머신러닝 기반 선별적 프로그램 분석
Machine Learning-Guided Adaptive Program Analysis

오학주
고려대학교 정보대학 컴퓨터학과

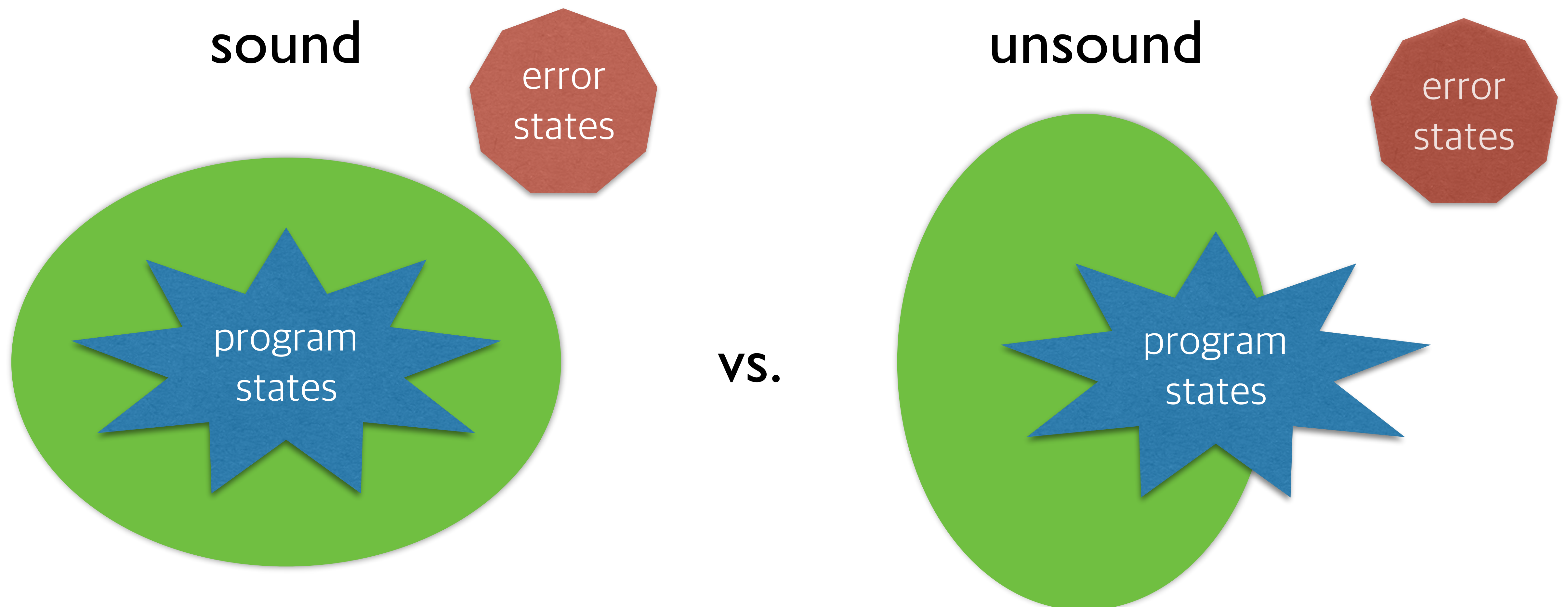
(with 허기홍, 채권수, 차수영, 정세훈, 홍성준, 양홍석, 이광근)

Aug. 19, 2016 @SIGPL Summer School

Static Program Analysis

- Predict program behavior statically and automatically
  - **static**: before execution, at compile-time
  - **automatic**: sw is analyzed by sw (“static analyzers”)
- Applications
  - **bug-finding**. e.g., find runtime failures of programs
  - **security**. e.g., is this app malicious or benign?
  - **verification**. e.g., does the program meet its specification?
  - **compiler optimization**, e.g., automatic parallelization

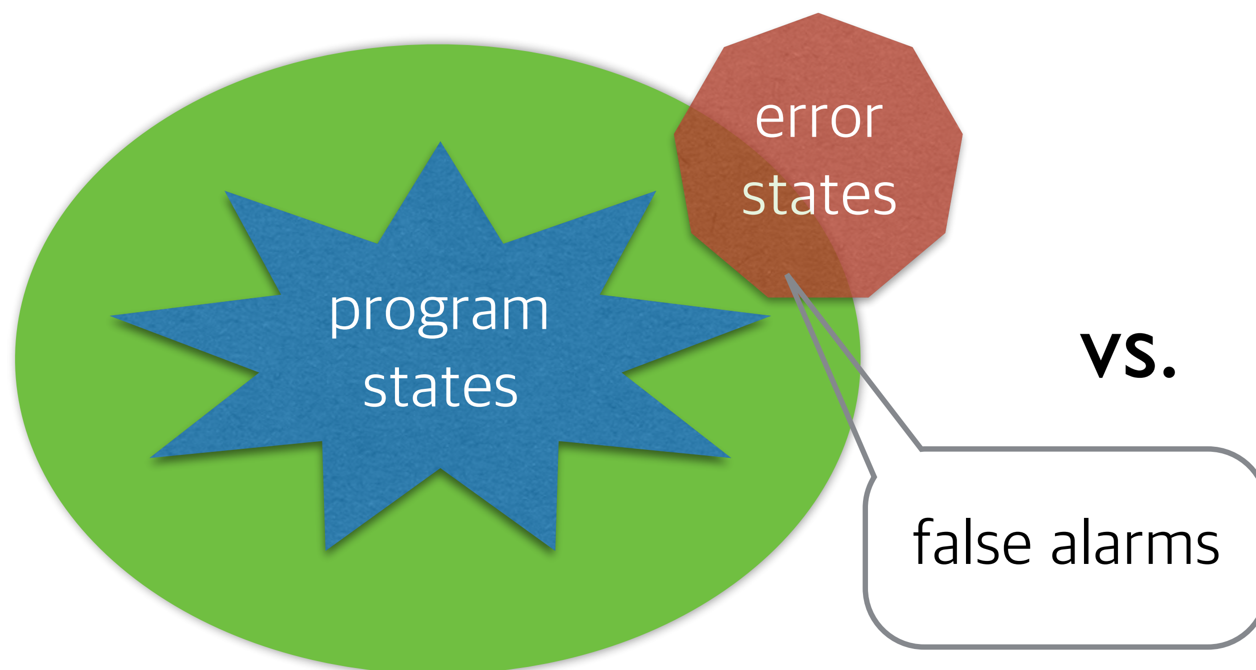
Principle of Program Analysis



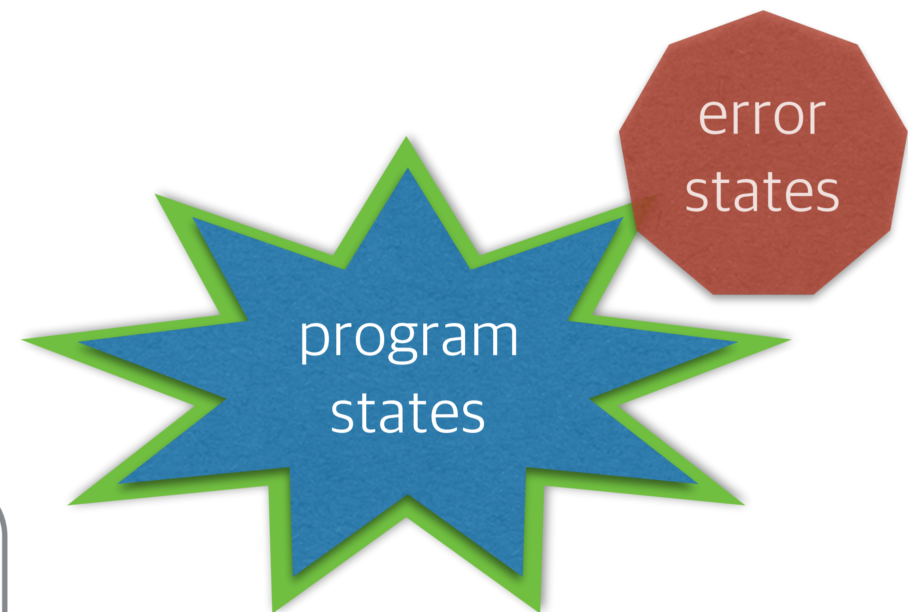
Principle of Program Analysis

imprecise

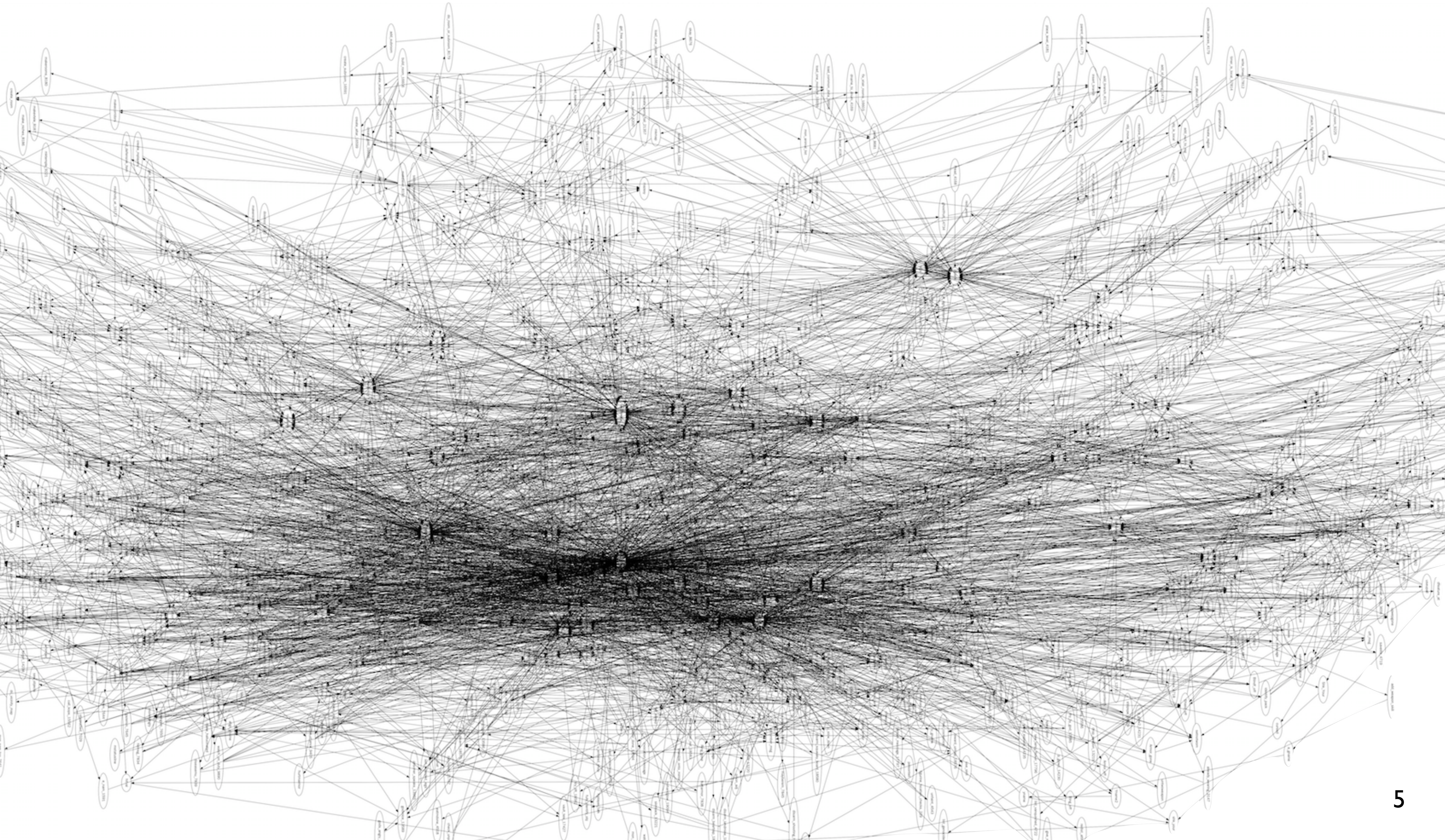
precise



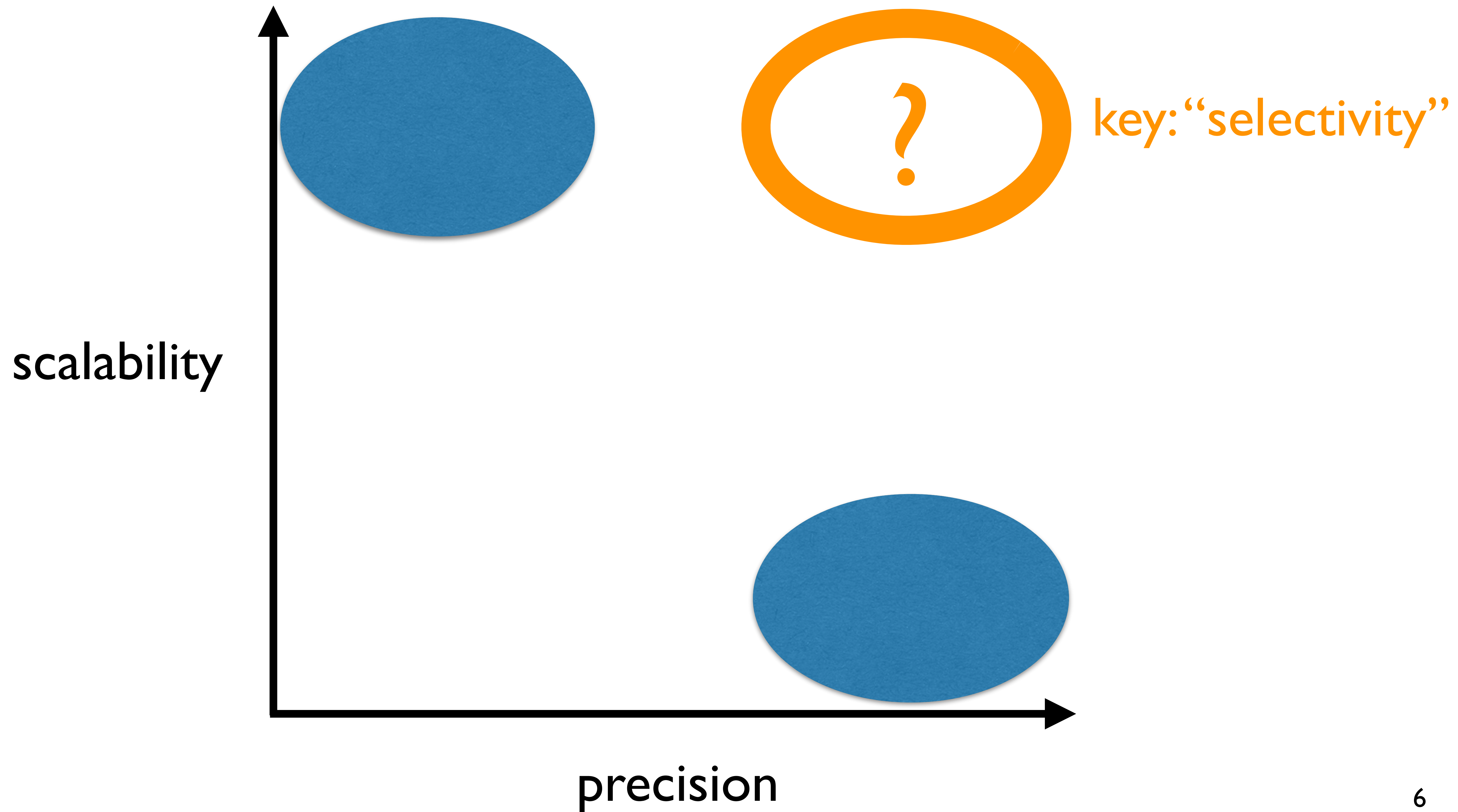
vs.



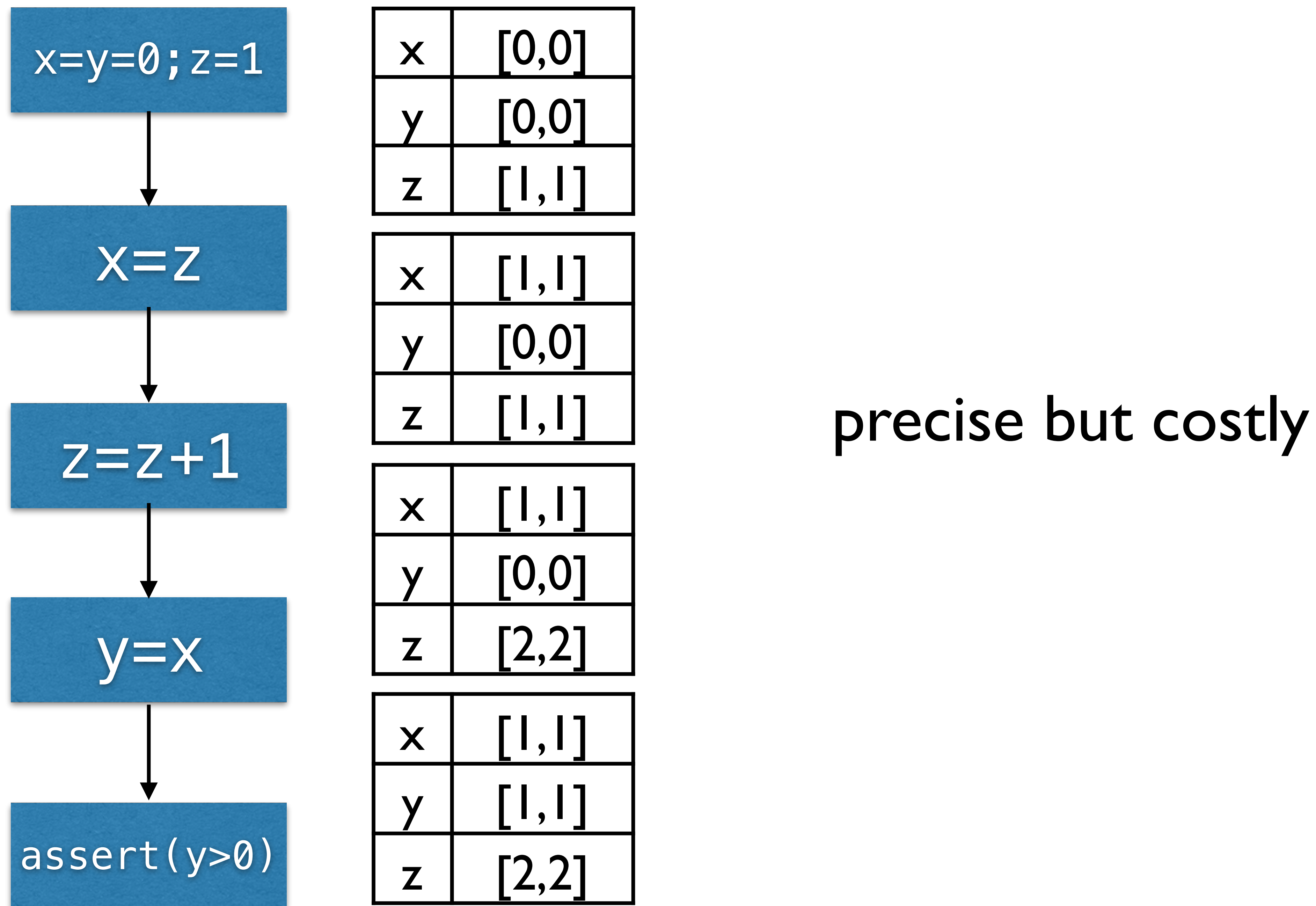
Principle of Program Analysis



Challenge in Static Analysis

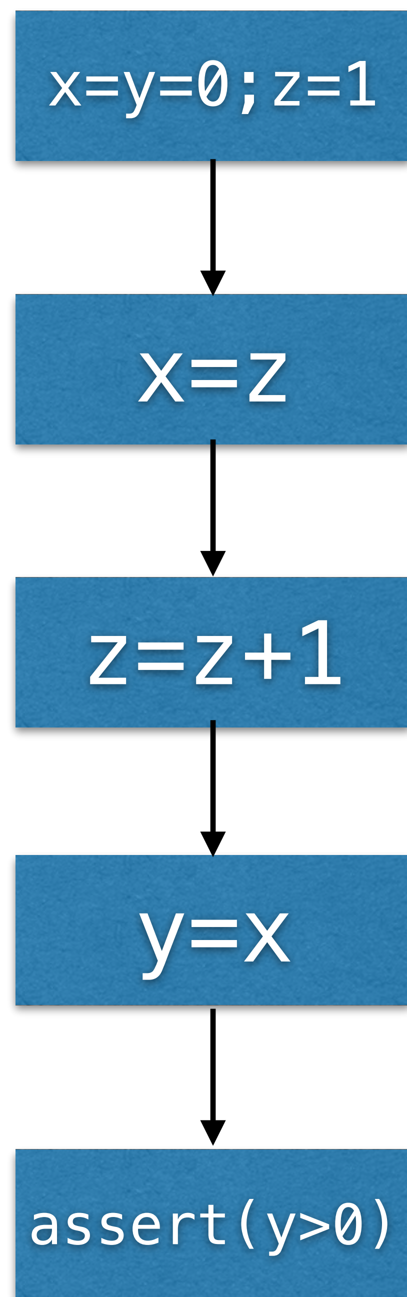


Flow-Sensitivity



precise but costly

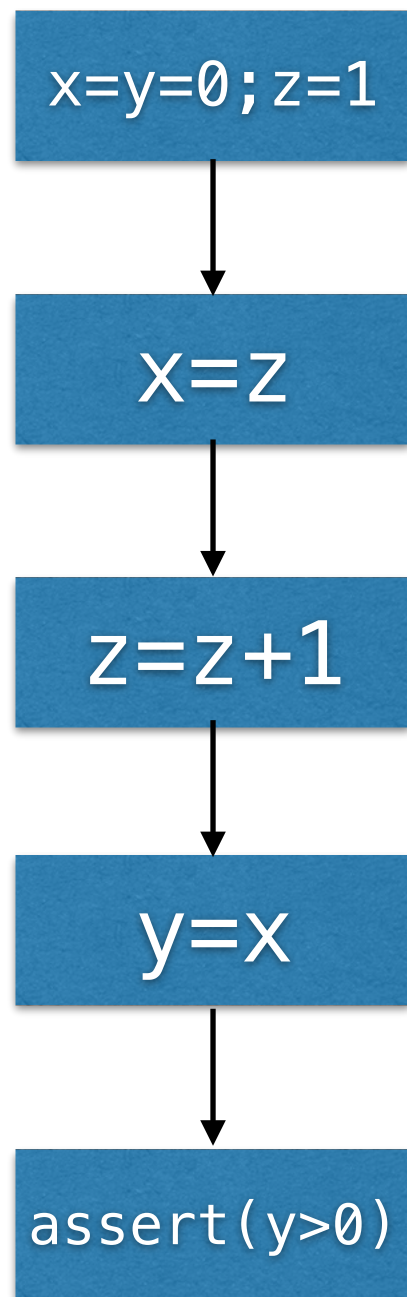
Flow-Insensitivity



x	$[0,+\infty]$
y	$[0,+\infty]$
z	$[1,+\infty]$

cheap but imprecise

Selective Flow-Sensitivity



FS : {x,y}

x	[0,0]
y	[0,0]

x	[1,+∞]
y	[0,0]

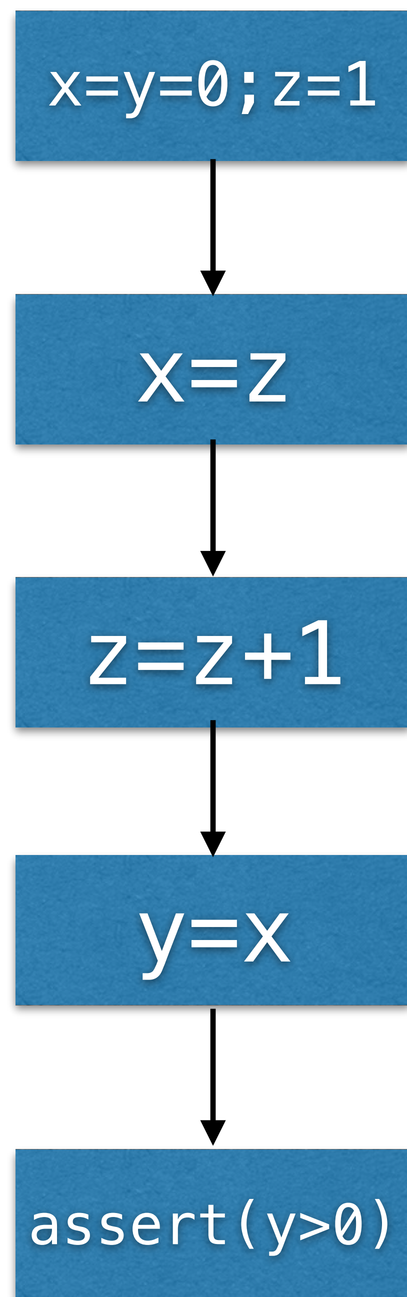
x	[1,+∞]
y	[0,0]

x	[1,+∞]
y	[1,+∞]

FI : {z}

z	[1,+∞]

Selective Flow-Sensitivity



FS : {y,z}

y	[0,0]
z	[1,1]

y	[0,0]
z	[1,1]

y	[0,0]
z	[2,2]

y	$[0,+\infty]$
z	[2,2]

fail to prove

FI : {x}

x	$[0,+\infty]$

Context-Sensitivity

```
    int h(n) {ret n;}

    void f(a) {
c1:   x = h(a);
      assert(x > 1);  // Q1
c2:   y = h(input());
      assert(y > 1);  // Q2
    }

c3: void g() {f(8);}

    void m() {
c4:   f(4);
c5:   g();
c6:   g();
    }
```

Q1: always holds

Q2: does not always hold

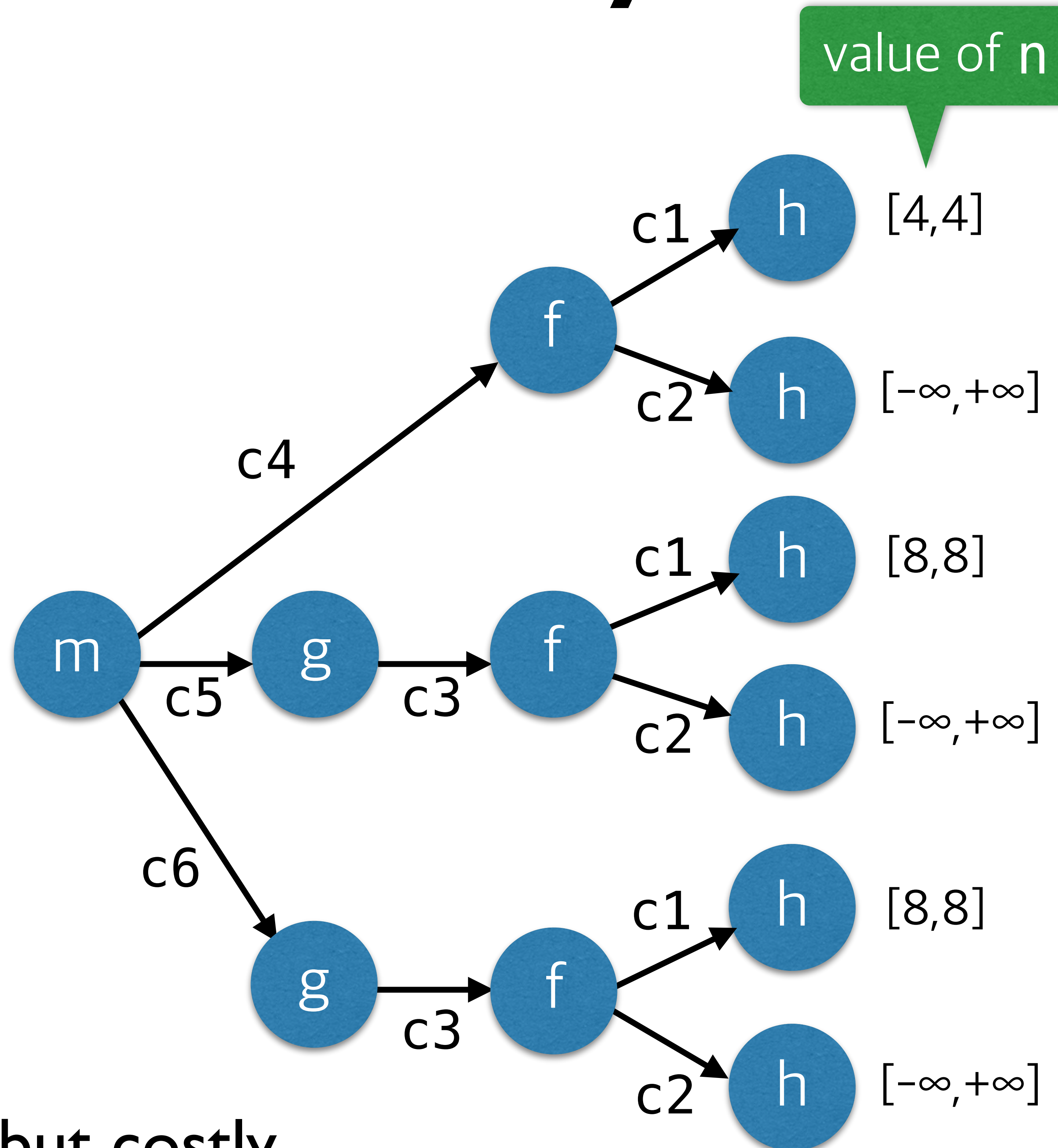
Context-Sensitivity

```
    int h(n) {ret n;}

    void f(a) {
c1:   x = h(a);
      assert(x > 1);  // Q1
c2:   y = h(input());
      assert(y > 1);  // Q2
    }

c3: void g() {f(8);}

    void m() {
c4:   f(4);
c5:   g();
c6:   g();
    }
```



precise but costly

Context-Insensitivity

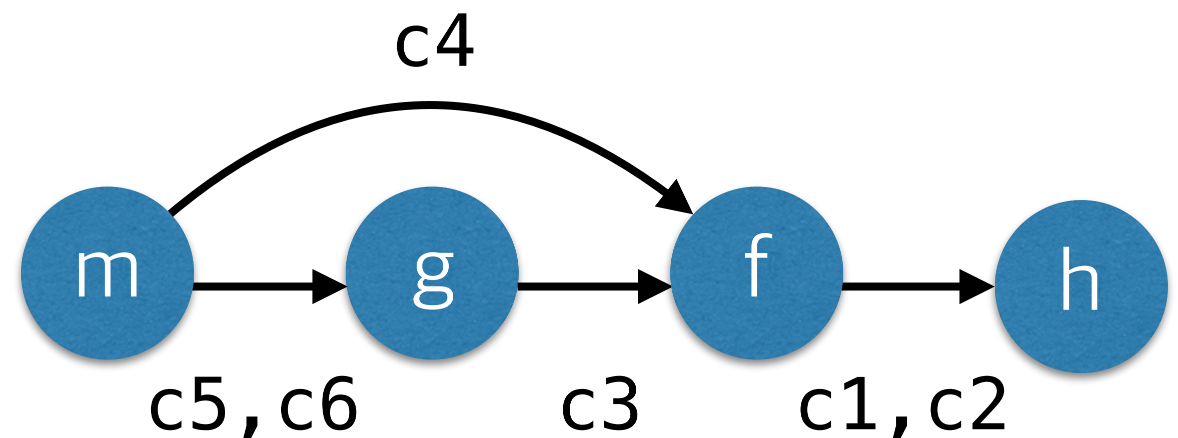
```
    int h(n) {ret n;}

    void f(a) {
c1:   x = h(a);
      assert(x > 1);   // Q1
c2:   y = h(input());
      assert(y > 1);   // Q2
    }

c3: void g() {f(8);}

    void m() {
c4:   f(4);
c5:   g();
c6:   g();
    }
```

[-∞,+∞]



cheap but imprecise

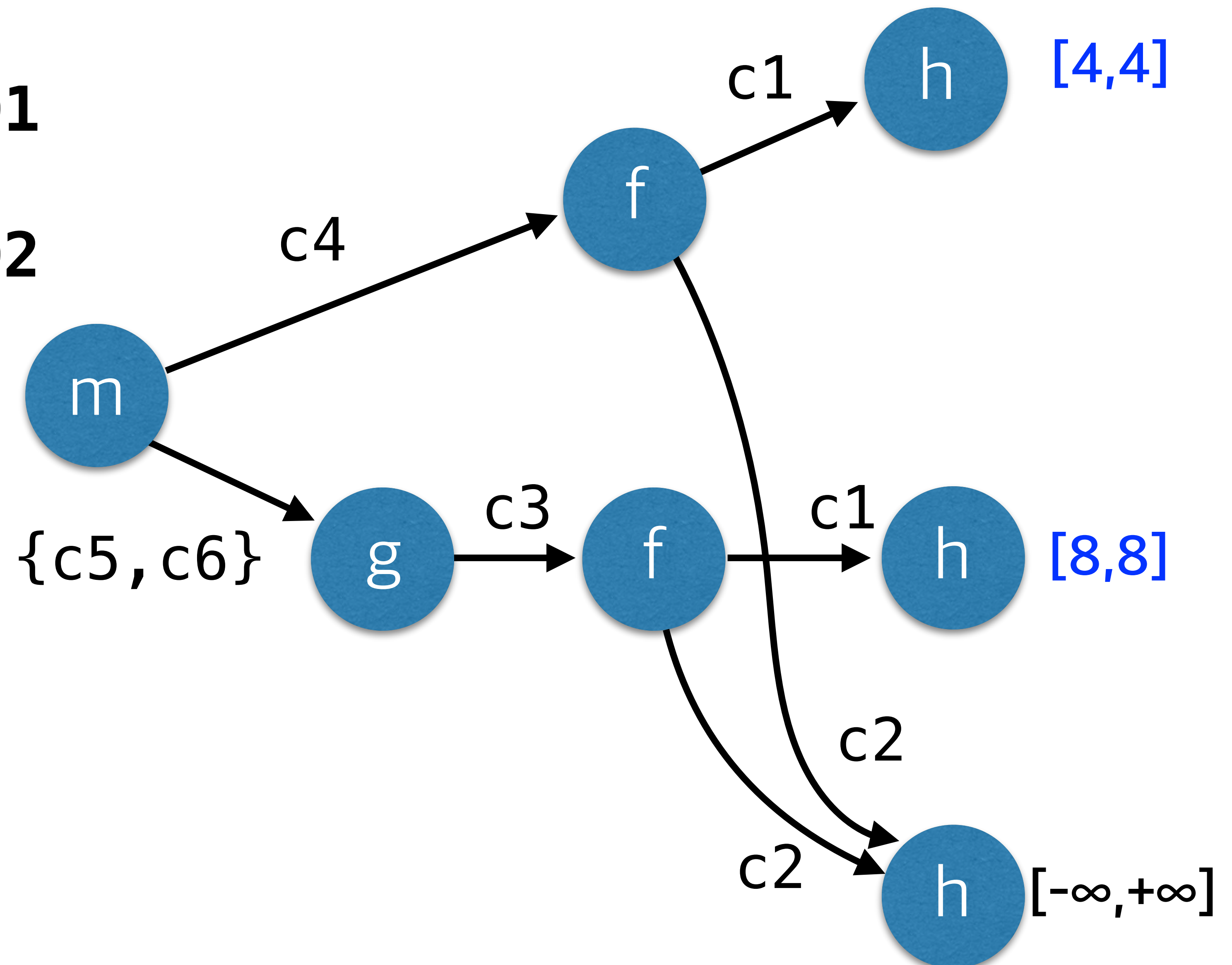
Selective Context-Sensitivity

```
    int h(n) {ret n;}

    void f(a) {
c1:   x = h(a);
      assert(x > 1);  // Q1
c2:   y = h(input());
      assert(y > 1);  // Q2
    }

c3: void g() {f(8);}

    void m() {
c4:   f(4);
c5:   g();
c6:   g();
    }
```



How to select?

- Often done manually by analysis designers
- Finding a good selection strategy is an art:
  - Intractably large space, if not infinite:
    ex) 2^{Var} different abstractions for FS
  - Most of them are too imprecise or costly
    ex) P({x,y,z}) = {~~∅,{x},{y},{z}~~,{x,y},~~{y,z},{x,z}~~,{x,y,z}}

Our Research

- Develop techniques for automatically finding the selection strategies
  - [PLDI'14, OOPSLA'15, TOPLAS'16, SAS'16, APLAS'16]
- Use machine learning techniques to learn a good strategy from freely available data.

Contents

- Learning via black-box optimization [OOPSLA'15]
- Learning via white-box optimization [APLAS'16]
- Learning from automatically labelled data [SAS'16]
- Learning with automatically generated features (in progress)
- Learning unsoundness strategy (in progress)
- Learning search strategy of concolic testing (in progress)
- Learning static analyzers (in progress)

Learning via Blackbox Optimization (OOPSLA'15)

Static Analyzer

$$F(p, a) \Rightarrow n$$

number of
proved assertions

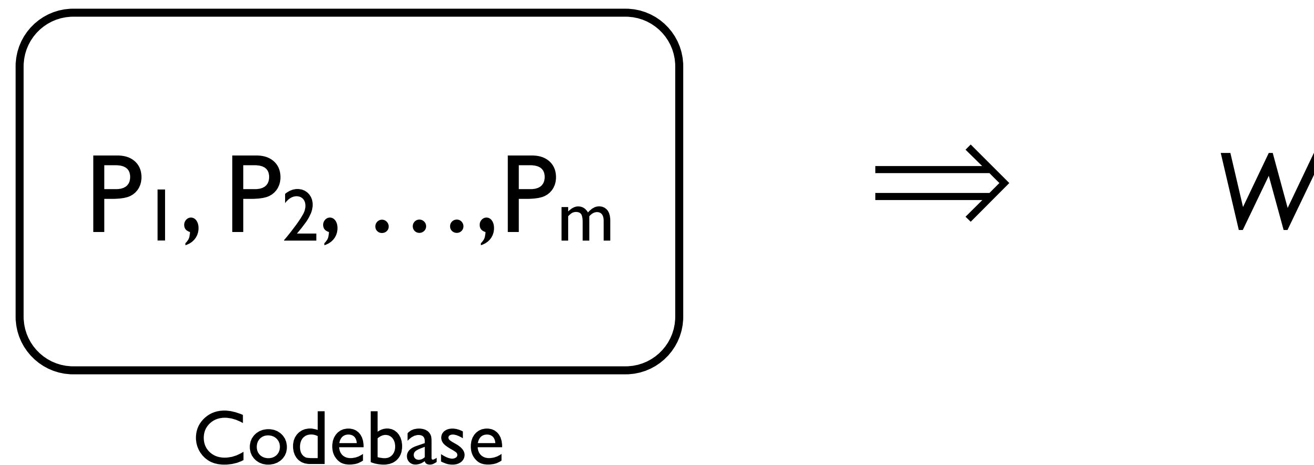
abstraction
(e.g., a set of variables)

Overall Approach

- Parameterized adaptation strategy

$$S_W : \text{pgm} \rightarrow 2^{\text{Var}}$$

- Learn a good parameter *W* from existing codebase



- For new program P, run static analysis with $S_W(P)$

1. Parameterized Strategy

$$S_w : \text{pgm} \rightarrow 2^{\text{Var}}$$

(1) Represent program variables as feature vectors.

(2) Compute the score of each variable.

(3) Choose the top-k variables based on the score.

(1) Features

- Predicates over variables:

$$f = \{f_1, f_2, \ldots, f_5\} \qquad (f_i : Var \rightarrow \{0,1\})$$

- 45 simple syntactic features for variables: e.g,
  - local / global variable, passed to / returned from malloc, incremented by constants, etc

- Represent each variable as a feature vector:

$$f(x) = \langle f_1(x), f_2(x), f_3(x), f_4(x), f_5(x) \rangle$$

(2) Scoring

- The parameter **w** is a real-valued vector: e.g.,

$$\mathbf{w} = \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle$$

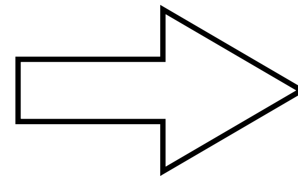
- Compute scores of variables:

$$\text{score}(x) = \langle 1,0,1,0,0 \rangle \cdot \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle = 0.3$$
$$\text{score}(y) = \langle 1,0,1,0,1 \rangle \cdot \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle = 0.6$$
$$\text{score}(z) = \langle 0,0,1,1,0 \rangle \cdot \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle = 0.1$$

(3) Choose Top-k Variables

- Choose the top-k variables based on their scores: e.g., when k=2,

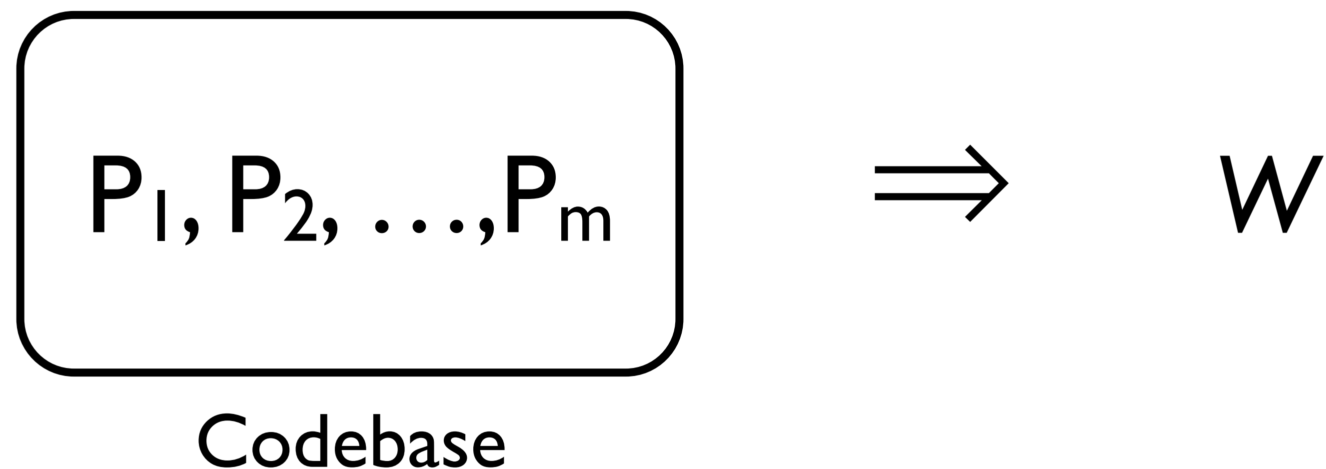
score(x) = 0.3
score(y) = 0.6
score(z) = 0.1



{x,y}

- In experiments, we chosen 10% of variables with highest scores.

2. Learn a Good Parameter



- Solve the optimization problem:

Find **w** that maximizes $\sum_{P_i} F(P_i, S_{\mathbf{w}}(P_i))$

Learning via Random Sampling

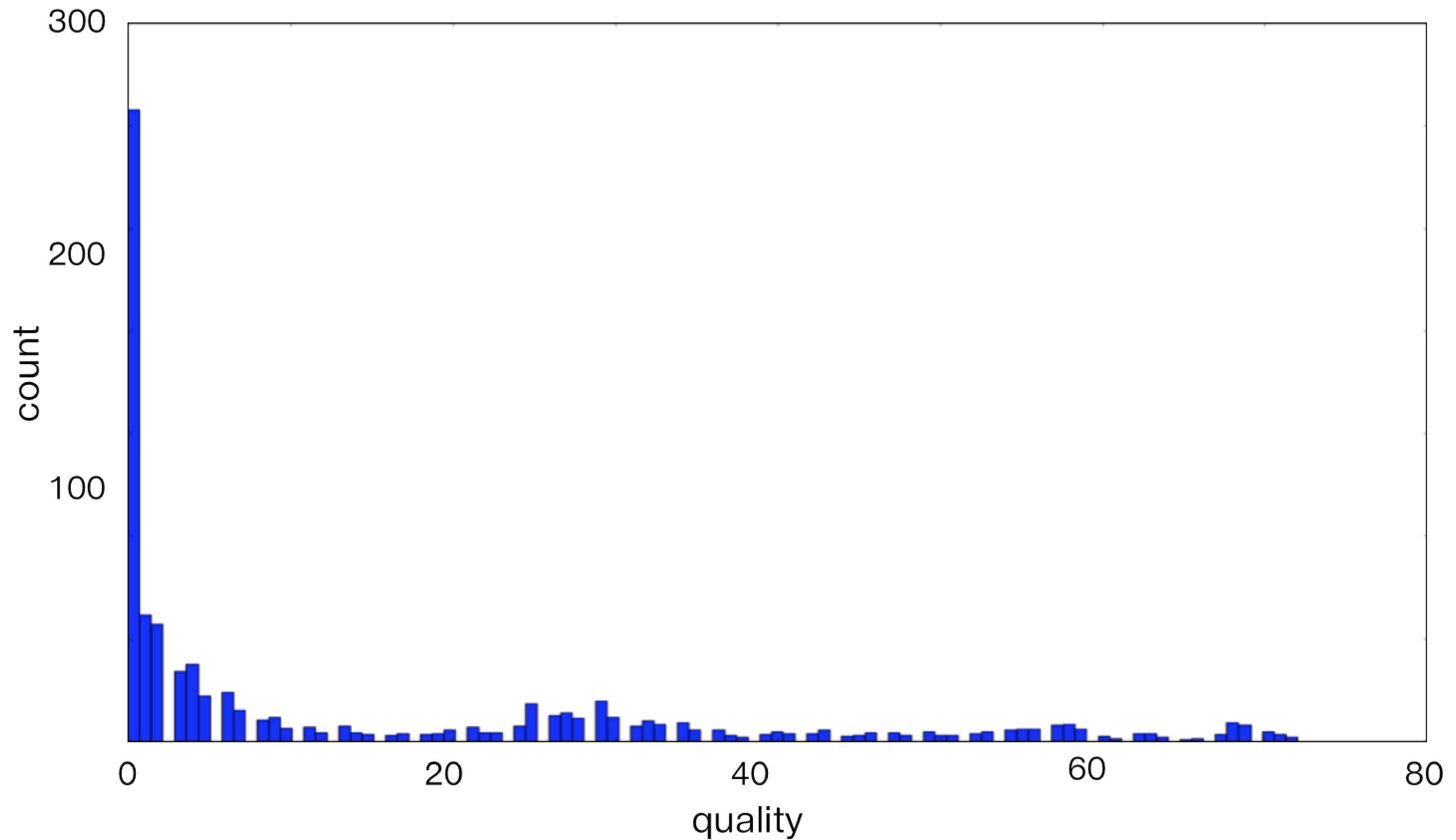
repeat N times

pick $\mathbf{w} \in R^n$ randomly

evaluate $\sum_{P_i} F(P_i, S_{\mathbf{w}}(P_i))$

return best $\mathbf{w}$ found

Learning via Random Sampling



Bayesian Optimization

- A powerful method for solving difficult black-box optimization problems.
- Especially powerful when the objective function is expensive to evaluate.
- Key idea: use a probabilistic model to reduce the number of objective function evaluations.

Learning via Bayesian Optimization

repeat N times

select a promising w using the model

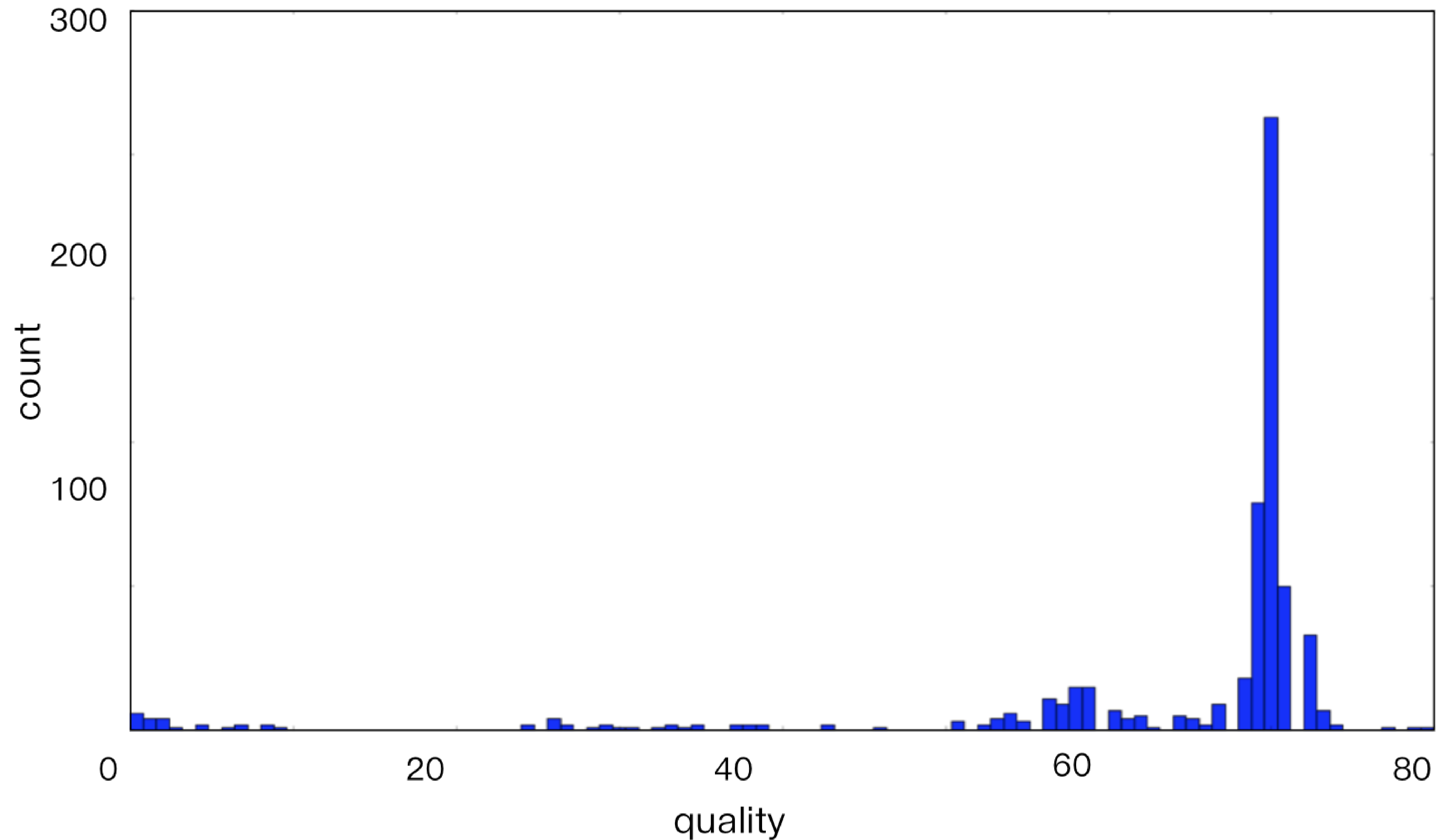
evaluate $\sum_{P_i} F(P_i, S_{\mathbf{w}}(P_i))$

update the probabilistic model

return best w found

- Probabilistic model: Gaussian processes
- Selection strategy: Expected improvement

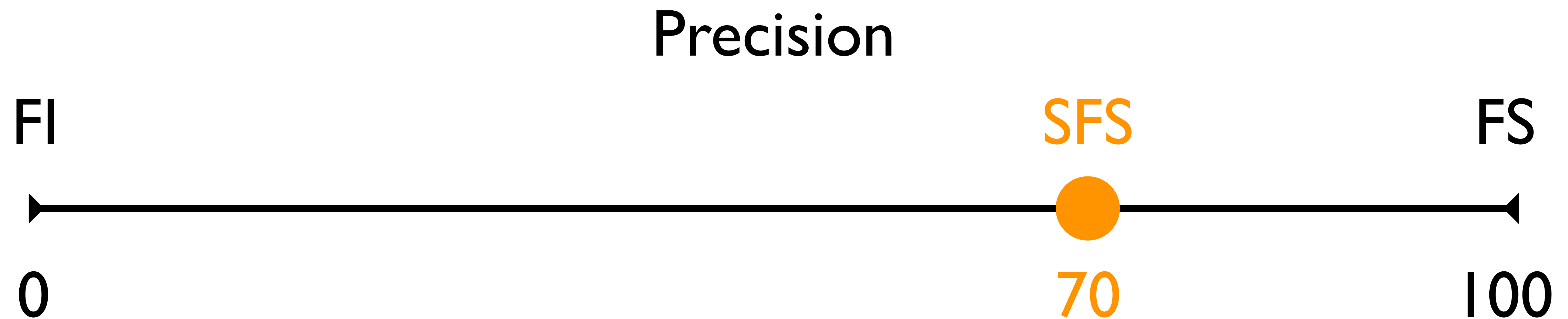
Learning via Bayesian Optimization



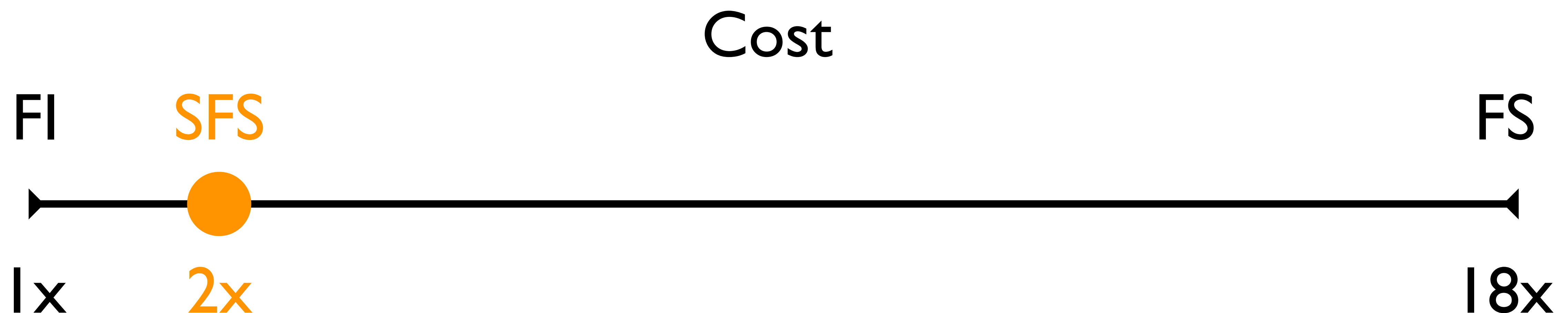
Effectiveness

- Implemented in Sparrow, an interval analyzer for C
- Evaluated on open-source benchmarks

Precision



Cost



Automatically Generating Features (In Progress)

Limitation: Feature Engineering

- The success of ML heavily depends on the "features"
- Feature engineering is nontrivial and time-consuming
- Features do not generalize to other domains

Type	#	Features
A	1	local variable
	2	global variable
	3	structure field
	4	location created by dynamic memory allocation
	5	defined at one program point
	6	location potentially generated in library code
	7	assigned a constant expression (e.g., x = c1 + c2)
	8	compared with a constant expression (e.g., x < c)
	9	compared with an other variable (e.g., x < y)
	10	negated in a conditional expression (e.g., if (!x))
	11	directly used in malloc (e.g., malloc(x))
	12	indirectly used in malloc (e.g., y = x; malloc(y))
	13	directly used in realloc (e.g., realloc(x))
	14	indirectly used in realloc (e.g., y = x; realloc(y))
	15	directly returned from malloc (e.g., x = malloc(e))
	16	indirectly returned from malloc
	17	directly returned from realloc (e.g., x = realloc(e))
	18	indirectly returned from realloc
	19	incremented by one (e.g., x = x + 1)
	20	incremented by a constant expr. (e.g., x = x + (1+2))
	21	incremented by a variable (e.g., x = x + y)
	22	decremented by one (e.g., x = x - 1)
	23	decremented by a constant expr (e.g., x = x - (1+2))
	24	decremented by a variable (e.g., x = x - y)
	25	multiplied by a constant (e.g., x = x * 2)
	26	multiplied by a variable (e.g., x = x * y)
	27	incremented pointer (e.g., p++)
	28	used as an array index (e.g., a[x])
	29	used in an array expr. (e.g., x[e])
	30	returned from an unknown library function
	31	modified inside a recursive function
	32	modified inside a local loop
	33	read inside a local loop
B	34	$1 \wedge 8 \wedge (11 \vee 12)$
	35	$2 \wedge 8 \wedge (11 \vee 12)$
	36	$1 \wedge (11 \vee 12) \wedge (19 \vee 20)$
	37	$2 \wedge (11 \vee 12) \wedge (19 \vee 20)$
	38	$1 \wedge (11 \vee 12) \wedge (15 \vee 16)$
	39	$2 \wedge (11 \vee 12) \wedge (15 \vee 16)$
	40	$(11 \vee 12) \wedge 29$
	41	$(15 \vee 16) \wedge 29$
	42	$1 \wedge (19 \vee 20) \wedge 33$
	43	$2 \wedge (19 \vee 20) \wedge 33$
	44	$1 \wedge (19 \vee 20) \wedge \neg 33$
	45	$2 \wedge (19 \vee 20) \wedge \neg 33$

flow-sensitivity

Type	#	Features
A	1	leaf function
	2	function containing malloc
	3	function containing realloc
	4	function containing a loop
	5	function containing an if statement
	6	function containing a switch statement
	7	function using a string-related library function
	8	write to a global variable
	9	read a global variable
	10	write to a structure field
	11	read from a structure field
	12	directly return a constant expression
	13	indirectly return a constant expression
	14	directly return an allocated memory
	15	indirectly return an allocated memory
	16	directly return a reallocated memory
	17	indirectly return a reallocated memory
	18	return expression involves field access
	19	return value depends on a structure field
	20	return void
	21	directly invoked with a constant
	22	constant is passed to an argument
	23	invoked with an unknown value
	24	functions having no arguments
	25	functions having one argument
	26	functions having more than one argument
	27	functions having an integer argument
	28	functions having a pointer argument
	29	functions having a structure as an argument
B	30	$2 \wedge (21 \vee 22) \wedge (14 \vee 15)$
	31	$2 \wedge (21 \vee 22) \wedge \neg(14 \vee 15)$
	32	$2 \wedge 23 \wedge (14 \vee 15)$
	33	$2 \wedge 23 \wedge \neg(14 \vee 15)$
	34	$2 \wedge (21 \vee 22) \wedge (16 \vee 17)$
	35	$2 \wedge (21 \vee 22) \wedge \neg(16 \vee 17)$
	36	$2 \wedge 23 \wedge (16 \vee 17)$
	37	$2 \wedge 23 \wedge \neg(16 \vee 17)$
	38	$(21 \vee 22) \wedge \neg 23$

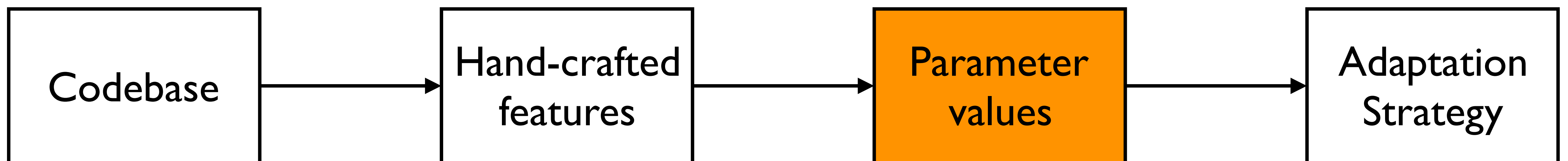
context-sensitivity

Type	#	Features
A	1	used in array declarations (e.g., a[c])
	2	used in memory allocation (e.g., malloc(c))
	3	used in the righthand-side of an assignment (e.g., x = c)
	4	used with the less-than operator (e.g, x < c)
	5	used with the greater-than operator (e.g., x > c)
	6	used with $\leq$ (e.g., x $\leq$ c)
	7	used with $\geq$ (e.g., x $\geq$ c)
	8	used with the equality operator (e.g., x == c)
	9	used with the not-equality operator (e.g., x ! = c)
	10	used within other conditional expressions (e.g., x < c+y)
	11	used inside loops
	12	used in return statements (e.g., return c)
	13	constant zero
B	14	$(1 \vee 2) \wedge 3$
	15	$(1 \vee 2) \wedge (4 \vee 5 \vee 6 \vee 7)$
	16	$(1 \vee 2) \wedge (8 \vee 9)$
	17	$(1 \vee 2) \wedge 11$
	18	$(1 \vee 2) \wedge 12$
	19	$13 \wedge 3$
	20	$13 \wedge (4 \vee 5 \vee 6 \vee 7)$
	21	$13 \wedge (8 \vee 9)$
	22	$13 \wedge 11$
	23	$13 \wedge 12$

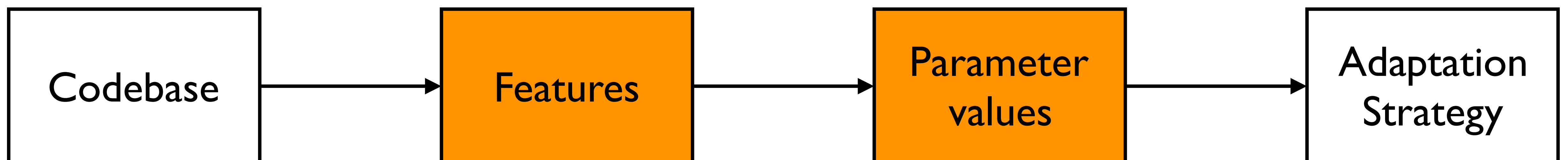
widening thresholds

Automatic Feature Generation

Before



New method



(analogous to representation learning, deep learning, etc in ML)

Example: Flow-Sensitive Analysis

- A query-based, partially flow-sensitive interval analysis
- The analysis uses a query-classifier C : Query → {1,0}

```
1  x = 0; y = 0; z = input(); w = 0;
2  y = x; y++;
3  assert (y > 0);    // Query 1  provable
4  assert (z > 0);    // Query 2  unprovable
5  assert (w == 0);   // Query 3  unprovable
```

flow-sensitive result		flow-insensitive result
line	abstract state	abstract state
1	$\{x \mapsto [0,0], y \mapsto [0,0]\}$	$\{z \mapsto [0,0], w \mapsto [0,0]\}$
2	$\{x \mapsto [0,0], y \mapsto [1,1]\}$	
3	$\{x \mapsto [0,0], y \mapsto [1,1]\}$	
4	$\{x \mapsto [0,0], y \mapsto [1,1]\}$	
5	$\{x \mapsto [0,0], y \mapsto [1,1]\}$	

Learning a Query Classifier

Standard binary classification:

$$\{(q_i, b_i)\}_{i=1}^{n} \longrightarrow \{(v_i, b_i)\}_{i=1}^{n} \longrightarrow \mathcal{C} : \mathbb{B}^k \to \mathbb{B}$$

$$(v_i \in \mathbb{B}^k)$$

feature extraction

standard learning algorithms

- Feature extraction is a key to success
- Raw data should be converted to suitable representations from which classification algorithms could find useful patterns

We aim to automatically find the right representation

Feature Extraction

A feature describes a property of queries

- Features and matching algorithm:
  - a set of features: $\Pi = \{\pi_1, \ldots, \pi_k\}$
  - match : $Query \times Feature \rightarrow \mathbb{B}$
- Transform the query q into the feature vector:

$$\langle \mathsf{match}(q, \pi_1), \ldots, \mathsf{match}(q, \pi_k) \rangle$$

Generating Features

$$\Pi = \{\pi_1, \ldots, \pi_k\}$$

- A feature is a graph that describes data flows of queries
- What makes good features?
  - selective to key aspects for discrimination
  - invariant to irrelevant aspects for generalization
- Generating features:
  - Generate *feature programs* by running reducer
  - Represent the feature programs by data-flow graphs
- Π is the set of all data flow graphs generated from the codebase

Generating Features

- Feature program *P* is a minimal program such that

$$\phi(P) \equiv FI(P) = unproven \wedge FS(P) = proven$$

- Generic program reducer: e.g., C-Reduce [PLDI'12]

$$\text{reduce} : \mathbb{P} \times (\mathbb{P} \rightarrow \mathbb{B}) \rightarrow \mathbb{P}$$

- Reducing programs while preserving the condition

$$\text{reduce}(P, \phi)$$

  generates feature programs.

Generating Features

- Reduce programs while preserving the condition

$$\phi(P) \equiv FI(P) = unproven \wedge FS(P) = proven$$

```
1  a = 0; b = 0;
2  while (1) {
3   b = unknown();
4   if (a > b)
5     if (a < 3)
6       assert (a < 5);
7   a++;
8  }
```

reduce(P, ϕ) $\Rightarrow$

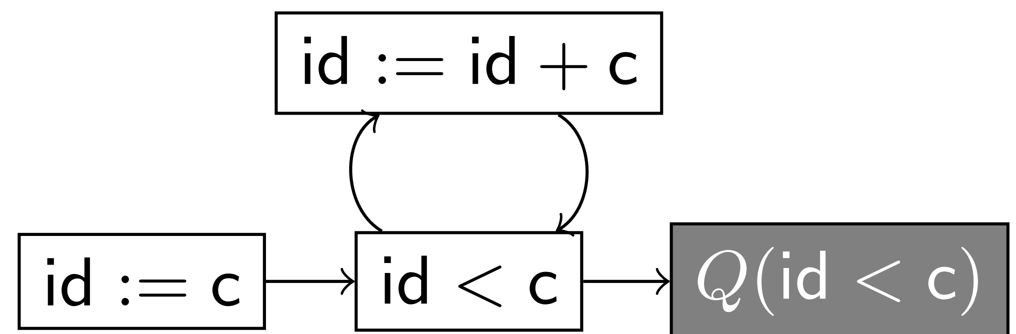
```
1  a = 0;
2  while (1) {
3   if (a < 3)
4     assert (a < 5);
5   a++;
6  }
```

Generating Features

- Represent the features by abstract data flow graphs

```
1   a = 0;
2   while (1) {
3    if (a < 3)
4      assert (a < 5);
5    a++;
6   }
```

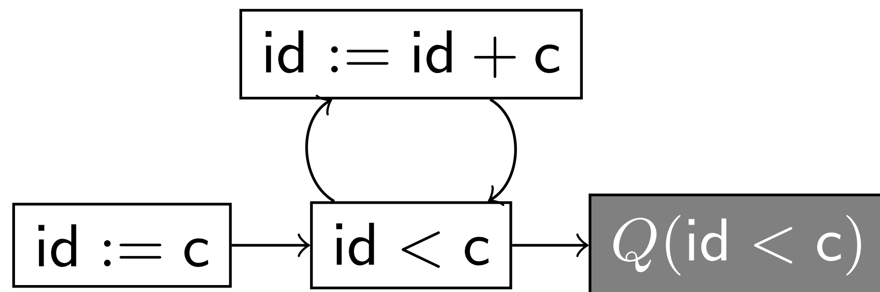
$\overset{\alpha}{\Longrightarrow}$



- The right level of abstraction is learned from codebase

Matching Algorithm

match : $Query \times Feature \rightarrow \mathbb{B}$



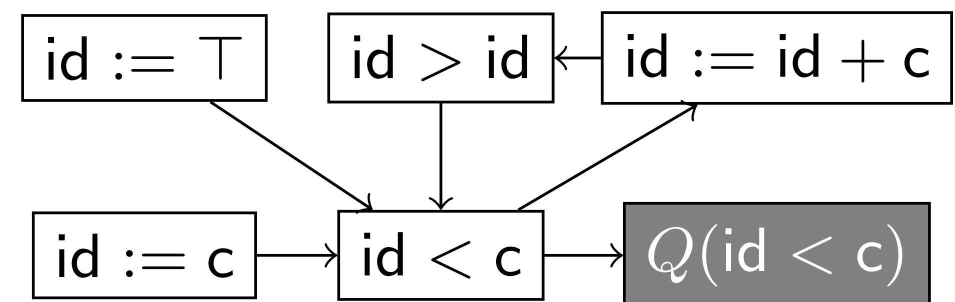
$\overset{?}{\subseteq}$

```
1   a = 0; b = 0;
2   while (1) {
3    b = unknown();
4    if (a > b)
5      if (a < 3)
6        assert (a < 5);
7    a++;
8   }
```

$\subseteq$

Subgraph inclusion:

$(N_1, E_1) \subseteq (N_2, E_2) \iff N_1 \subseteq N_2 \wedge E_1 \subseteq E_2^*$



Performance

- Partially flow-sensitive interval analysis
- Partially relational octagon analysis

	Query Selection		Analysis							
			Prove			Sec				
Trial	Precision	Recall	ITV	IMPCT	ML	ITV	IMPCT	ML	Quality	Cost
1	59.8 %	71.2 %	7128	7192	7181	772	4496.8	1098.8	82.8 %	8.8 %
2	70.3 %	92.0 %	6792	6926	6918	376.7	8568.4	958.3	94.0 %	7.1 %
3	68.0 %	90.3 %	1014	1129	1126	324.0	972.3	453.0	97.4 %	19.9 %
4	82.8 %	72.7 %	6877	6962	6940	370.5	8838	984.6	74.1 %	7.3 %
5	68.1 %	67.1 %	2585	2657	2636	418.1	1392.7	624.3	70.8 %	21.2 %
TOTAL	70.5 %	81.5 %	24396	24866	24801	2261.3	24268.2	4119.0	86.2 %	8.4 %

Other PA + ML Approaches

Learning via White-box Optimization [APLAS'16]

- The black-box optimization method is too slow when the codebase is large
- Replace it to an easy-to-solve white-box problem by using oracle:

$$\mathcal{O}_P : \mathbb{J}_P \rightarrow \mathbb{R}.$$

$$\text{Find } \mathbf{w}^* \text{ that minimizes } \sum_{j \in \mathbb{J}_P} (score_P^{\mathbf{w}}(j) - \mathcal{O}(j))^2$$

- Oracle is obtained from a single run of codebase
- 26x faster to learn a comparable strategy

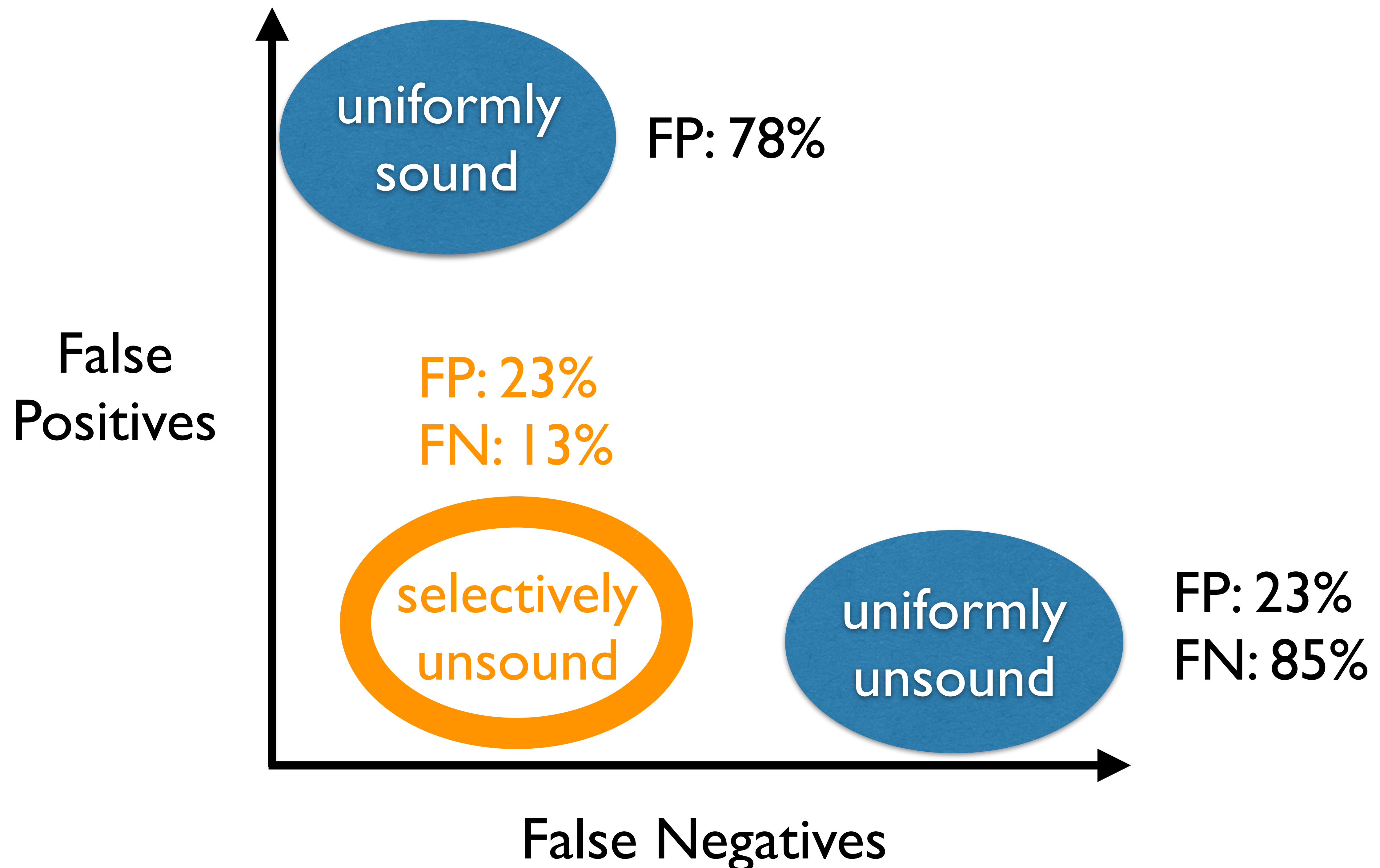
Learning from Automatically Labelled Data [SAS'16]

- Learning a variable clustering strategy for Octagon is too difficult to solve with black-box optimization
- Replace it to a (much easier) supervised-learning problem:

	a	−a	b	−b	c	−c	i	−i
a	★	⊤	★	⊤	⊤	⊤	★	⊤
−a	⊤	★	⊤	★	⊤	⊤	⊤	⊤
b	★	⊤	★	⊤	⊤	⊤	★	⊤
−b	⊤	★	⊤	★	⊤	⊤	⊤	⊤
c	⊤	⊤	⊤	⊤	★	⊤	⊤	⊤
−c	⊤	⊤	⊤	⊤	⊤	★	⊤	⊤
i	⊤	⊤	⊤	⊤	⊤	⊤	★	⊤
−i	⊤	★	⊤	★	⊤	⊤	⊤	★

- Who label the data? by impact pre-analysis [PLDI'14].
- The ML-guided Octagon analysis is 33x faster than the pre-analysis-guided one with 2% decrease in precision.

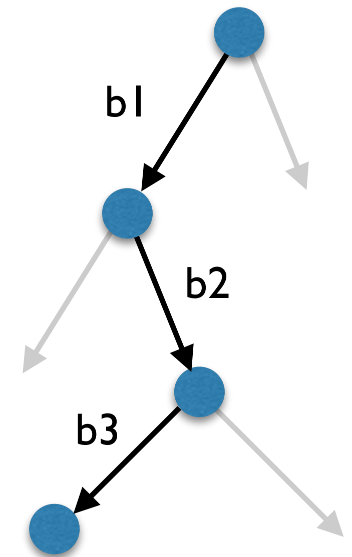
Learning Unsoundness Strategies (in submission)



Data-Driven Concolic Testing (in progress)

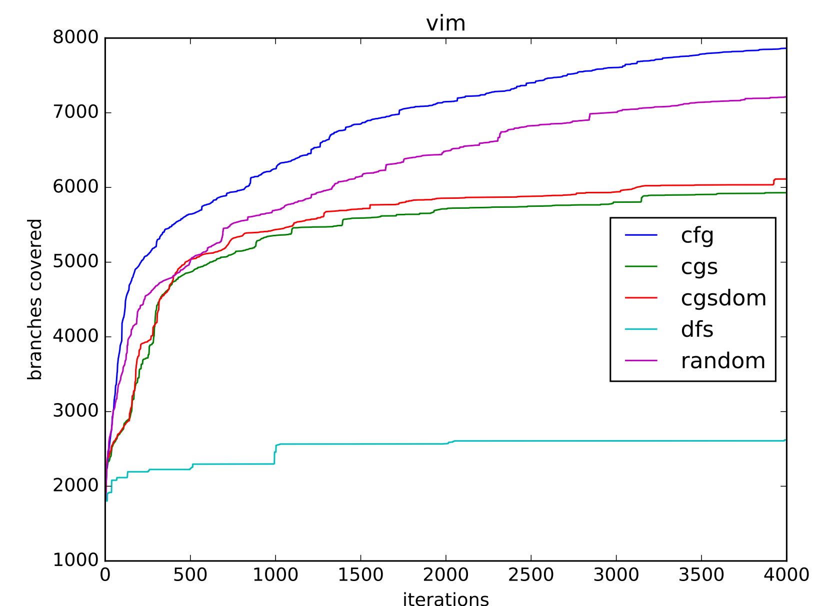
- The efficacy of concolic testing heavily depends on the search strategy

$$\mathcal{S} \in \mathit{Strategy} = \mathit{Path} \rightarrow \mathit{Branch}$$



- Search strategies are manually designed (heuristics)
  - e.g., $\mathcal{S}_{rand}, \mathcal{S}_{dfs}, \mathcal{S}_{cfg}, \mathcal{S}_{cgs}, \ldots$
  - a huge amount of engineering efforts
  - sub-optimal performance
- Automate the process: $\mathcal{S}_\theta : \mathit{Path} \rightarrow \mathit{Branch}$

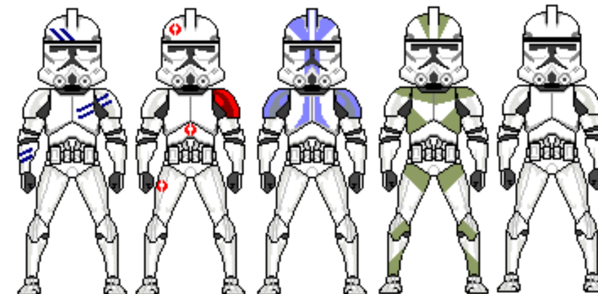
$$\text{Find } \theta^* \text{ that maximizes } \sum_{P_i \in \mathbb{P}} \mathcal{C}(P_i, \mathcal{S}_{\theta^*})$$



Learning Static Analyzers (in progress)

- The usage of static analyzers is limited in extreme (yet daily in practice) situations:
  - It cannot analyze unparsable programs.
  - It does not scale to the entire linux package.
  - A C analyzer cannot be used even for C++ code.
  - A source code analyzer cannot be used for binary code.

$\overset{learn}{\Rightarrow}$



Summary

- Adaptation is a key problem in static analysis
- Using ML is a promising and exciting direction
- Something is done with hand-tuning?
  - Parameterize it
  - Learn the best parameters from data

Thank you