

How to find a good program abstraction automatically?

Hongseok Yang
University of Oxford

Joint work with Ravi Mangal, Mayur Naik, Xin Zhang (Georgia Tech), Kihong Heo, Wonchan Lee, Hakjoo Oh, Kwangkeun Yi (SNU), Radu Grigore (Oxford)
Mooly Sagiv, Ghila Castelnovo (Tel-Aviv)

How to find a good program abstraction automatically? + some old story

Hongseok Yang
University of Oxford

Joint work with Ravi Mangal, Mayur Naik, Xin Zhang (Georgia Tech), Kihong Heo, Wonchan Lee, Hakjoo Oh, Kwangkeun Yi (SNU), Radu Grigore (Oxford)
Mooly Sagiv, Ghila Castelnovo (Tel-Aviv)

How to find a good program abstraction automatically? + some old story

Hongseok Yang
University of Oxford

Joint work with Ravi Mangal, Mayur Naik, Xin Zhang (Georgia Tech), ~~Kihong Heo, Wonchan Lee, Hakjoo Oh, Kwangkeun Yi (SNU)~~, Radu Grigore (Oxford)
Mooly Sagiv, Ghila Castelnovo (Tel-Aviv)

A problem has been detected and windows has been shut down to prevent damage to your computer.

DRIVER_IRQL_NOT_LESS_OR_EQUAL

If this is the first time you've seen this Stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup Options, and then select Safe Mode.

Technical information:

*** STOP: 0x000000D1 (0x0000000C,0x00000002,0x00000000,0xF86B5A89)

*** gv3.sys - Address F86B5A89 base at F86B5000, DateStamp 3dd991eb

Beginning dump of physical memory

Physical memory dump complete.

Contact your system administrator or technical support group for further assistance.

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。

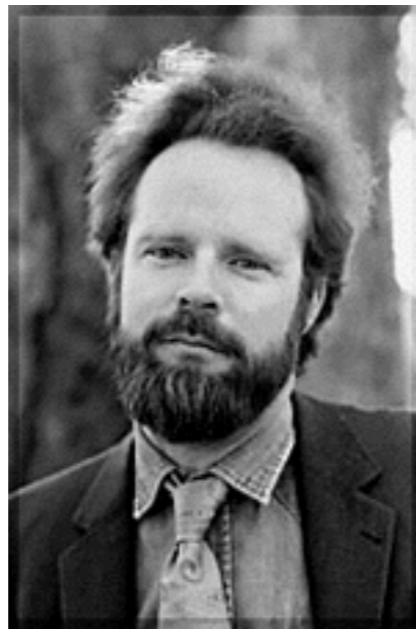
Software verification

- Active research area in computer science.
- Aims at verifying “no blue screen”, i.e., programs do not have errors.
- Develops methods for such verification.

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



Hoare



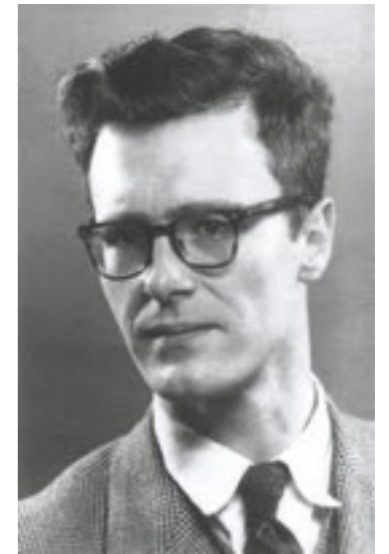
Floyd



Turing



Majumdar



Dijkstra

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



Hoare



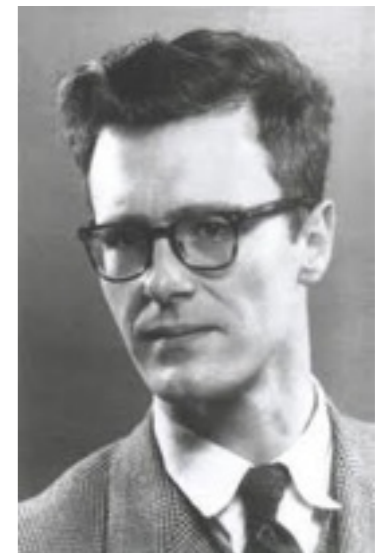
Floyd



Turing



Majumdar



Dijkstra

Turing in June 1949

Friday, 24th June.

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

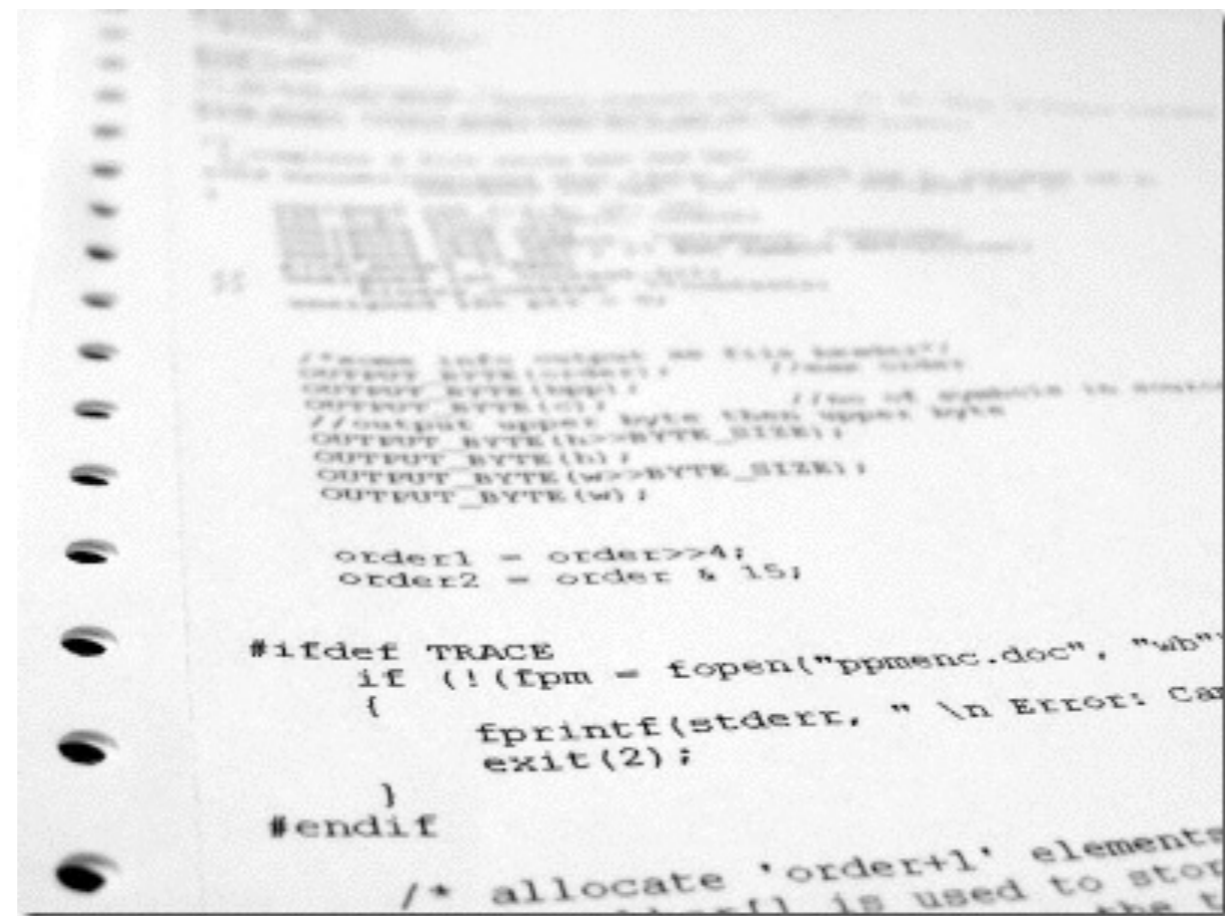
How can one check a routine in the sense of making sure that it is right?

Turing's idea

Use intermediate assertions.

Turing's idea

Use intermediate assertions.



```
// output info output as this format!
// output BYTE (0-255)
// output BYTE (0-255)
// output BYTE (0-255)
// output upper byte then upper byte
// output BYTE (0-255)
// output BYTE (0-255)
// output BYTE (0-255)

order1 = order >> 4;
order2 = order & 15;

#ifdef TRACE
if (!(fpm = fopen("ppmenc.doc", "wb"))
{
    fprintf(stderr, " \n Error: Can't
    exit(2);
}
#endif

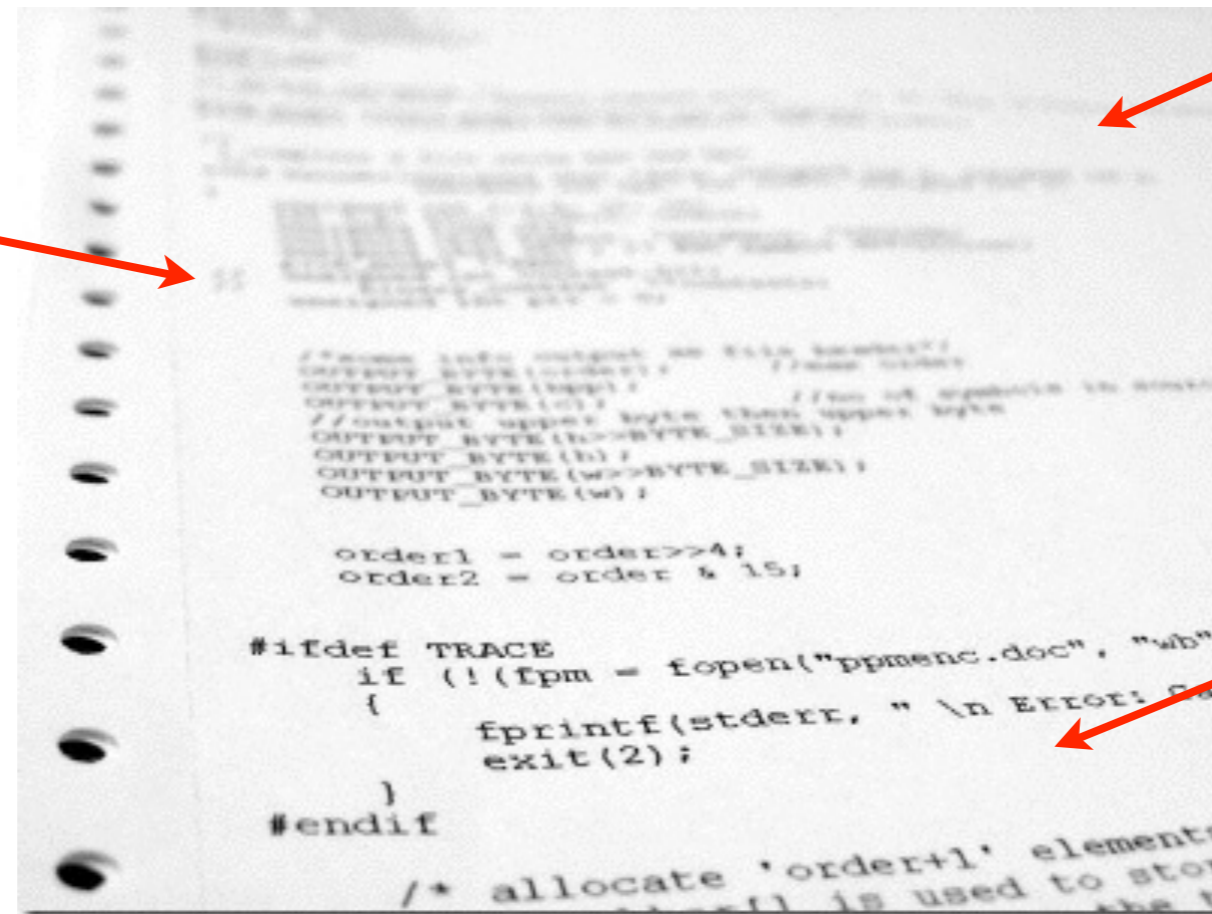
/* allocate 'order+1' elements
   ... is used to store the t
```

Verify that this program computes “100 * 30”.

Turing's idea

Use intermediate assertions.

$x = 60$
and $i = 2$



```
/* ... */
// ...
// output upper byte then upper byte
OUTPUT_BYTE(h>>BYTE_SIZE);
OUTPUT_BYTE(h);
OUTPUT_BYTE(w>>BYTE_SIZE);
OUTPUT_BYTE(w);

order1 = order>>4;
order2 = order & 15;

#ifdef TRACE
if (!(fpm = fopen("ppmenc.doc", "wb"))
{
    fprintf(stderr, " \n Error: can't open file\n");
    exit(2);
}
#endif

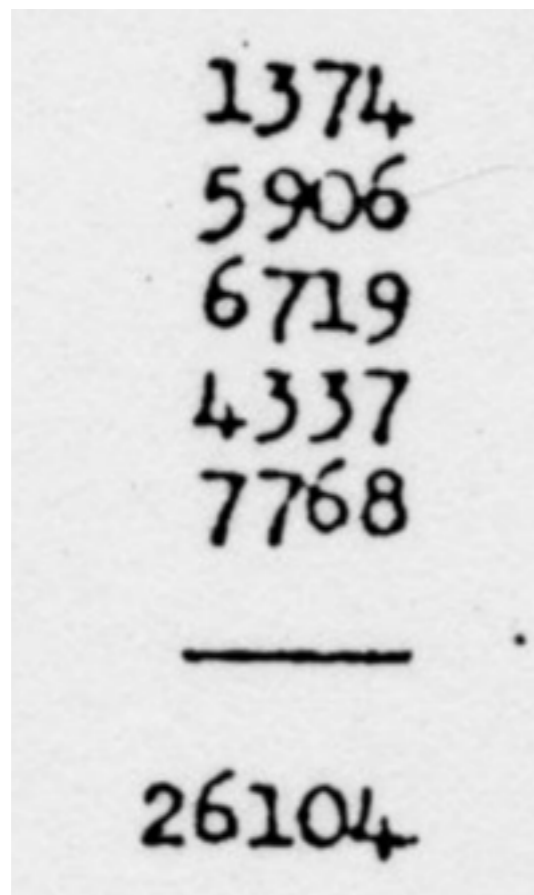
/* allocate 'order+1' elements
... is used to store the t
```

$x = 30$
and $i = 1$

$x = 2970$
and $i = 29$

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

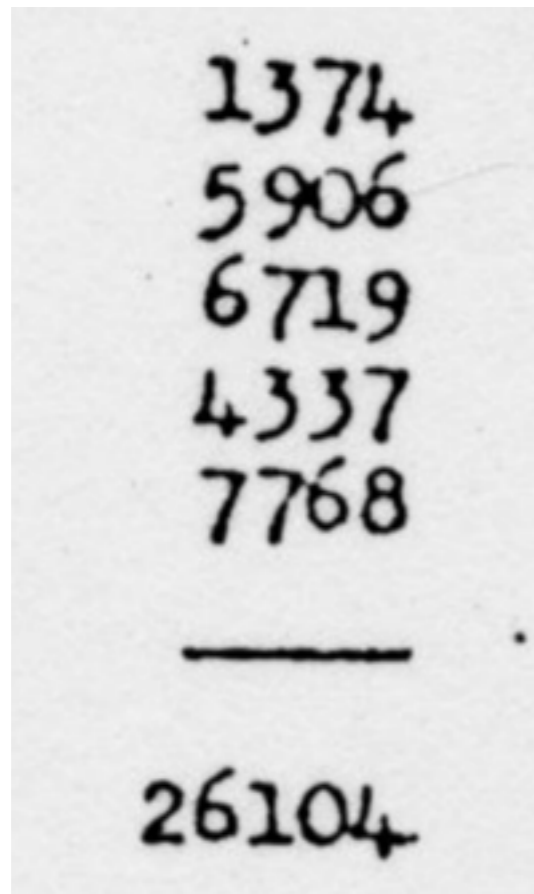
Turing's example



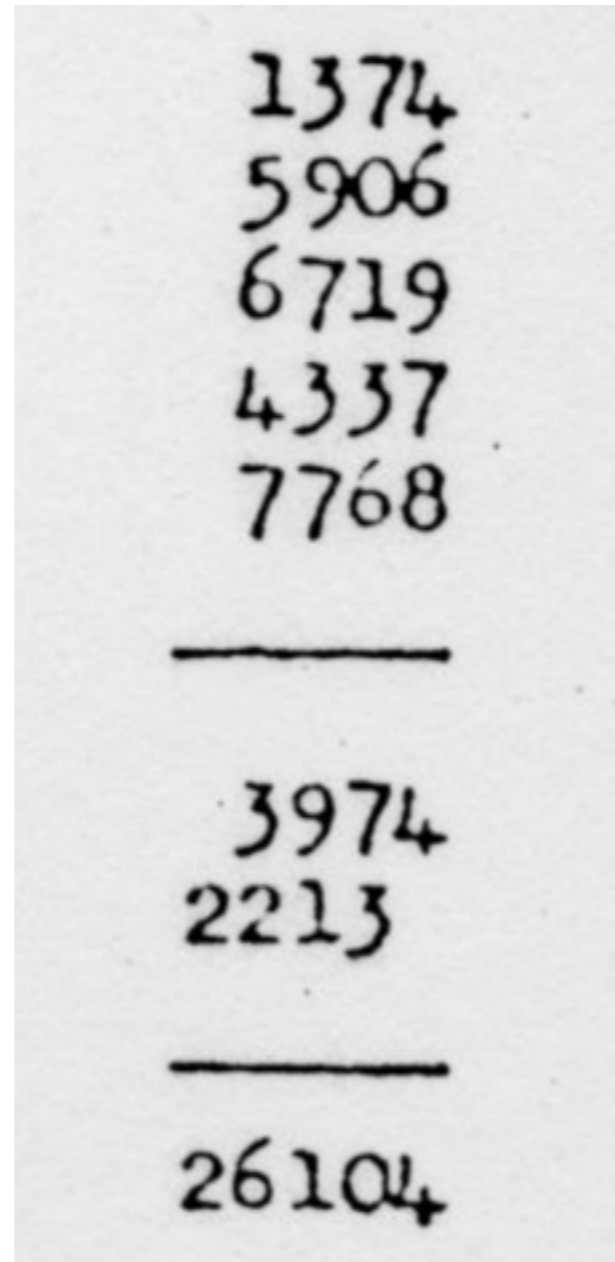
1374
5906
6719
4337
7768
—
26104

A photograph of a piece of paper with handwritten numbers in a vertical column. The numbers are 1374, 5906, 6719, 4337, and 7768. Below these is a horizontal line, and then the number 26104. The handwriting is in a simple, slightly slanted font.

Turing's example



1374
5906
6719
4337
7768
—
26104



1374
5906
6719
4337
7768
—
3974
2213
—
26104

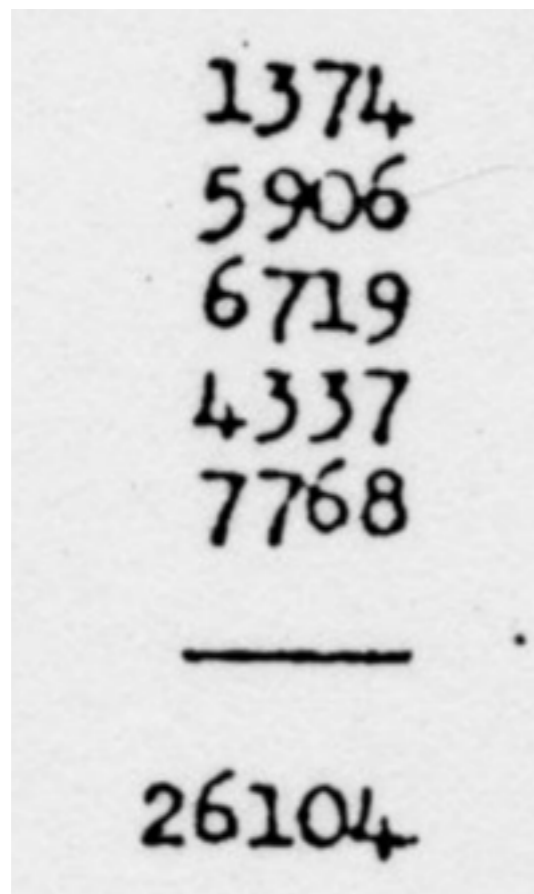
Turing's example

1374
5906
6719
4337
7768
—
26104

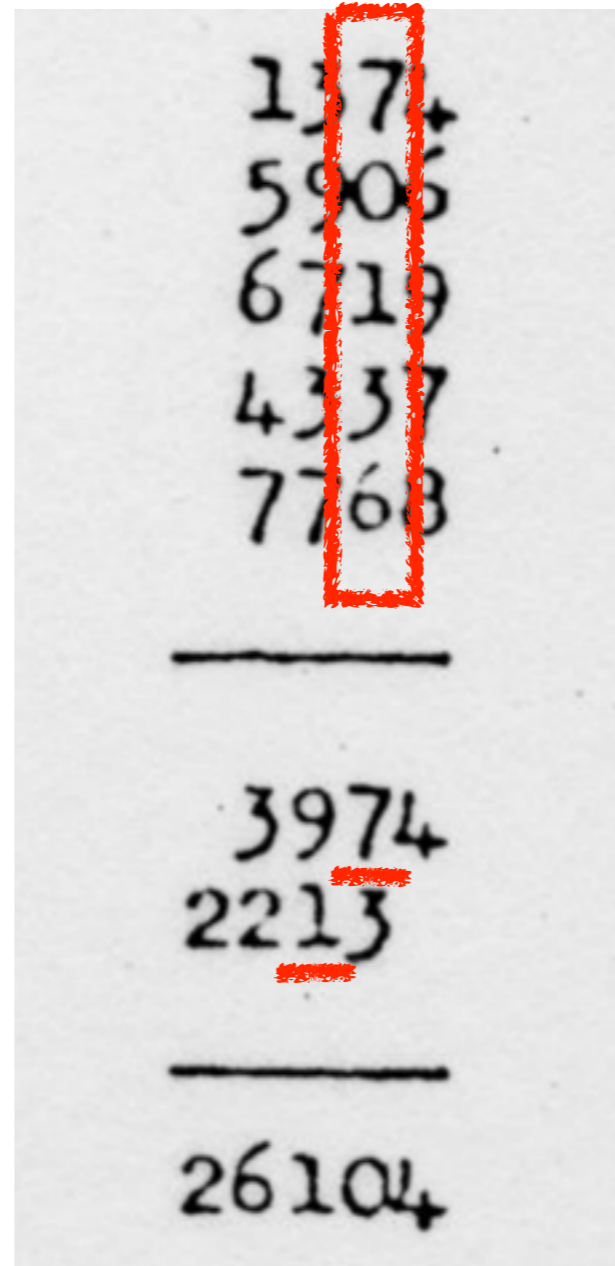
1374
5906
6719
4337
7768
—
3974
2213
—
26104

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

Turing's example



Handwritten addition of two 5-digit numbers:

$$\begin{array}{r} 1374 \\ 5906 \\ 6719 \\ 4337 \\ 7768 \\ \hline 26104 \end{array}$$


Handwritten addition of two 5-digit numbers, with a red box highlighting the first column:

$$\begin{array}{r} 1374 \\ 5906 \\ 6719 \\ 4337 \\ 7768 \\ \hline 3974 \\ 2213 \\ \hline 26104 \end{array}$$

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

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

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

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

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

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

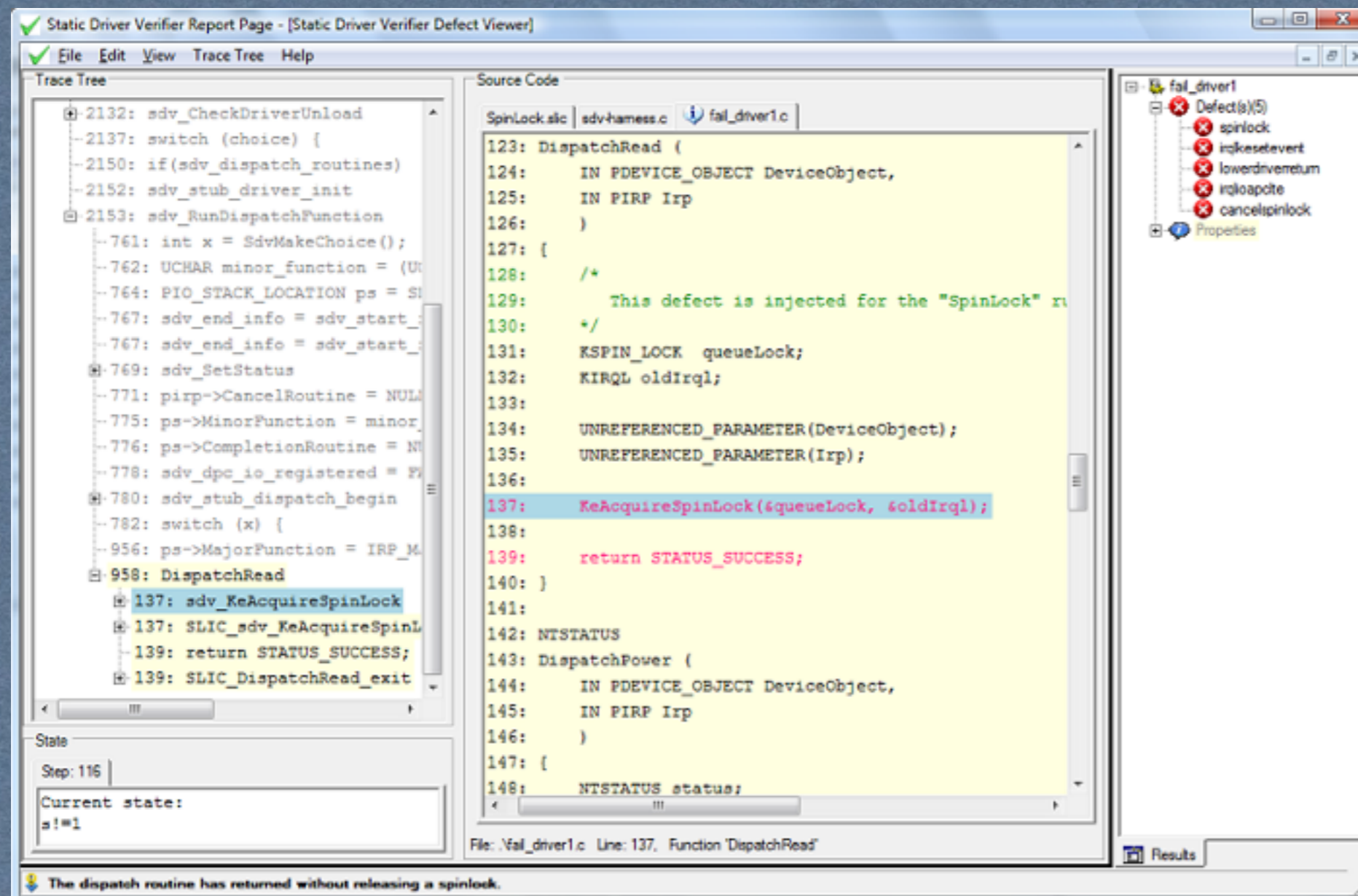
$$1+5+6+4+7 = 23$$

Intermediate assertions

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

Commercial tools in 2014

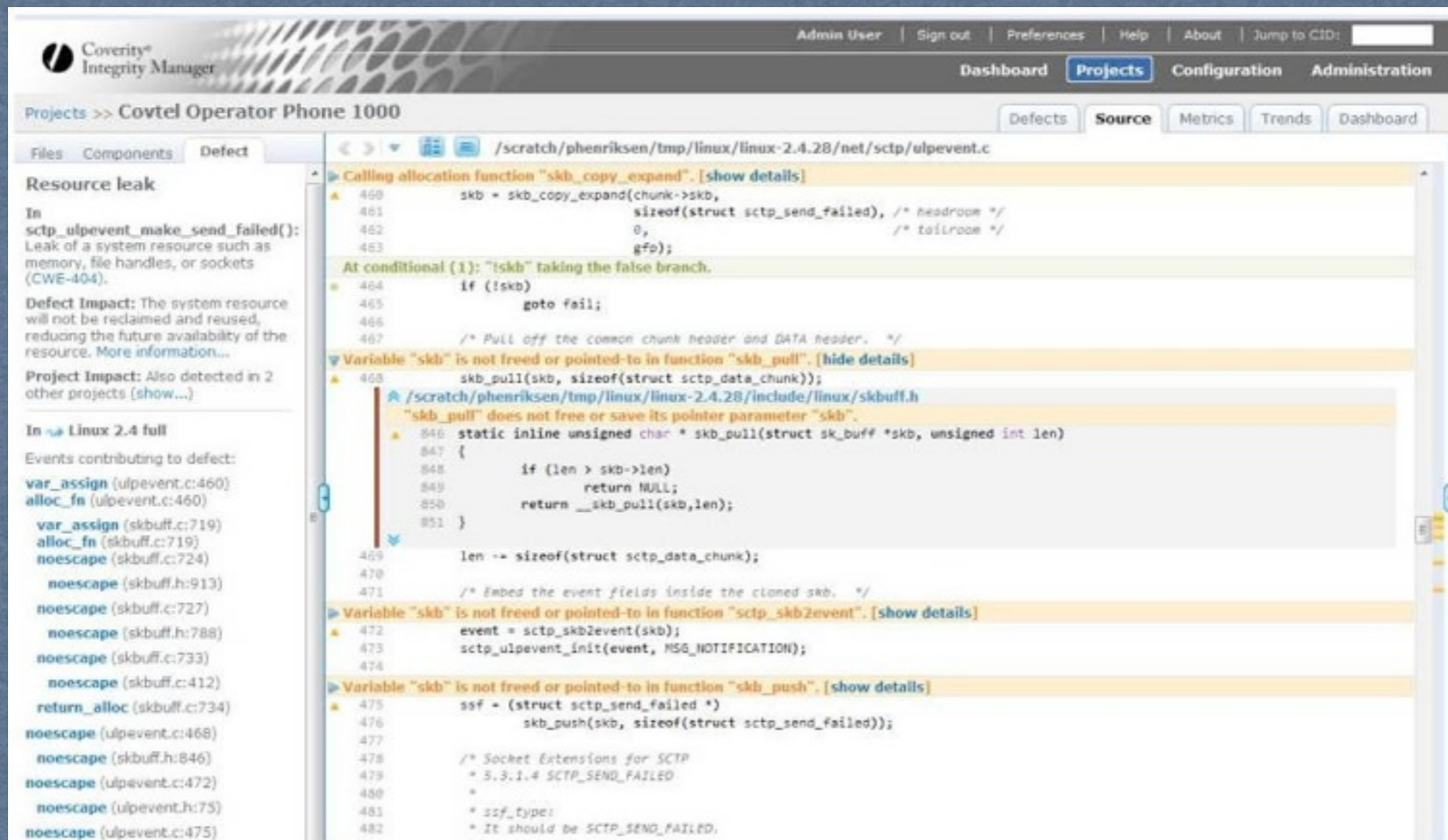
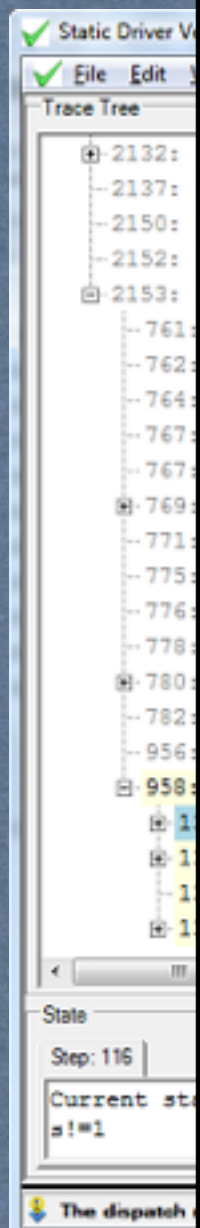
Microsoft SDV



Commercial tools in 2014

Micro

Coverity Prevent

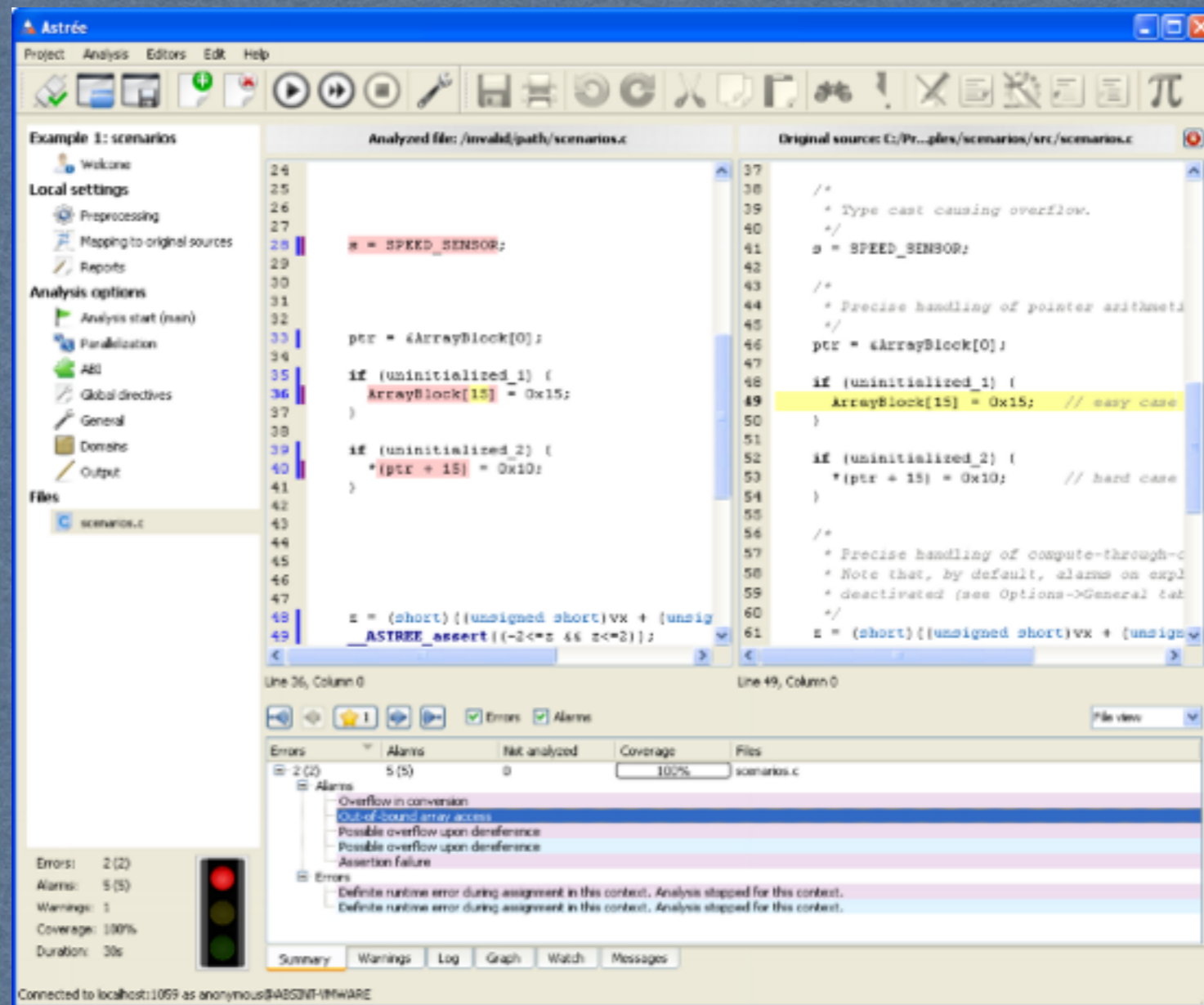


Commercial tools in 2014

Micro

Coverage

AbsInt Astree



Abstraction

- Key idea behind automation.
- Keeps only important properties of programs. Forgets all the rest.

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

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

```
1:  assert(n >= 1);    [n:3,x:0,y:0]
2:  x = 1;              ↓
3:  y = 1;              [n:3,x:1,y:1]  [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 bottom state $[n:2, x:1, y:2]$ to a red state $[n:2, x:1, y:2]$ to the 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;        [n:2,x:1,y:2]  [n:1,x:2,y:3]
7:  }
```

The diagram illustrates the state transitions of a Fibonacci algorithm. It starts with the initial state $[n:3, x:0, y:0]$. A black arrow points down to $[n:3, x:1, y:1]$. From there, a black arrow points down to $[n:2, x:1, y:2]$. A black arrow also points from $[n:3, x:1, y:1]$ to $[n:2, x:1, y:2]$. A red arrow points from $[n:2, x:1, y:2]$ to $[n:1, x:2, y:3]$. A red arrow also points from $[n:2, x:1, y:2]$ to $[n:2, x:1, y:2]$.

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;              [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 control flow graph for the Fibonacci algorithm. The graph shows the state of variables n , x , and y at different points in the code. The initial state is $[n:3, x:0, y:0]$. After line 2, the state is $[n:3, x:1, y:1]$. After line 3, the state is $[n:3, x:1, y:1]$. After line 4, the state is $[n:2, x:1, y:2]$. After line 5, the state is $[n:2, x:1, y:2]$. After line 6, the state is $[n:1, x:2, y:3]$. The final state is $[n:1, x:2, y:3]$.

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;
7:  }
```

The diagram illustrates the state transitions of the Fibonacci algorithm. The states are represented as $[n, x, y]$ where n is the loop counter, x is the current Fibonacci number, and y is the previous Fibonacci number. The transitions are as follows:

- 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 the first iteration of the while loop: $[n:2, x:1, y:2]$
- After the second iteration of the while loop: $[n:1, x:2, y:3]$
- Final state (red): $[n:1, x:2, y:3]$

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 line 2: $[n:3, x:1, y:1]$
- After line 3: $[n:2, x:1, y:2]$
- After line 4: $[n:1, x:2, y:3]$
- After line 5: $[n:2, x:1, y:2]$
- After line 6: $[n:1, x:2, y:3]$
- After line 7: $[n:1, x:2, y:3]$

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
|
+

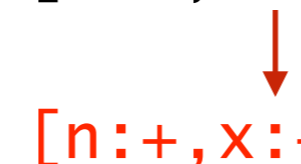
- An abstract state is a map from variables to abstract values. E.g. $[n:\top, x:+, y:+]$.

Analysing Fibonacci with simple sign abstraction

```
1:  assert(n >= 1);    [n:+,x:T,y:T]
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:τ,y:τ]
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 >= 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);    [n:⊤,x:+,y:+]

      ↓
      ↓
      ↓
```

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:	$(x, y) = (y, x+y)$;	$[n:\top, x:+, y:+] \rightarrow [n:\top, x:+, y:+] \text{ (red arrow)}$
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 initialization: $[n:+, x:+, y:+]$
- Inside the while loop body: $[n:\tau, x:+, y:+]$
- After the while loop body: $[n:\tau, x:+, y:+]$

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);
```

```
graph TD; A["[n:+, x:τ, y:τ]"] --> B["[n:+, x:+, y:+]"]; B --> C["[n:τ, x:+, y:+]"]; B --> D["[n:τ, x:+, y:+]"]; C --> E["[n:τ, x:+, y:+]"]; D --> E; E -- red --> F["[n:τ, 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);
```

Diagram illustrating the sign abstraction of the Fibonacci code:

- Concrete state: $[n:+, x:\tau, y:\tau]$
- Abstract state: $[n:+, x:+, y:+] \rightarrow [n:\tau, x:+, y:+] \rightarrow [n:\tau, x:+, \underline{y:~}]$

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

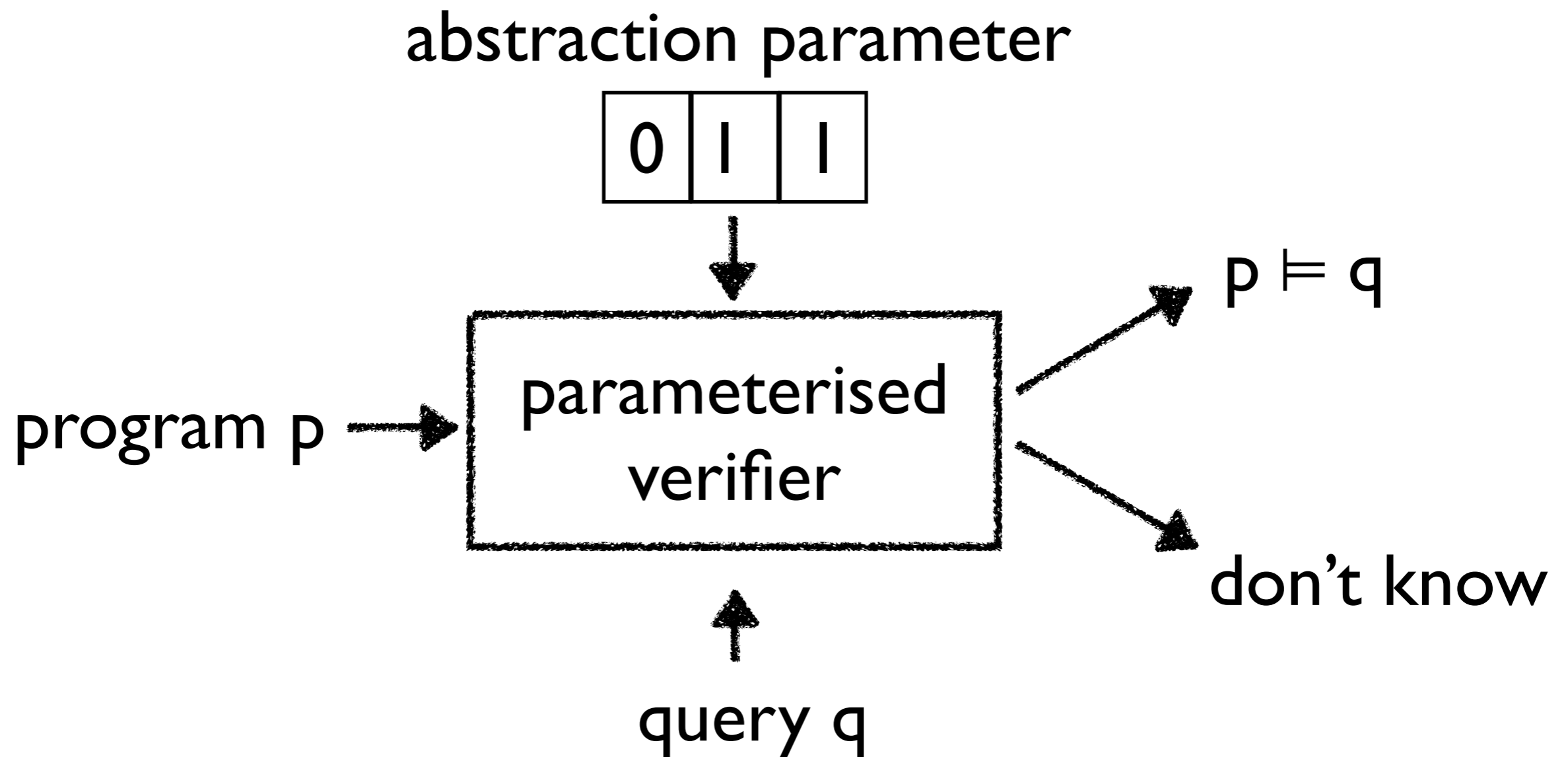
```
1:  assert(n >= 1);
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  y = y-1;
9:  assert(y >= 0);
```

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

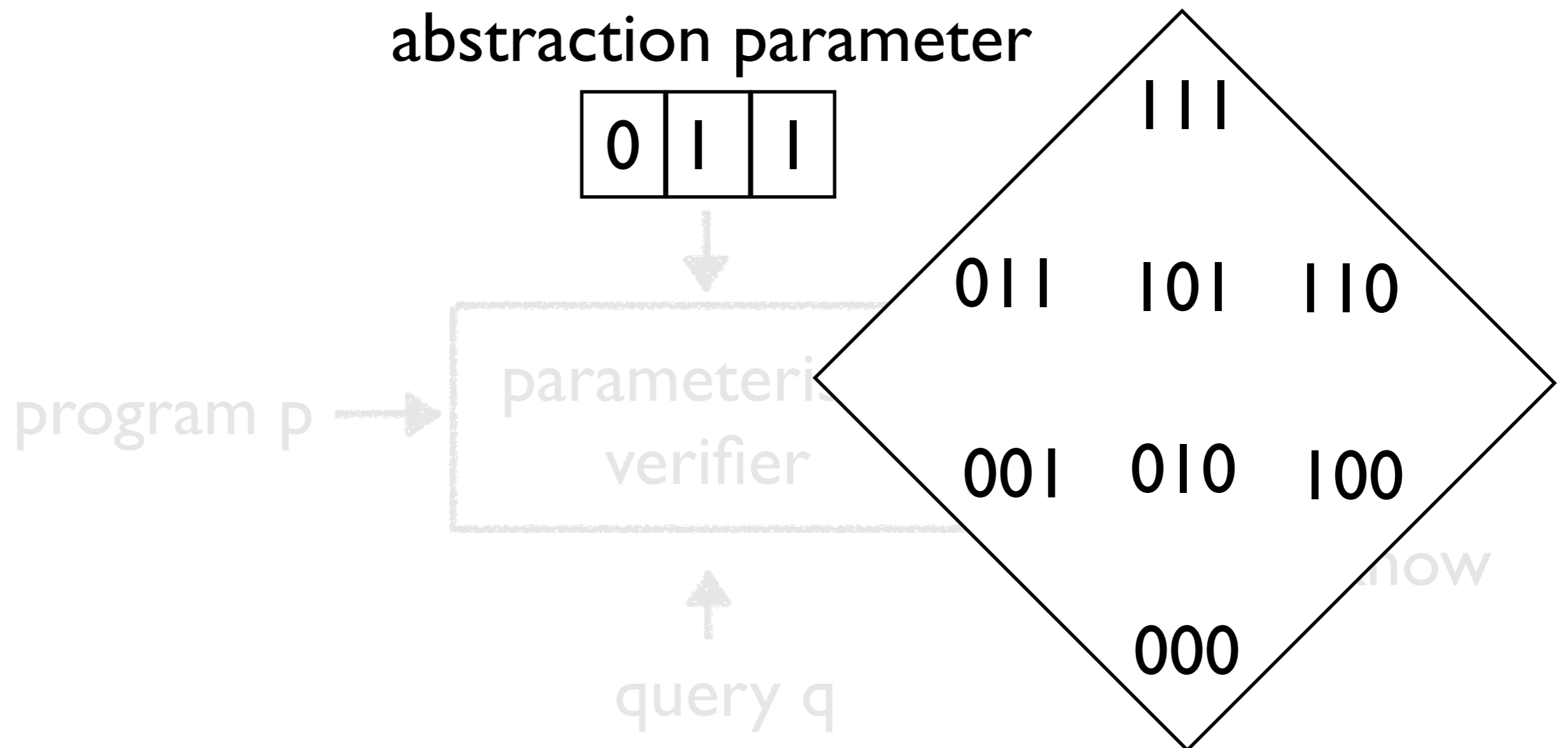
Our approach since 2011

- Formulate abstraction finding as a search problem [POPL'12, PLDI'13, PLDI14a, PLDI14b].
- Choose search space carefully.
- Develop an efficient search algorithm.

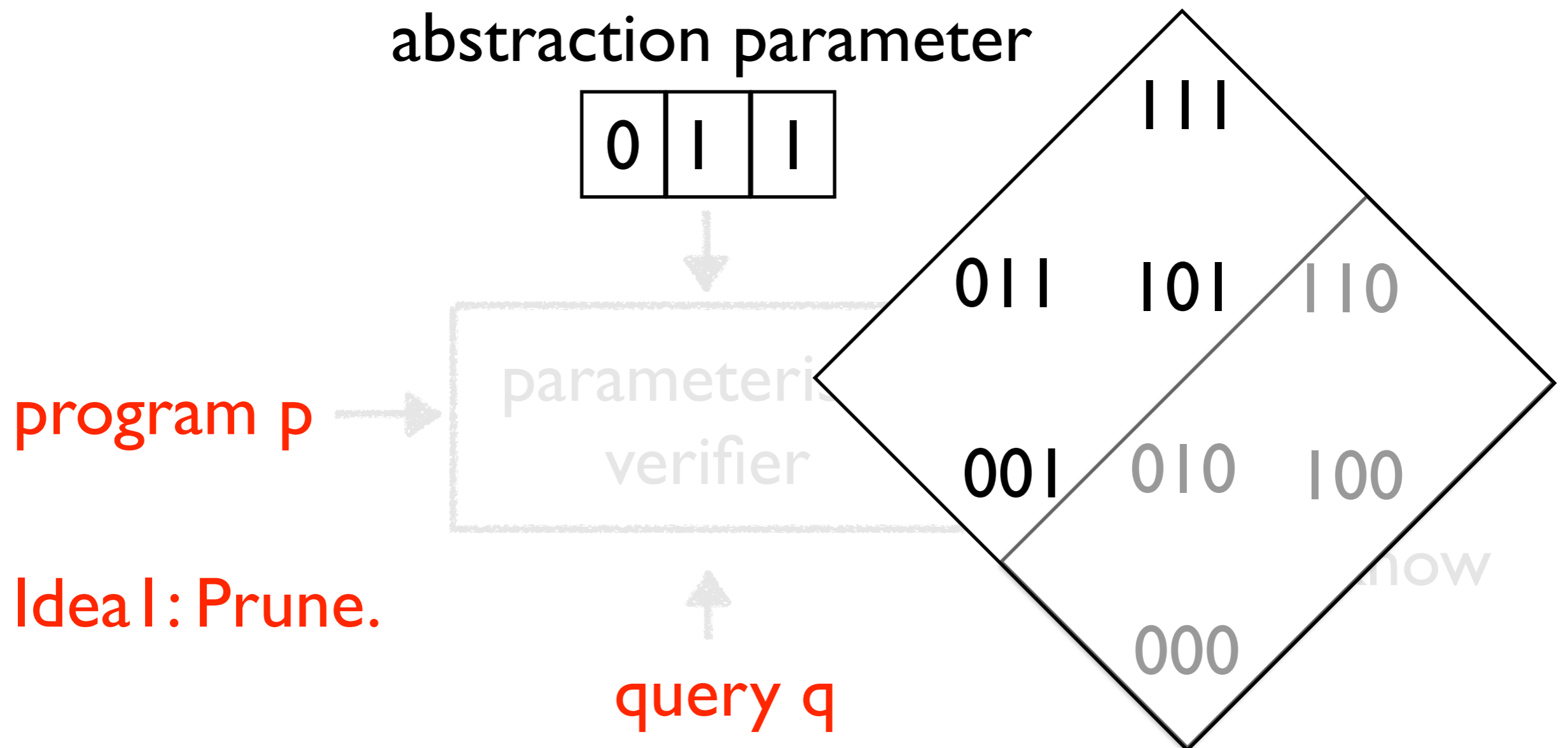
Verifier with explicit abstraction parameters



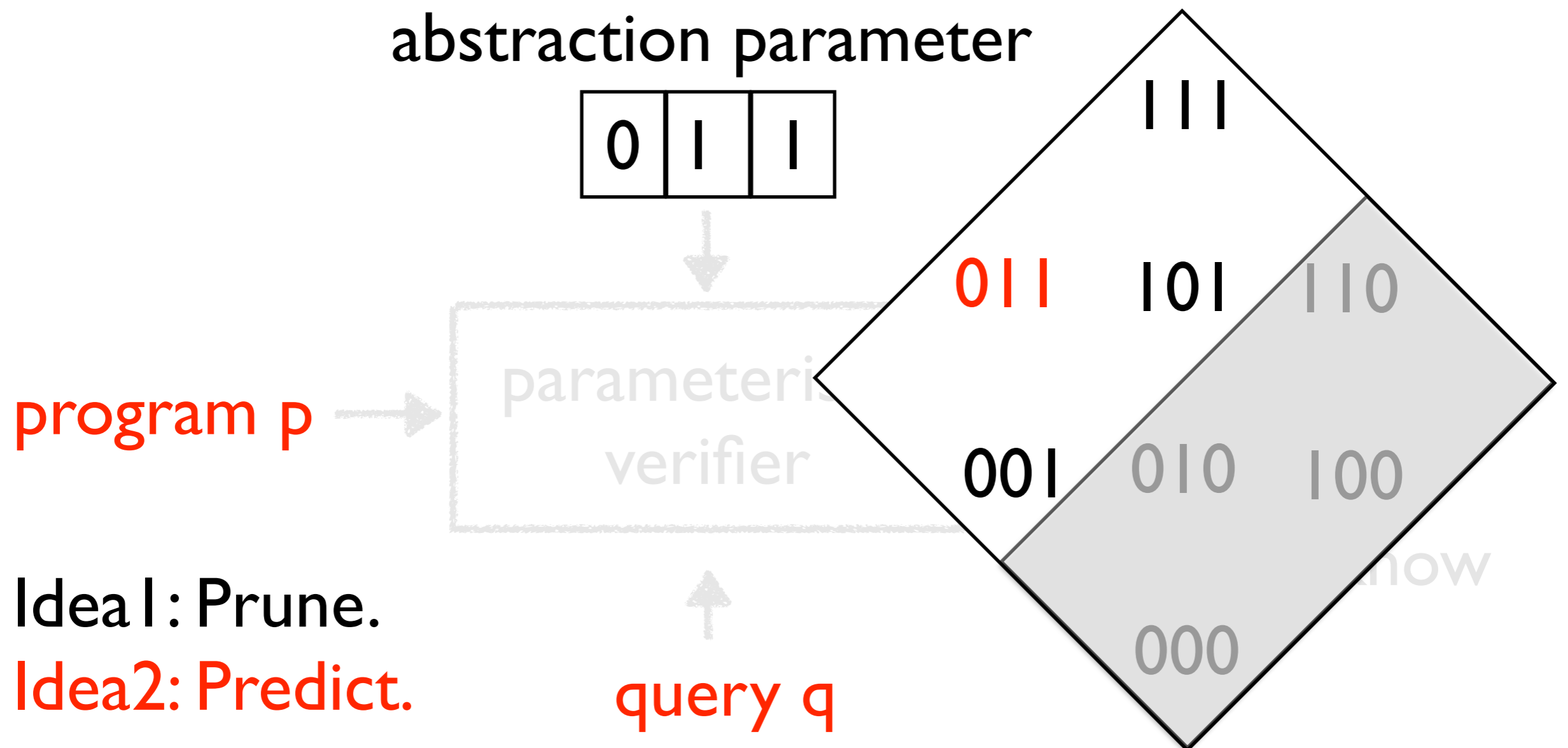
Verifier with explicit abstraction parameters



Verifier with explicit abstraction parameters



Verifier with explicit abstraction parameters



Pruning based on
testing results

Two sign abstractions

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

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

Abstraction parameters

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

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

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

$$\text{abs}_0 = [n:0, x:0, y:0]$$

$$\text{abs}_1 = [n:1, x:1, y:1]$$

$$\text{abs}_2 = [n:0, x:0, y:1]$$

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);
```

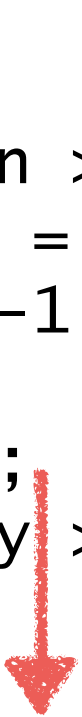
Testing and pruning

```
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);

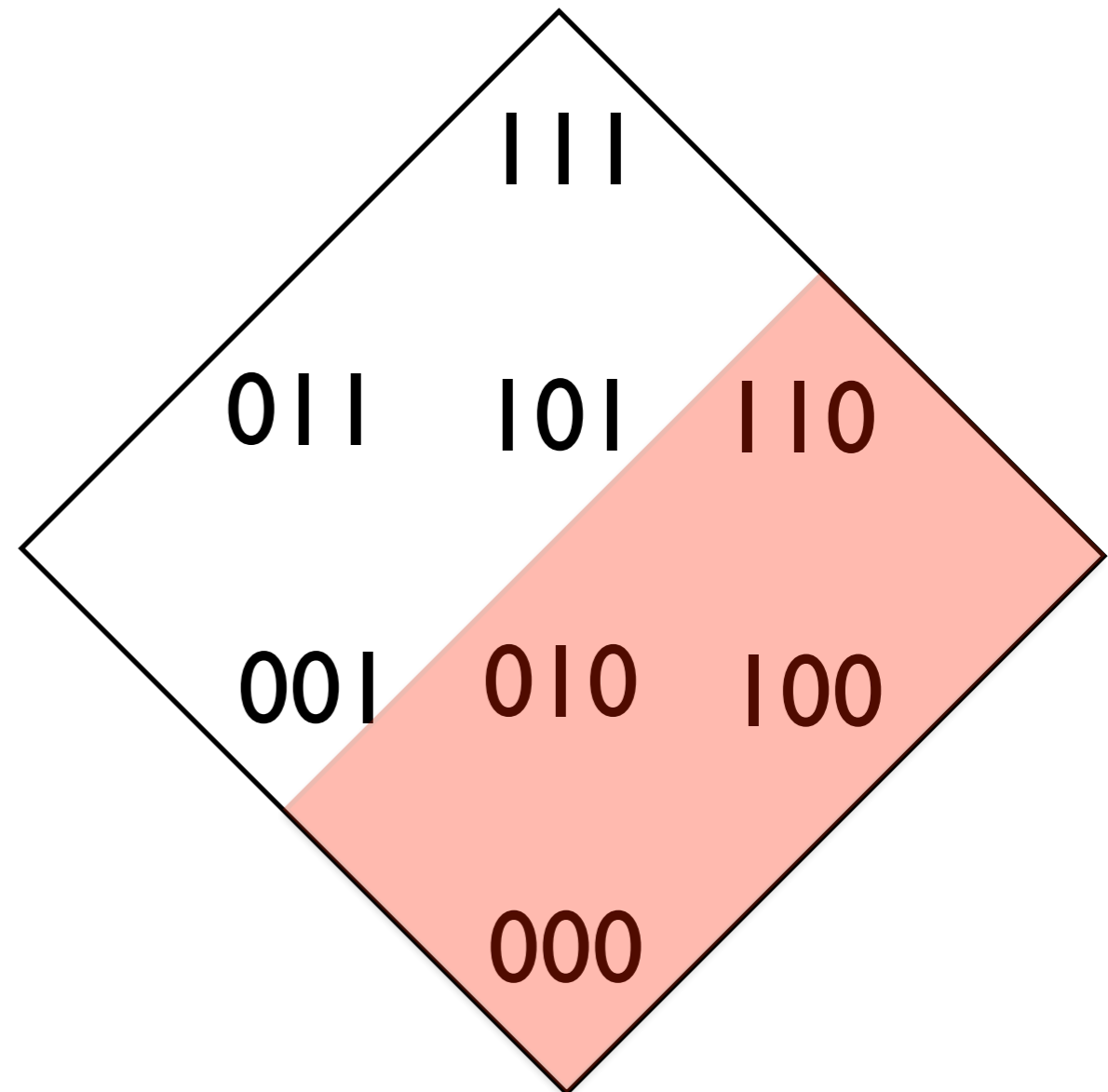
```

↓

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

↓

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



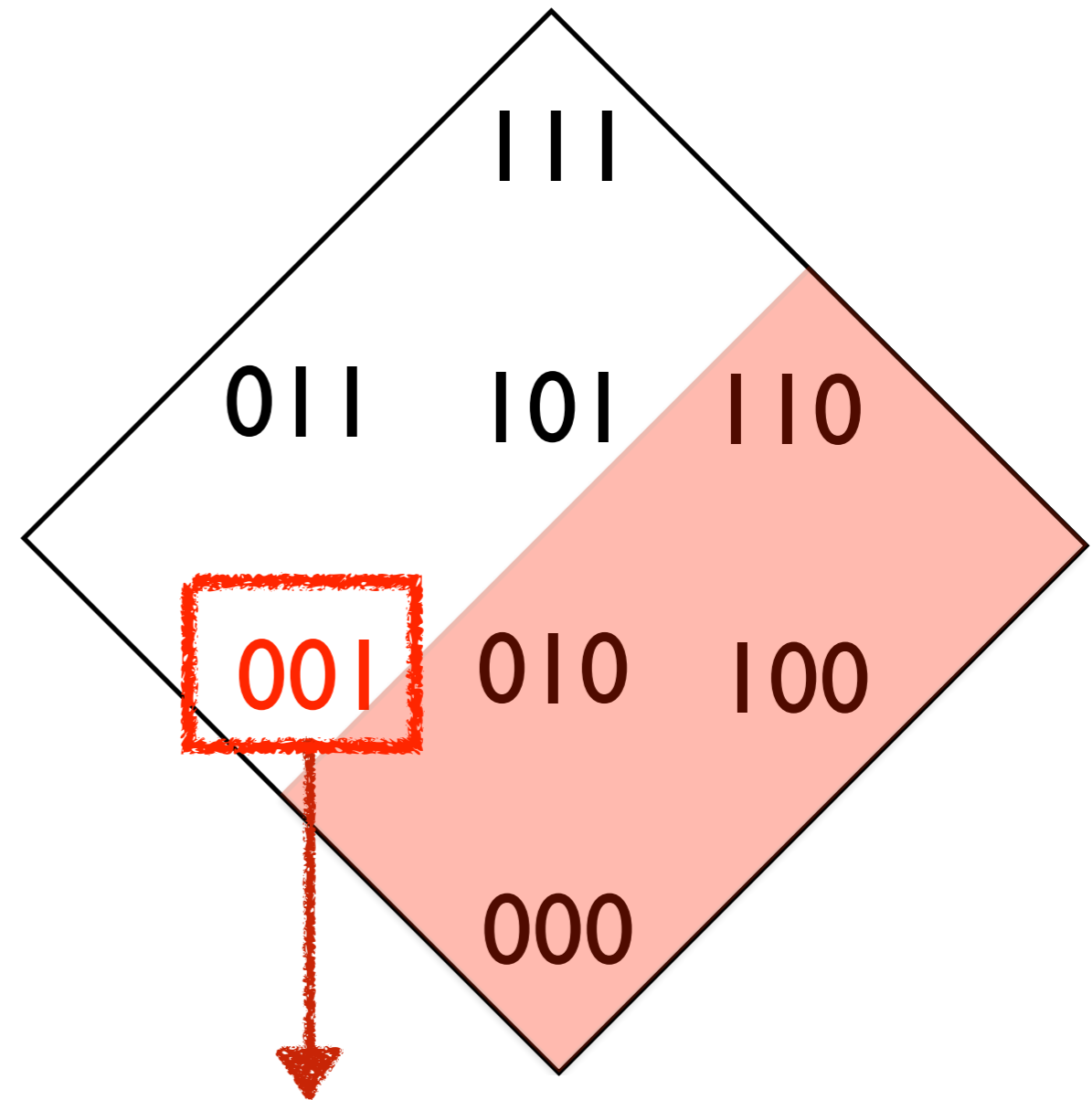
Testing and pruning

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




Choose a minimal abs.

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:  y = y-1;
10: 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;
9:  y = y-1;          [n:1,x:1,y:1]
10:  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;
9:  y = y-1;
10: assert(y >= 0);
```



[n:1,x:1,y:1(dec)]

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:  y = y-1;          [n:1,x:1,y:1(dec)]
10: assert(y >= 0);
```



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

Because dec results in T in $S_0=\{+, T\}$.

Testing and pruning

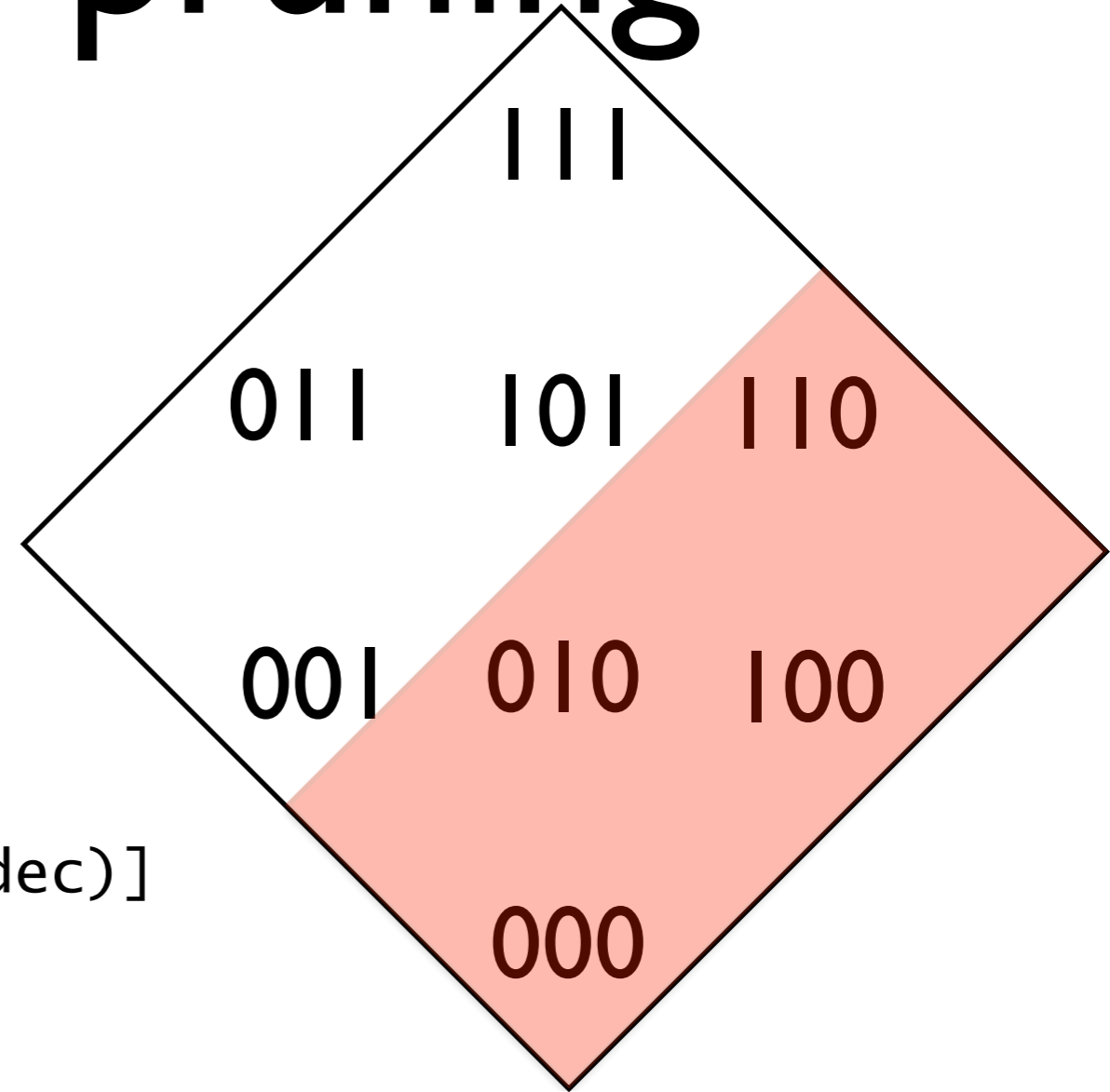
```

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:  y = y-1;
10: assert(y >= 0);

```

↓

[n:1,x:1,y:1(dec)]



[n:_, x:_, y:⊤] if abs(y)=0.

Because dec results in ⊤ in $S_0 = \{+, \top\}$.

Testing and pruning

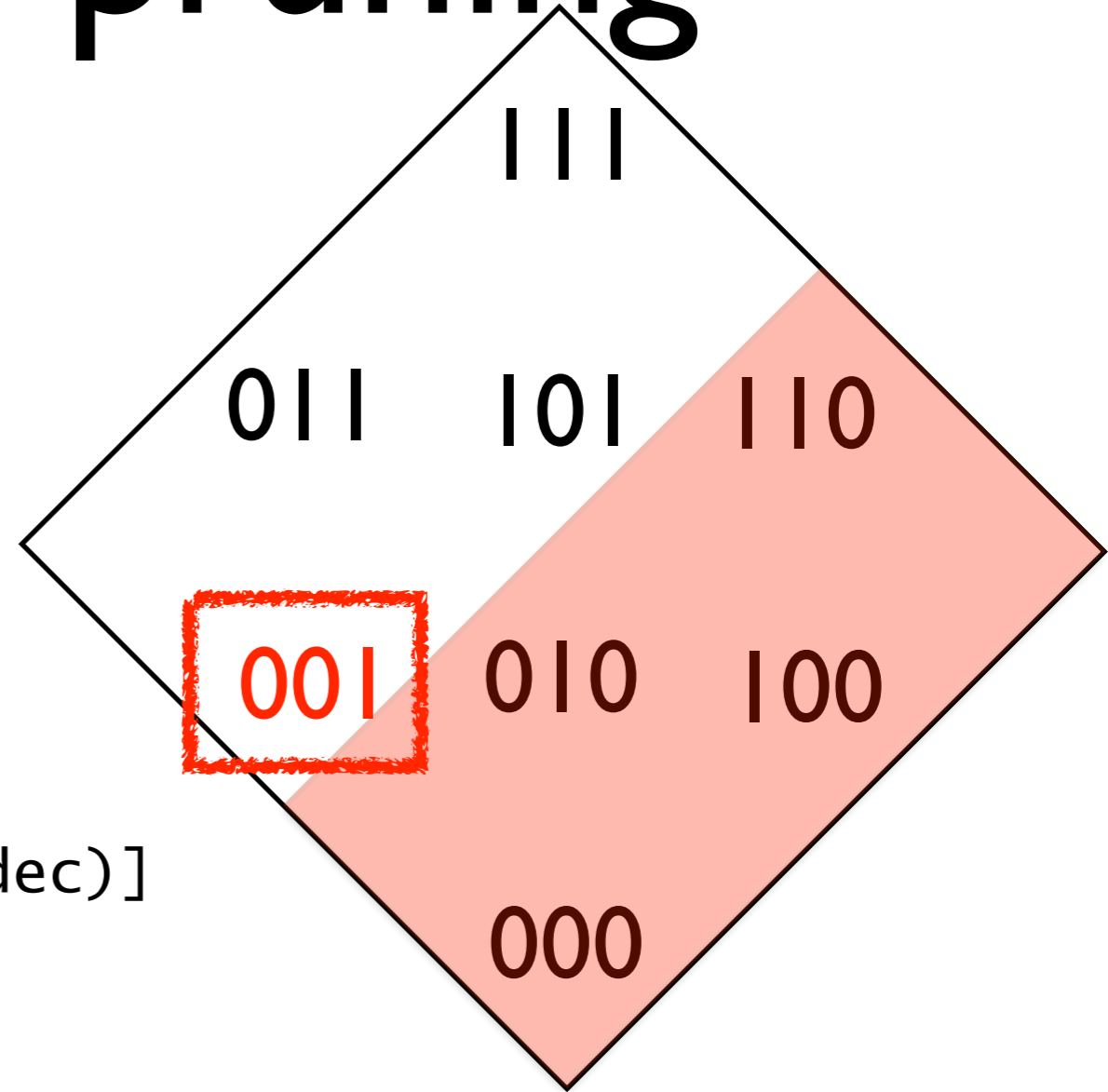
```
1:  assert(n >= 1);  [n:1,x:0,y:0]
2:  x = 1;
3:  y = 1;
4:  while (n > 1) {
5:      (x,y) = (y,x+y);
6:      n = n-1;
7:  }
8:  y = y+1;
9:  y = y-1;
10: assert(y >= 0);
```

↓

[n:1,x:1,y:1(dec)]

[n:_, x:_, y:⊤] if $\text{abs}(y)=0$.

Because dec results in ⊤ in $S_0=\{+, \top\}$.



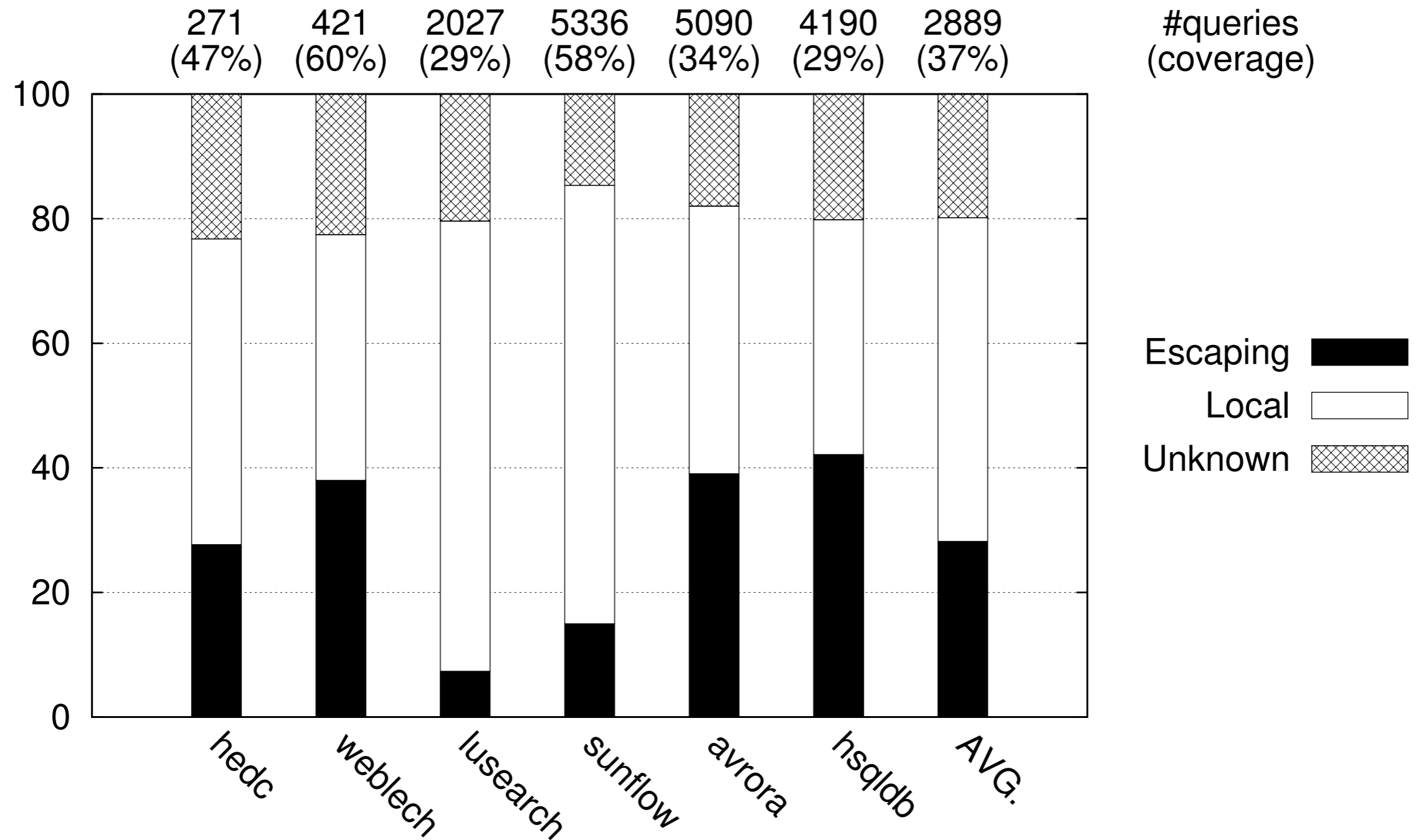


Figure 3. Precision results for our thread-escape analysis.

[POPL'12]

Pruning based on refinement

Limitation of testing

```
1: assert(n >= 1);  
2: x = 1; y = 1;  
3: while (n > 1) {  
4:     (x,y) = (y,x+y);  
5:     n = n-1;  
6: }  
7: x = x-1;  
8: if (y == 832040)  
9:     assert(x >= 0)
```

Limitation of testing

```
1: assert(n >= 1);  
2: x = 1; y = 1;  
3: while (n > 1) {  
4:     (x,y) = (y,x+y);  
5:     n = n-1;  
6: }  
7: x = x-1;  
8: if (y == 832040)  
9:     assert(x >= 0)
```



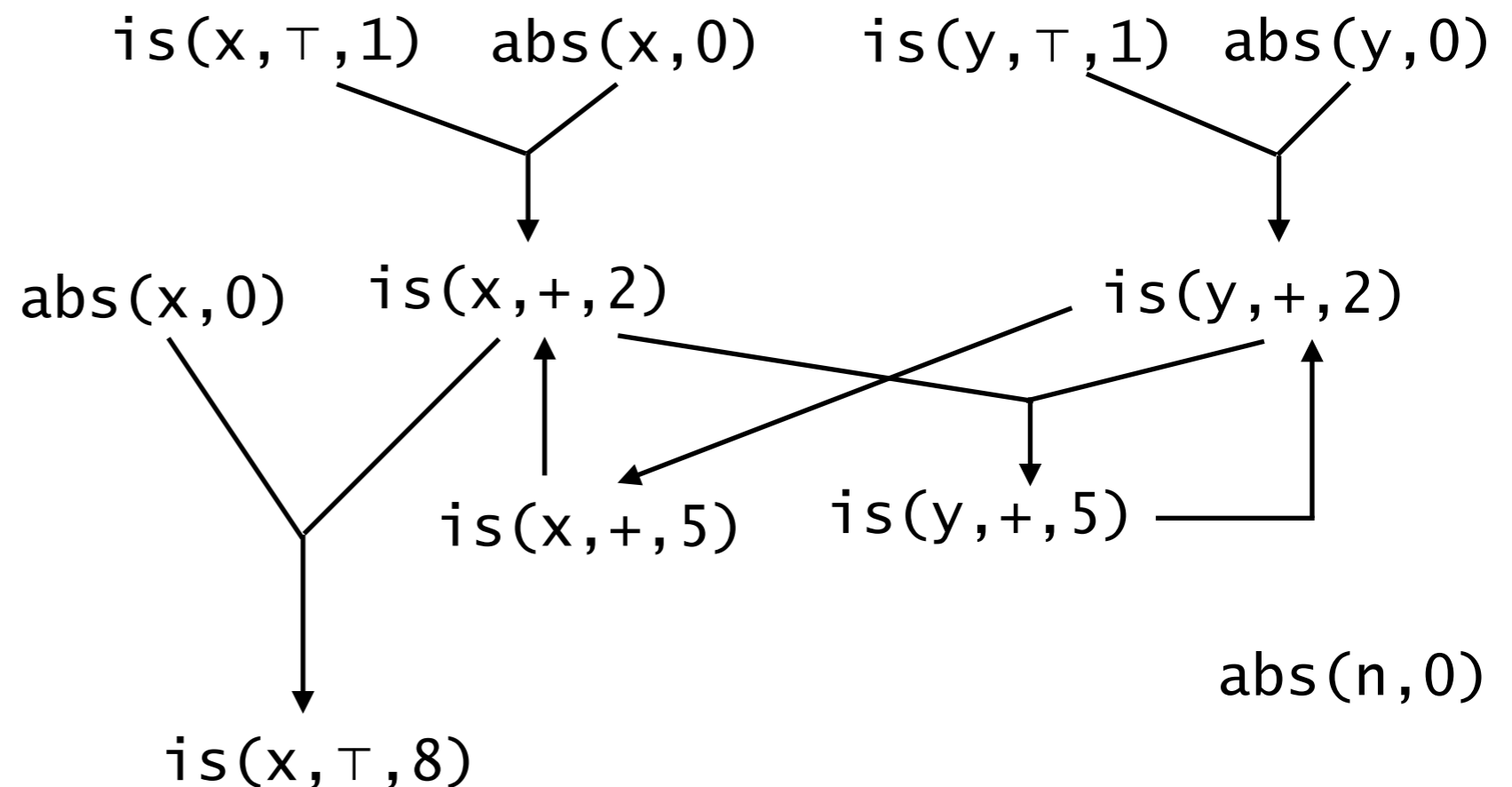
Reaching assert(...) by testing is not easy.

Iterative refinement

- Run a verifier with a cheap abstraction.
- Prune all abstractions that lead to similar verification failures.

Result with [n:0,x:0,y:0]

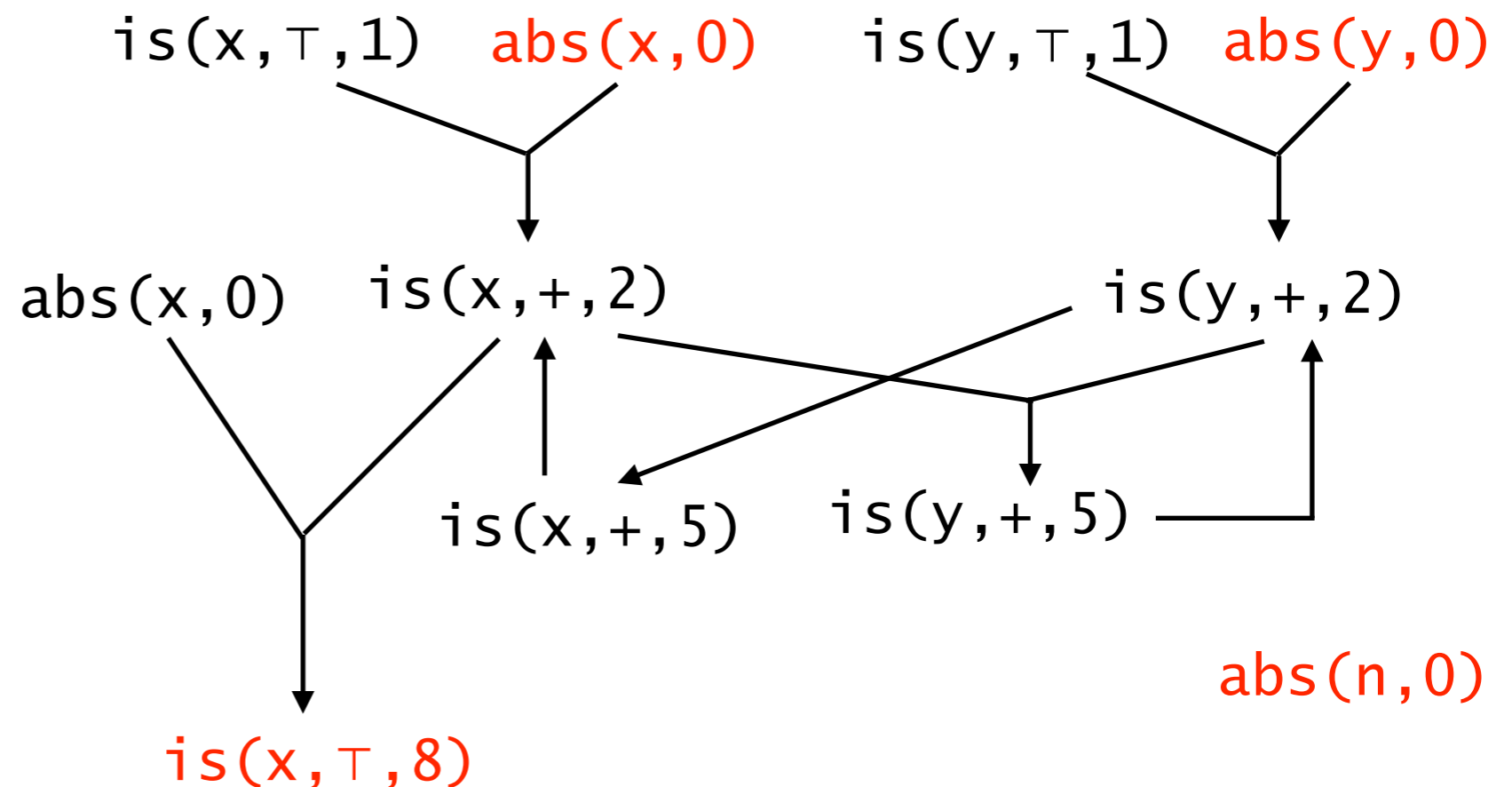
```
1: assert(n >= 1);  
2: x = 1; y = 1;  
3: while (n > 1) {  
4:   (x,y) = (y,x+y);  
5:   n = n-1;  
6: }  
7: x = x-1;  
8: if (y == 832040)  
9:   assert(x >= 0)
```



Result with $[n:0, x:0, y:0]$

[Goal] Destroy all derivations of $is(x, \top, 8)$.

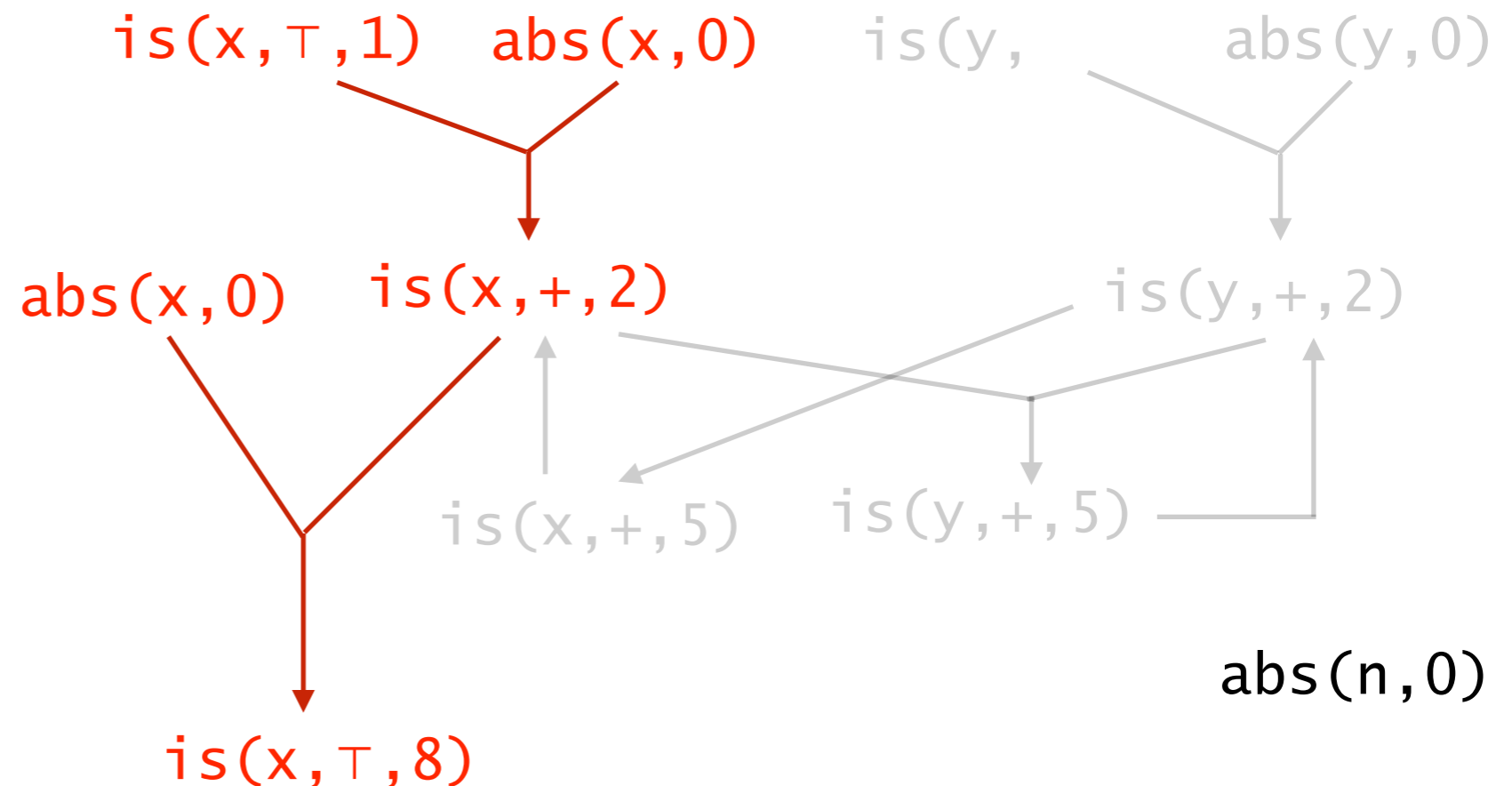
```
1: assert(n >= 1);  
2: x = 1; y = 1;  
3: while (n > 1) {  
4:   (x,y) = (y,x+y);  
5:   n = n-1;  
6: }  
7: x = x-1;  
8: if (y == 832040)  
9:   assert(x >= 0)
```



Result with $[n:0, x:0, y:0]$

[Goal] Destroy all derivations of $is(x, \top, 8)$.

```
1: assert(n >= 1);  
2: x = 1; y = 1;  
3: while (n > 1) {  
4:   (x,y) = (y,x+y);  
5:   n = n-1;  
6: }  
7: x = x-1;  
8: if (y == 832040)  
9:   assert(x >= 0)
```

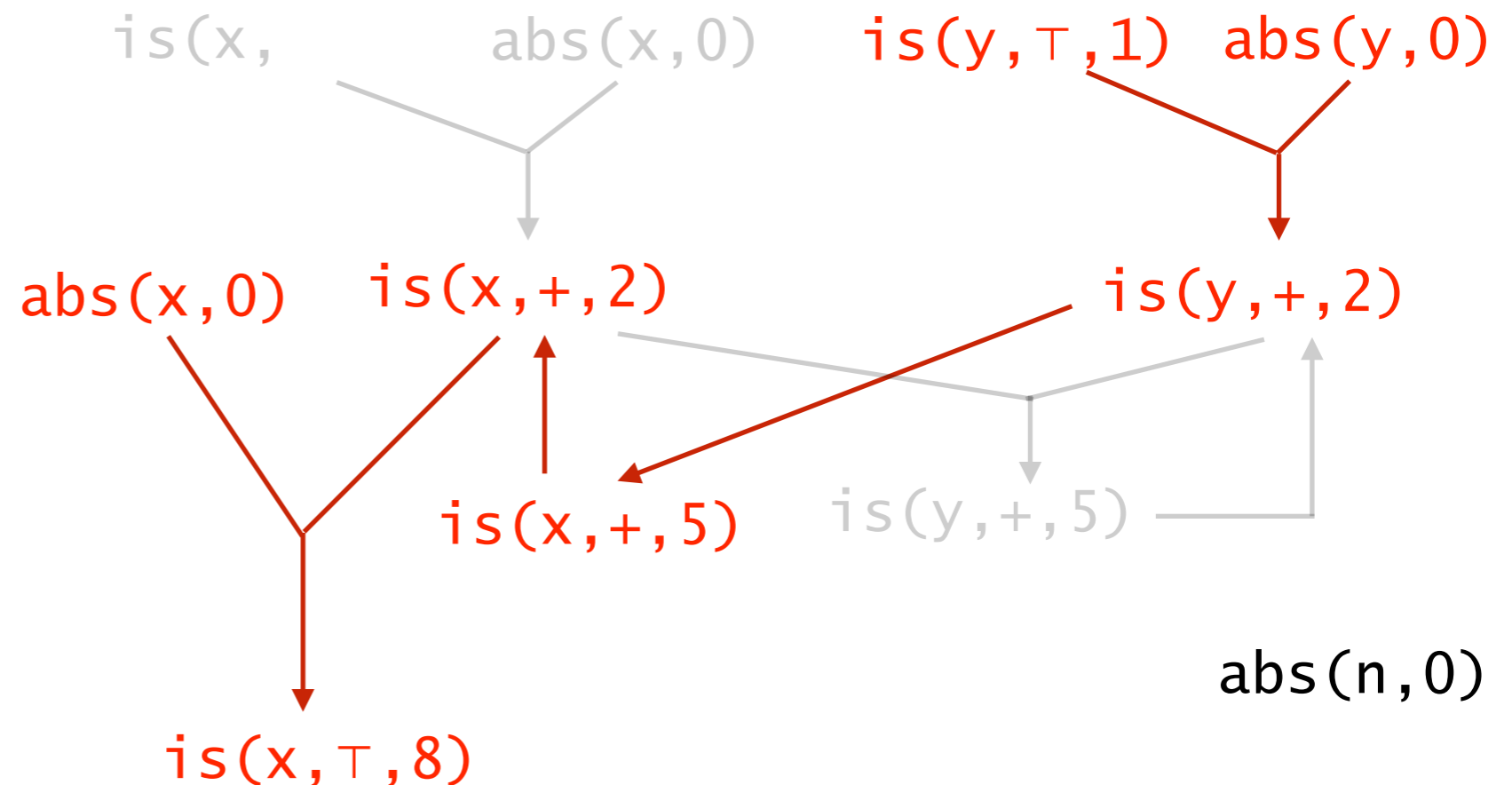


If $abs(x, 0)$, we cannot prove the query.

Result with [n:0,x:0,y:0]

[Goal] Destroy all derivations of $\text{is}(x, \top, 8)$.

```
1: assert(n >= 1);  
2: x = 1; y = 1;  
3: while (n > 1) {  
4:   (x,y) = (y,x+y);  
5:   n = n-1;  
6: }  
7: x = x-1;  
8: if (y == 832040)  
9:   assert(x >= 0)
```



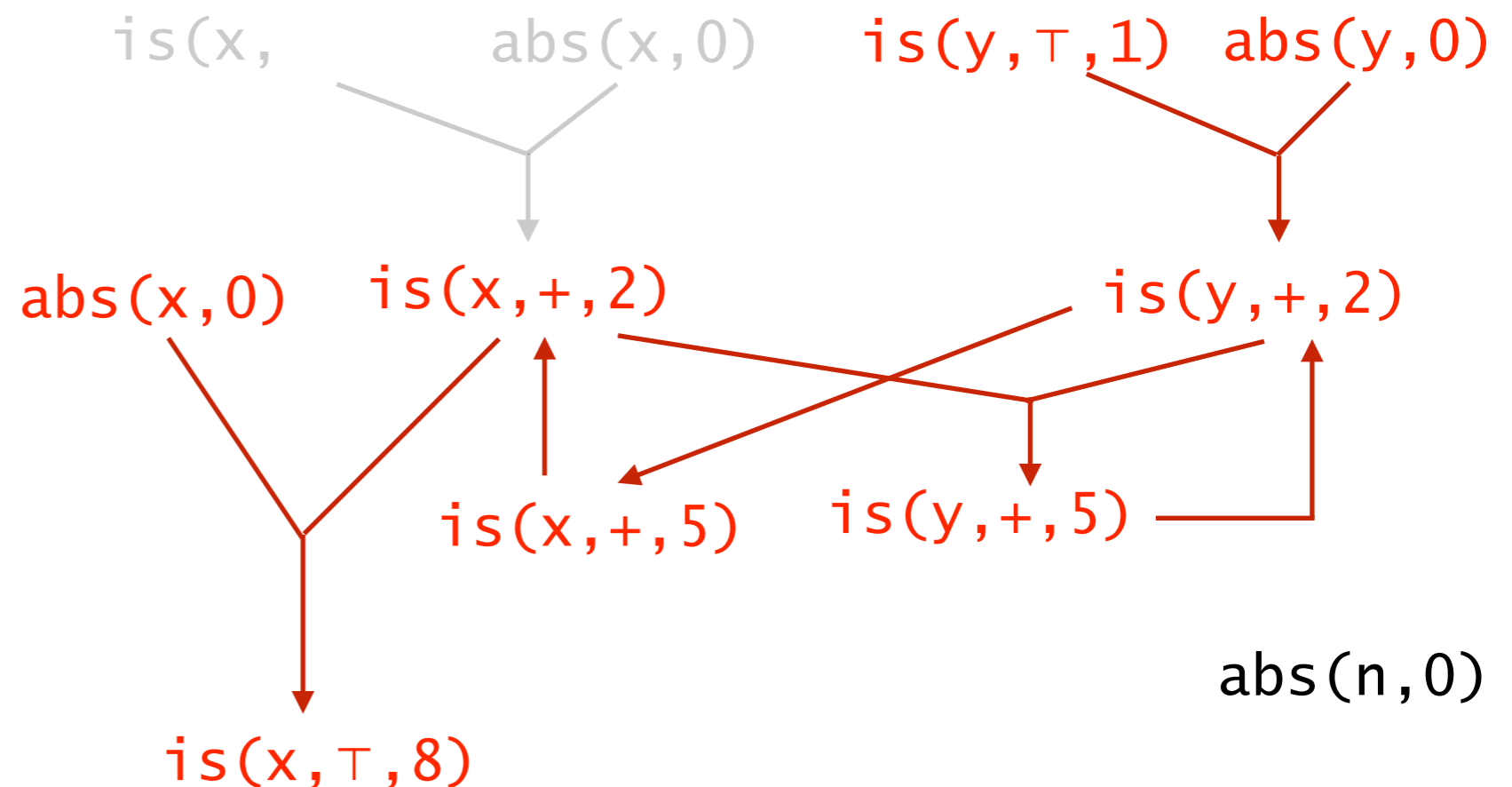
If $\text{abs}(x, 0)$, we cannot prove the query.

If $\text{abs}(x, 0)$ or $\text{abs}(y, 0)$, we cannot prove the query.

Result with $[n:0, x:0, y:0]$

[Goal] Destroy all derivations of $is(x, \top, 8)$.

```
1: assert(n >= 1);  
2: x = 1; y = 1;  
3: while (n > 1) {  
4:   (x, y) = (y, x+y);  
5:   n = n-1;  
6: }  
7: x = x-1;  
8: if (y == 832040)  
9:   assert(x >= 0)
```



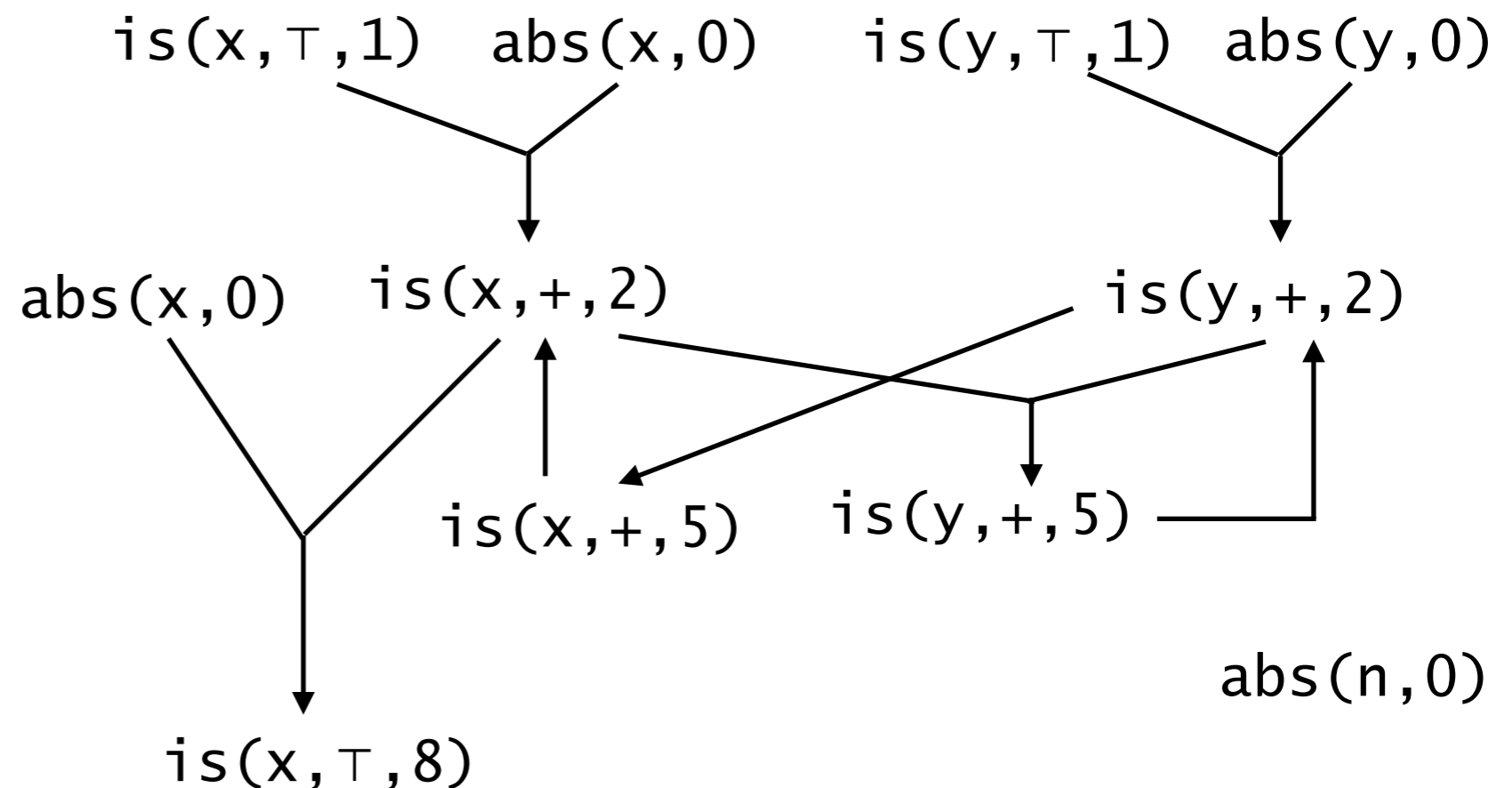
If $abs(x, 0)$, we cannot prove the query.

If $abs(x, 0)$ or $abs(y, 0)$, we cannot prove the query.

Result with $[n:0, x:0, y:0]$

[Goal] Destroy all derivations of $is(x, \top, 8)$.

```
1: assert(n >= 1);  
2: x = 1; y = 1;  
3: while (n > 1) {  
4:   (x,y) = (y,x+y);  
5:   n = n-1;  
6: }  
7: x = x-1;  
8: if (y == 832040)  
9:   assert(x >= 0)
```



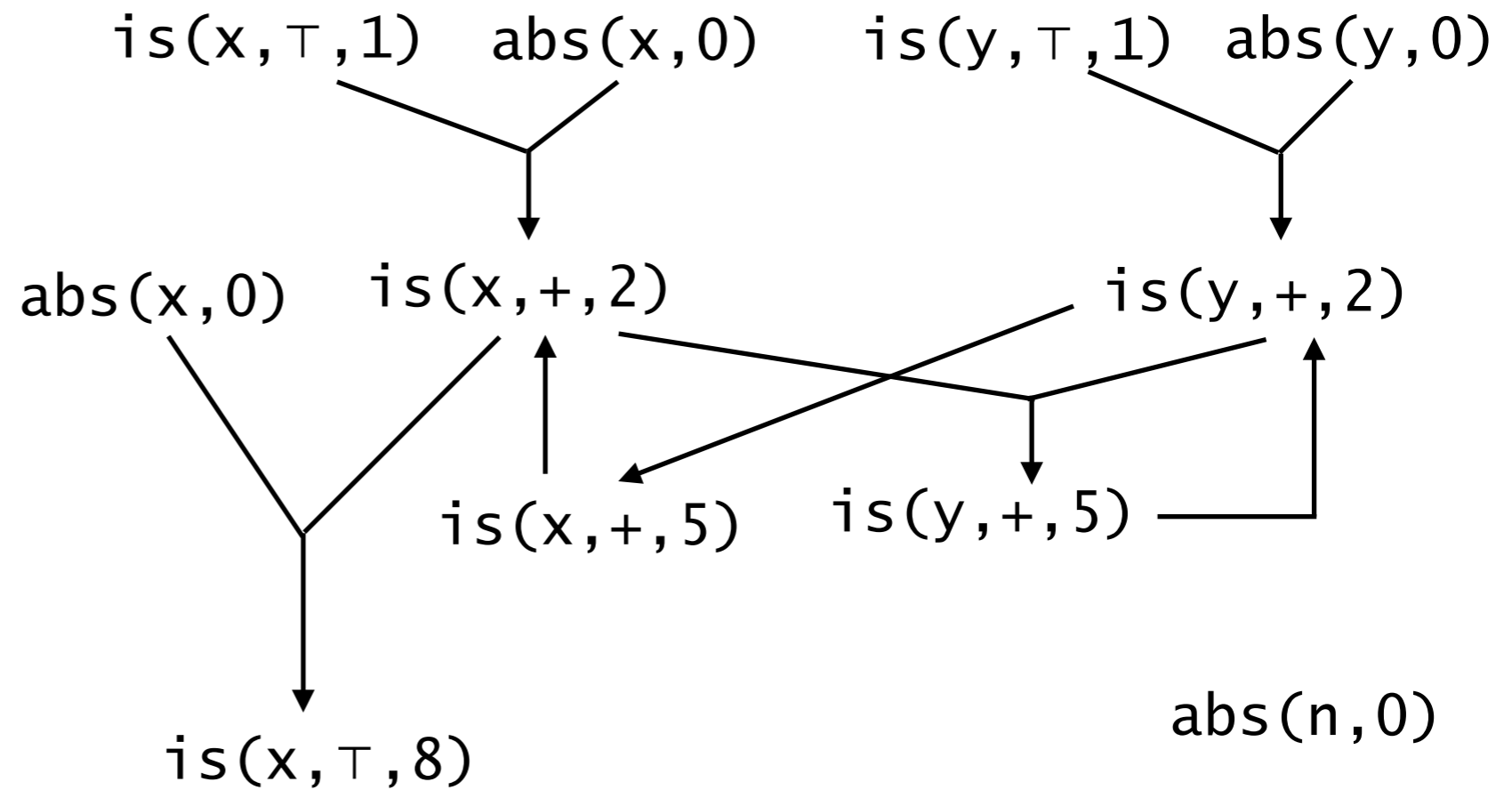
If $abs(x, 0)$, we cannot prove the query.

If $abs(x, 0)$ or $abs(y, 0)$, we cannot prove the query.

```

1: assert(n >= 1);
2: x = 1; y = 1;
3: while (n > 1) {
4:   (x,y) = (y,x+y);
5:   n = n-1;
6: }
7: x = x-1;
8: if (y == 832040)
9:   assert(x >= 0)

```



MaxSat Encoding:

Hard clauses:

$(is(x, T, 1) \ \& \ abs(x, 0) \Rightarrow is(x, +, 2))$
 $\& \ (abs(x, 0) \ \& \ is(x, +, 2) \Rightarrow is(x, T, 8))$
 $\& \ \dots$
 $\& \ (not \ is(x, T, 8))$

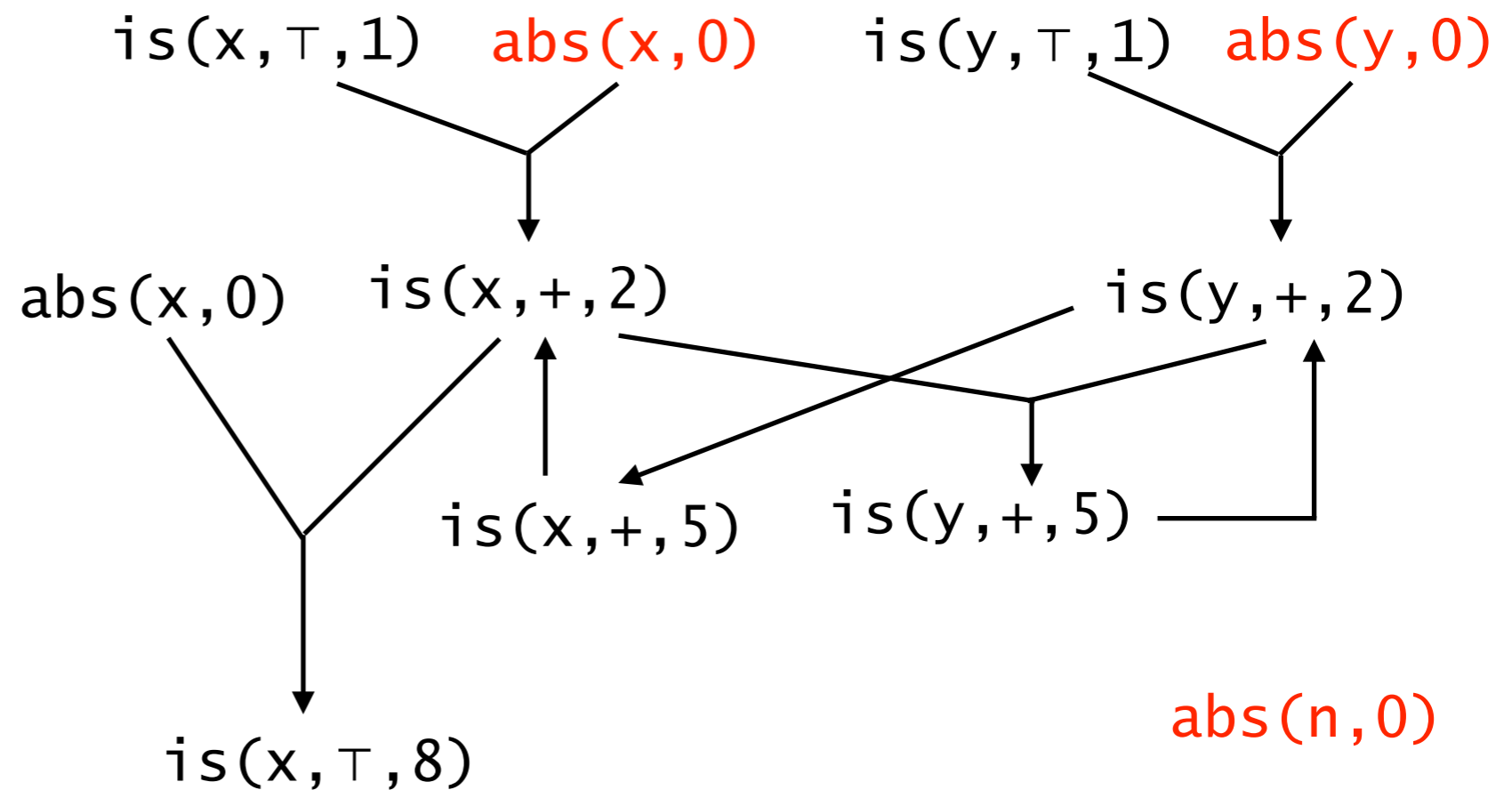
Soft clauses:

$abs(n, 0)$
 $\& \ abs(x, 0)$
 $\& \ abs(y, 0)$

```

1: assert(n >= 1);
2: x = 1; y = 1;
3: while (n > 1) {
4:   (x,y) = (y,x+y);
5:   n = n-1;
6: }
7: x = x-1;
8: if (y == 832040)
9:   assert(x >= 0)

```



MaxSat Encoding:

Hard clauses:

$(is(x, \top, 1) \ \& \ abs(x, 0) \Rightarrow is(x, +, 2))$
 $\& \ (abs(x, 0) \ \& \ is(x, +, 2) \Rightarrow is(x, \top, 8))$
 $\& \ \dots$
 $\& \ (not \ is(x, \top, 8))$

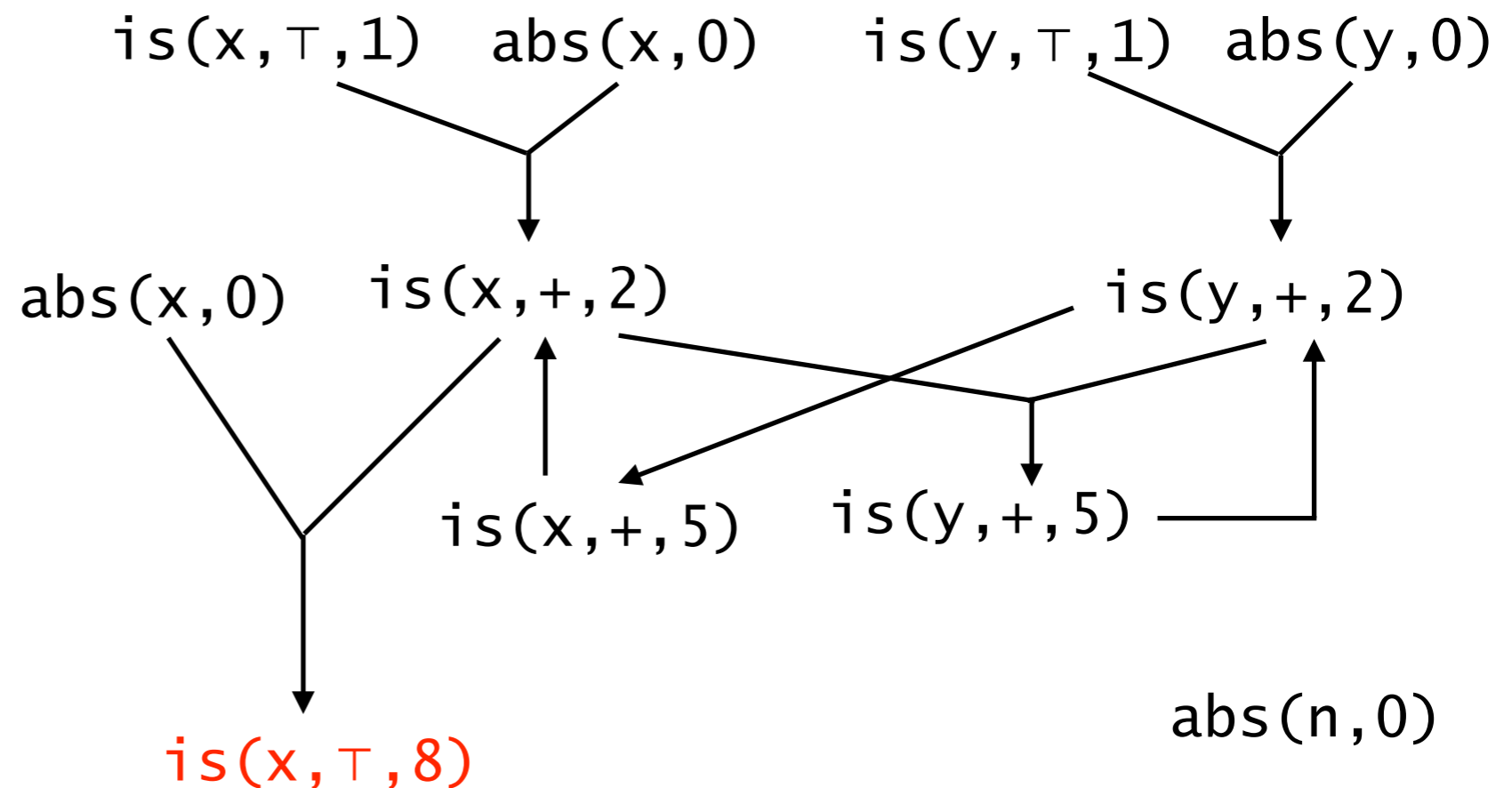
Soft clauses:

$abs(n, 0)$
 $\& \ abs(x, 0)$
 $\& \ abs(y, 0)$

```

1: assert(n >= 1);
2: x = 1; y = 1;
3: while (n > 1) {
4:     (x,y) = (y,x+y);
5:     n = n-1;
6: }
7: x = x-1;
8: if (y == 832040)
9:     assert(x >= 0)

```



MaxSat Encoding:

Hard clauses:

$(is(x, \tau, 1) \ \& \ abs(x, 0) \Rightarrow is(x, +, 2))$
 $\& \ (abs(x, 0) \ \& \ is(x, +, 2) \Rightarrow is(x, \tau, 8))$
 $\& \ \dots$
 $\& \ (not \ is(x, \tau, 8))$

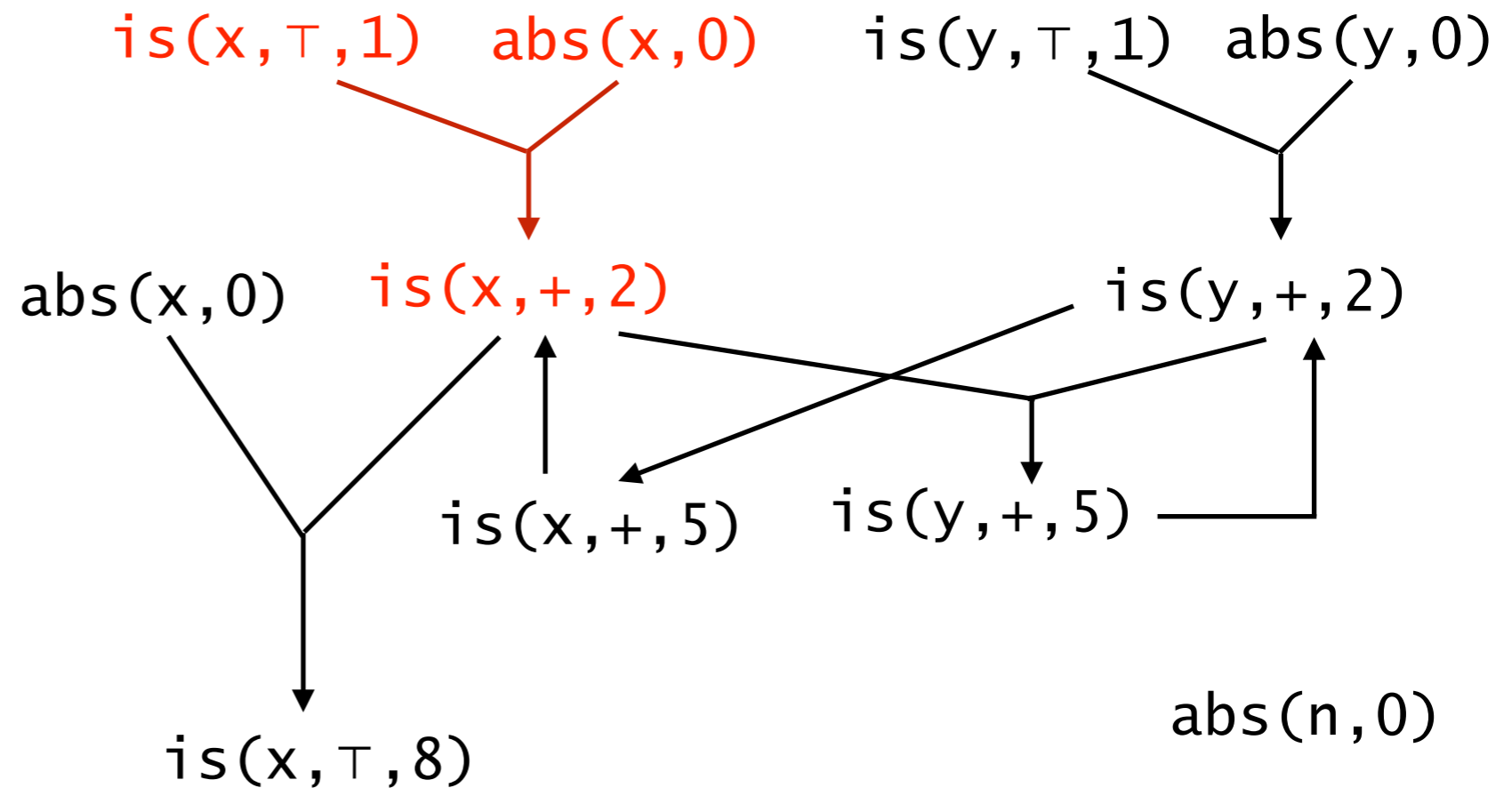
Soft clauses:

$abs(n, 0)$
 $\& \ abs(x, 0)$
 $\& \ abs(y, 0)$

```

1: assert(n >= 1);
2: x = 1; y = 1;
3: while (n > 1) {
4:   (x,y) = (y,x+y);
5:   n = n-1;
6: }
7: x = x-1;
8: if (y == 832040)
9:   assert(x >= 0)

```



MaxSat Encoding:

Hard clauses:

$(is(x, \top, 1) \ \& \ abs(x, 0) \Rightarrow is(x, +, 2))$
 $\& \ (abs(x, 0) \ \& \ is(x, +, 2) \Rightarrow is(x, \top, 8))$
 $\& \ \dots$
 $\& \ (\text{not } is(x, \top, 8))$

Soft clauses:

$abs(n, 0)$
 $\& \ abs(x, 0)$
 $\& \ abs(y, 0)$

Full story

- The process is repeated until we prune the whole search space or prove the query.
- Implemented in the context of program analyses (or verifiers) written in Datalog.
- See PLDI'14a for details.

	pointer analysis						typestate analysis					
	queries			abstraction size		iterations	queries		abstraction size		iterations	
	total	resolved		final	max.		total	resolved	final	max.	iterations	
		CURRENT	BASELINE								CURRENT	BASELINE
toba-s	7	7	0	17	1,782	10	543	543	62	14,781	15	159
javasrc-p	46	46	0	47	1,845	13	159	159	89	13,653	14	92
weblech	5	5	2	14	3,095	10	13	13	33	25,781	14	16
hedc	47	47	6	73	2,948	18	24	24	14	23,622	7	10
antlr	143	143	5	97	2,917	15	77	77	66	24,815	12	45
luindex	138	138	67	116	4,055	26	248	248	79	33,835	16	72
lusearch	322	322	29	146	3,936	17	45	45	74	33,526	13	52
schroeder-m	51	51	25	45	5,826	15	194	194	71	54,741	9	49

Table 3: Results showing statistics of queries, abstractions, and iterations of our approach (**CURRENT**) and the baseline approaches (**BASELINE**).

	running time of the Datalog solver (in seconds)							running time of the MAXSAT solver (in seconds)					
	pointer analysis				typestate analysis			pointer analysis			typestate analysis		
	BASELINE	min.	max.	avg.	min.	max.	avg.	min.	max.	avg.	min.	max.	avg.
toba-s	11	5	7	6	49	82	68.1	2	7	3.1	1	6	3.1
javasrc-p	29	7	11	9	76	152	120.8	<1	4	1.6	2	19	6.4
weblech	2,574	44	54	47.5	121	172	146.6	5	11	6.7	3	8	5.3
hedc	5,058	21	37	27.9	52	58	54.3	1	23	3.7	1	2	1.7
antlr	3,723	30	55	39.3	193	325	264.8	11	44	24.1	5	27	13.25
luindex	913	59	84	76.4	311	512	426.7	8	48	16.3	6	26	14.7
lusearch	7,040	59	85	72.7	238	437	343.9	7	62	23.9	6	29	15.9
schroeder-m	23,038	192	428	289.6	1,778	2,681	2,304.6	34	257	114	37	308	138.6

Table 4: Running time of the Datalog and MAXSAT solvers in each iteration.

	pointer analysis		typestate analysis	
	# variables	# clauses	# variables	# clauses
toba-s	784k	1,485k	741k	938k
javasrc-p	470k	877k	1,022k	1,333k
weblech	1,620k	3,307k	1,374k	1,807k
hedc	1,245k	2,664k	606k	751k
antlr	3,621k	6,875k	2,318k	3,009k
luindex	2,406k	5,643k	2,829k	3,784k
lusearch	2,103k	5,011k	2,626k	3,524k
schroeder-m	6,706k	23,680k	16,293k	22,257k

Table 5: Statistics of MAXSAT formula in the final iteration.

How to find a good program abstraction automatically?

- Formulate it as a search problem.
- Develop a good pruning strategy.
- Predict based on the knowledge of a verifier.