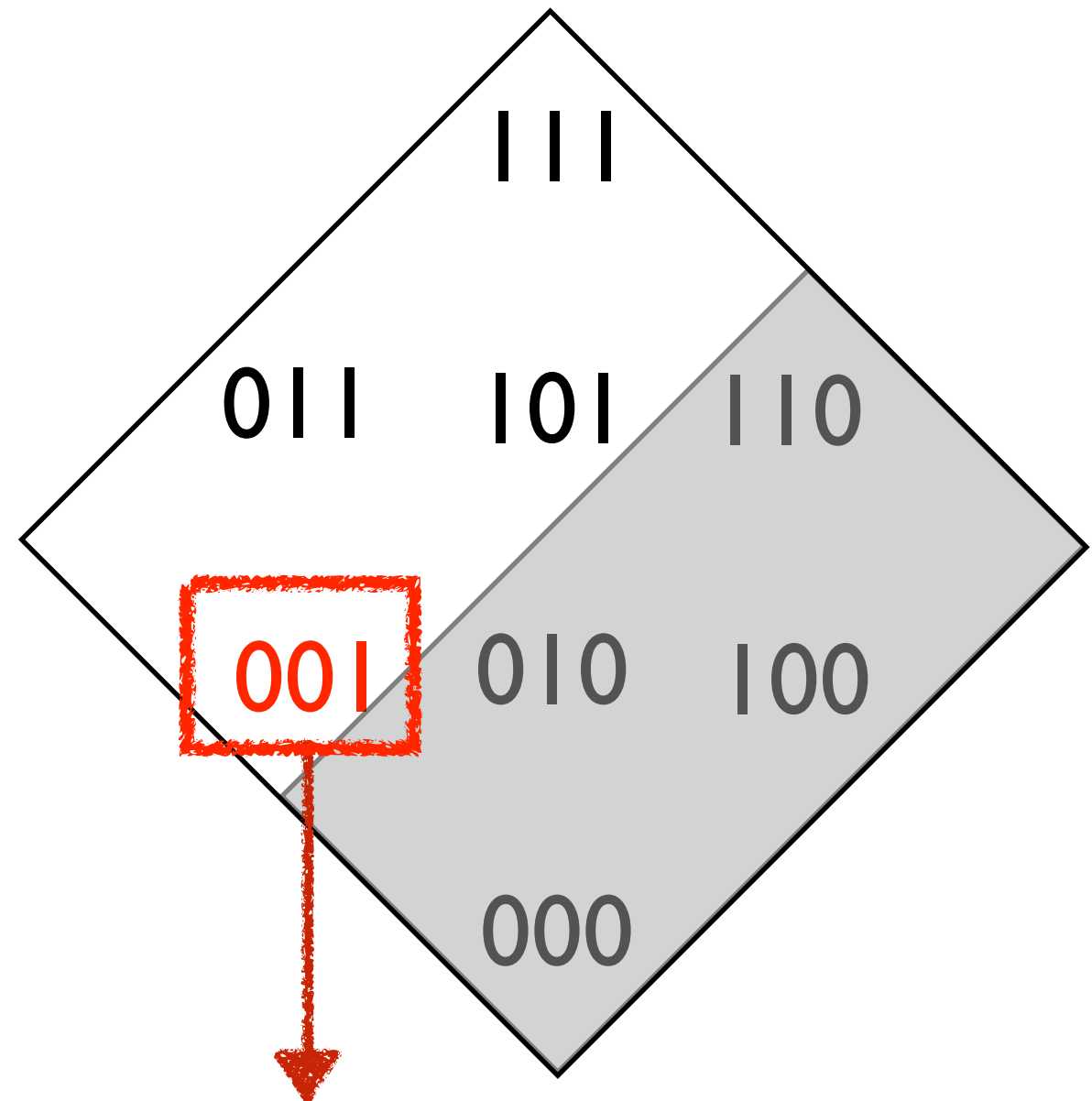
Testing and pruning

```
1:  assert(n >= 1);    [n:1,x:0,y:0]
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  y = y-1;
9:  assert(y >= 0);
```

↓

[n:1,x:1,y:0]

[n:_, x:_, y:⊤] if $\text{abs}(y)=0$.
Because $S_0 = \{+, \top\}$.



Choose a minimal abs.

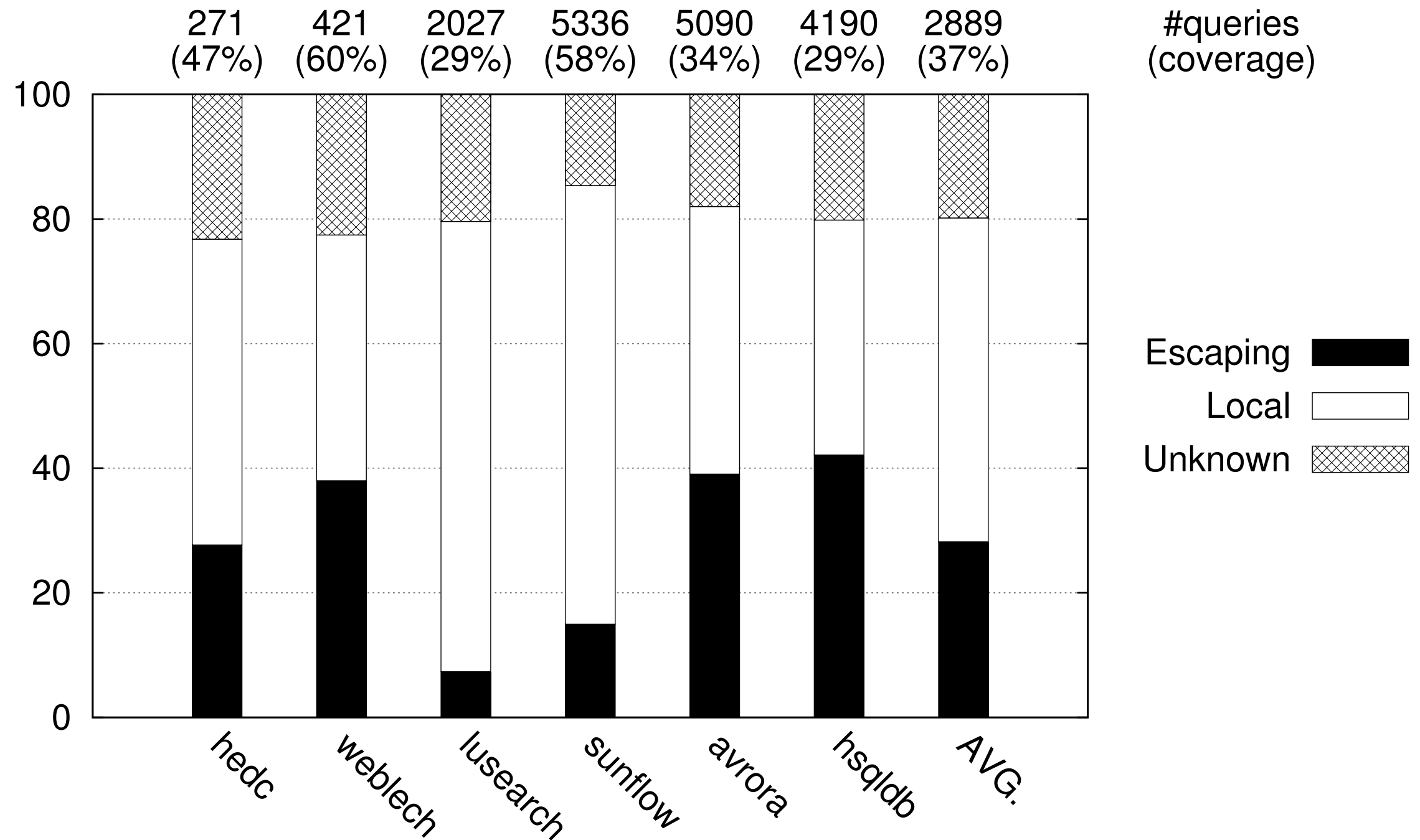


Figure 3. Precision results for our thread-escape analysis.

[POPL'12]

**Intermediate assertions.
Abstraction.**