Machine-Learning-Guided Adaptive Program Analysis

Hakjoo Oh

Korea University

Kihong Heo
Kwangkeun Yi

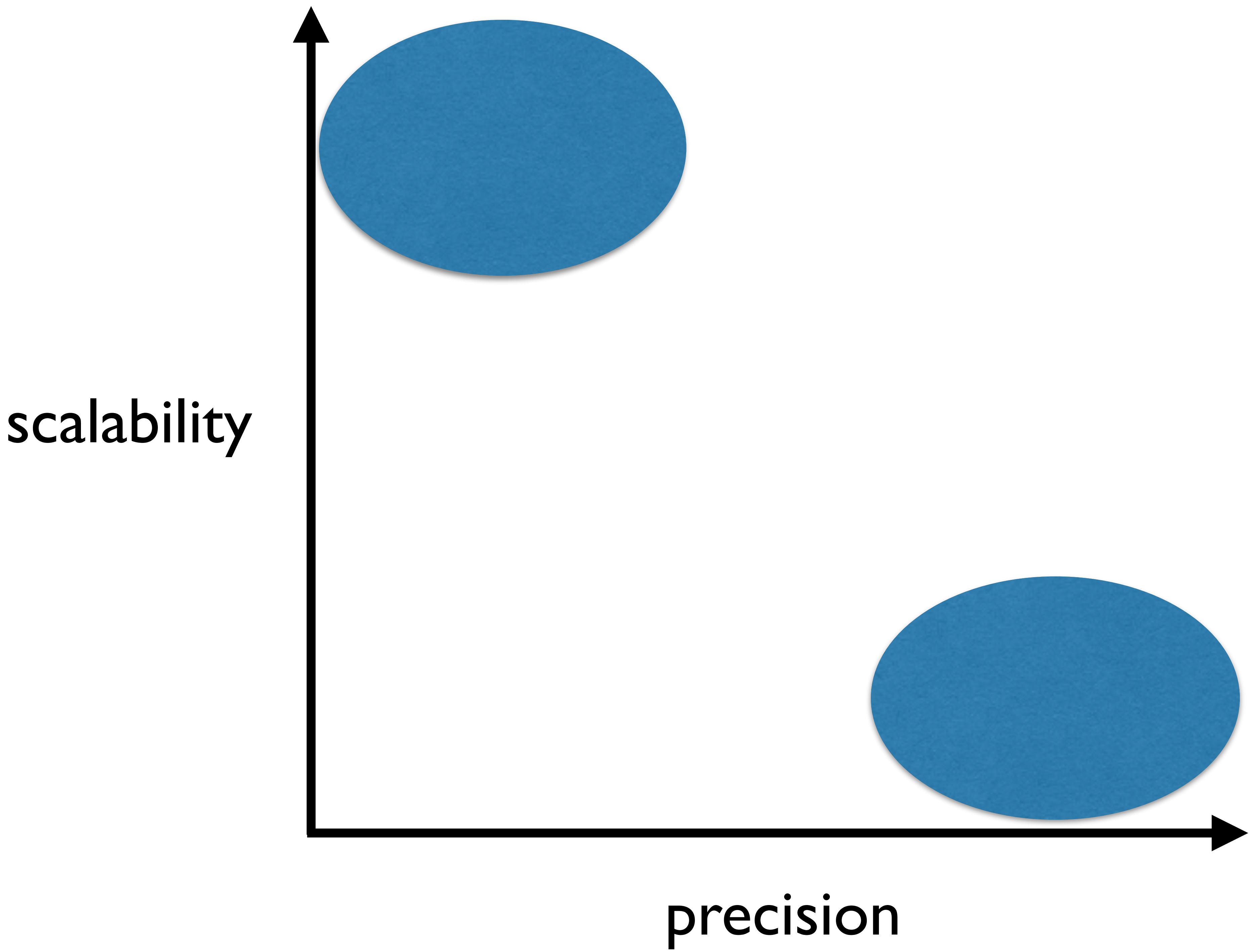
Seoul National University

Hongseok Yang

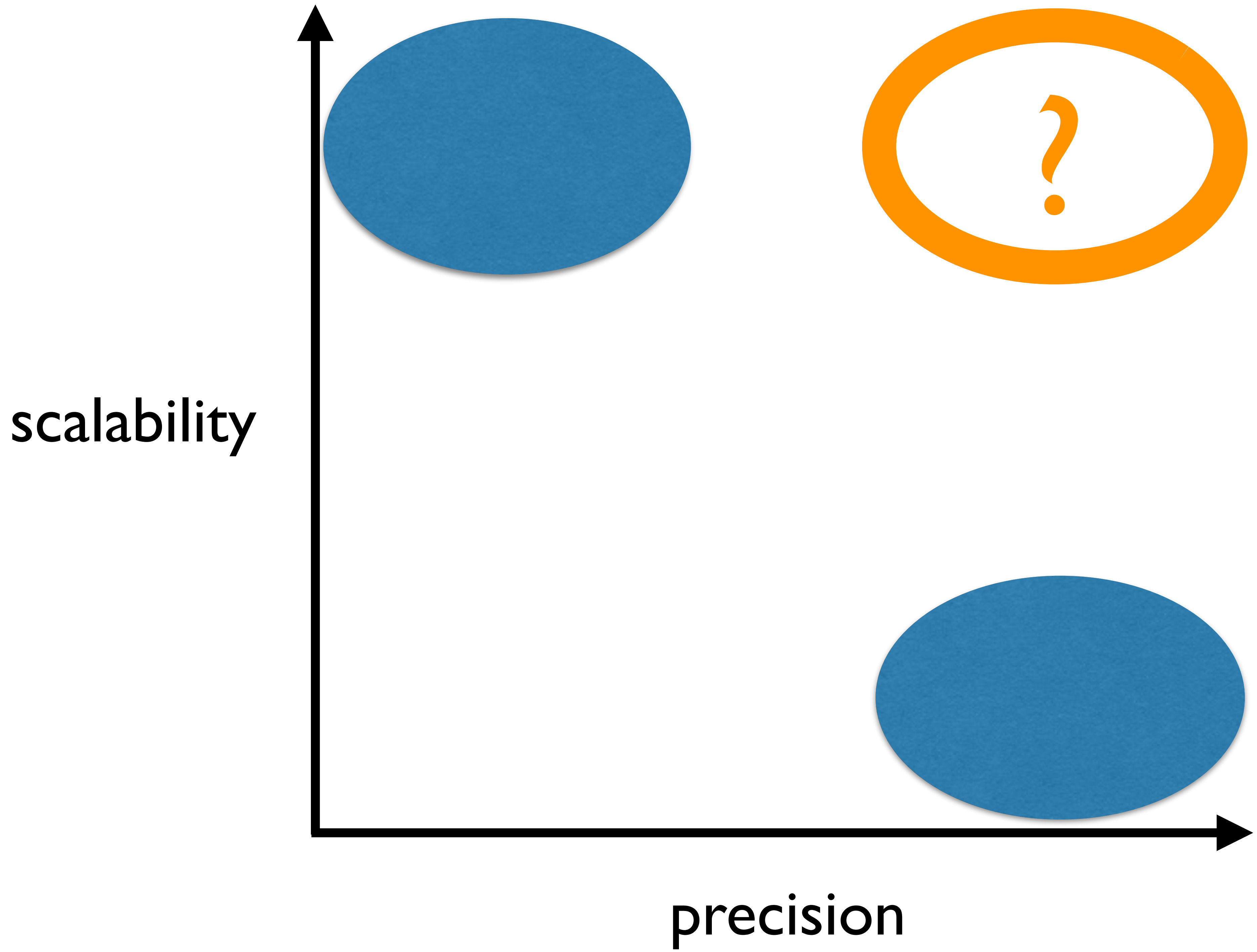
Oxford University

7 September 2016
TAPAS 2016 @ Edinburgh, Scotland

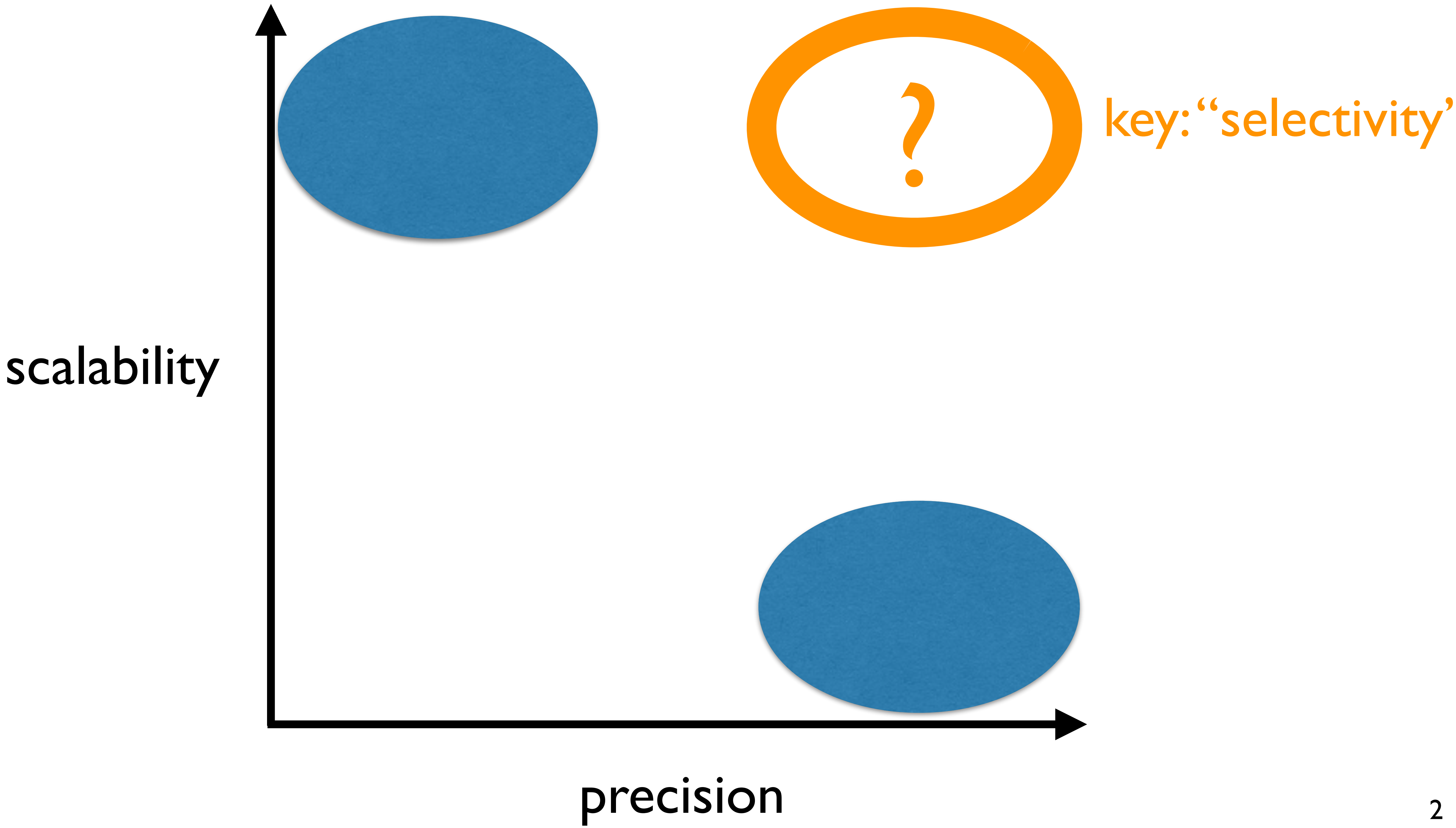
Challenge in Static Analysis



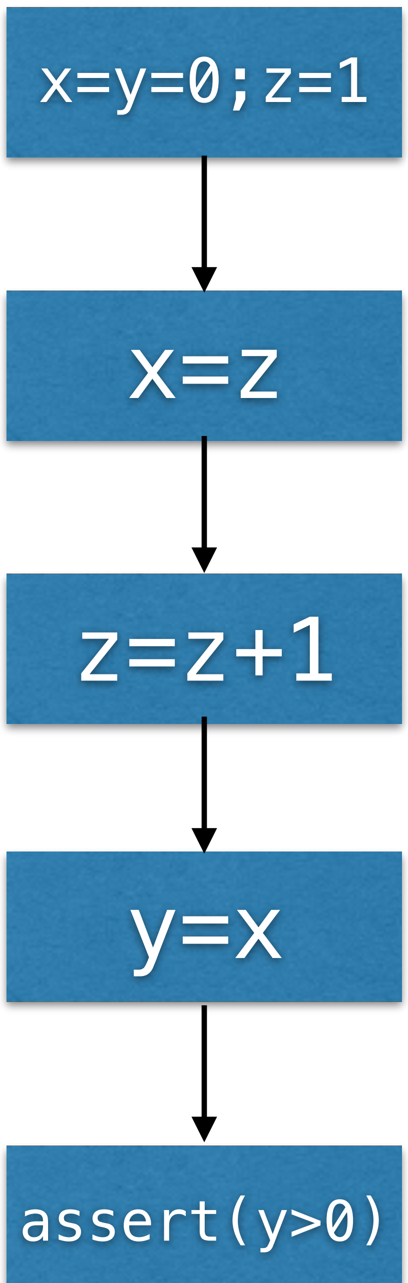
Challenge in Static Analysis



Challenge in Static Analysis



Flow-Sensitivity



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

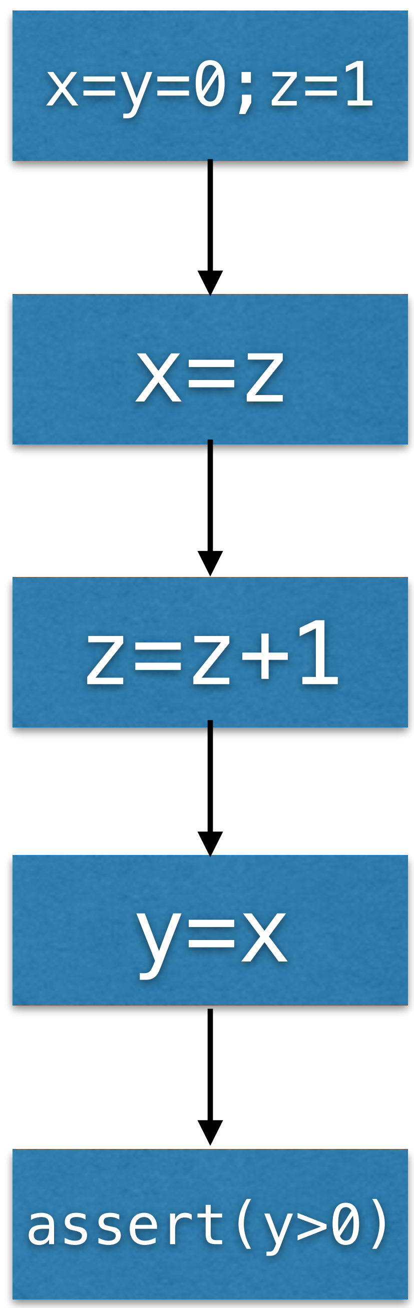
x	[1,1]
y	[0,0]
z	[1,1]

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

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

precise but costly

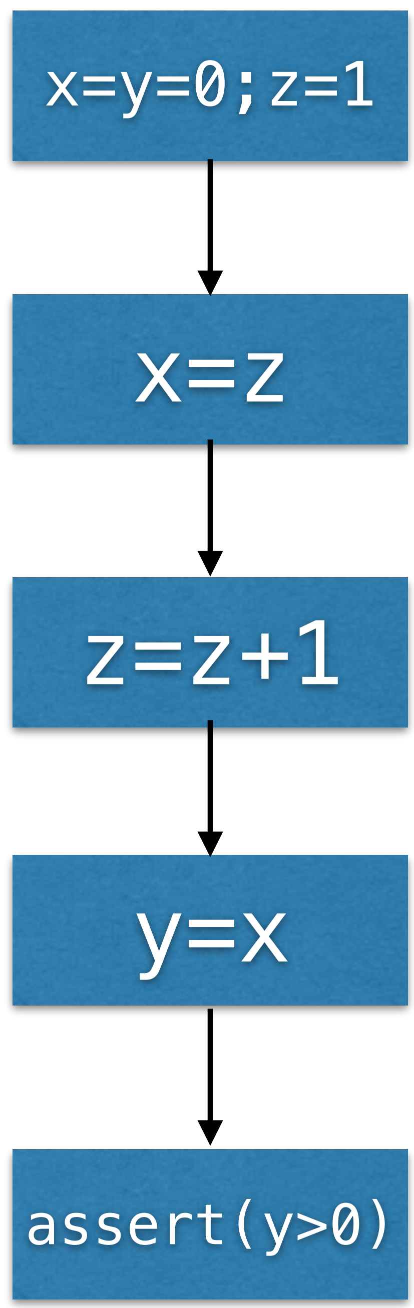
Flow-Insensitivity



x	$[0,+\infty]$
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]

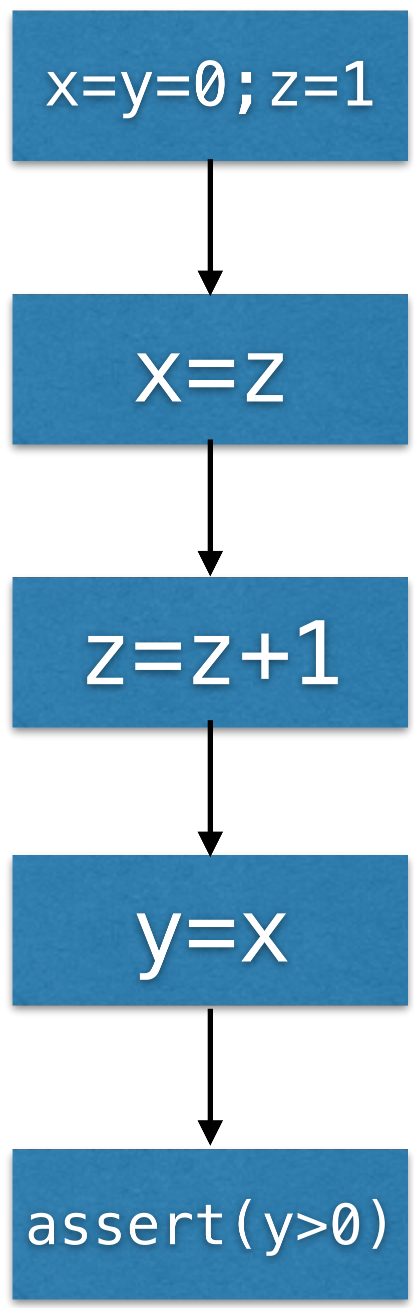
x	[1,+∞]
y	[0,0]

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,+∞]
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 [PLDI'14, TOPLAS'16]

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

  - machine learning techniques [OOPSLA'15, SAS'16, APLAS'16]

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

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

number of
proved assertions

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

OOPSLA'15

Our Learning Approach

OOPSLA'15

Our Learning Approach

- Parameterized adaptation strategy

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

OOPSLA'15

Our Learning 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

OOPSLA'15

Our Learning Approach

- Parameterized adaptation strategy

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

- Learn a good parameter W from existing codebase

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

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

(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

$$P_1, P_2, \ldots, P_m \Rightarrow w$$

Codebase

- Solve the optimization problem:

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

Learning via Random Sampling

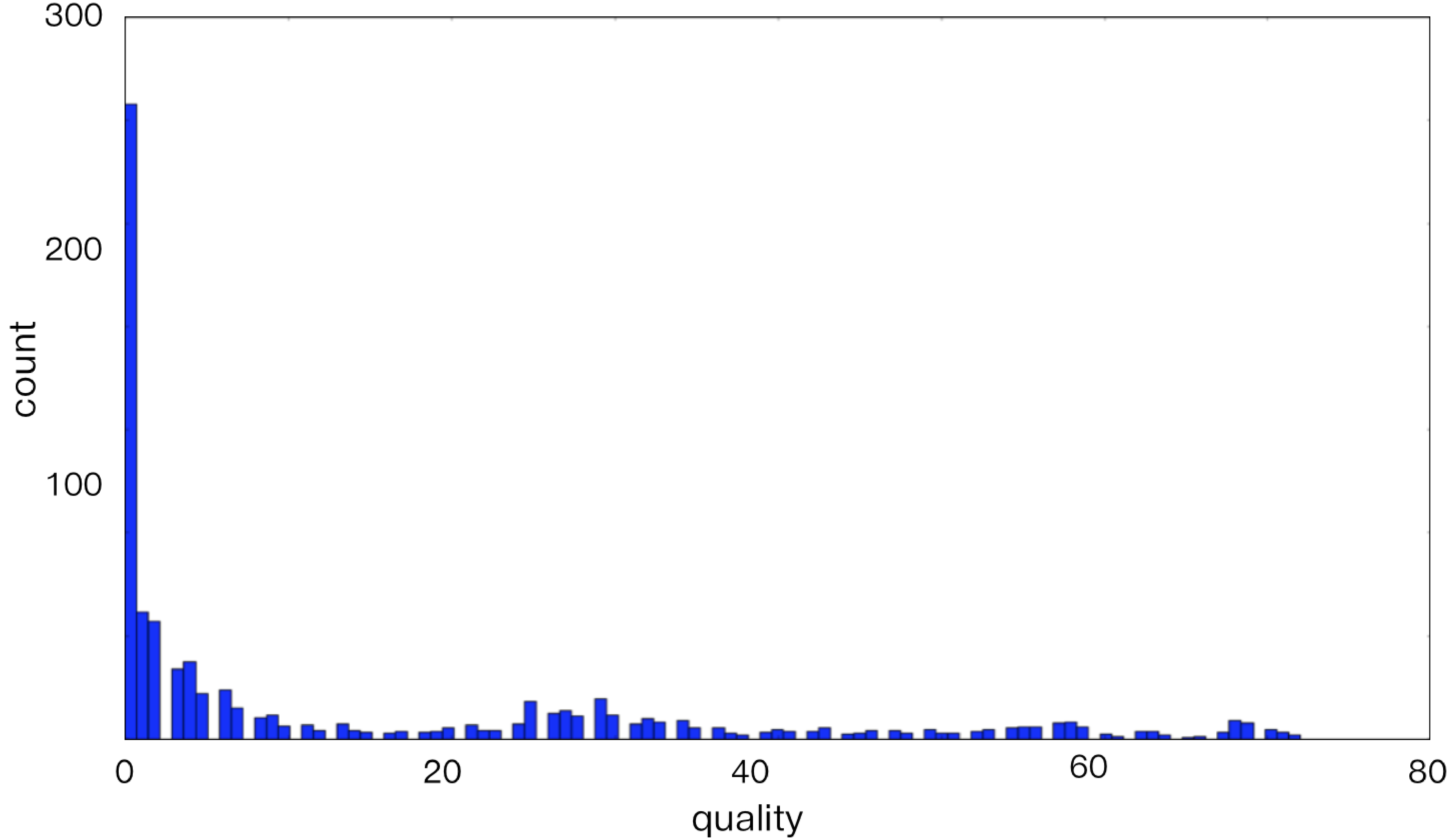
repeat N times

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

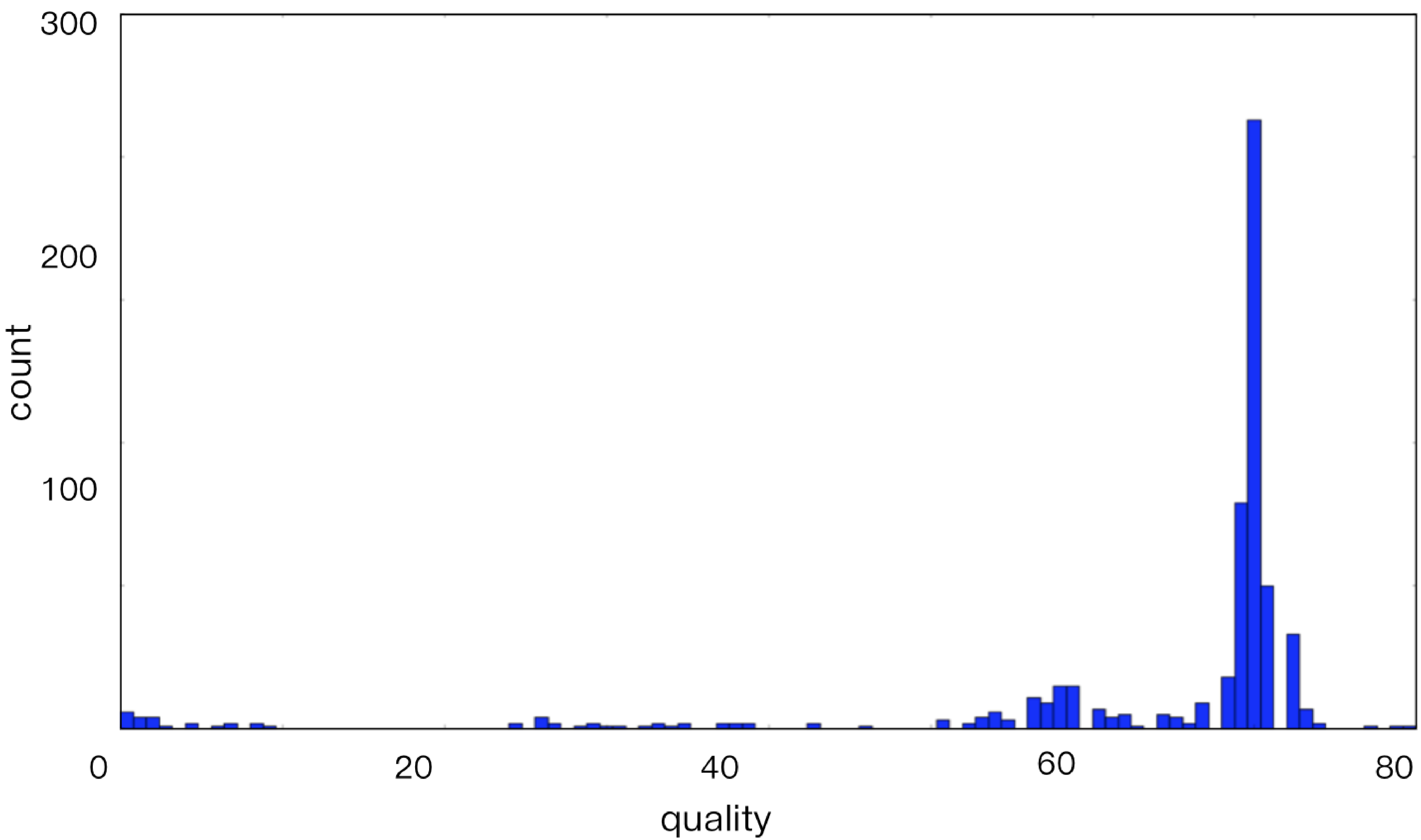
  evaluate $\sum_{P_i} F(P_i, S_{\mathbf{w}}(P_i))$

  update the probabilistic model

return best w found

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

Learning via Bayesian Optimization

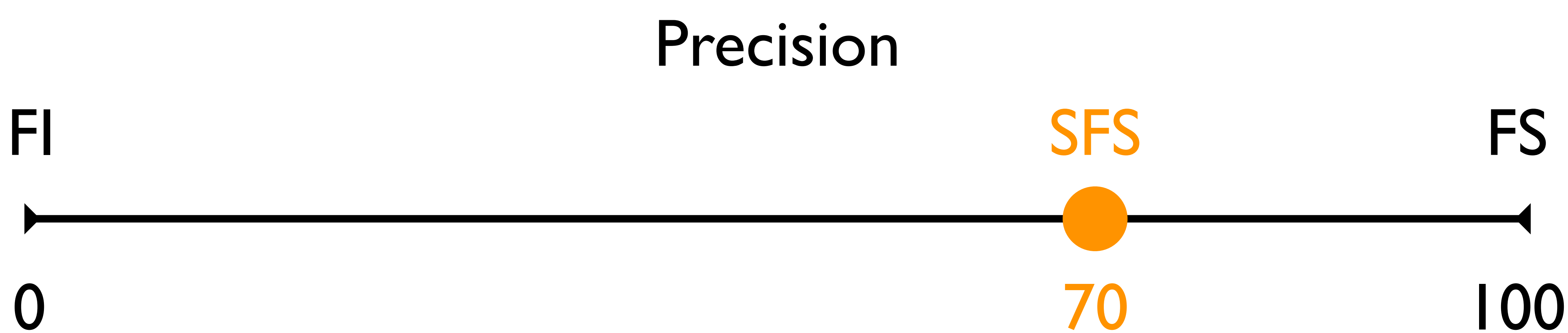


Effectiveness

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

Effectiveness

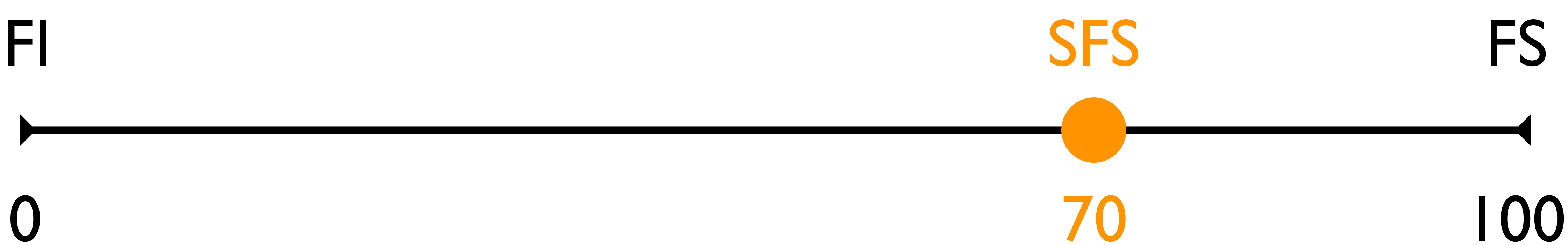
- Implemented in Sparrow, an interval analyzer for C
- Evaluated on 30 open-source programs
  - 20 for training, 10 for testing



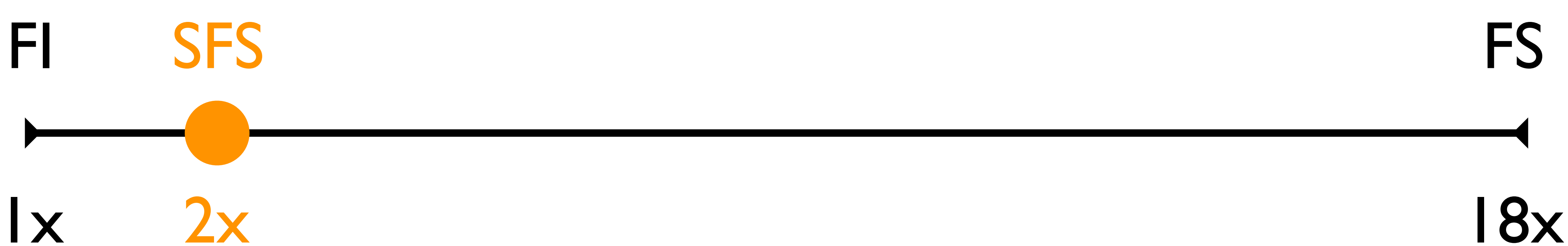
Effectiveness

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

Precision



Cost



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	$\star$	$\top$	$\star$	$\top$	$\top$	$\top$	$\star$	$\top$
-a	$\top$	$\star$	$\top$	$\star$	$\top$	$\top$	$\top$	$\top$
b	$\star$	$\top$	$\star$	$\top$	$\top$	$\top$	$\star$	$\top$
-b	$\top$	$\star$	$\top$	$\star$	$\top$	$\top$	$\top$	$\top$
c	$\top$	$\top$	$\top$	$\top$	$\star$	$\top$	$\top$	$\top$
-c	$\top$	$\top$	$\top$	$\top$	$\top$	$\star$	$\top$	$\top$
i	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\star$	$\top$
-i	$\top$	$\star$	$\top$	$\star$	$\top$	$\top$	$\top$	$\star$

Manual Feature Engieering

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

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 ≤ (e.g., x ≤ c)
	7	used with ≥ (e.g., x ≥ 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

on-going

Automating Feature Engineering

Before [OOPSLA’15,SAS’16,APLAS’16]

Codebase → Hand-crafted features → Parameter values → Adaptation Strategy

New method

Codebase → Features → Parameter values → Adaptation Strategy on-going

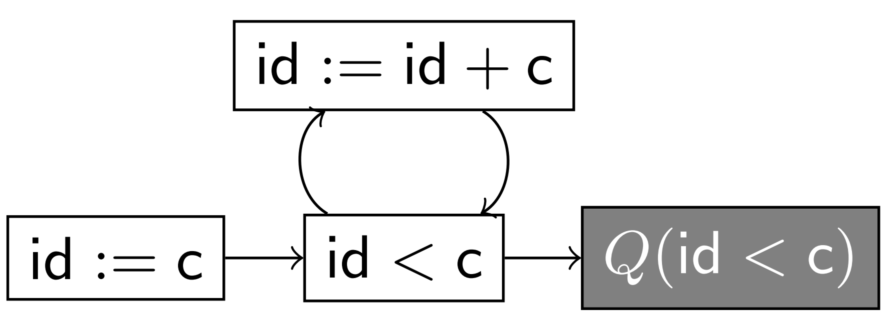
Key Ideas

- Use a program reducer to generate ***feature programs*** that capture the key reason why FS succeeds but FI fails.

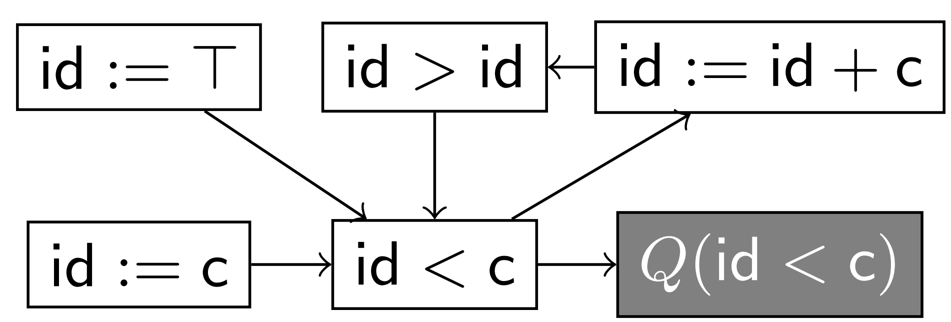
```
int j = 0;
main() {
  j++;
  assert (j>0);
}
```

```
main() {
 for (int i=1;i<50; i++) {
    assert (i<100);
  }
}
```

- Generalize the programs by abstract data flow graphs



$\overset{?}{\subseteq}$



Summary

- Challenges in selective static analysis
- Using machine learning is promising
    - [OOPSLA'15, SAS'16, APLAS'16,…]
    - flow-sensitivity, context-sensivitiy, relational domain, widening thresholds, soudness, etc
- Generally applicable beyond static analysis
    - e.g., concolic testing

Summary

- Challenges in selective static analysis
- Using machine learning is promising
  - [OOPSLA'15, SAS'16, APLAS'16,…]
  - flow-sensitivity, context-sensivitiy, relational domain, widening thresholds, soudness, etc
- Generally applicable beyond static analysis
  - e.g., concolic testing

Thank you