Data-Driven Program Analysis

Hakjoo Oh

Programming Research Laboratory
Korea University

(co-work with Kihong Heo, Kwonsoo Chae, Hongseok Yang, Kwangkeun Yi)

Nov. 9, 2016 @POSTECH

Research Areas

- Program Analysis derives specifications from code
- Program Synthesis derives code from specifications

```
int f(int n) {
  int i = 0;
  int r = 1;
  while (i < n)
  {
    r = r * i;
    i = i + 1;
  }
  return r;
}
```

program analysis →

← program synthesis

$f(1) = 1$
$f(2) = 2$
$f(3) = 6$
$\dots$
$f(n) = n!$

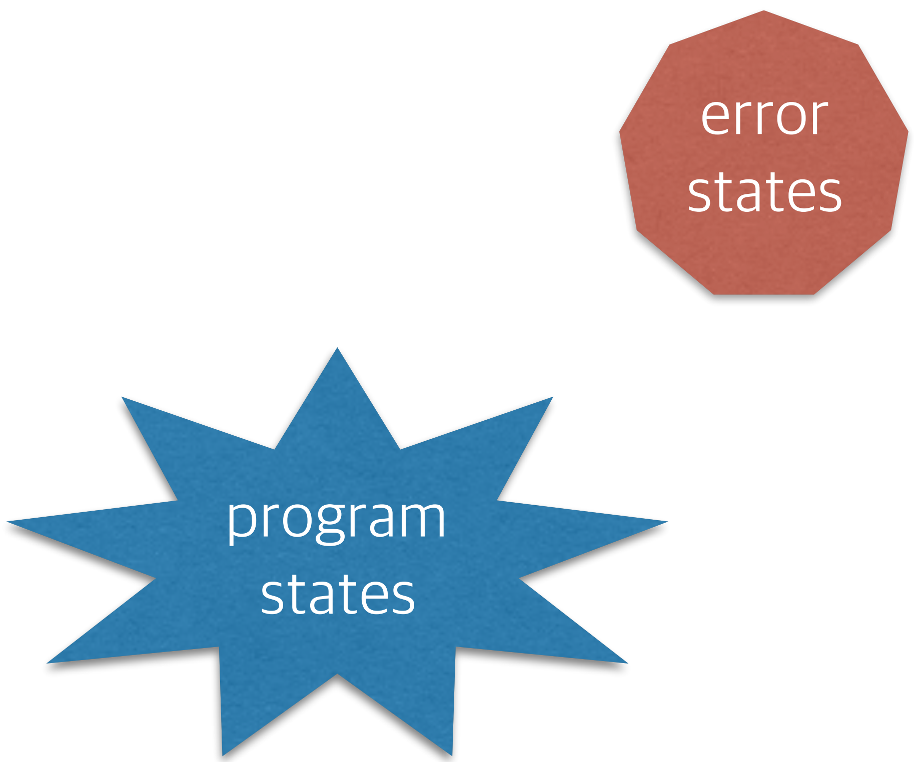
Program Analysis

- Predict program behavior automatically
  - **static** or **dynamic**: before execution at compile-time / at runtime
  - **automatic**: sw is analyzed by sw ("program 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

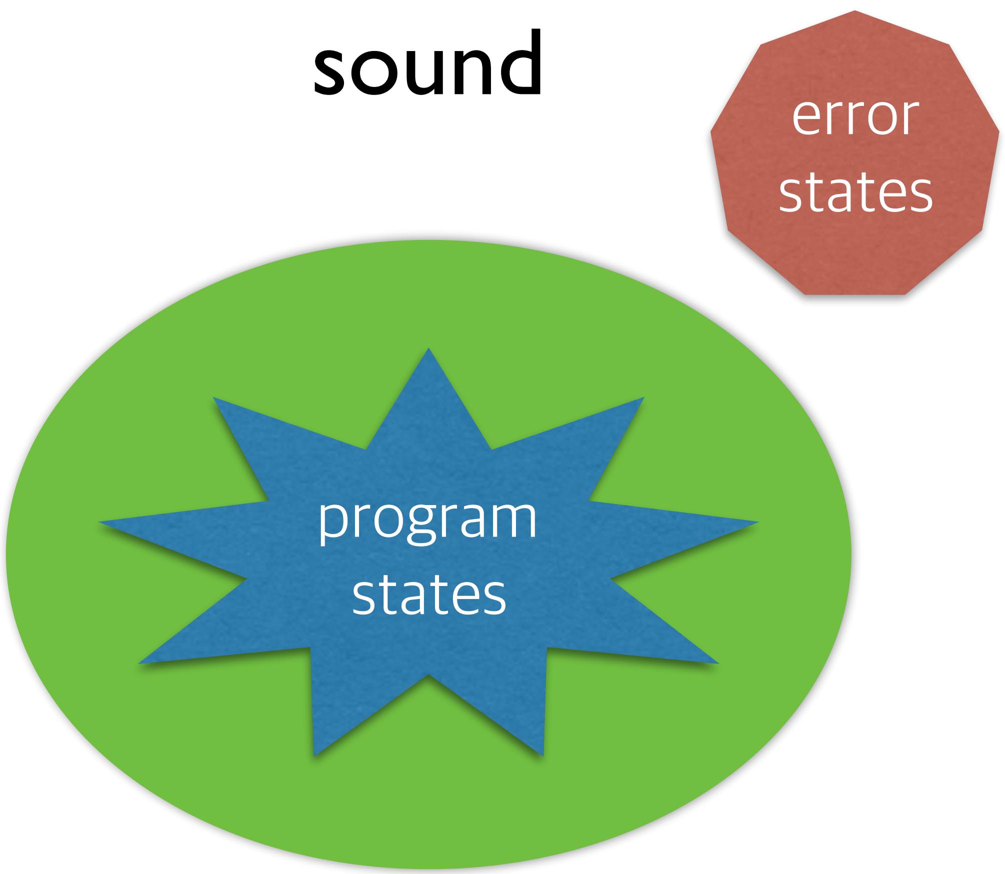
Program Synthesis

- Generate program code from specifications automatically
  - **specification**: logics, examples, implementation, etc
  - **automatic**: sw is generated by sw ("program synthesizers")
- Applications
  - **programming assistance:** e.g., complete tricky parts of programs
  - **end-user programming:** e.g., automate repetitive tasks
  - **algorithm discovery:** find a new solution for a problem
  - **program optimization:** find a more efficient implementation
  - **automatic patch generation**: automatically fix software bugs

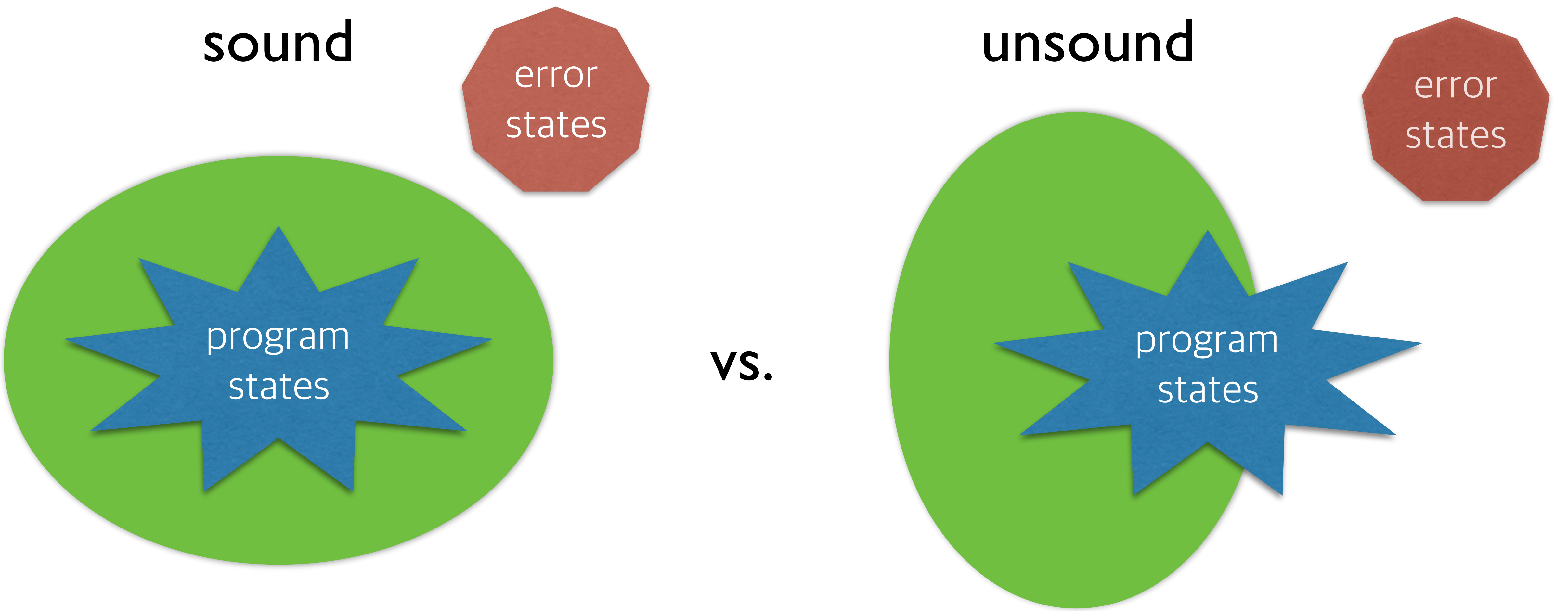
Static Program Analysis



Static Program Analysis

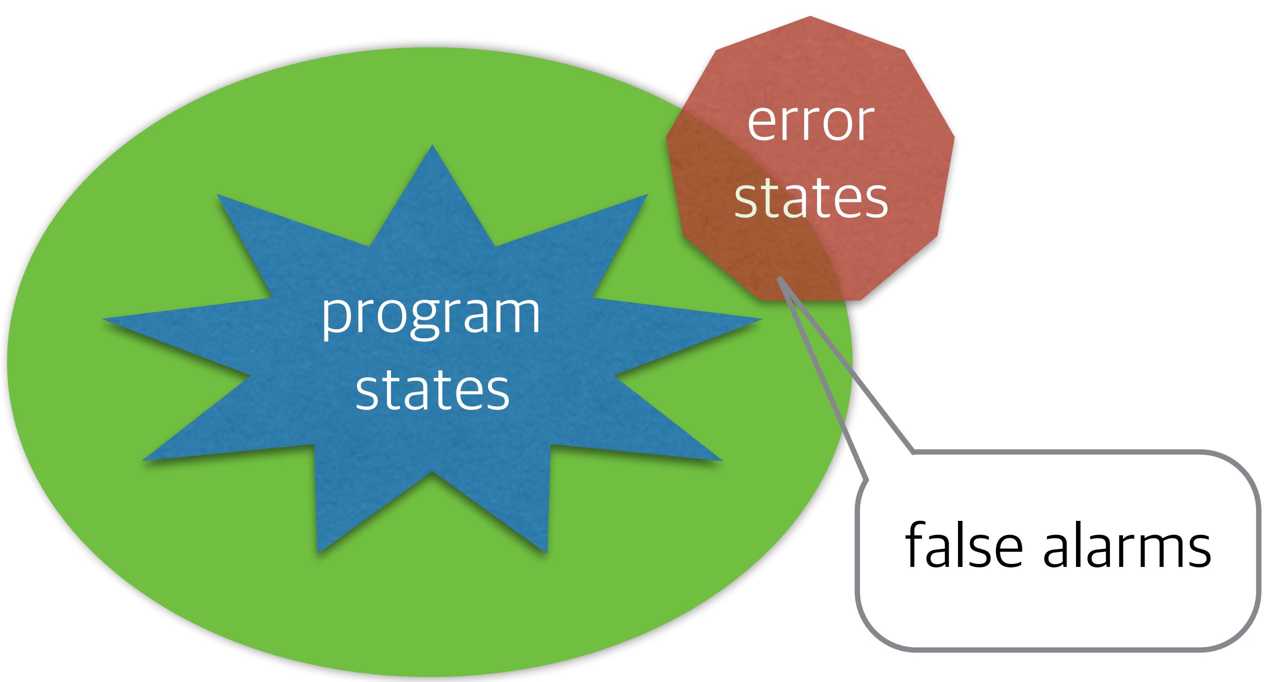


Static Program Analysis

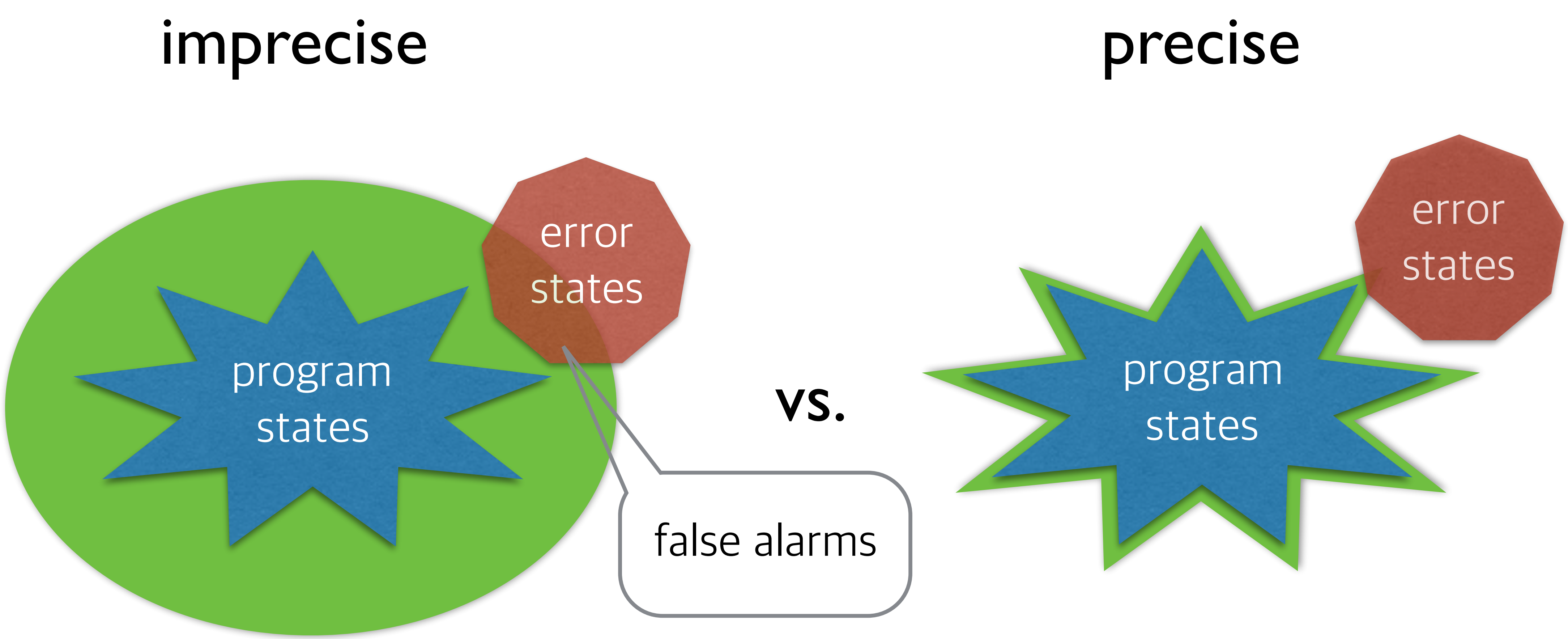


Static Program Analysis

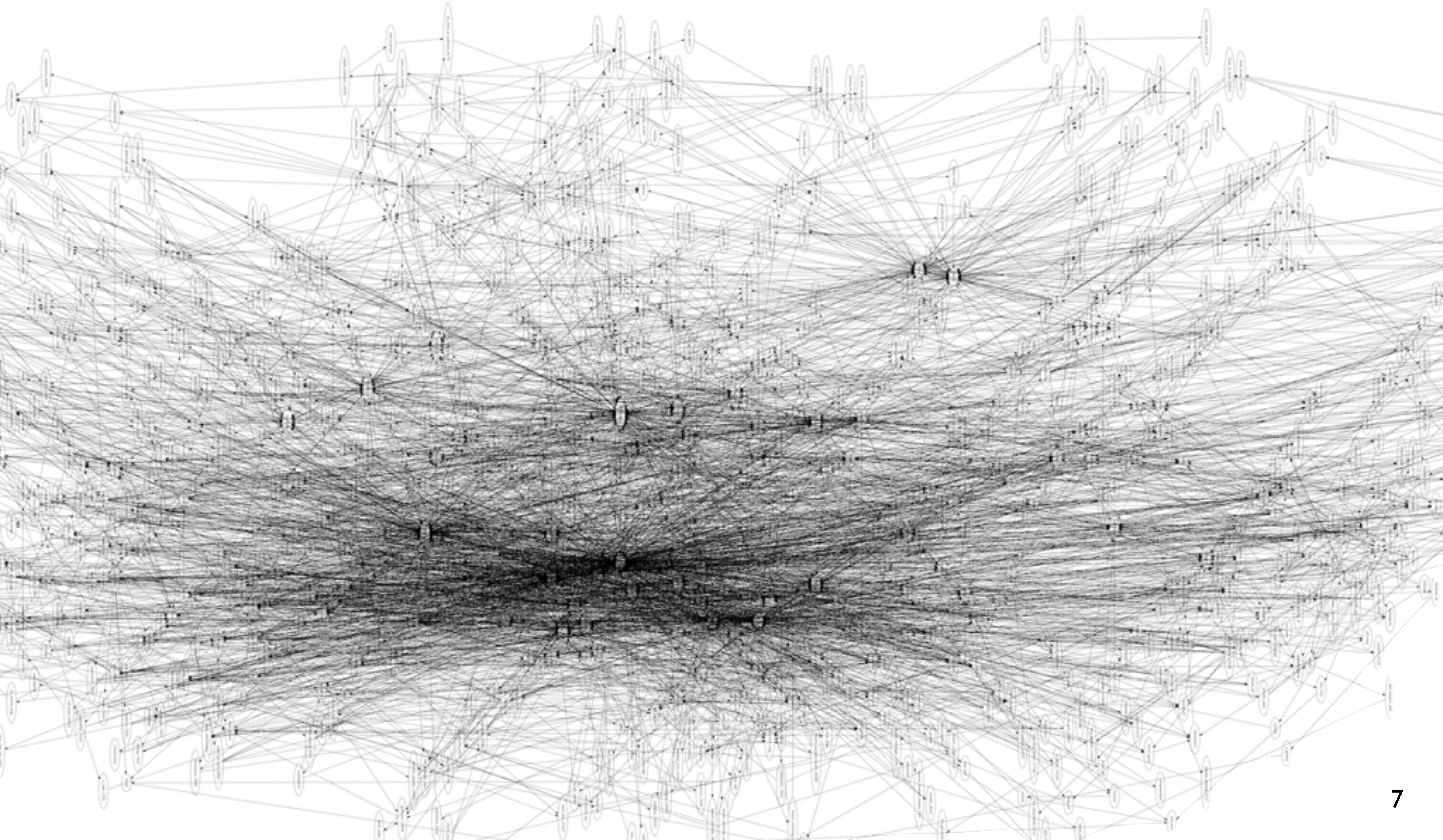
imprecise



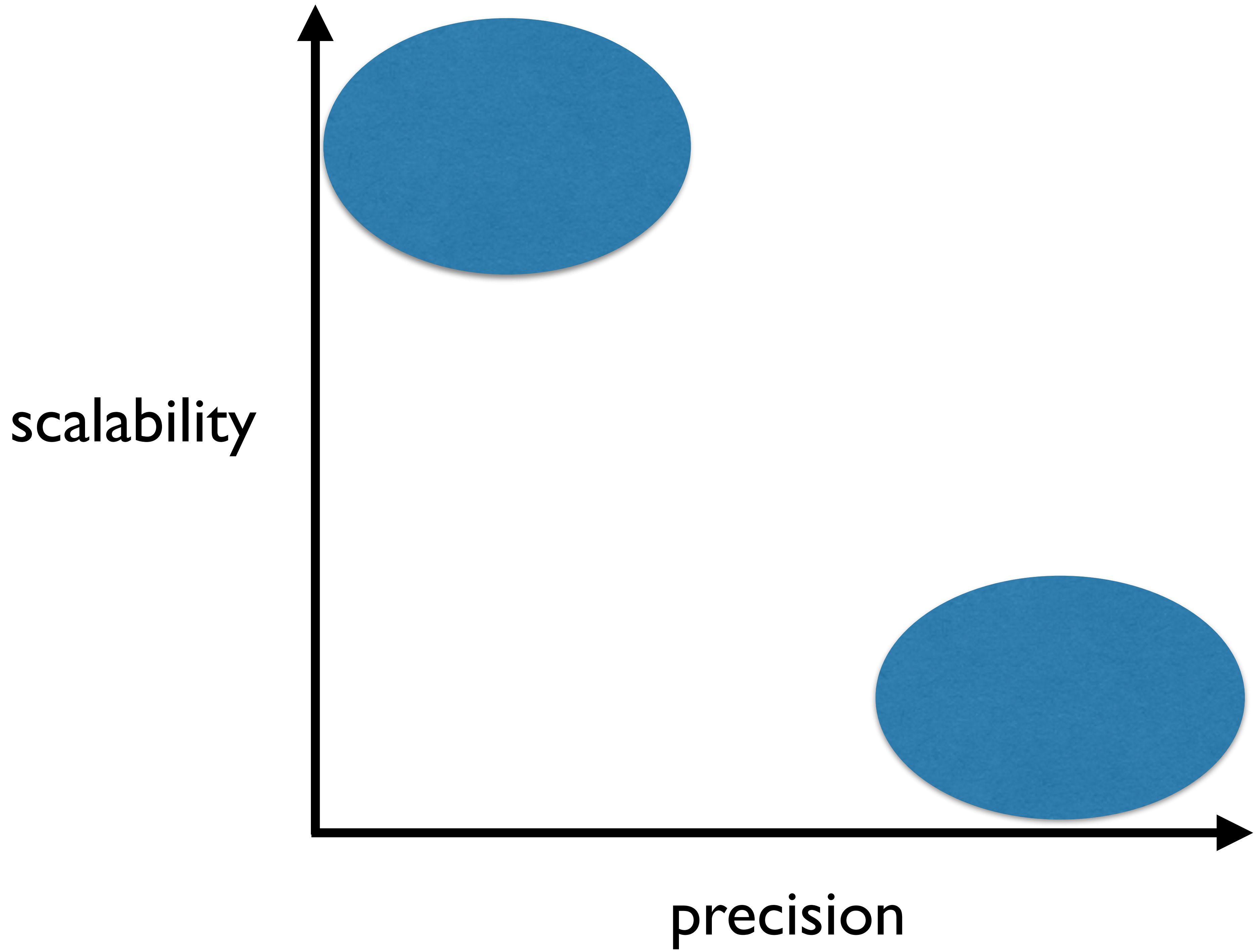
Static Program Analysis



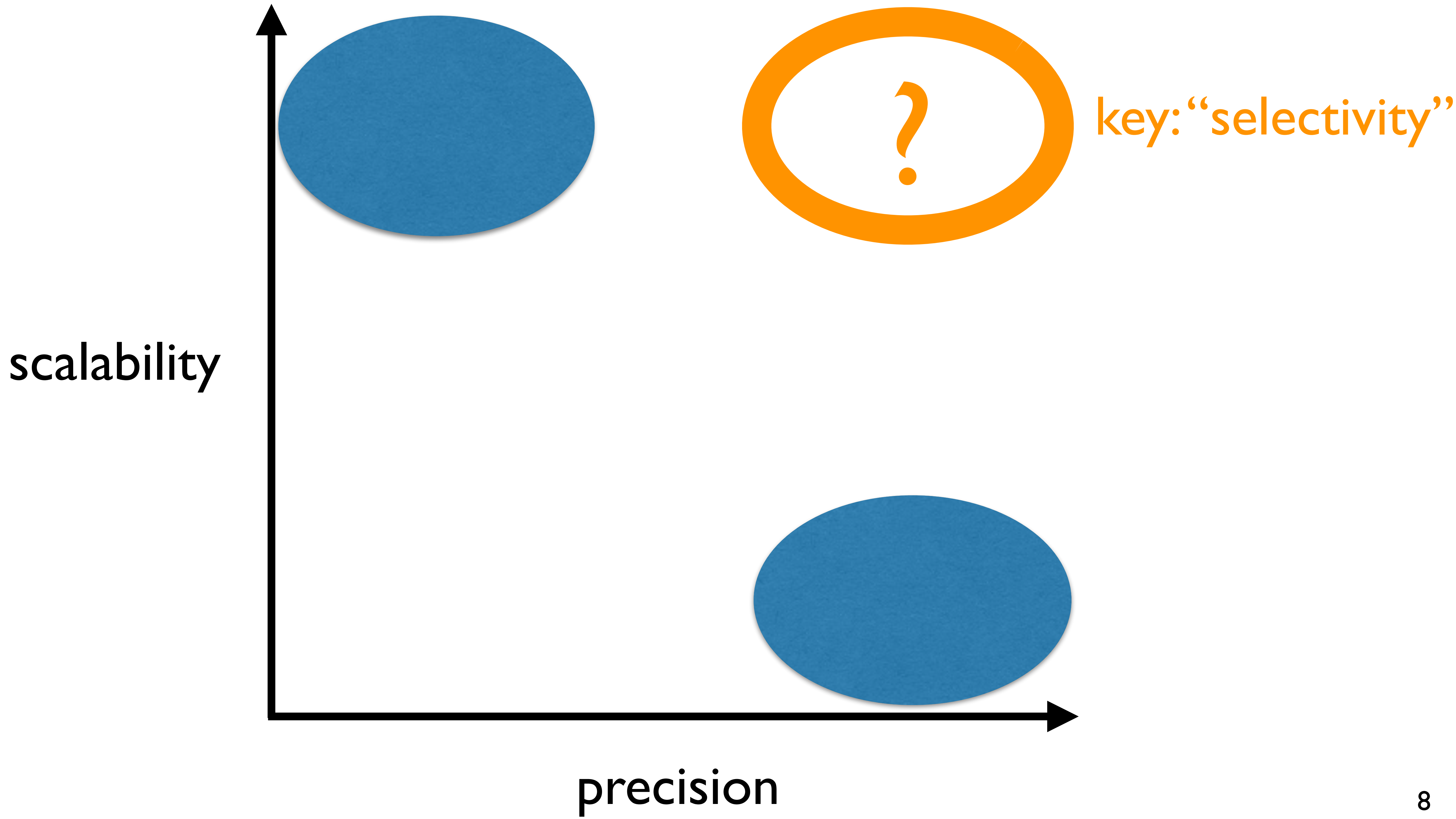
Static Program Analysis



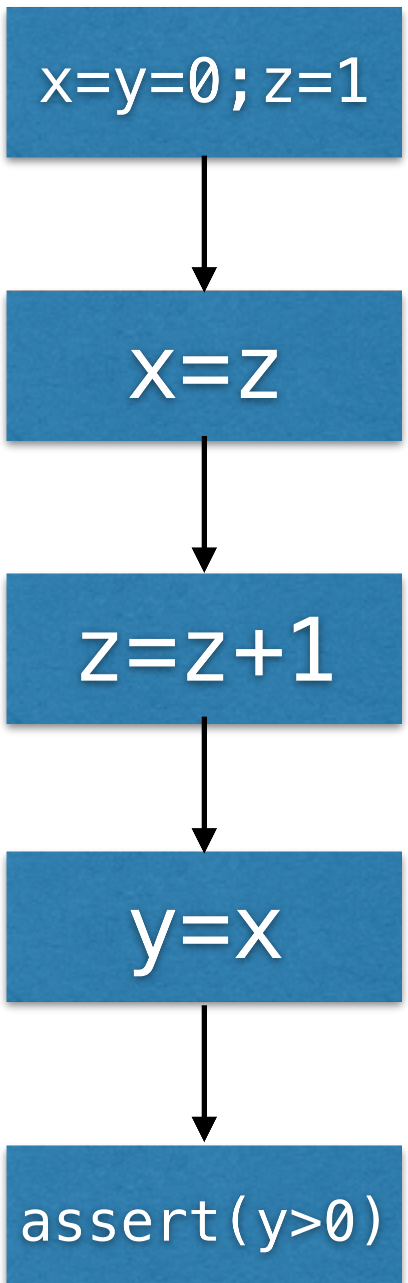
Challenge in Static Analysis



Challenge in Static Analysis



Flow-Sensitivity



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

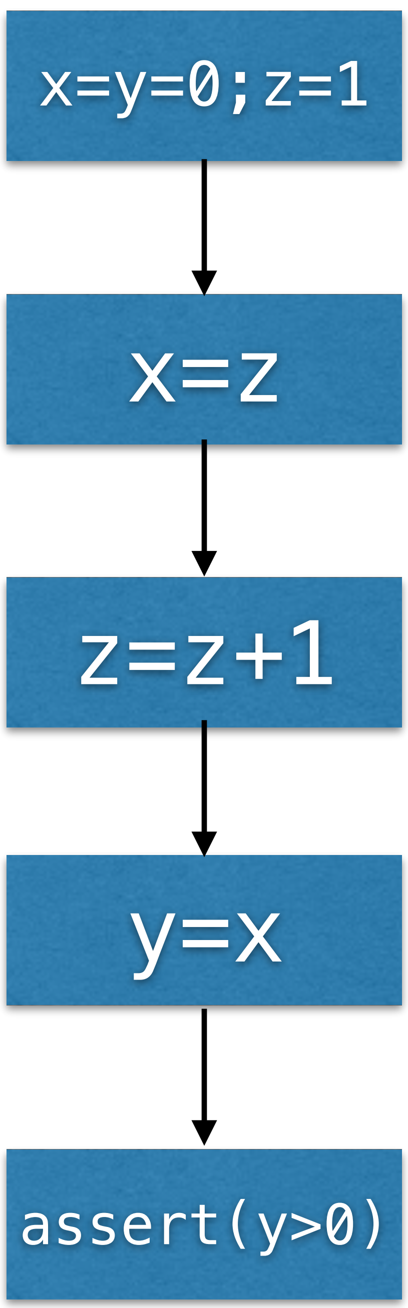
x	[1,1]
y	[0,0]
z	[1,1]

x	[1,1]
y	[0,0]
z	[2,2]

x	[1,1]
y	[1,1]
z	[2,2]

precise but costly

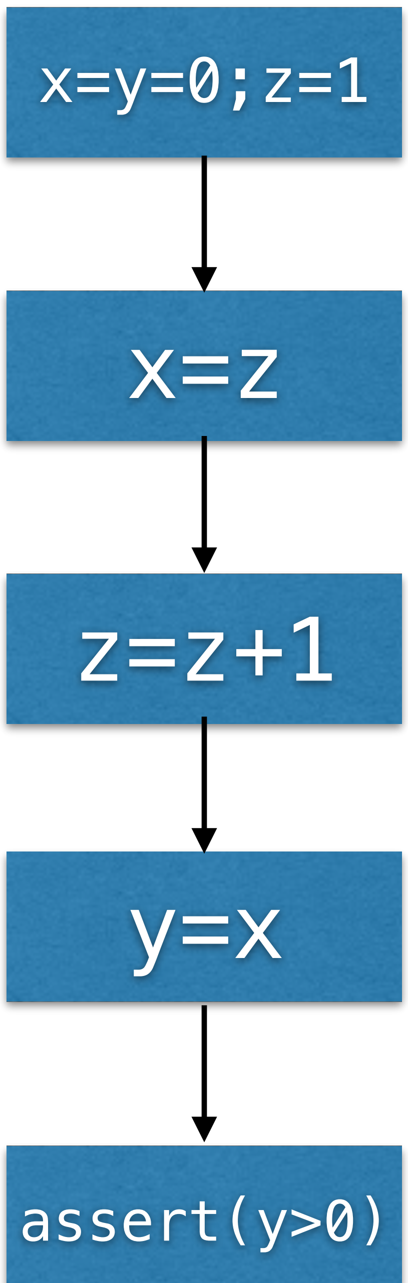
Flow-Insensitivity



x	[0,+∞]
y	[0,+∞]
z	[1,+∞]

cheap but imprecise

Selective Flow-Sensitivity



FS : {x,y}

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

x	[1,+∞]
y	[0,0]

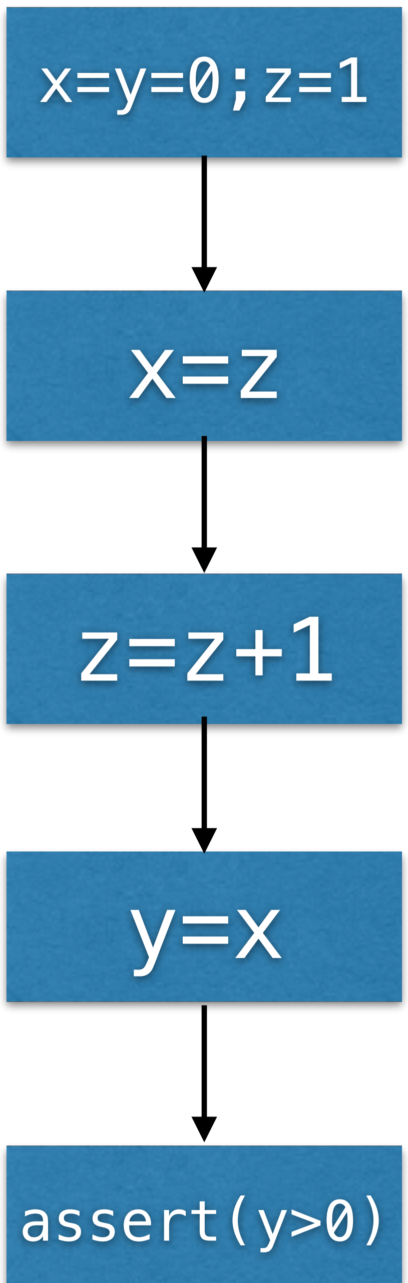
x	[1,+∞]
y	[0,0]

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,+∞]
z	[2,2]

fail to prove

FI : {x}

x	[0,+∞]

Hard Search Problem

- Intractably large space, if not infinite
  - 2^{Var} different abstractions for FS
- Most of them are too imprecise or costly
  - $P(\{x,y,z\}) = \{\cancel{\emptyset}, \cancel{\{x\}}, \cancel{\{y\}}, \cancel{\{z\}}, \{x,y\}, \cancel{\{y,z\}}, \cancel{\{x,z\}}, \{x,y,z\}\}$

Our Research

- How to automatically find a good abstraction?
  - pre-analysis approach [PLDI'14, TOPLAS'16]

pre-analysis $\xrightarrow{\{x,y,z,\ldots\}}$ main analysis

  - data-driven approaches [OOPSLA'15, SAS'16, APLAS'16]

GitHub

sourceforge

Bitbucket

learn a good strategy from data
via machine learning techniques

Our Learning Approaches

- 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)
- …

Static Analyzer

number of
proved assertions

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

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

Overall Approach

Overall Approach

- Parameterized adaptation strategy

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

Overall Approach

- Parameterized adaptation strategy

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

- Learn a good parameter W from existing codebase

$$\boxed{P_1, P_2, \ldots, P_m} \quad \Rightarrow \quad W$$

Codebase

Overall Approach

- Parameterized adaptation strategy

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

- Learn a good parameter W from existing codebase

$$\boxed{P_1, P_2, \ldots, P_m} \Rightarrow W$$

Codebase

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

1. Parameterized Strategy

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

(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

(1) Features

- Represent each variable as a feature vector:

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

$$f(x) = \langle 1,0,1,0,0 \rangle$$
$$f(y) = \langle 1,0,1,0,1 \rangle$$
$$f(z) = \langle 0,0,1,1,0 \rangle$$

(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

$$\boxed{P_1, P_2, \ldots, P_m} \Rightarrow w$$

Codebase

- Solve the optimization problem:

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

Learning via Random Sampling

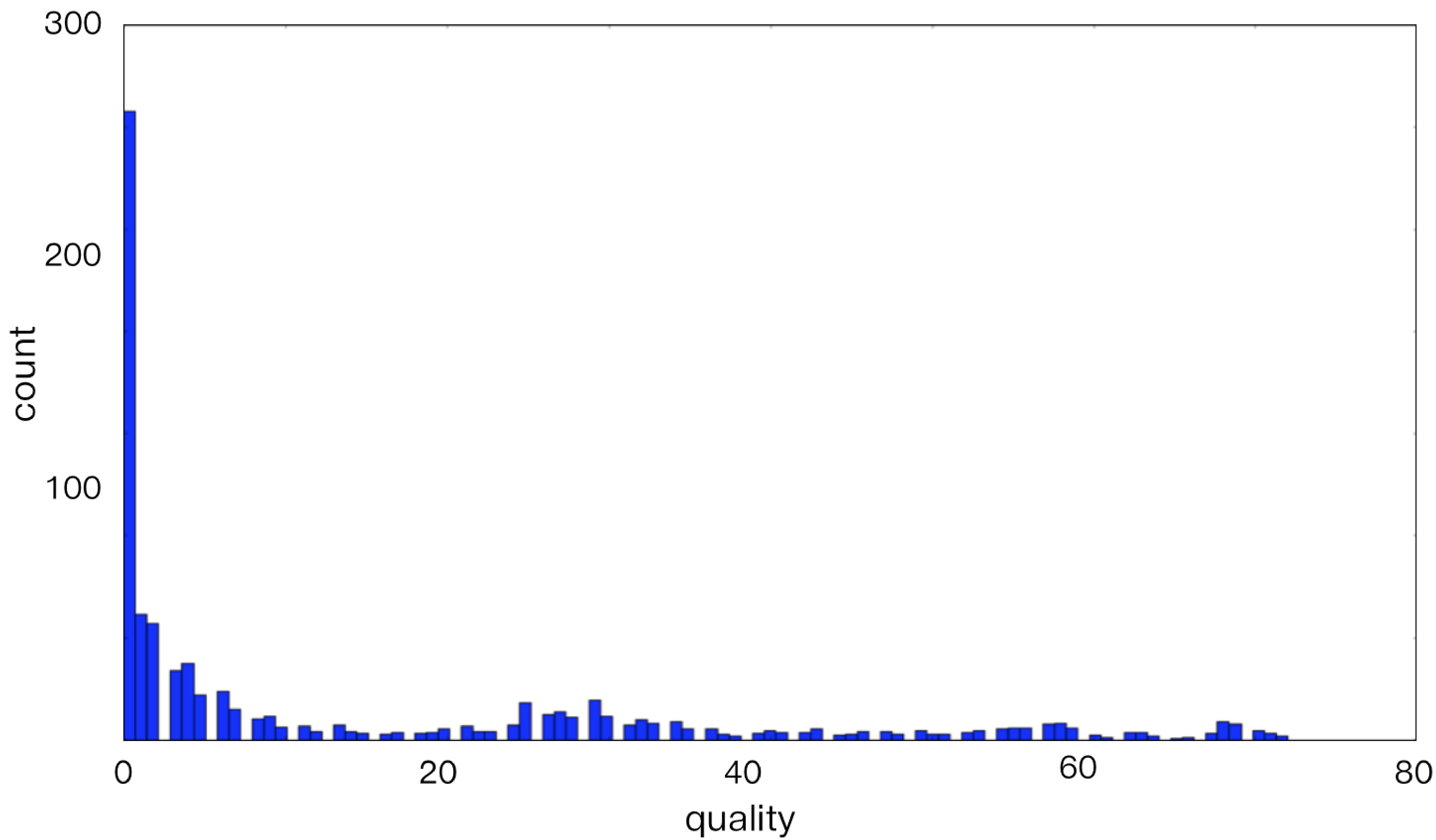
repeat N times

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

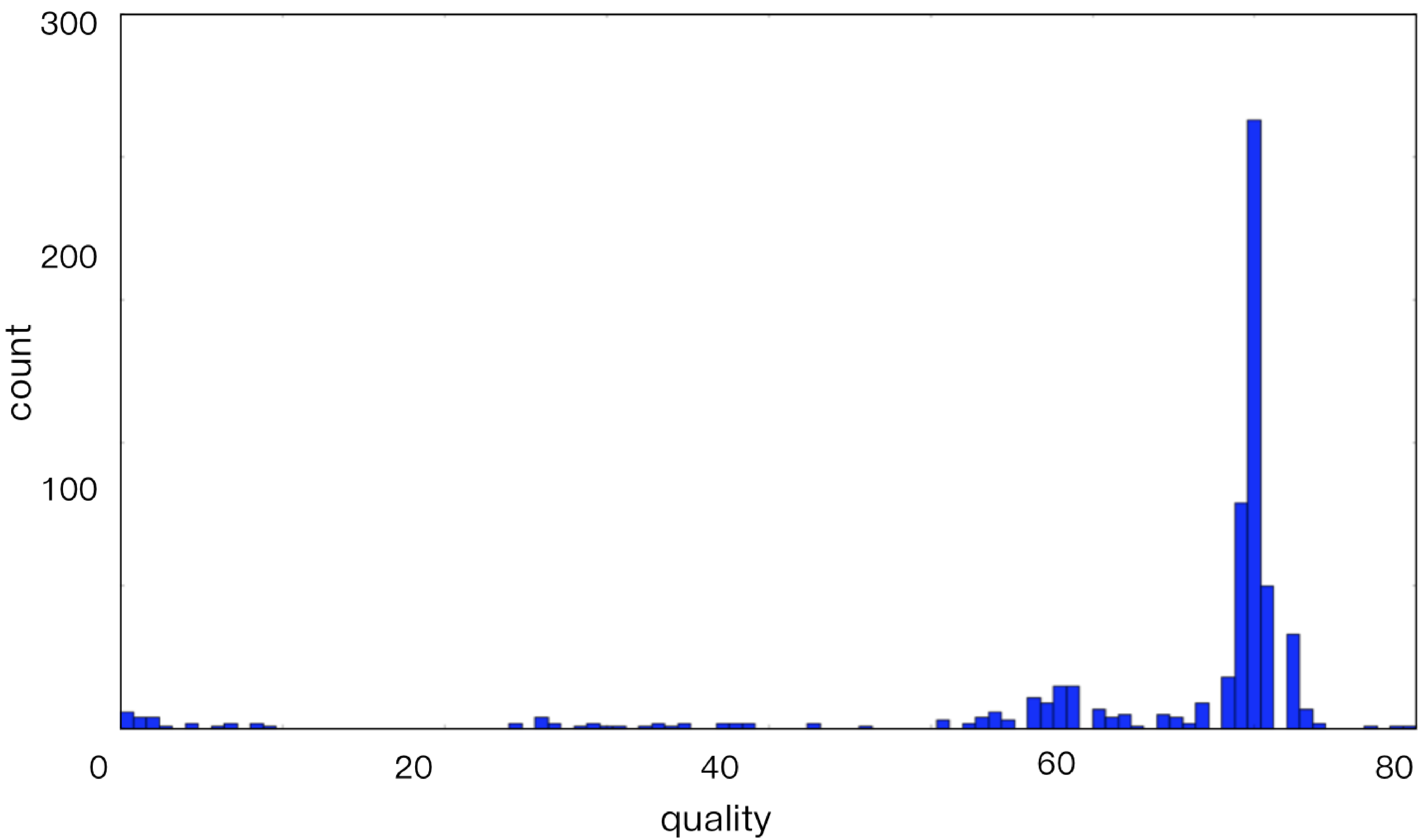
  evaluate $\sum_{P_i} F(P_i, S_{\mathbf{w}}(P_i))$

  update the probabilistic model

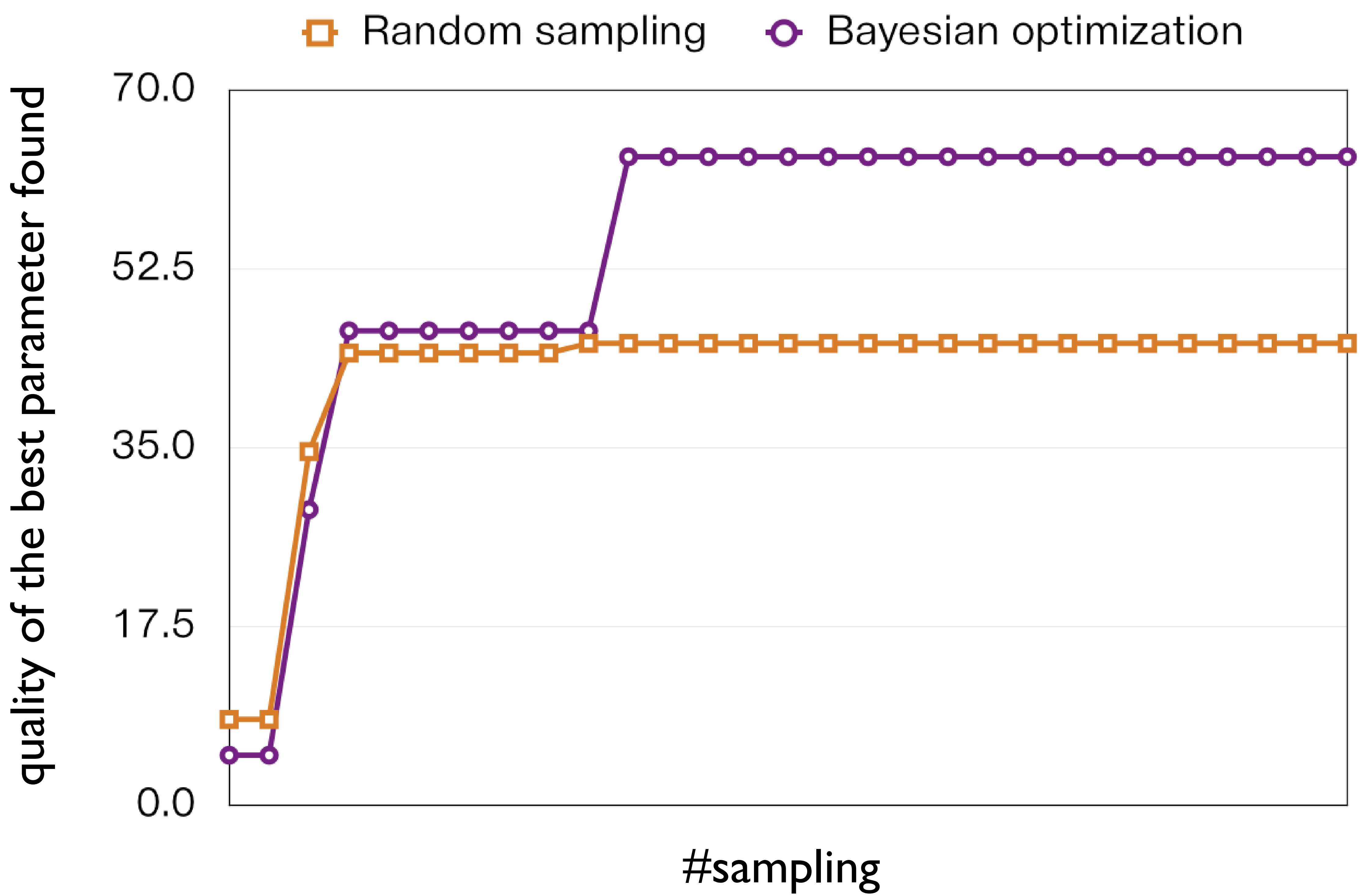
return best w found

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

Learning via Bayesian Optimization



Random Sampling vs Bayesian Optimization

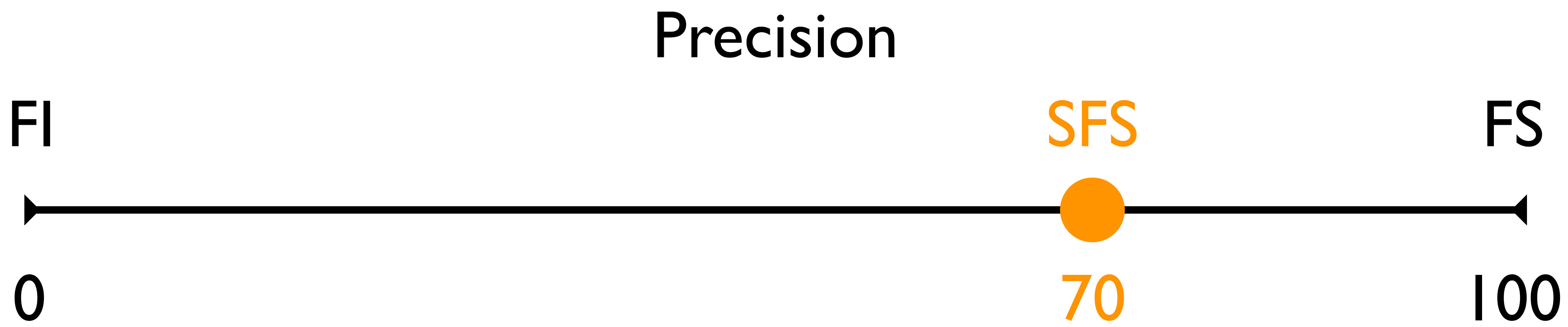


Effectiveness

- Implemented in Sparrow, an interval analyzer for C
- Evaluated on 30 open-source programs
  - 20 for training, 10 for testing

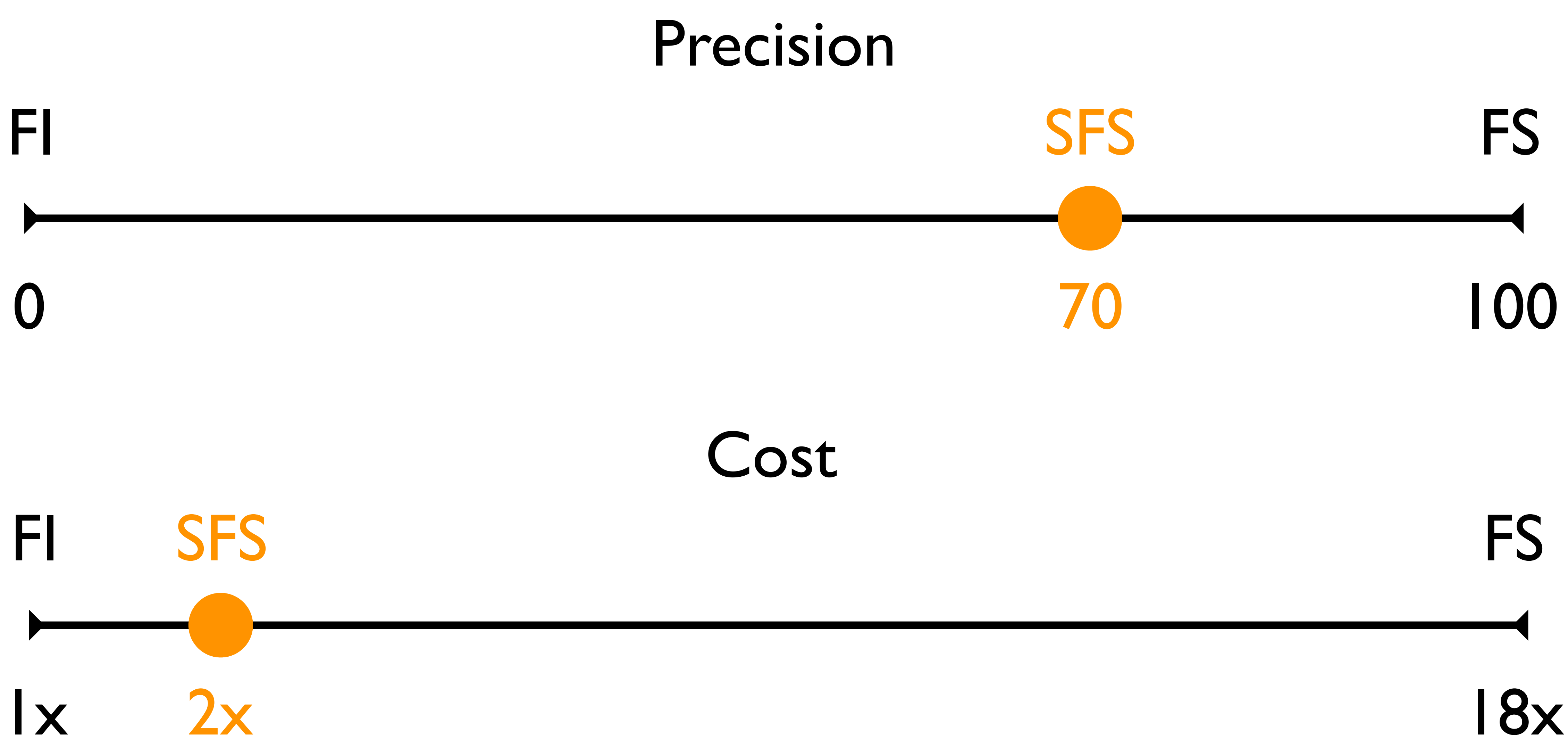
Effectiveness

- Implemented in Sparrow, an interval analyzer for C
- Evaluated on 30 open-source programs
  - 20 for training, 10 for testing



Effectiveness

- Implemented in Sparrow, an interval analyzer for C
- Evaluated on 30 open-source programs
  - 20 for training, 10 for testing



Limitations

- While promising, the method has limitations:
  - black-box optimization is inherently inefficient
  - manual feature engineering is needed
- Follow-up work to overcome the limitations:
  - improving the efficiency [APLAS'16, SAS'16]
  - automating feature engineering [on-going]

Improving Efficiency

- A white-box optimization method [APLAS'16]

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

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

- A supervised learning method [SAS'16]

	a	—a	b	—b	c	—c	i	—i
a	★	⊤	★	⊤	⊤	⊤	★	⊤
—a	⊤	★	⊤	★	⊤	⊤	⊤	⊤
b	★	⊤	★	⊤	⊤	⊤	★	⊤
—b	⊤	★	⊤	★	⊤	⊤	⊤	⊤
c	⊤	⊤	⊤	⊤	★	⊤	⊤	⊤
—c	⊤	⊤	⊤	⊤	⊤	★	⊤	⊤
i	⊤	⊤	⊤	⊤	⊤	⊤	★	⊤
—i	⊤	★	⊤	★	⊤	⊤	⊤	★

Manual Feature Engineering

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

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	$1 \land 8 \land (11 \lor 12)$
	35	$2 \land 8 \land (11 \lor 12)$
	36	$1 \land (11 \lor 12) \land (19 \lor 20)$
	37	$2 \land (11 \lor 12) \land (19 \lor 20)$
	38	$1 \land (11 \lor 12) \land (15 \lor 16)$
	39	$2 \land (11 \lor 12) \land (15 \lor 16)$
	40	$(11 \lor 12) \land 29$
	41	$(15 \lor 16) \land 29$
	42	$1 \land (19 \lor 20) \land 33$
	43	$2 \land (19 \lor 20) \land 33$
	44	$1 \land (19 \lor 20) \land \neg 33$
	45	$2 \land (19 \lor 20) \land \neg 33$

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	$2 \land (21 \lor 22) \land (14 \lor 15)$
	31	$2 \land (21 \lor 22) \land \neg(14 \lor 15)$
	32	$2 \land 23 \land (14 \lor 15)$
	33	$2 \land 23 \land \neg(14 \lor 15)$
	34	$2 \land (21 \lor 22) \land (16 \lor 17)$
	35	$2 \land (21 \lor 22) \land \neg(16 \lor 17)$
	36	$2 \land 23 \land (16 \lor 17)$
	37	$2 \land 23 \land \neg(16 \lor 17)$
	38	$(21 \lor 22) \land \neg 23$

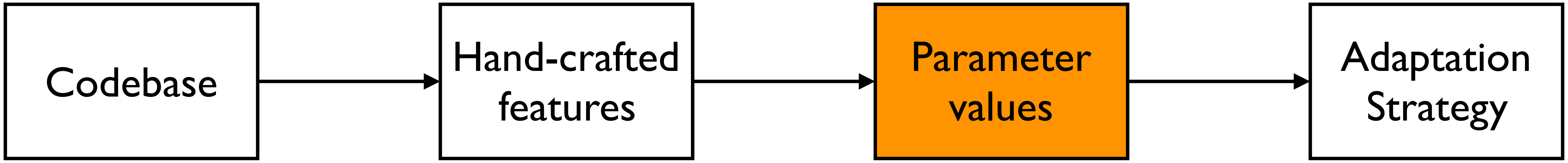
context-sensitivity

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	$(1 \lor 2) \land 3$
	15	$(1 \lor 2) \land (4 \lor 5 \lor 6 \lor 7)$
	16	$(1 \lor 2) \land (8 \lor 9)$
	17	$(1 \lor 2) \land 11$
	18	$(1 \lor 2) \land 12$
	19	$13 \land 3$
	20	$13 \land (4 \lor 5 \lor 6 \lor 7)$
	21	$13 \land (8 \lor 9)$
	22	$13 \land 11$
	23	$13 \land 12$

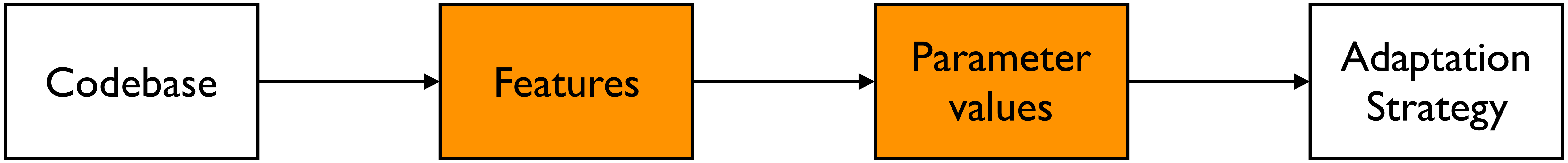
widening thresholds

Automatic Feature Generation

Before [OOPSLA'15,SAS'16,APLAS'16]



New method



Partial Flow-Sensitive Analysis

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

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

Partial Flow-Sensitive Analysis

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

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

Partial 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	$\{x \mapsto [0,0], y \mapsto [0,0]\}$	
2	$\{x \mapsto [0,0], y \mapsto [1,1]\}$	
3	$\{x \mapsto [0,0], y \mapsto [1,1]\}$	$\{z \mapsto [0,0], w \mapsto [0,0]\}$
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:

$$\{(q_i, b_i)\}_{i=1}^{n}$$

Learning a Query Classifier

Standard binary classification:

$$\{(q_i, b_i)\}_{i=1}^{n} \longrightarrow \{(v_i, b_i)\}_{i=1}^{n}$$

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

transform to
feature vectors

Learning a Query Classifier

Standard binary classification:

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

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

transform to
feature vectors

apply
standard learning
algorithms

Learning a Query Classifier

Standard binary classification:

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

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

transform to feature vectors

apply standard learning algorithms

- Success relies on how we convert queries to feature vectors
- This feature engineering has been done manually

Conversion from Queries to Feature Vectors

- A set of feature features $\Pi = \{\pi_1, \ldots, \pi_k\}$
  - a feature encodes a property about queries
- A procedure to check whether a query satisfies a feature

$$\text{match} : Query \times Feature \to \mathbb{B}$$

- The feature vector of a query q:

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

Automatic Feature Generation

- Generate *feature programs* by running reducer
  - small pieces of code that minimally describe when it is worth increasing the precision
- Represent them by *abstract data-flow graphs*
  - generalized form of feature programs

Generating Feature Programs

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

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

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

- By running a program reducer: e.g., C-Reduce [PLDI'12]

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

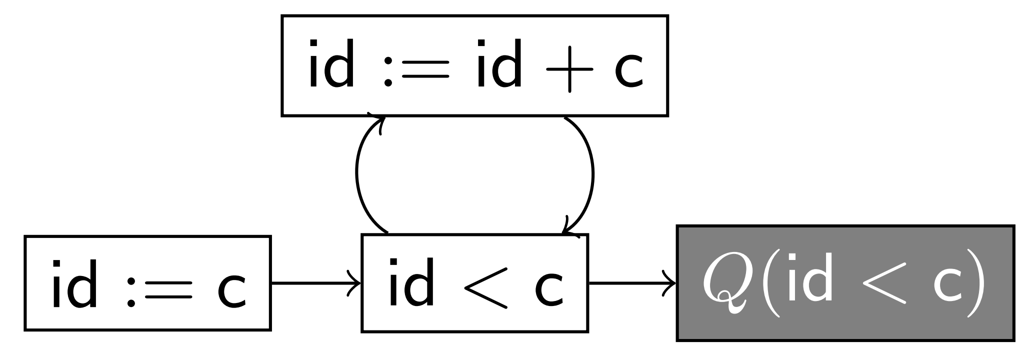
- Feature-preserving condition:

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

Generalize to Abstract Data-Flow Graphs

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

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



- The right level of abstraction depends on an analysis
- We choose the best abstraction using a combination of searching and cross-validation

Feature Generation

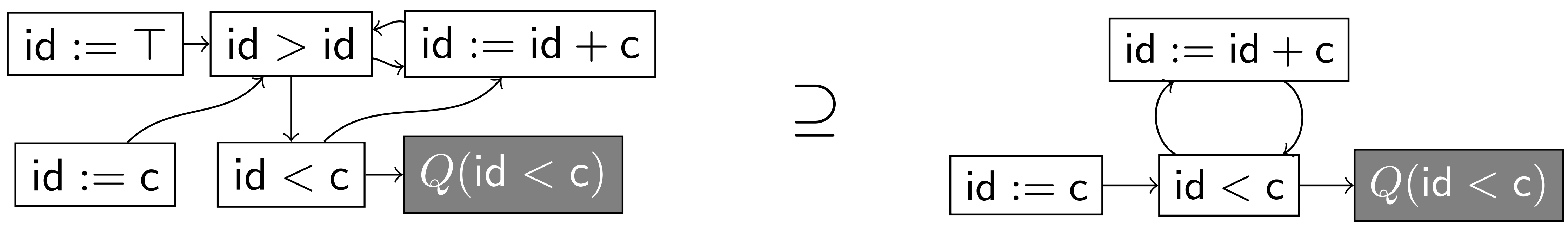
- Apply the method on codebases:

$$\boxed{P_1, P_2, \ldots, P_m} \quad \Rightarrow \quad \Pi = \{\pi_1, \ldots, \pi_k\}$$

Codebase

Matching Algorithm

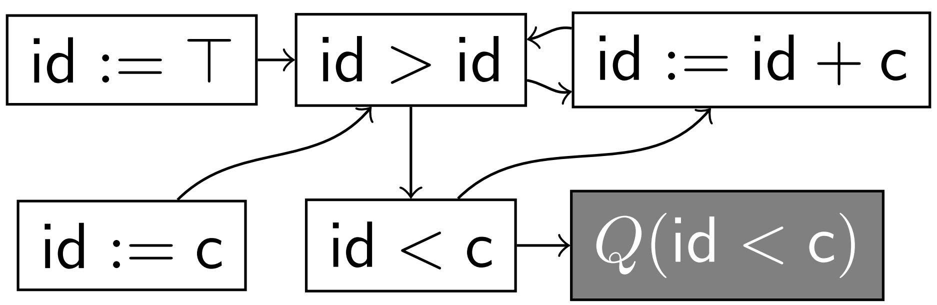
$$\text{match} : \mathit{Query} \times \mathit{Feature} \to \mathbb{B}$$



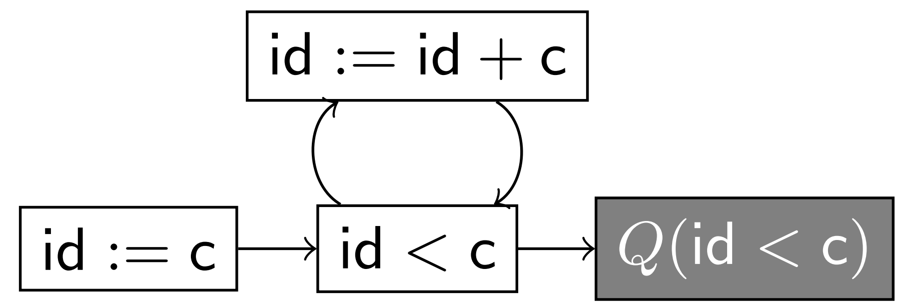

```
a = 0; b = 0;
while (1) {
 b = unknown();
 if (a > b)
   if (a < 3)
     assert (a < 5);
 a++;
}
```

Matching Algorithm

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



$\supseteq$




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

Subgraph inclusion:

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

Learning a Query Classifier

$P_1, P_2, \ldots, P_m$

Codebase

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

$$\Downarrow$$

$$\{(v_i, b_i)\}_{i=1}^{n}$$

$$\Downarrow$$

$$\mathcal{C} : \mathbb{B}^k \to \mathbb{B}$$

Experiments

Effectiveness of partially flow-sensitive analysis

Trial	Query Prediction		Analysis								Comparison		
			Prove			Sec						Oh et al. [38]	
	Precision	Recall	FIi	FSi	Ours	FIi	FSi	Ours	Quality	Cost	Self	Quality	Cost
1	92.6 %	77.9 %	5,340	6,053	5,973	38.2	564.0	55.3	88.7 %	1.4x	88.7 %	85.2%	1.5x
2	78.8 %	73.3 %	2,972	3,373	3,262	16.3	460.5	25.7	72.3 %	1.5x	72.0 %	41.6%	1.9x
3	66.7 %	73.3 %	3,984	4,668	4,559	27.3	1,635.6	176.2	84.0 %	6.4x	82.7 %	89.9%	3.2x
4	88.7 %	68.8 %	4,600	5,450	5,307	38.1	688.2	59.6	83.1 %	1.5x	83.5 %	60.7%	1.9x
5	89.9 %	79.4 %	2,517	2,971	2,945	10.9	325.9	18.9	94.2 %	1.7x	94.0 %	47.8%	2.1x
TOTAL	81.5 %	73.9 %	19,413	22,515	22,046	131.1	3,674.4	336.0	**84.8** %	**2.5x**	84.6 %	68.4%	2.1x

Effectiveness of partially relational analysis

Trial	Query Prediction		Analysis								Comparison		
			Prove			Sec						Heo et al. [21]	
	Precision	Recall	FSi	IMPCT	Ours	FSi	IMPCT	Ours	Quality	Cost	Self	Quality	Cost
1	74.8 %	81.3 %	3,678	3,806	3,789	140.7	389.8	189.5	86.7 %	1.3 x	54.2 %	100.0 %	3.0 x
2	84.1 %	82.6 %	5,845	6,004	5,977	613.5	18,022.9	775.5	83.0 %	1.3 x	65.5 %	30.2 %	0.9 x
3	82.8 %	73.0 %	1,926	2,079	2,036	315.2	2,396.9	460.2	71.9 %	1.5 x	95.7 %	92.2 %	1.1 x
4	77.6 %	85.2 %	2,221	2,335	2,313	72.7	495.1	141.2	80.7 %	1.9 x	67.2 %	100.0 %	2.0 x
5	71.6 %	78.4 %	2,886	2,962	2,946	148.9	557.2	210.2	78.9 %	1.4 x	59.9 %	96.1 %	2.3 x
TOTAL	79.0 %	79.9 %	16,556	17,186	17,061	1,291.0	21,861.9	1,776.6	**80.2** %	**1.4** x	67.7 %	80.0 %	1.4 x

Summary

- Choosing a good abstraction is a key challenge in static program analysis
- New data-driven approach is promising
- Further information:

http://prl.korea.ac.kr

Summary

- Choosing a good abstraction is a key challenge in static program analysis
- New data-driven approach is promising
- Further information:

http://prl.korea.ac.kr

Thank you