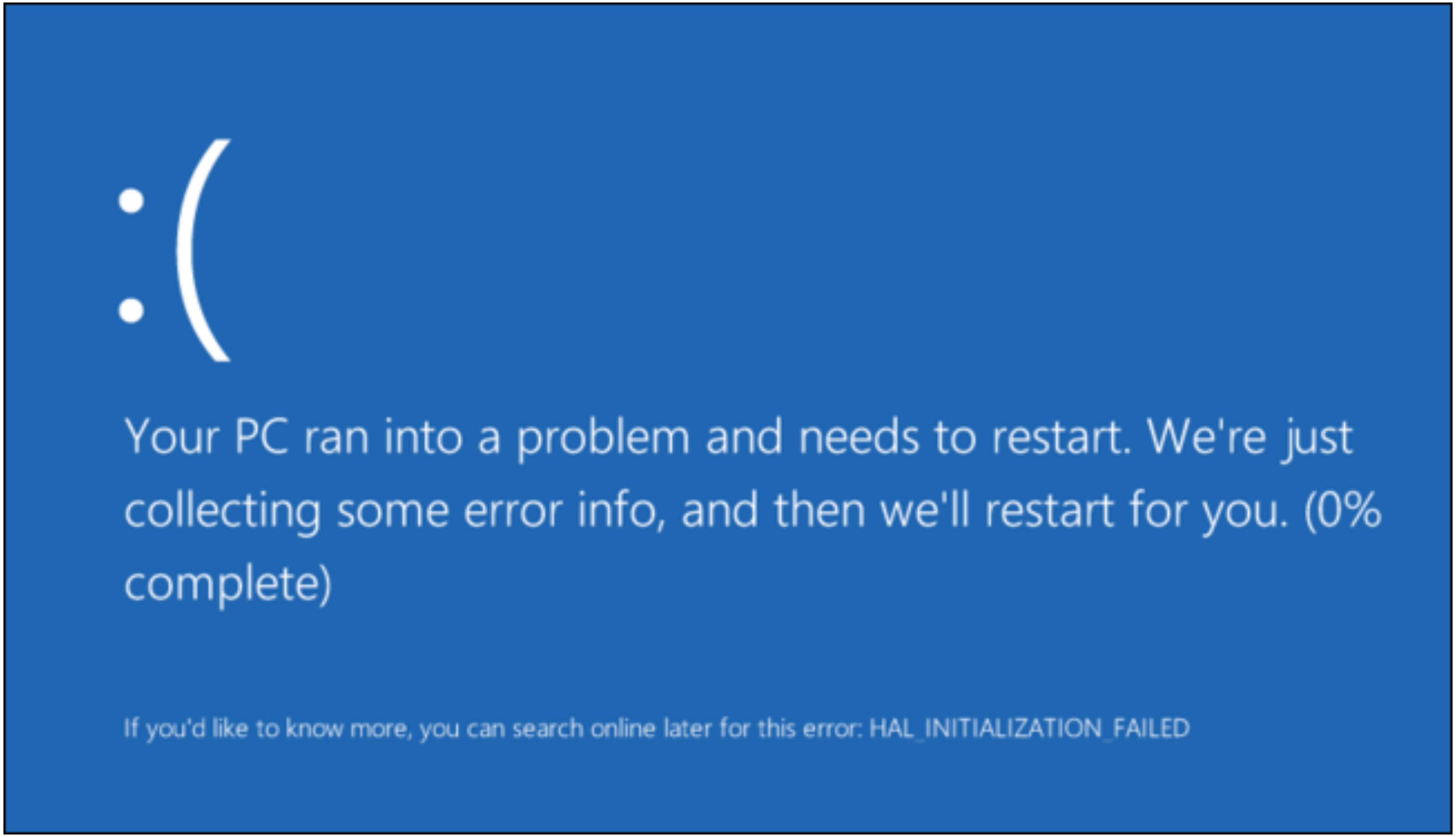
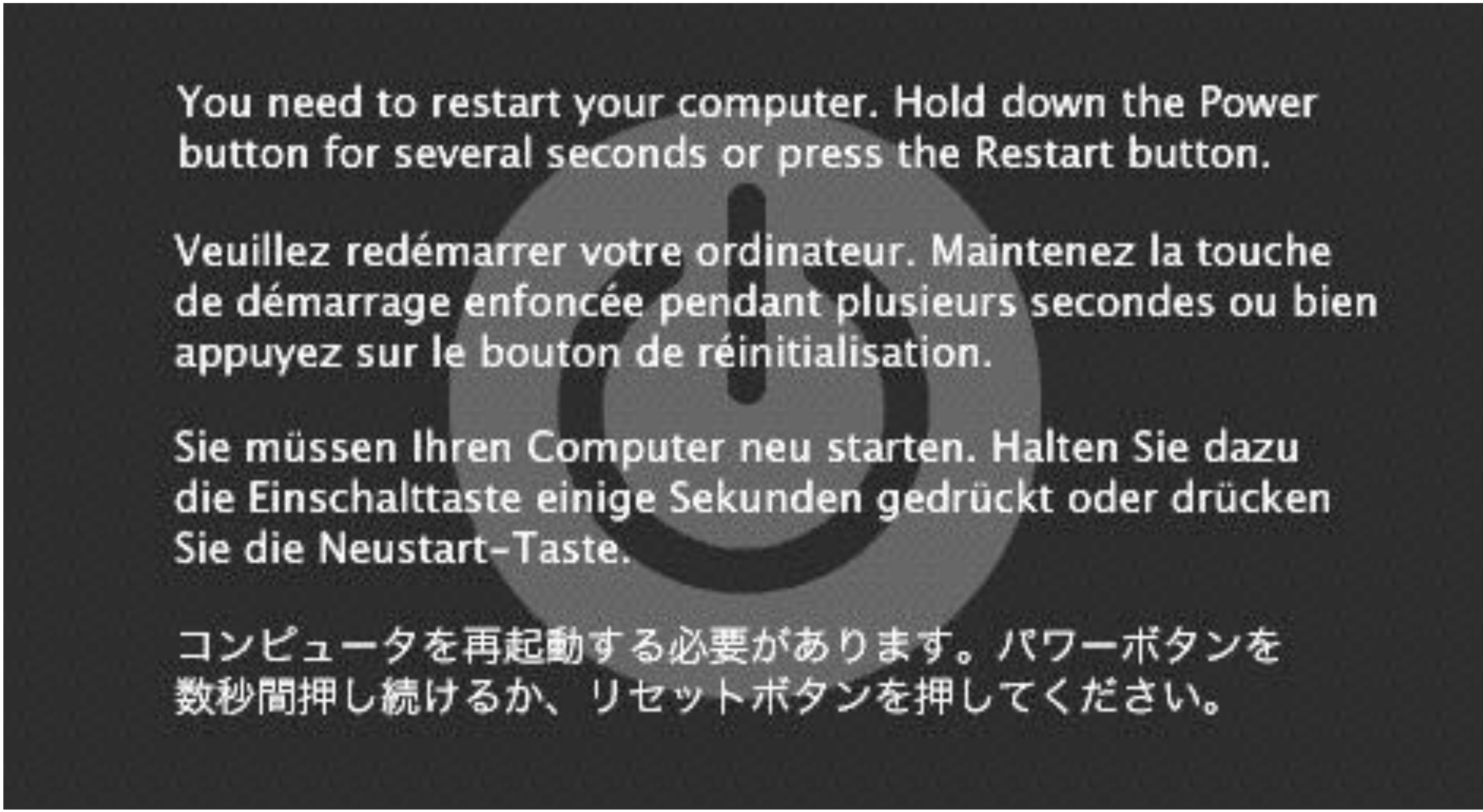
CS320: Programming Languages

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.

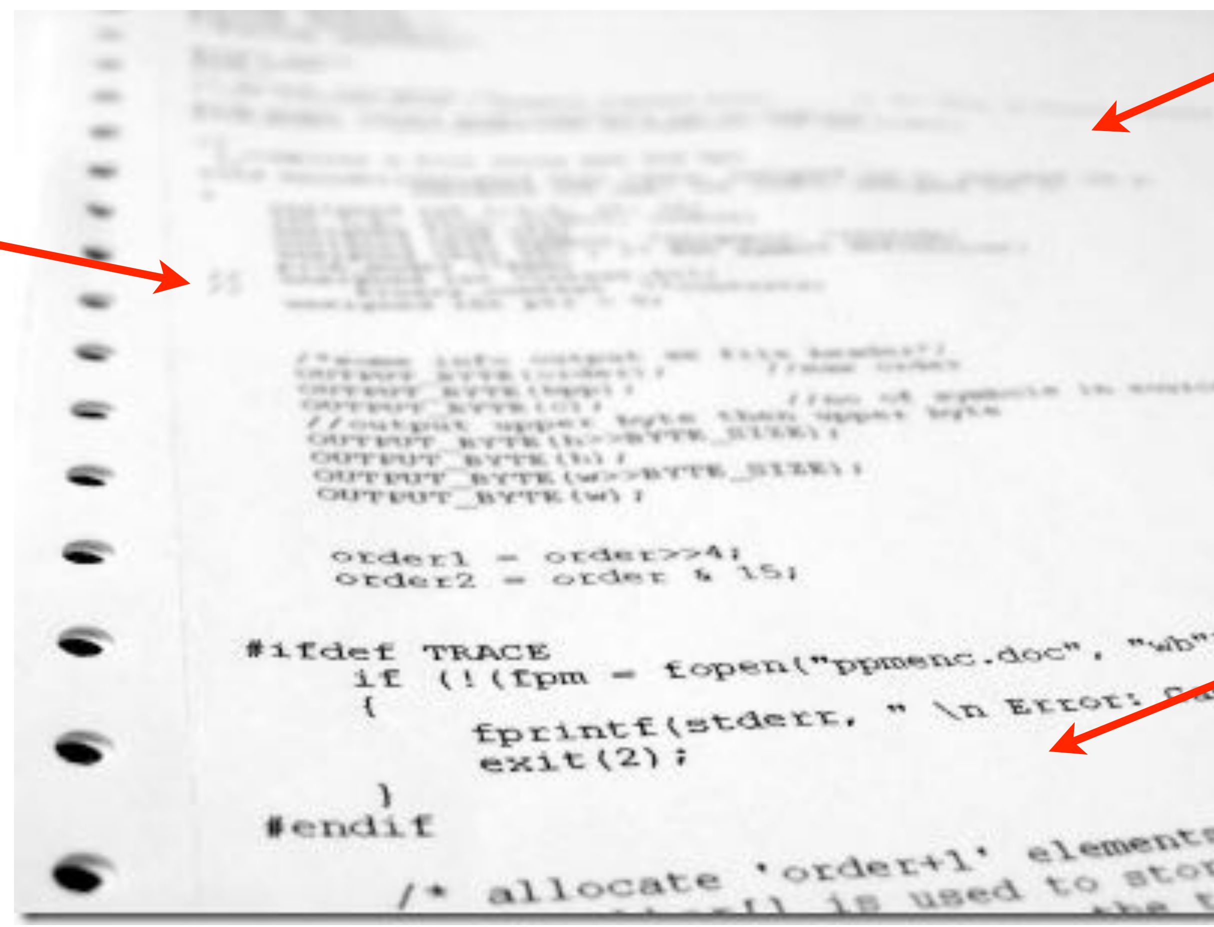
Verify that this program computes “100 * 30”.

Turing’s idea

Use intermediate assertions.

x = 60
and i = 2

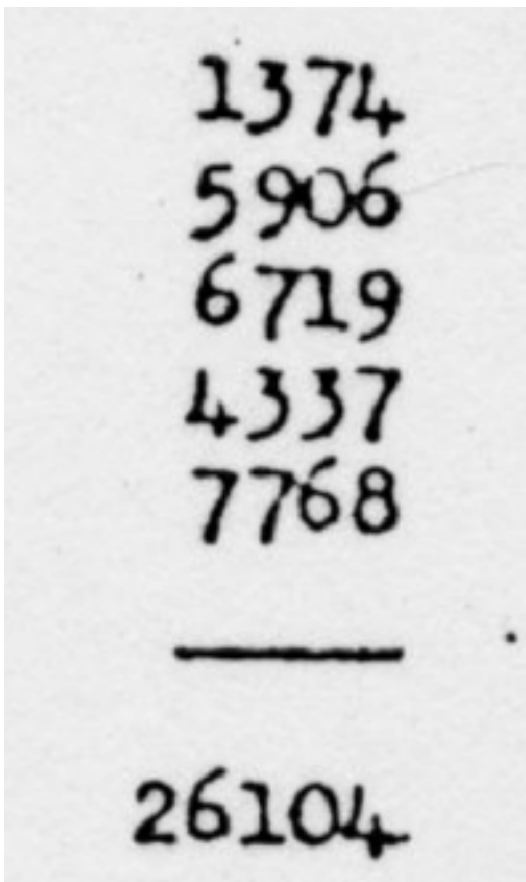
x = 30
and i = 1



x = 2970
and i = 29

Verify that this program computes “100 * 30”.

Turing's example



1374
5906
6719
4337
7768

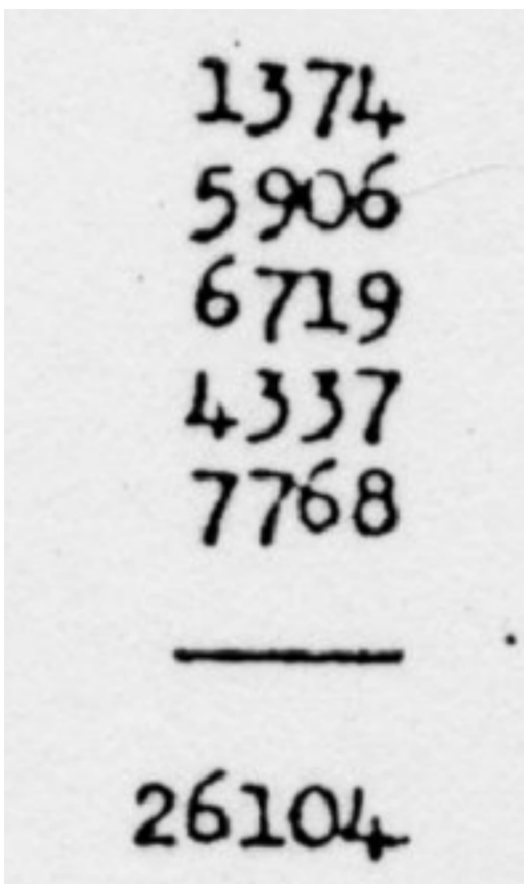
26104

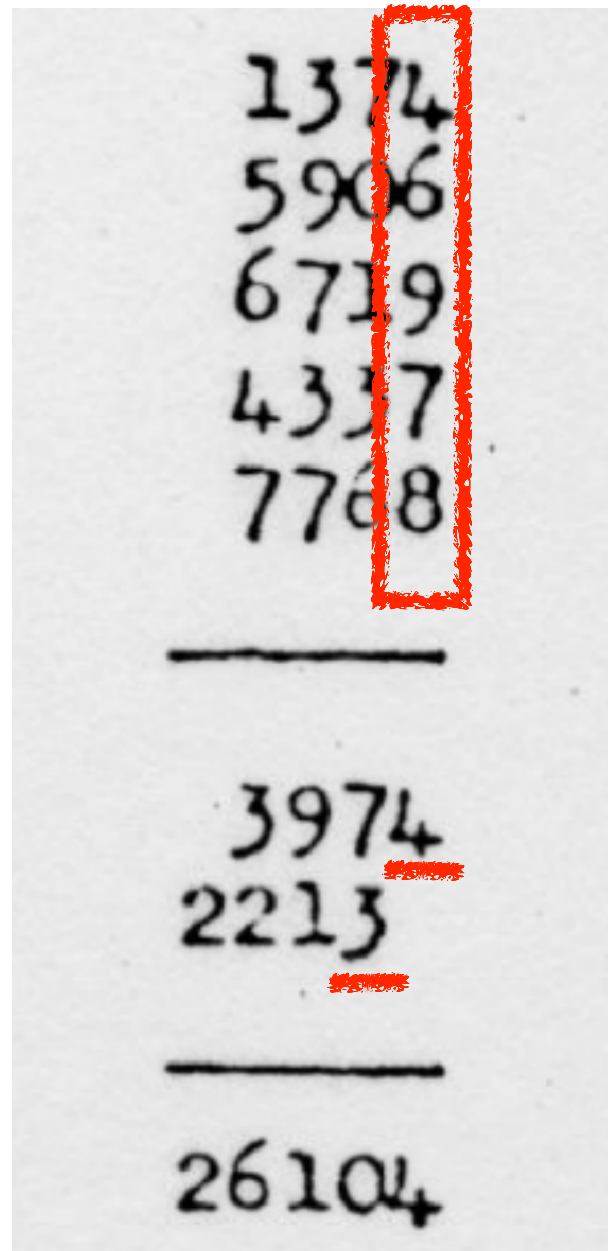
Turing’s example

```
 1374
 5906
 6719
 4337
 7768
 ----
26104
```

```
 1374
 5906
 6719
 4337
 7768
 ----
 3974
2213
 ----
26104
```

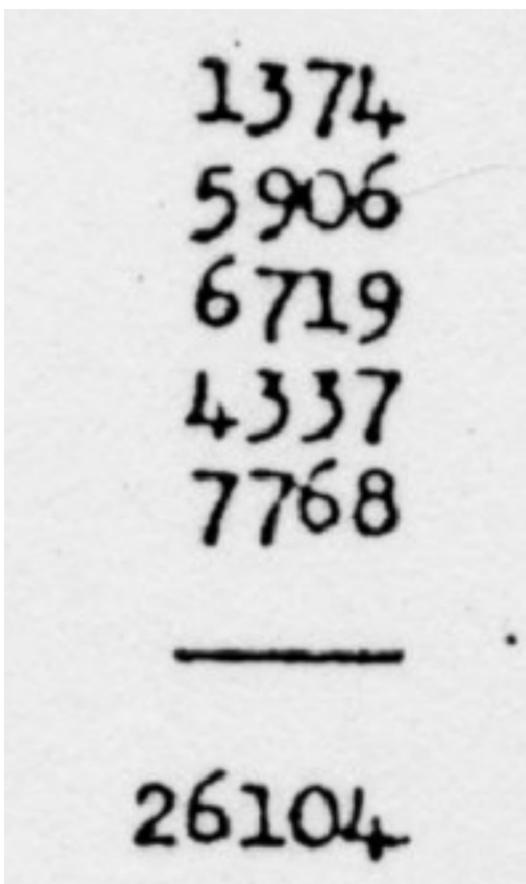
Turing's example

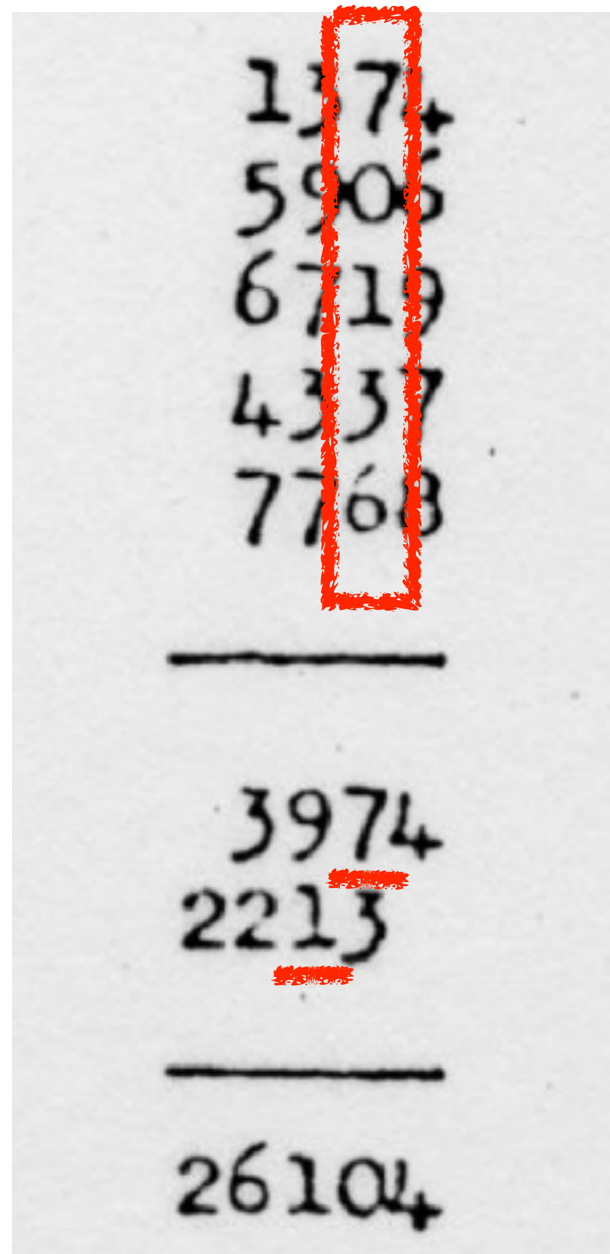




4+6+9+7+8 = 34

Turing's example

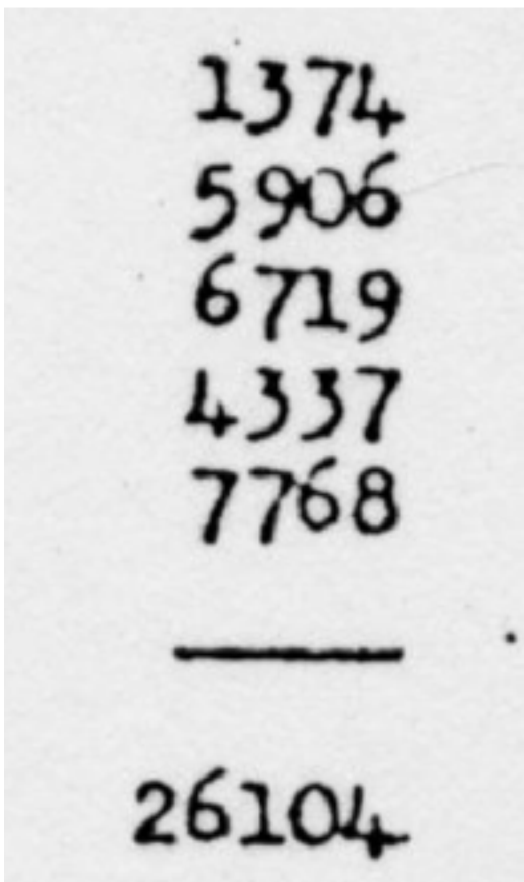


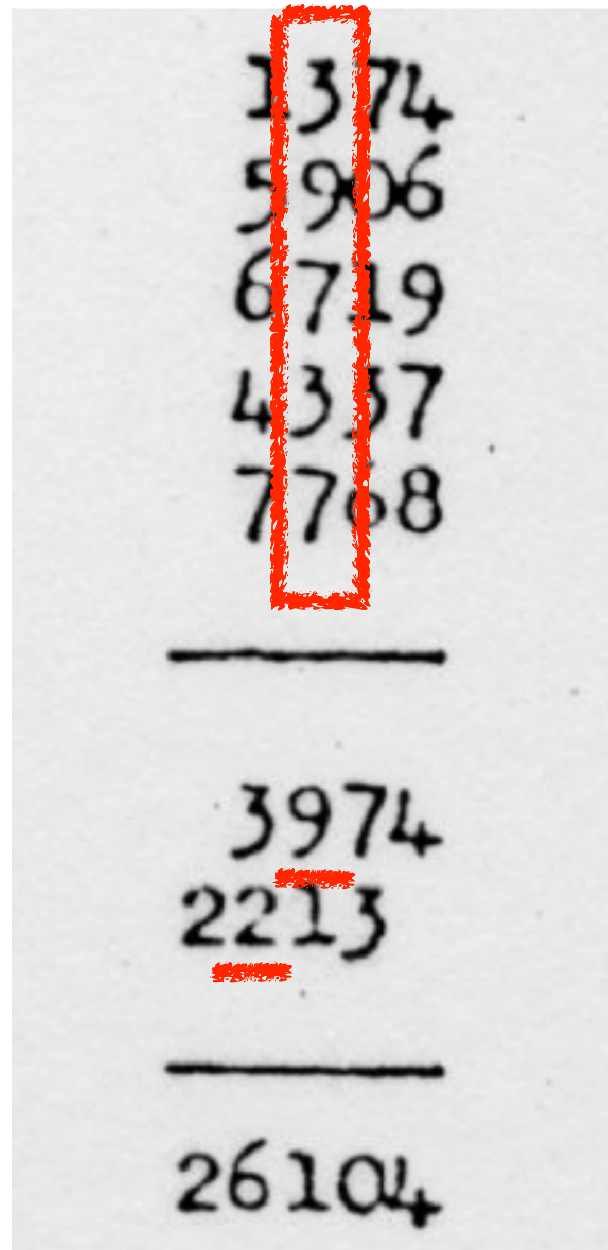


4+6+9+7+8 = 34

7+0+1+3+6 = 17

Turing's example



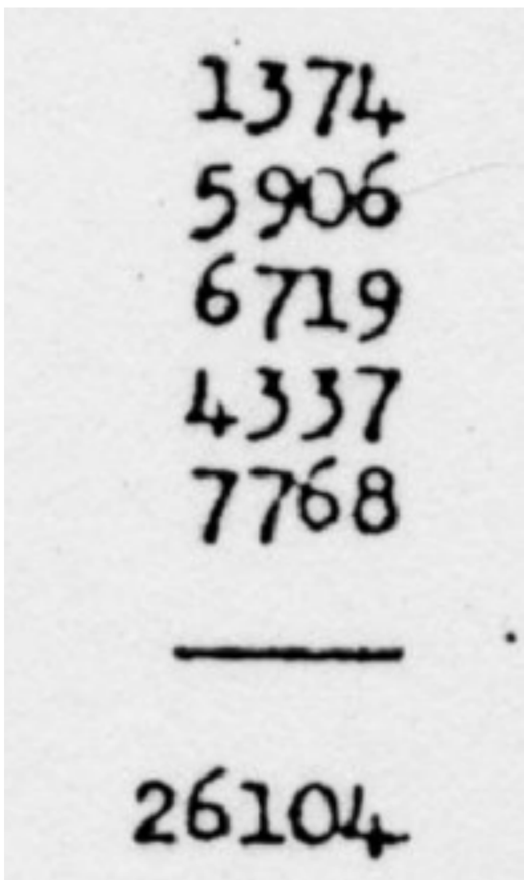


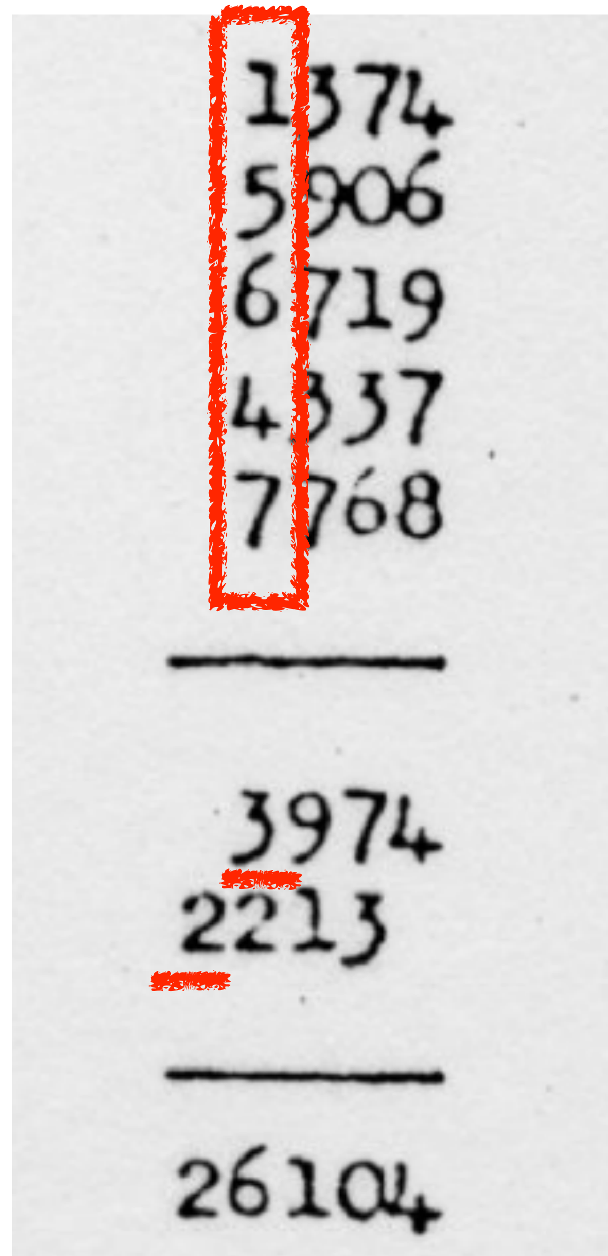
4+6+9+7+8 = 34

7+0+1+3+6 = 17

3+9+7+3+7 = 29

Turing's example





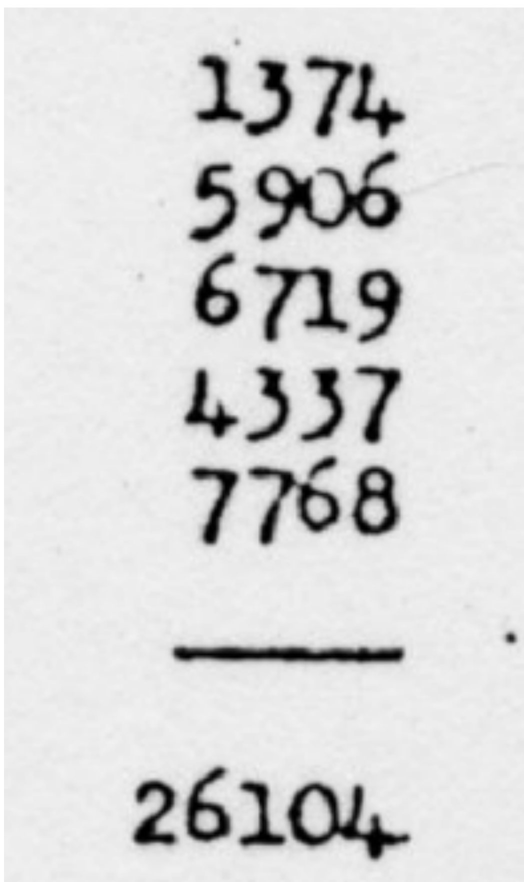
4+6+9+7+8 = 34

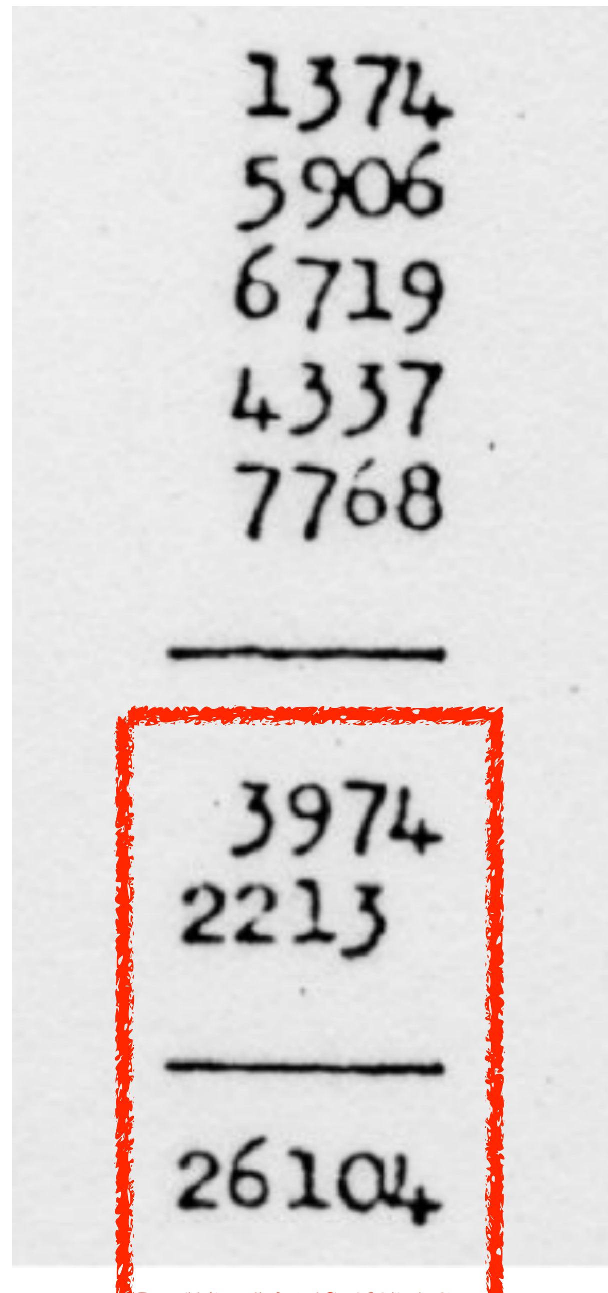
7+0+1+3+6 = 17

3+9+7+3+7 = 29

1+5+6+4+7 = 23

Turing’s example





4+6+9+7+8 = 34

7+0+1+3+6 = 17

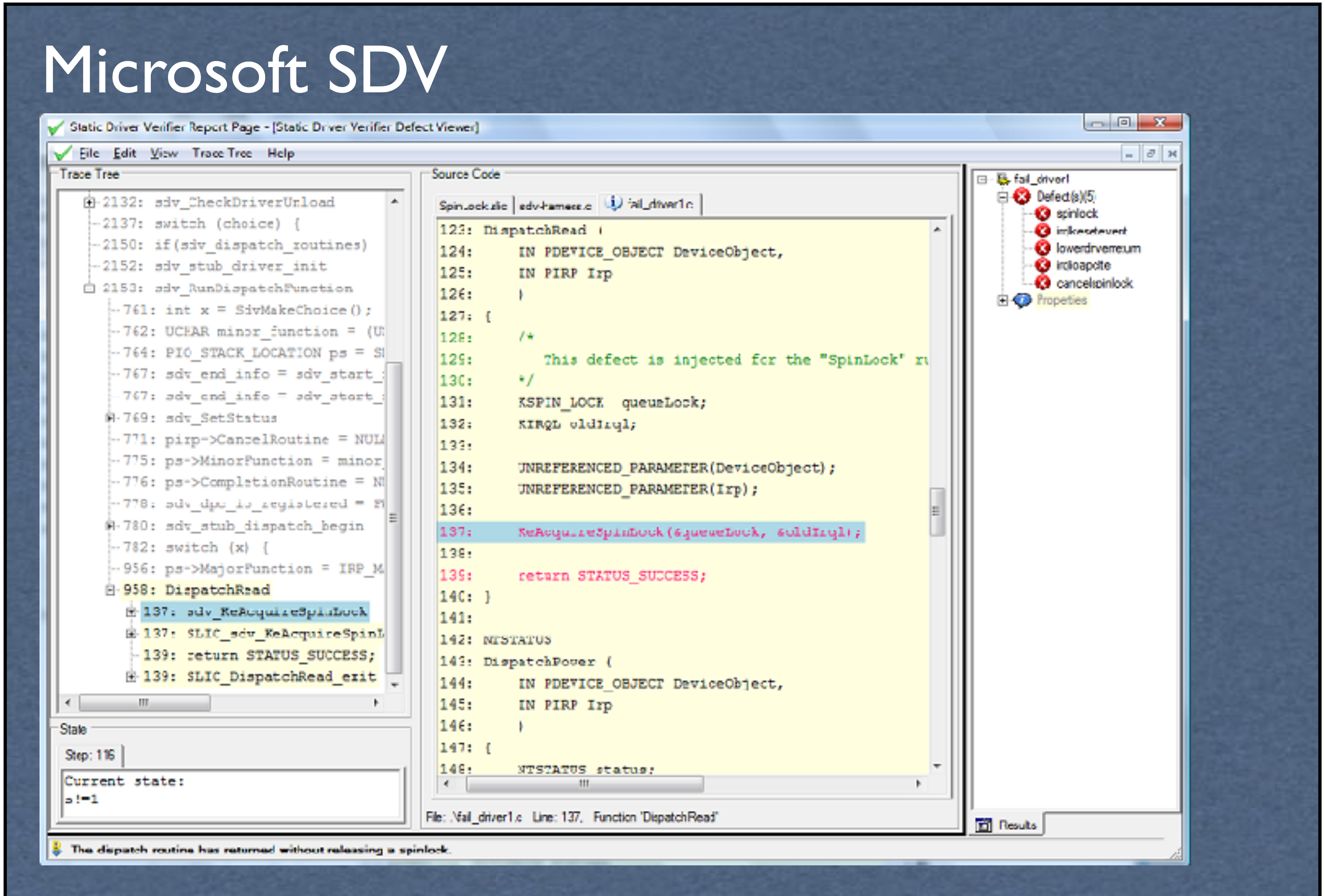
3+9+7+3+7 = 29

1+5+6+4+7 = 23

Intermediate assertions

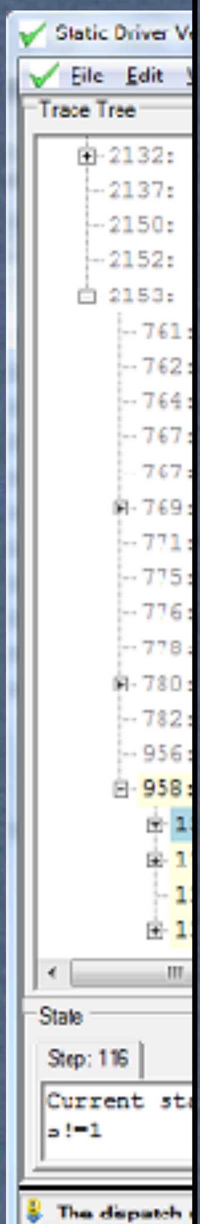
- Form the basis of modern verification methods.
- Inferred automatically by commercial tools nowadays.

Commercial tools in 2017

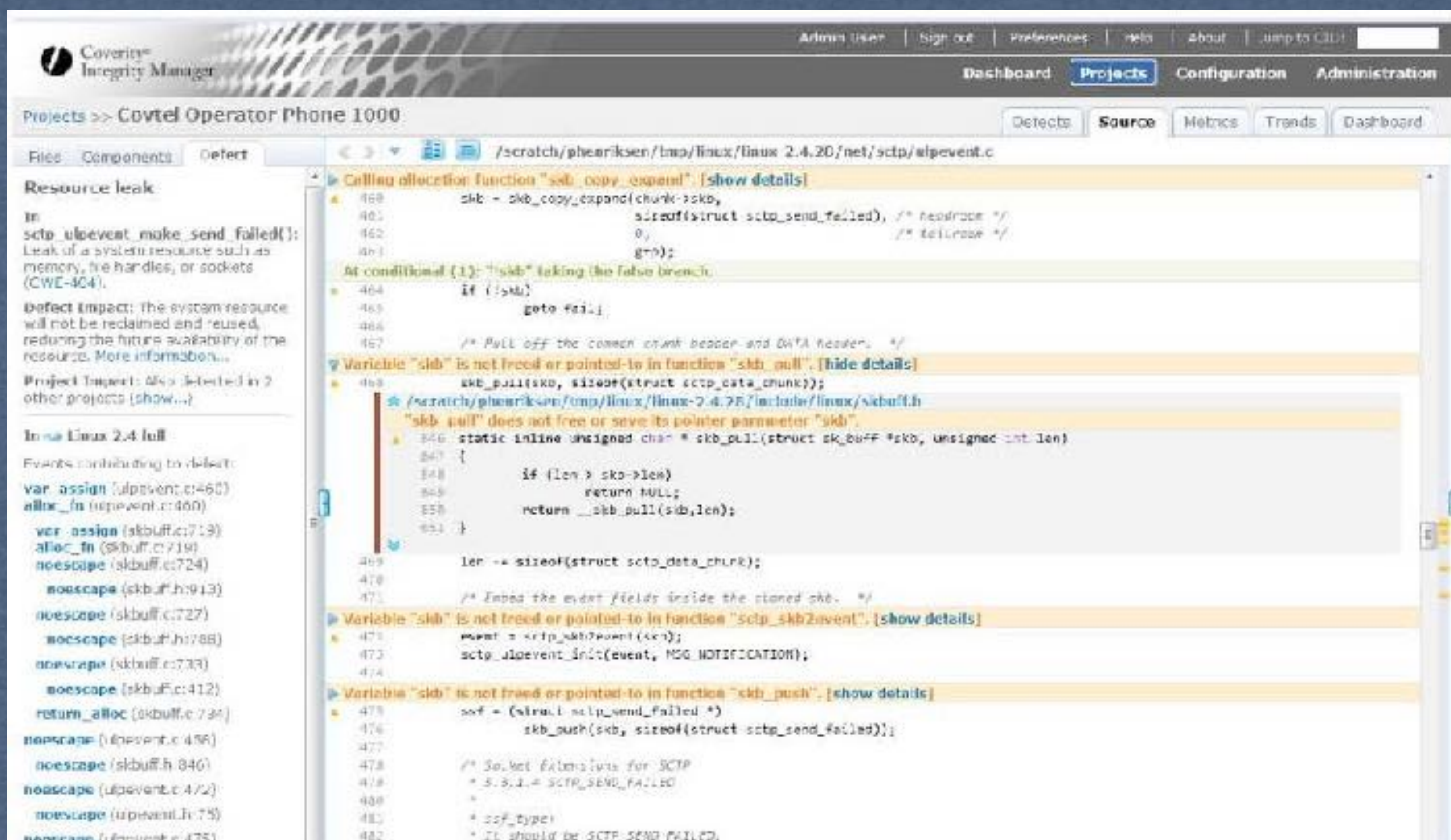


Commercial tools in 2017

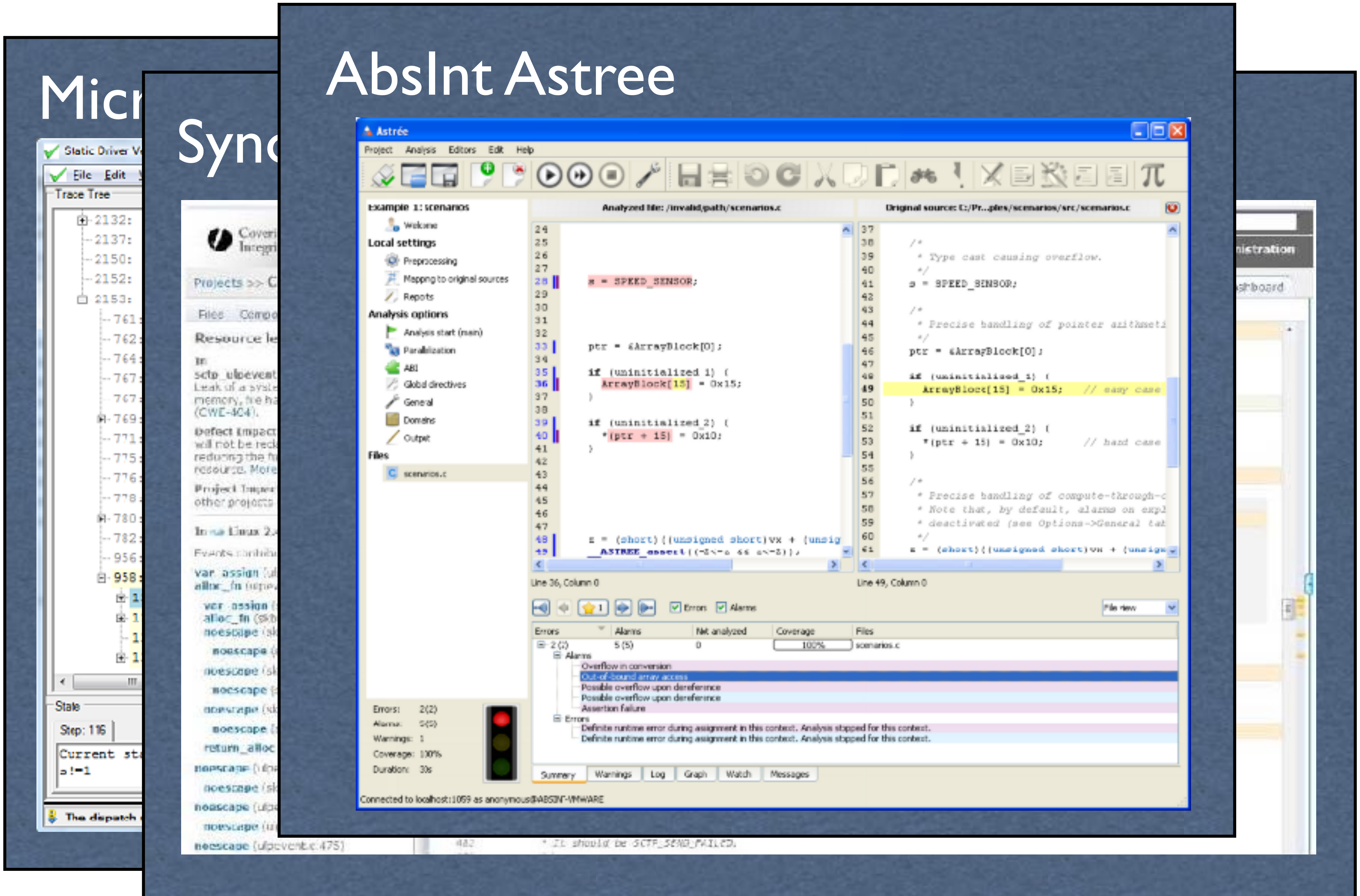
Micr



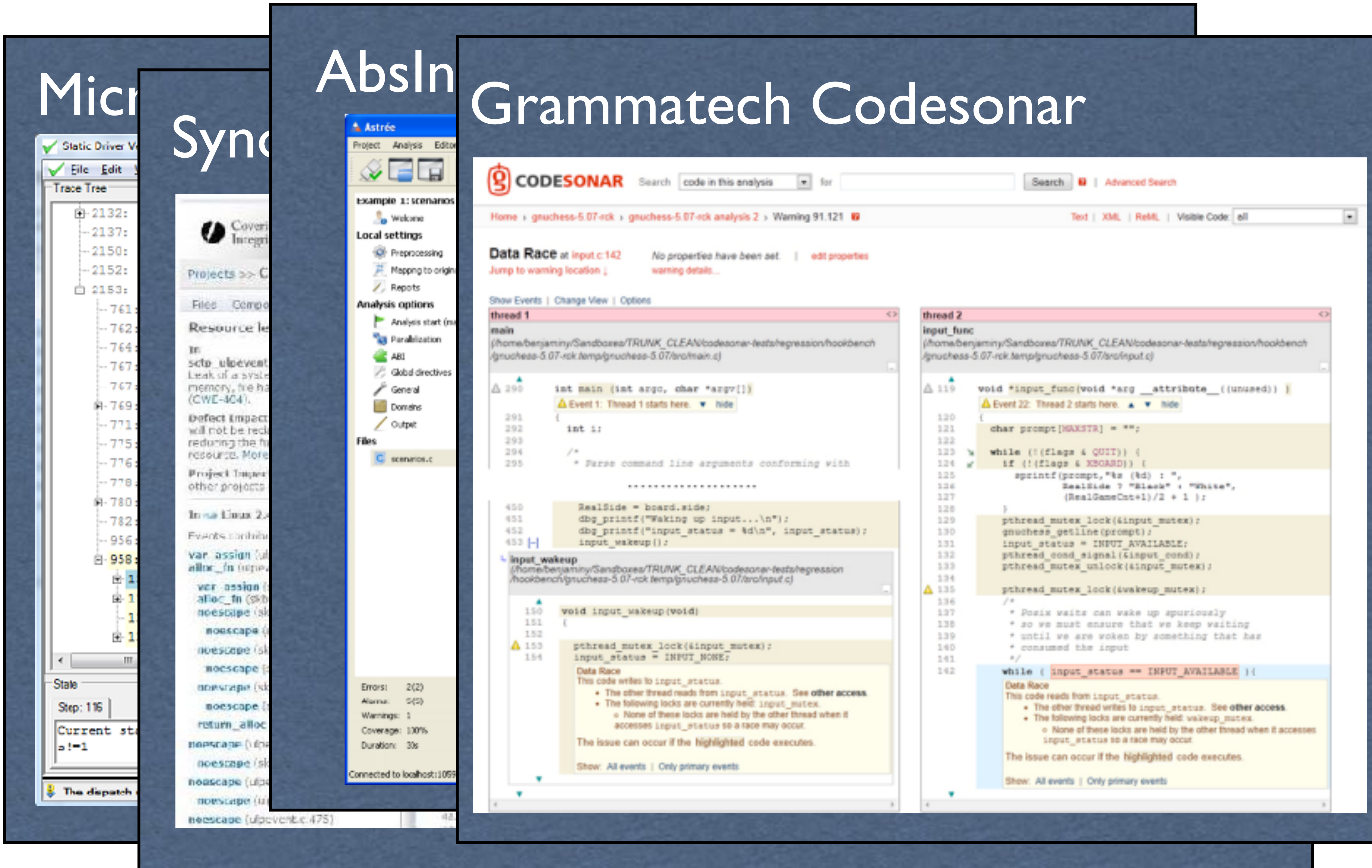
Synopsis Coverity



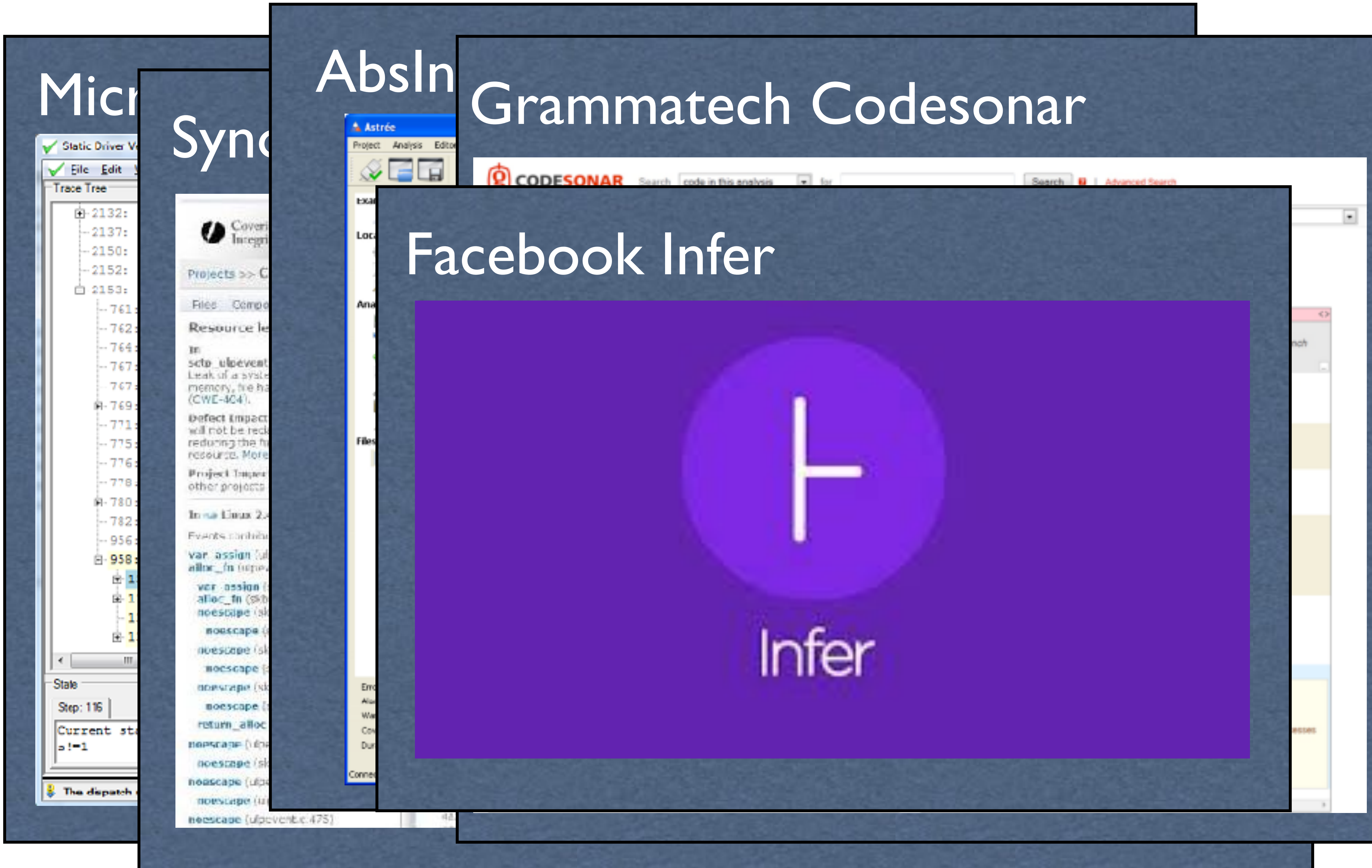
Commercial tools in 2017



Commercial tools in 2017



Commercial tools in 2017



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

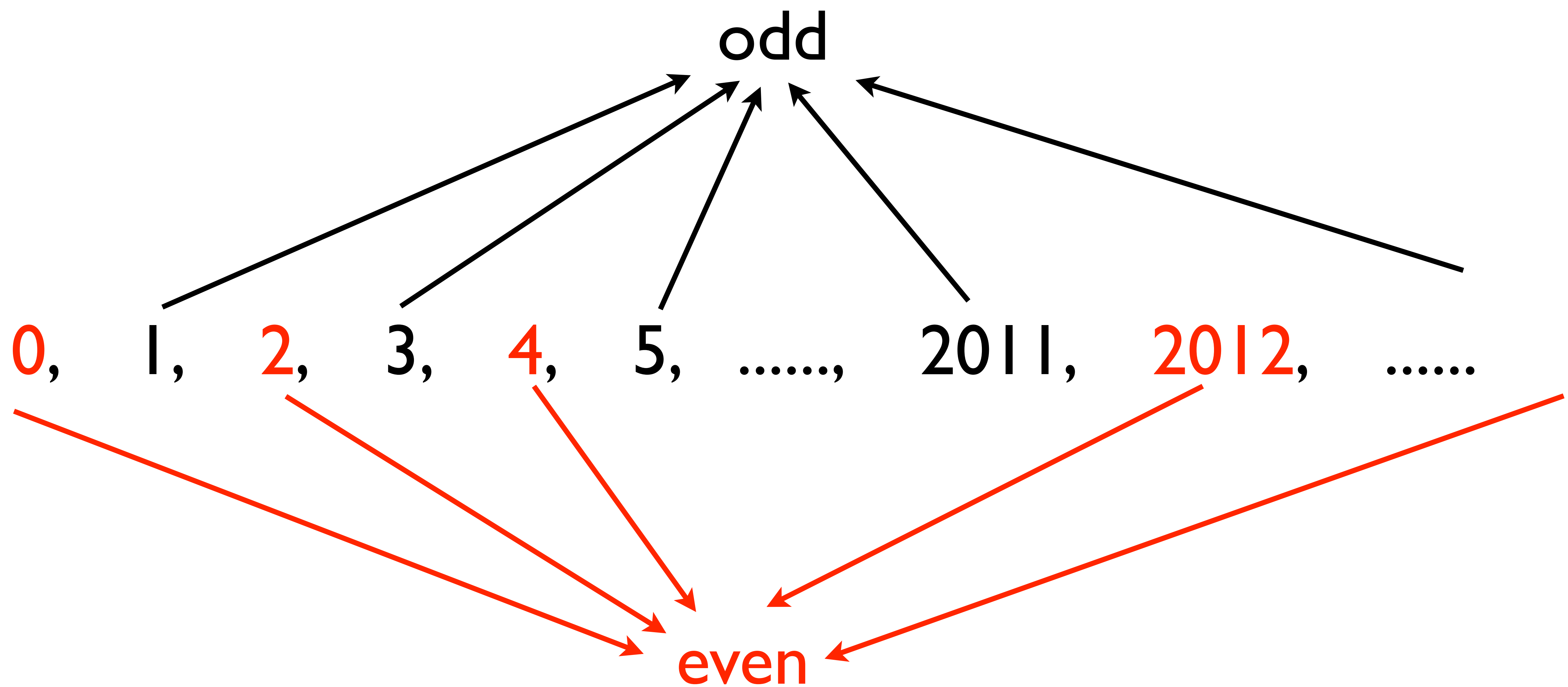
0, 1, 2, 3, 4, 5, ..., 2011, 2012, ...

Even/odd abstraction

odd

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

Even/odd abstraction



Even/odd abstraction

- even + even =
- even + odd =
- odd + even =
- odd + odd =

Even/odd abstraction

- even + even = even
- even + odd =
- odd + even =
- odd + odd =

Even/odd abstraction

- even + even = even
- even + odd = odd
- odd + even =
- odd + odd =

Even/odd abstraction

- even + even = even
- even + odd = odd
- odd + even = odd
- odd + odd =

Even/odd abstraction

- even + even = even
- even + odd = odd
- odd + even = odd
- odd + odd = 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

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

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
```

Fibonacci number

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

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

Fibonacci number

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

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

Fibonacci number

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

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

Fibonacci number

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

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

Fibonacci number

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

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
```

[n:3,x:0,y:0]

[n:3,x:1,y:1] [n:2,x:1,y:2]

[n:2,x:1,y:2] [n:1,x:2,y:3]

Fibonacci number

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

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

Fibonacci number

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

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

Fibonacci number

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

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

Fibonacci number

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

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
assert(y >= 0);
```

Fibonacci number

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

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
assert(y >= 0);
```

Because it computes fib. number.

Fibonacci number

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

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
   (x,y) = (y,x+y);
   n = n-1;
}
assert(y >= 0);
```

Because it computes fib. number.

Irrelevant n. No negative numbers nor minus.

Simple sign abstraction

- Abstract values:

$$\begin{array}{c} \top \\ | \\ + \end{array}$$

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

Analysing Fibonacci with simple sign abstraction

```
assert(n >= 1);   [n:+,x:⊤,y:⊤]
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
assert(y >= 0);
```

Analysing Fibonacci with simple sign abstraction

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
assert(y >= 0);
```

[n:+,x:⊤,y:⊤]

↓

[n:+,x:+,y:+]

Analysing Fibonacci with simple sign abstraction

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
assert(y >= 0);
```

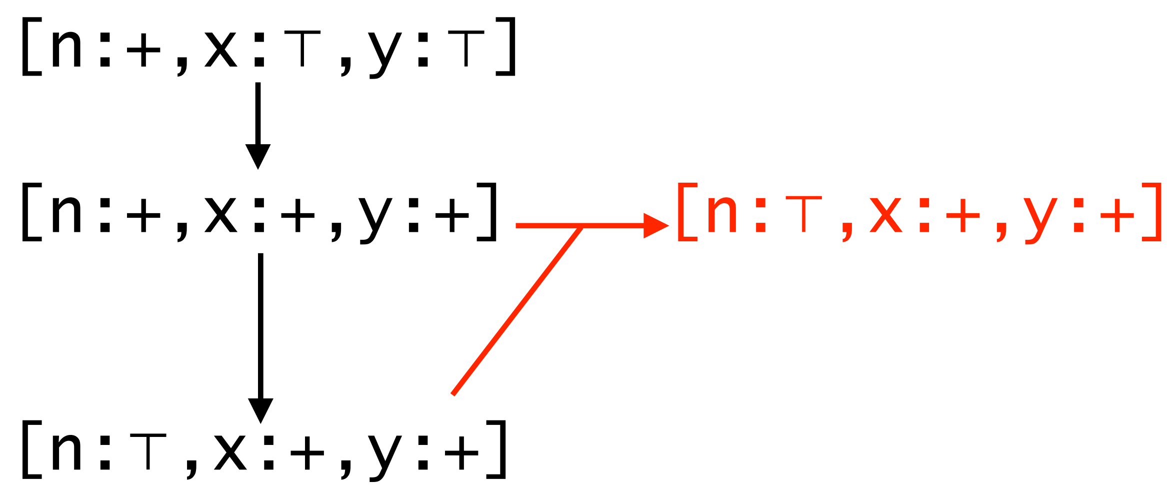
[n:+,x:⊤,y:⊤]

[n:+,x:+,y:+]

[n:⊤,x:+,y:+]

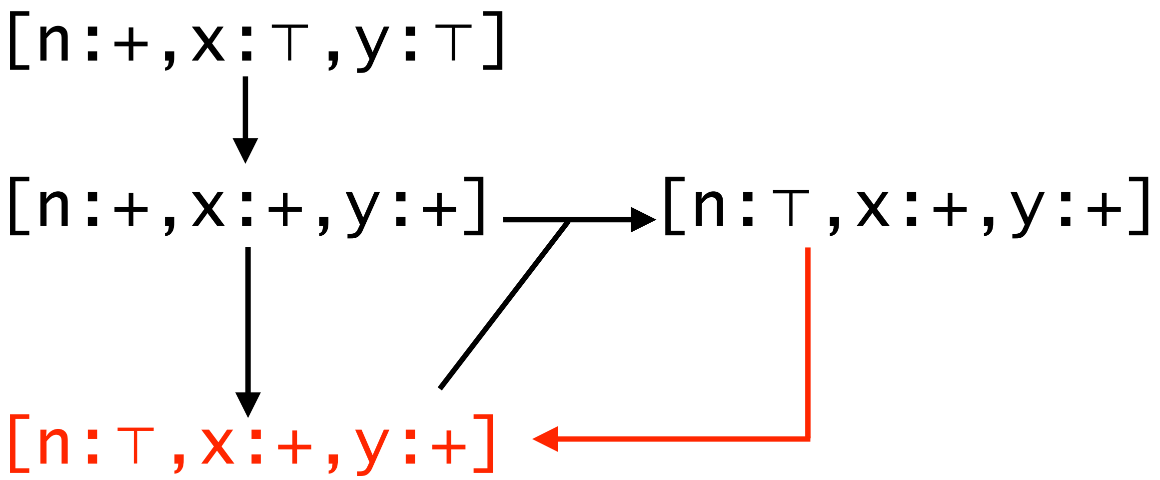
Analysing Fibonacci with simple sign abstraction

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
assert(y >= 0);
```



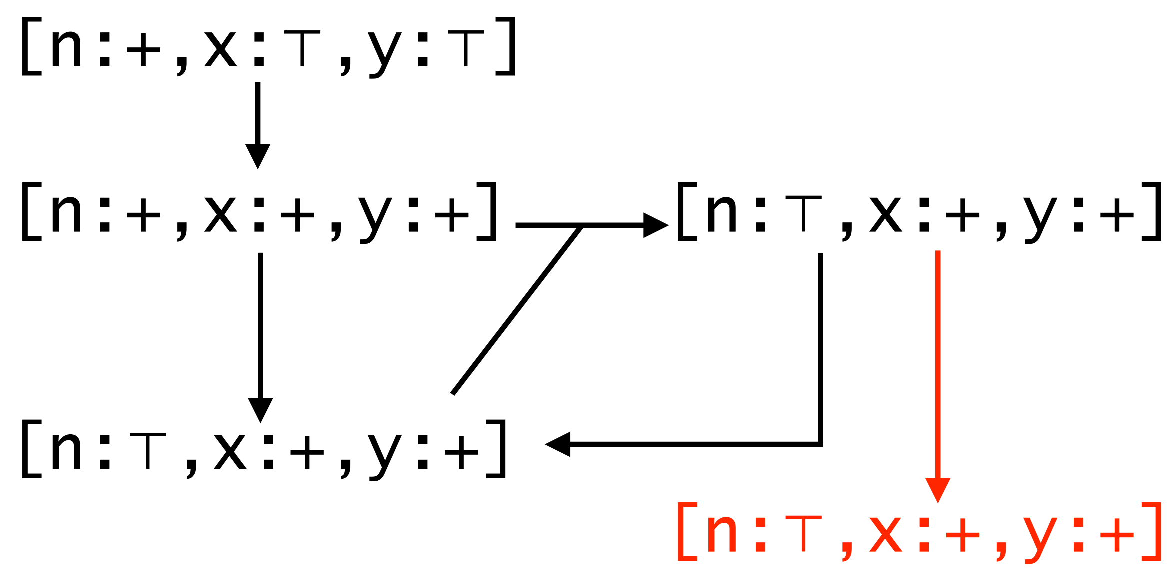
Analysing Fibonacci with simple sign abstraction

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
assert(y >= 0);
```



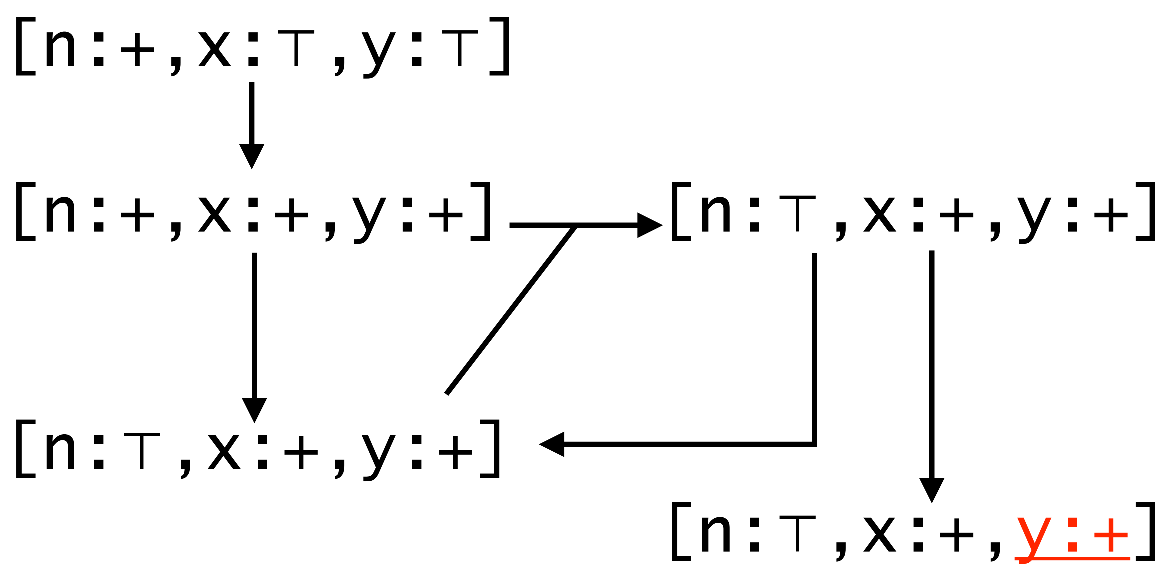
Analysing Fibonacci with simple sign abstraction

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
assert(y >= 0);
```



Analysing Fibonacci with simple sign abstraction

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
assert(y >= 0);
```



Finding a good abstraction

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
assert(y >= 0);
```

- Typically done by hand.

Finding a good abstraction

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
y = y-1;
assert(y >= 0);
```

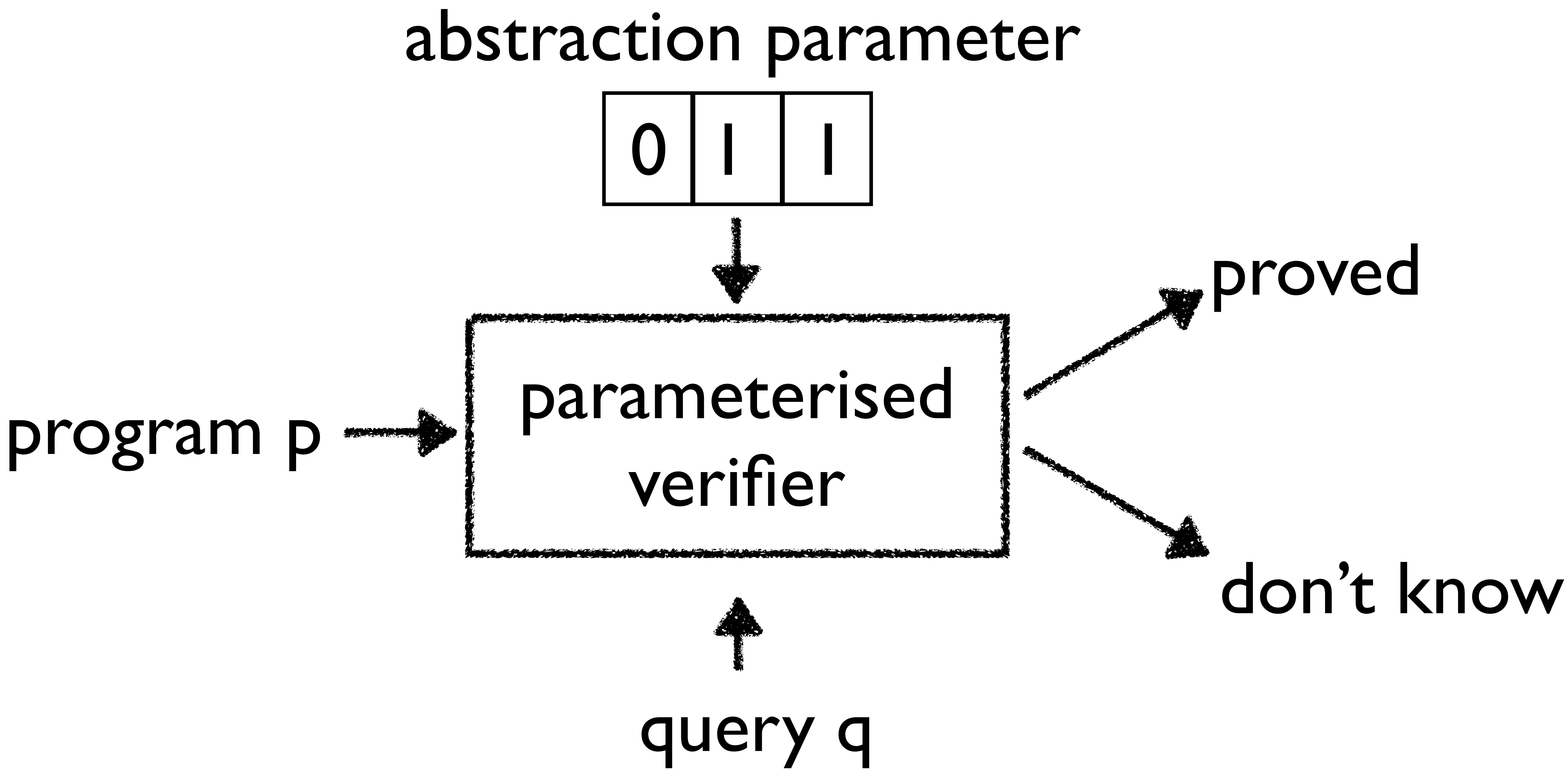
- Typically done by hand.
- Tricky.
- Active research area: how to automate this?

How to find a good program abstraction automatically?

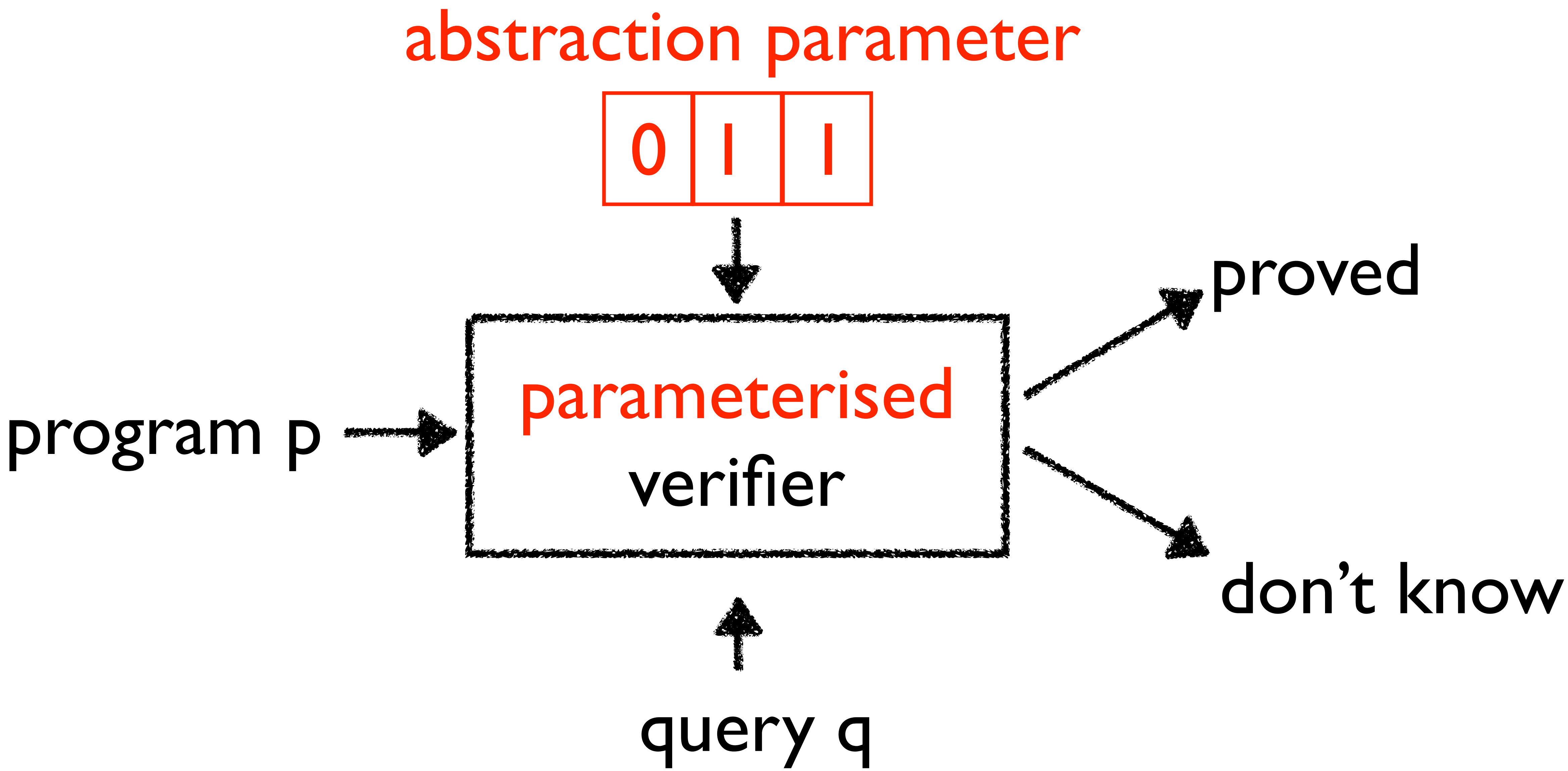
Abstraction finding as search problem

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

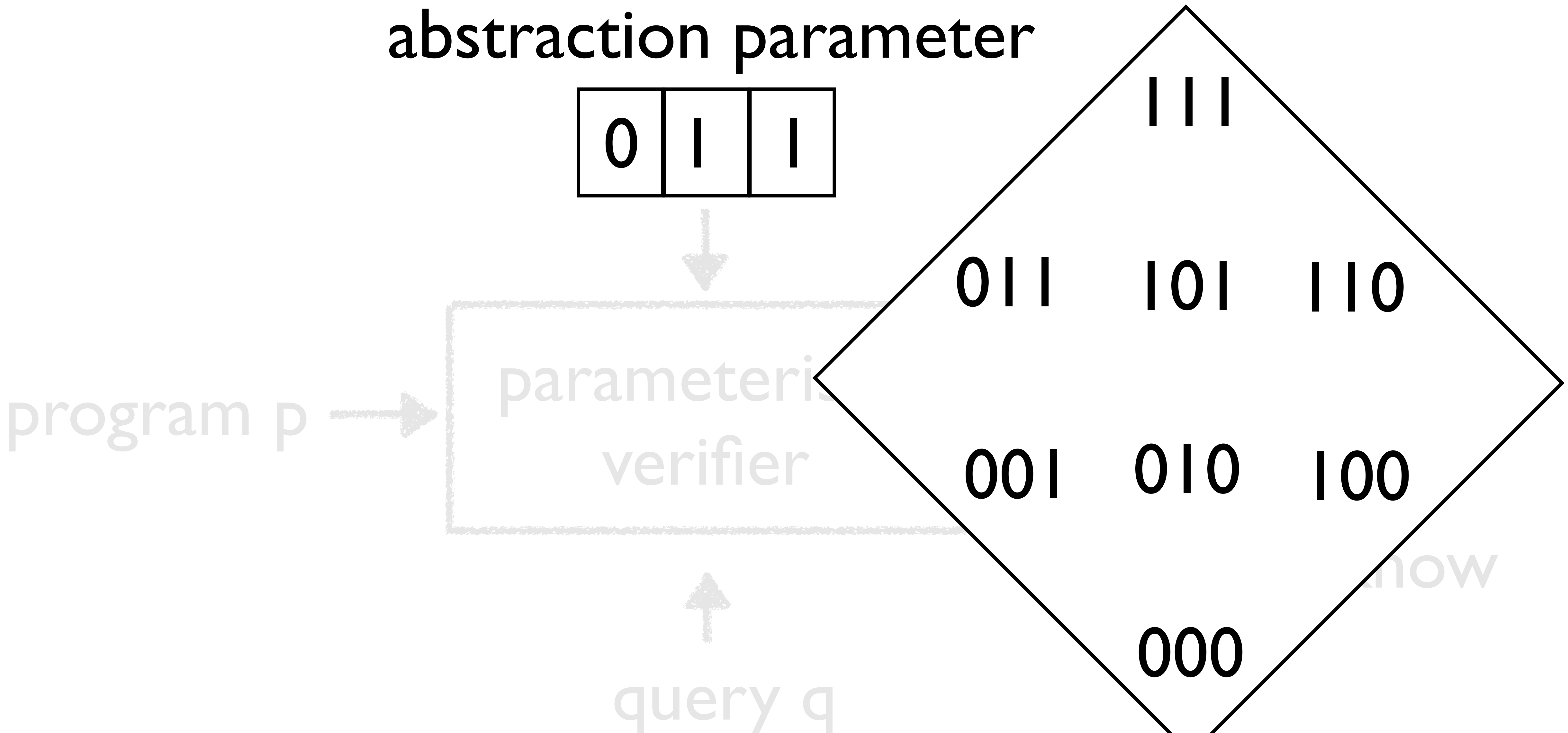
Verifier with explicit abstraction parameters



Verifier with explicit abstraction parameters

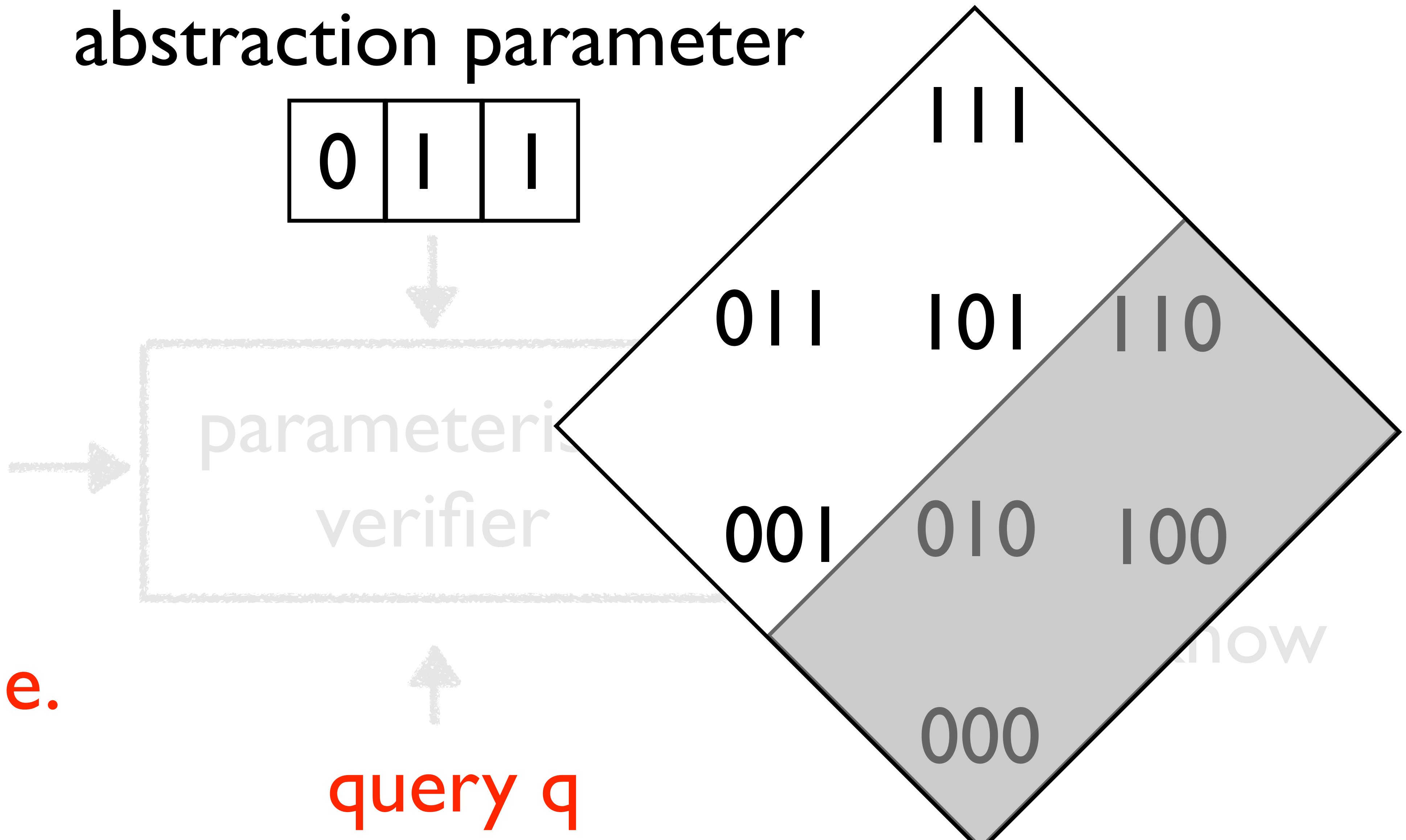


Verifier with explicit abstraction parameters



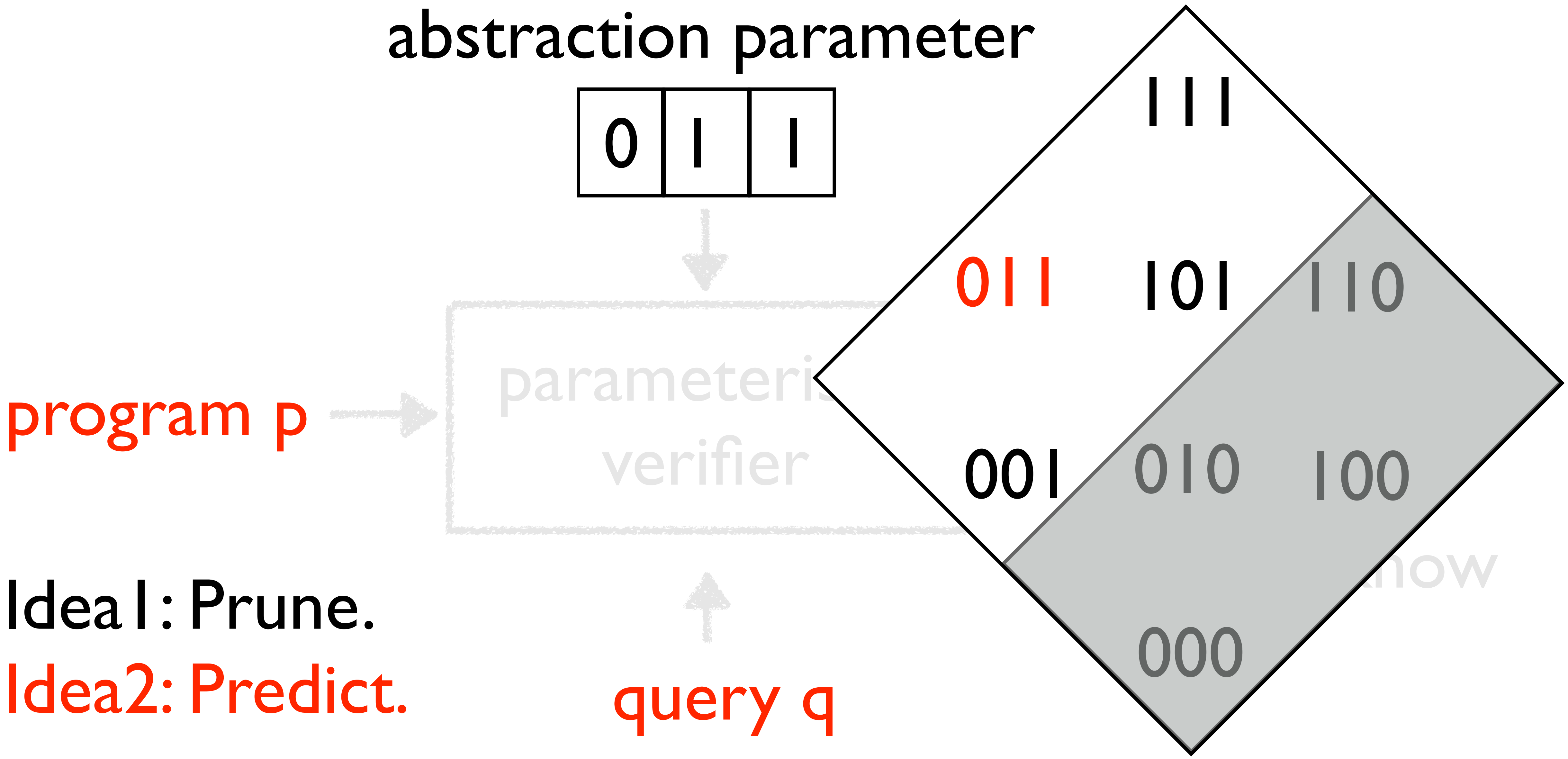
Verifier with explicit abstraction parameters

program p



Idea 1: Prune.

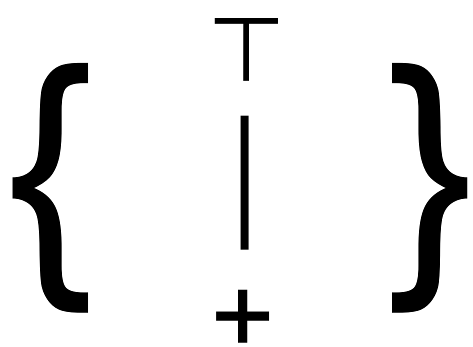
Verifier with explicit abstraction parameters

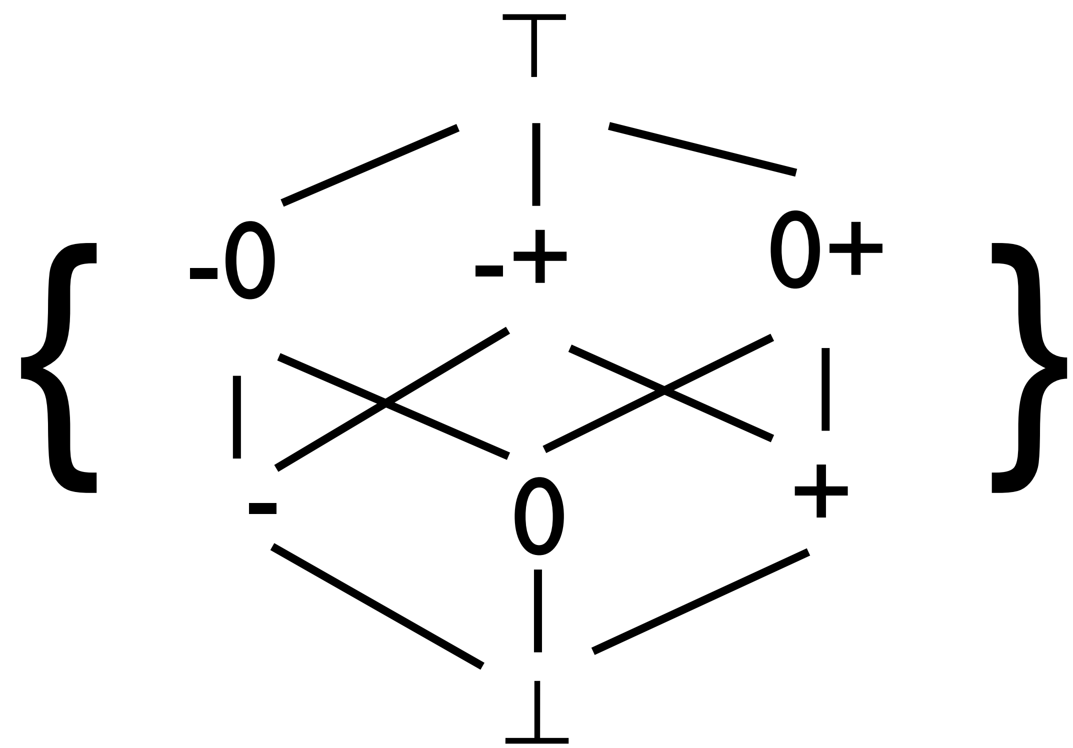


Idea1: Prune.
Idea2: Predict.

Pruning based on testing results [POPL'12]

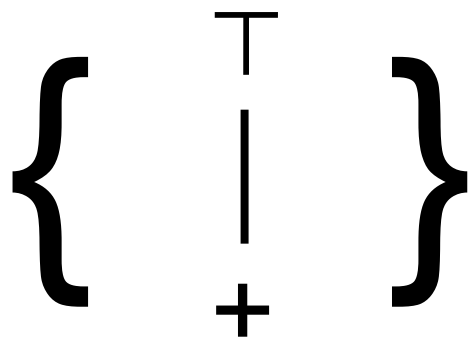
Two sign abstractions

$S_0 =$ 

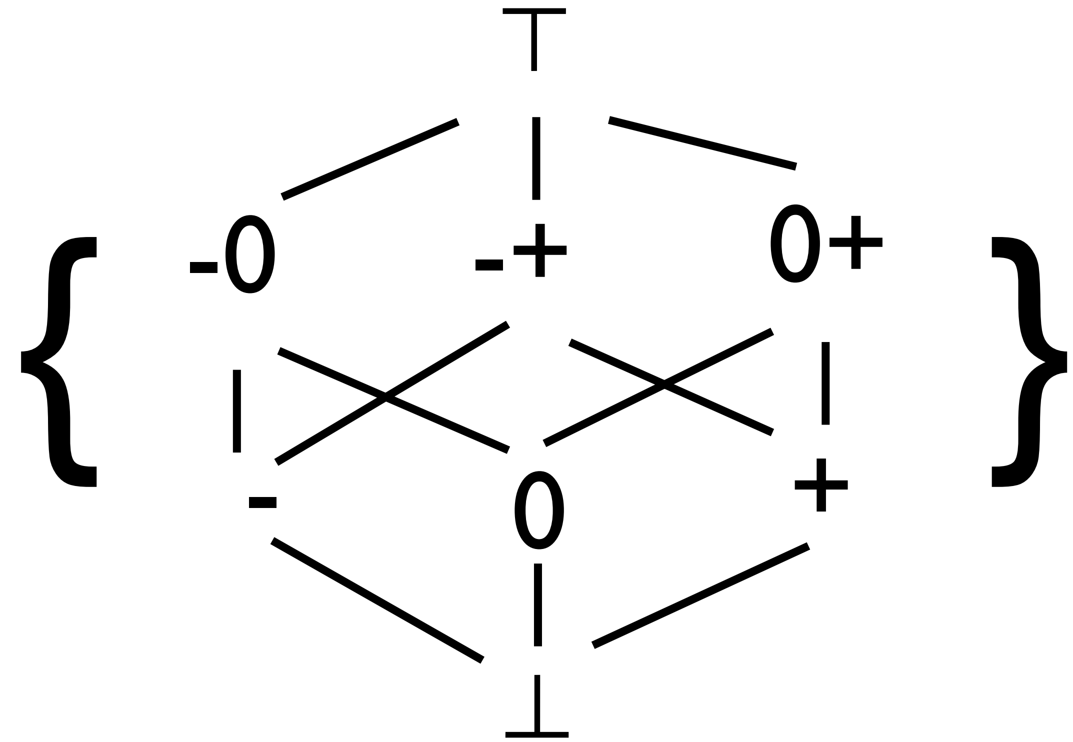
$S_1 =$ 

Abstraction parameters

$S_0 =$



$S_1 =$



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

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

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

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

Abstraction parameters

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

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
y = y-1;
assert(y >= 0);
```

Abstraction parameters

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

⟪n:1, x:1, y:0⟫

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
y = y-1;
assert(y >= 0);
```

[n:+,x:⊤,y:⊤]

4 iter

[n:⊤,x:+,y:+]
[n:⊤,x:+,y:⊤]

Abstraction parameters

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

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
y = y-1;
assert(y >= 0);
```

⟪n:1, x:1, y:0⟫

[n:+,x:⊤,y:⊤]

⋮

4 iter

[n:⊤,x:+,y:+]

[n:⊤,x:+,y:⊤]

⟪n:0, x:0, y:1⟫

[n:+,x:⊤,y:⊤]

⋮

2 iter

[n:⊤,x:+,y:+]

[n:⊤,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

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
y = y-1;
assert(y >= 0);
```

Testing and pruning

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
y = y-1;
assert(y >= 0);
```

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

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

Testing and pruning

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
y = y-1;
assert(y >= 0);
```

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

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

[n:_, x:_, y:$\top$] if abs(y)=0.

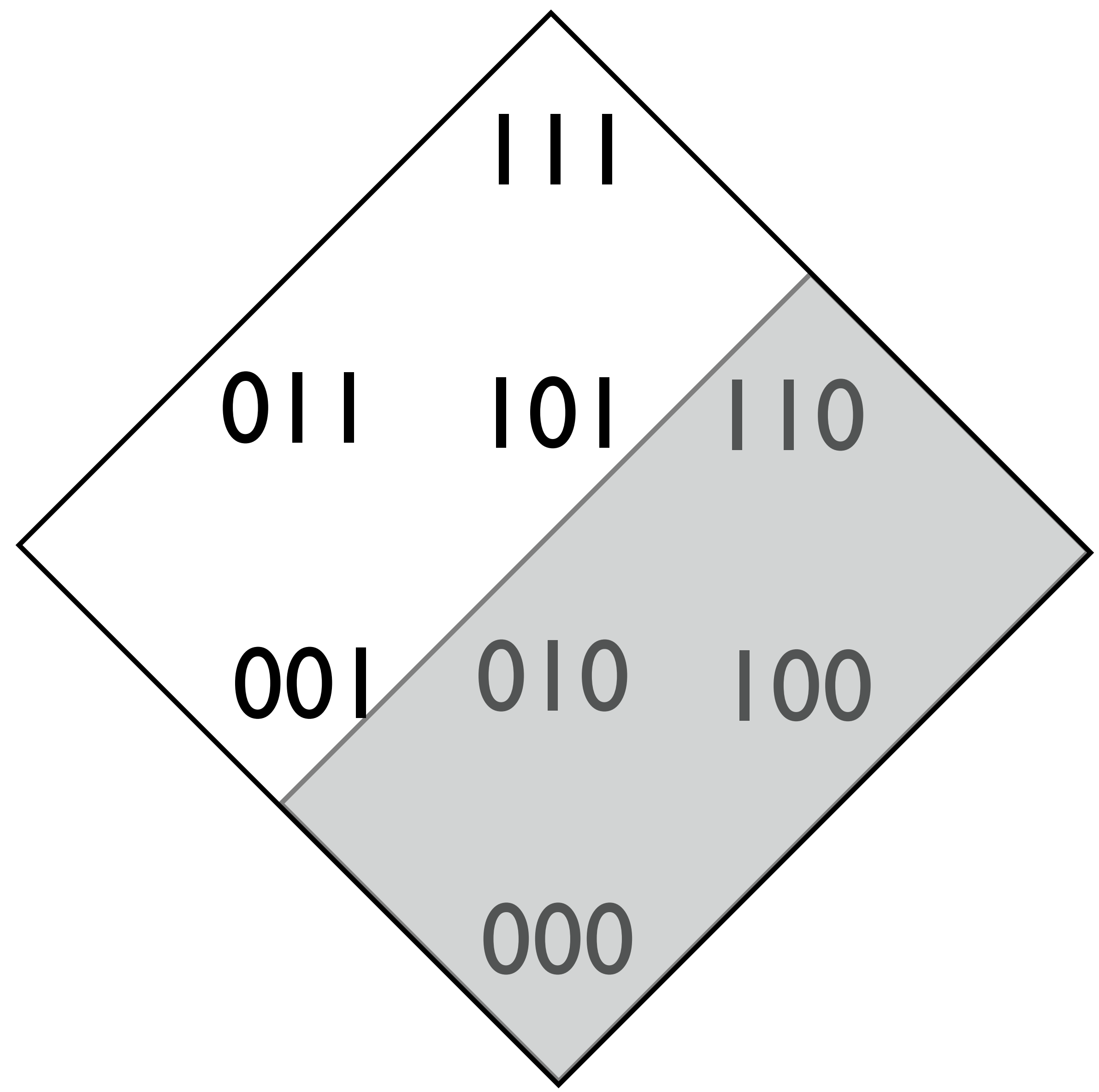
Because $S_0 = \{+, \top\}$.

Testing and pruning

```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
y = y-1;
assert(y >= 0);
```

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

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



[n:_, x:_, y:$\top$] if abs(y)=0.

Because S_0 = {+, $\top$}.

Testing and pruning

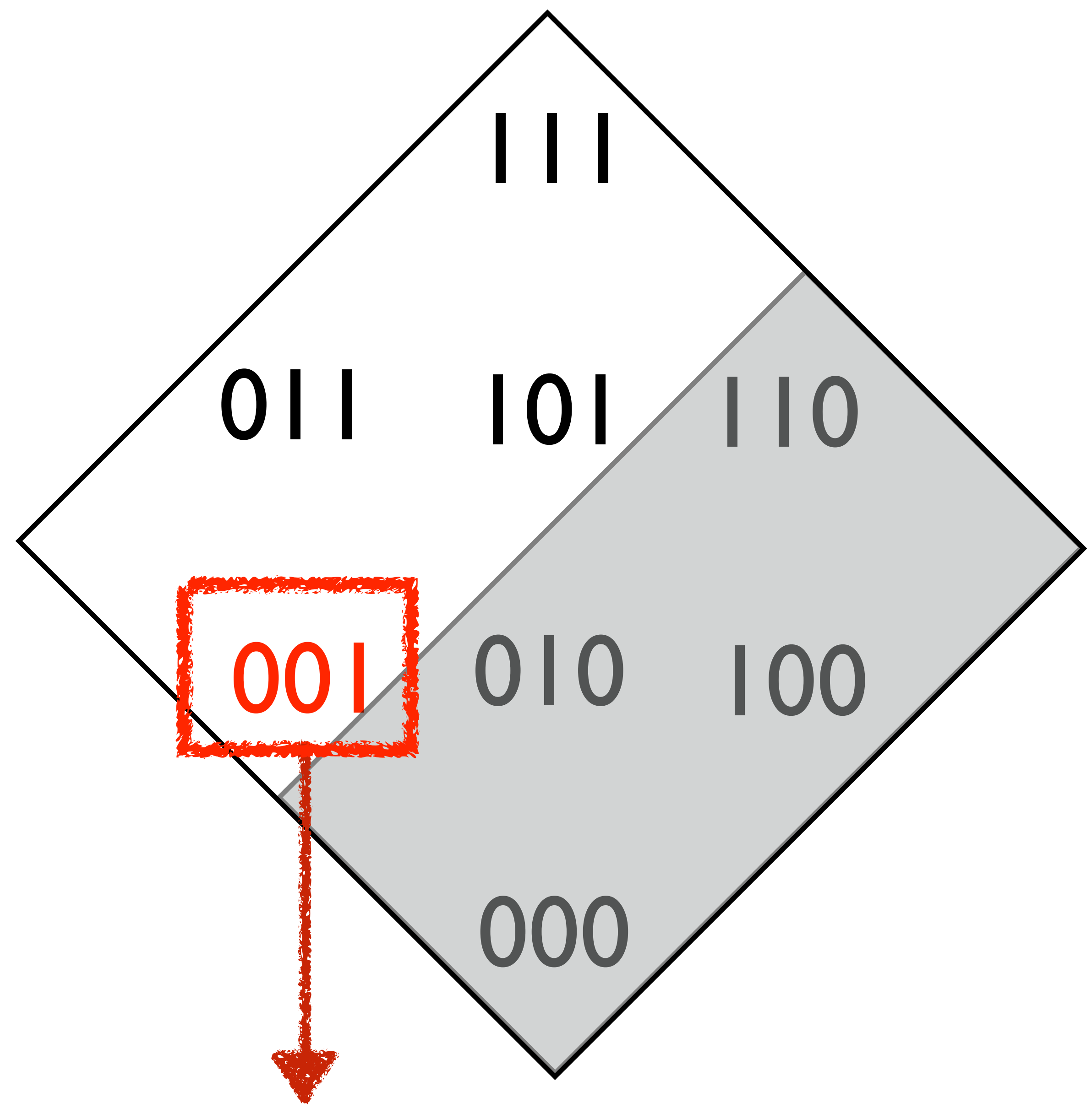
```
assert(n >= 1);
x = 1;
y = 1;
while (n > 1) {
  (x,y) = (y,x+y);
  n = n-1;
}
y = y-1;
assert(y >= 0);
```

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

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

[n:_, x:_, y:⊤] if abs(y)=0.

Because $S_0 = \{+, \top\}$.



Choose a minimal abs.

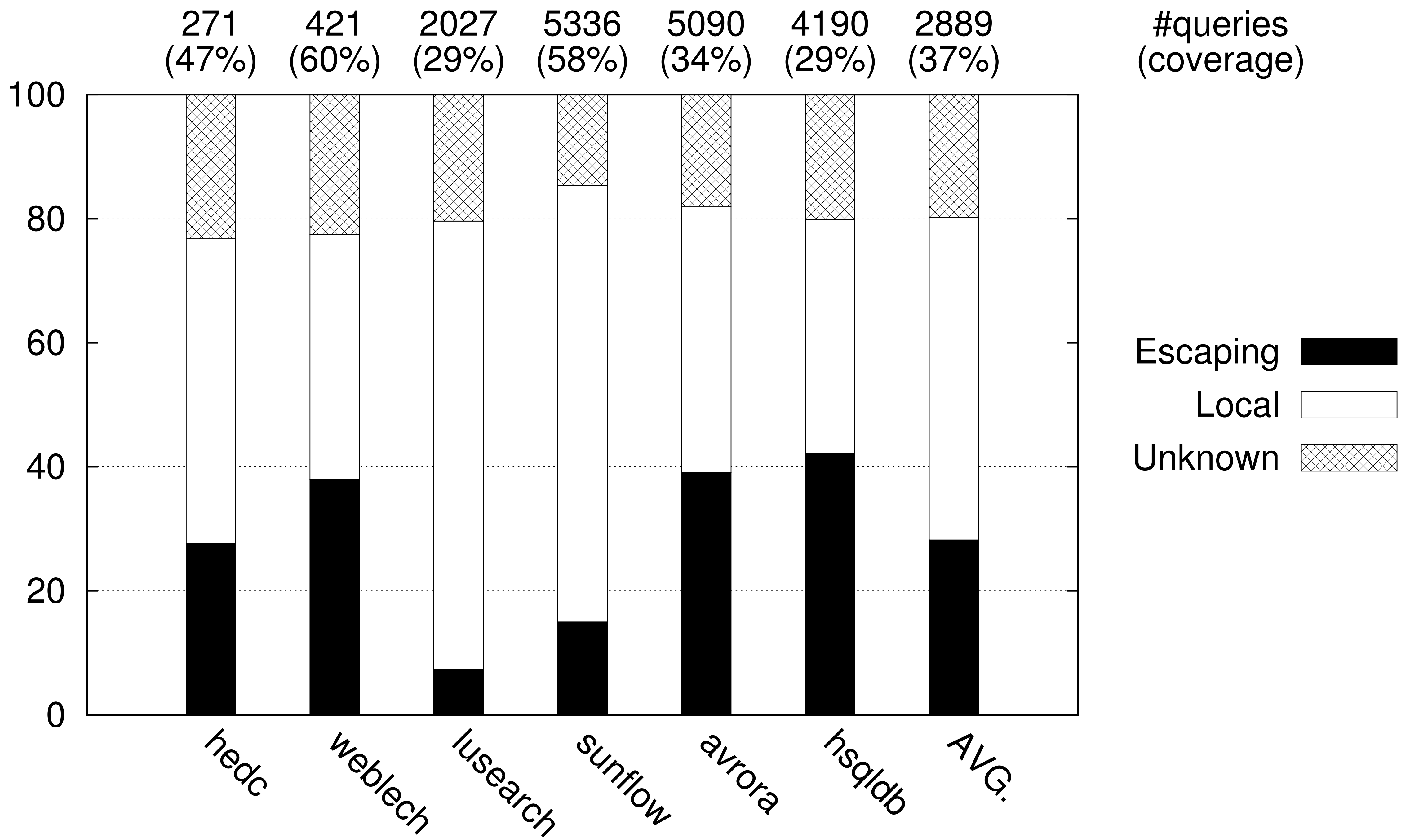


Figure 3. Precision results for our thread-escape analysis.

[POPL'12]

Intermediate assertions.
Abstraction.