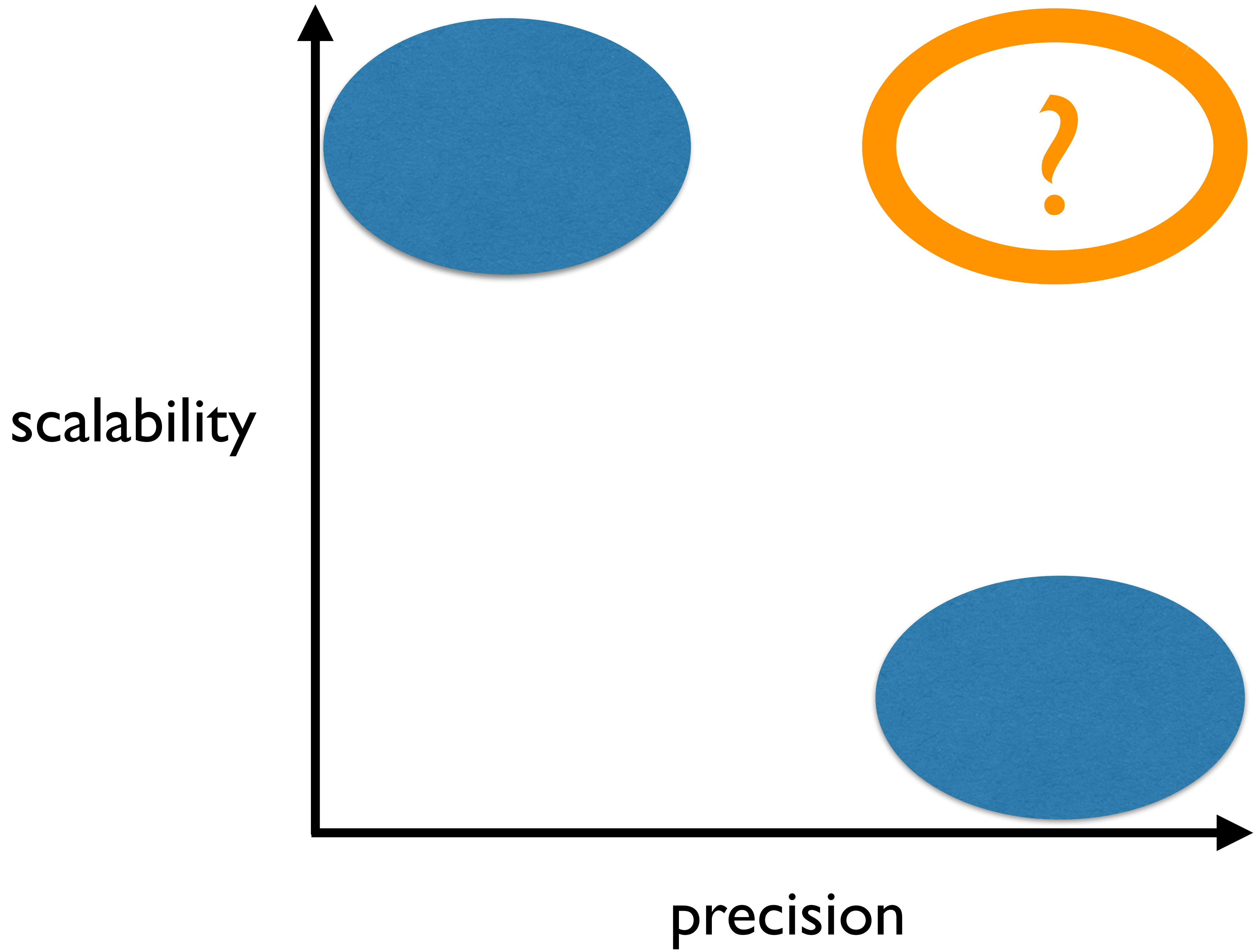
머신러닝 기반 선별적 정적 분석
Machine Learning Approaches to Selective Program Analyses

오학주
고려대학교 프로그래밍 연구실

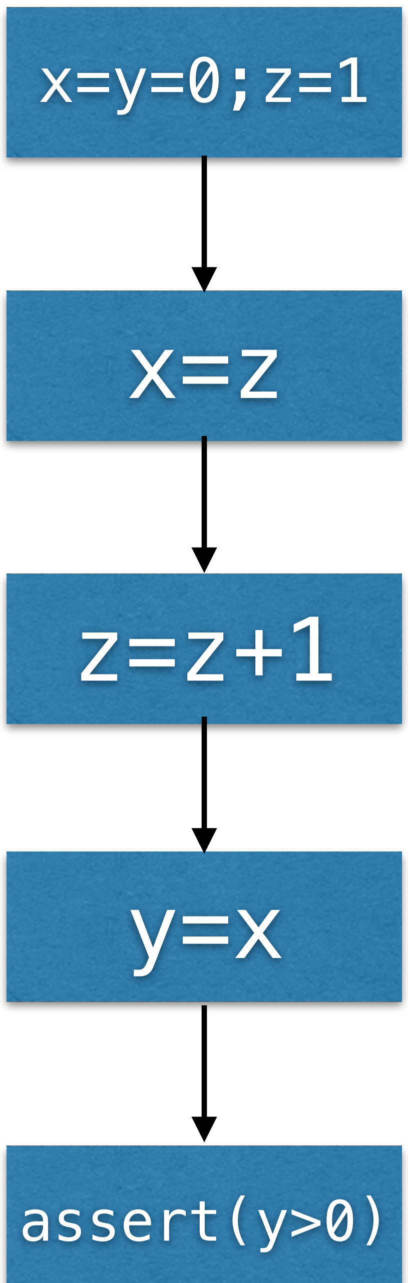
(with 채권수, 홍성준, 이민아, 양홍석, 이광근)

Feb. 17, 2016 @SIGPL Winter workshop

Scalability and Precision via Selectivity



Flow-Sensitivity



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

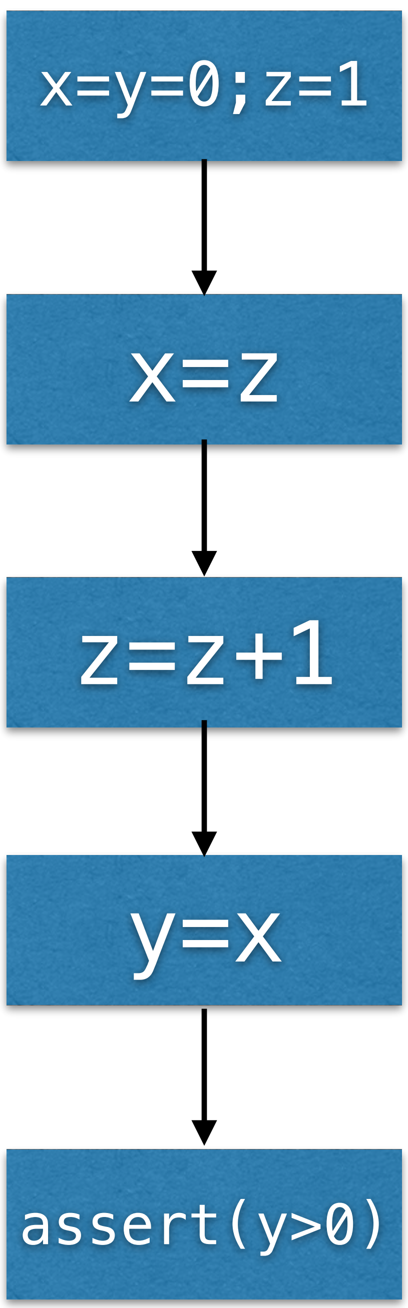
x	[1,1]
y	[0,0]
z	[1,1]

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

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

precise but costly

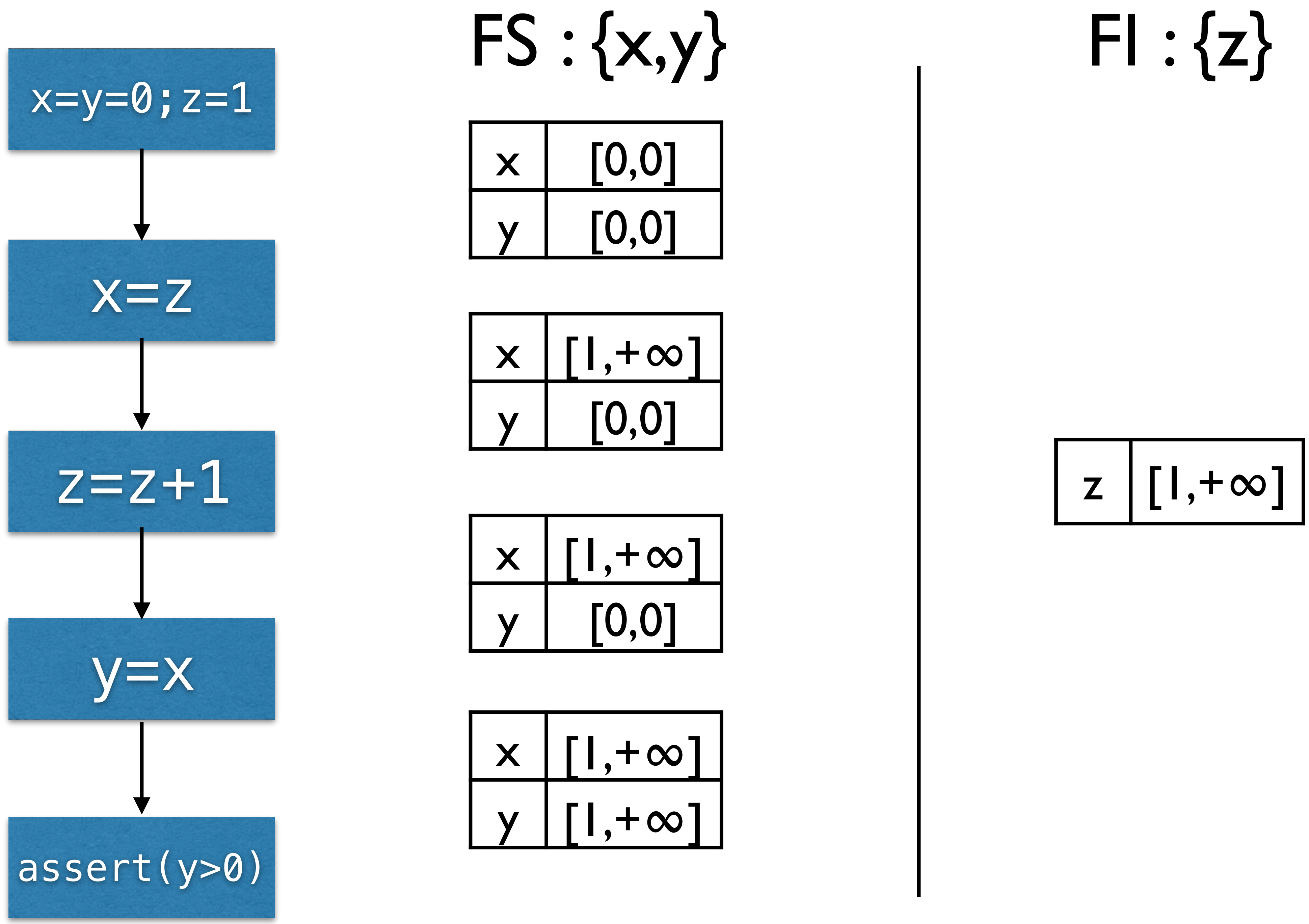
Flow-Insensitivity



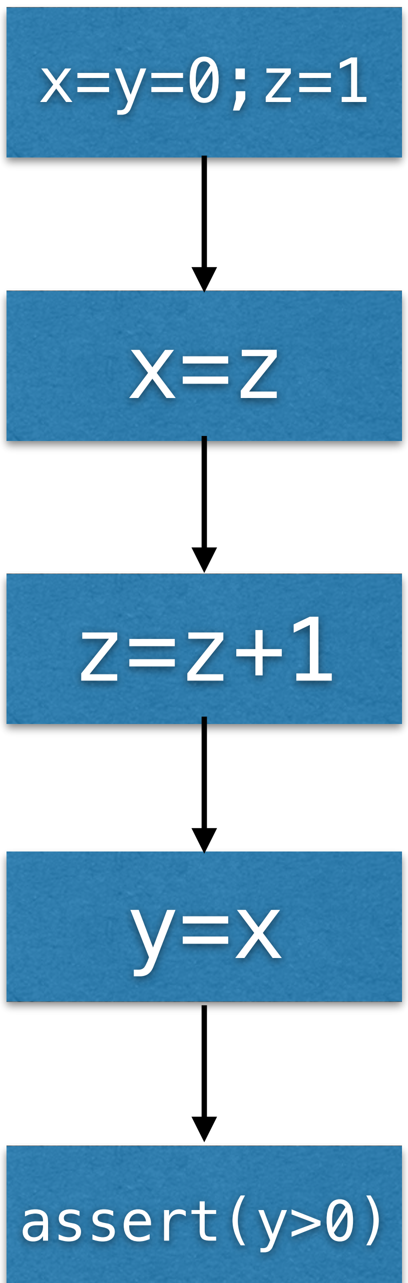
x	[0,+∞]
y	[0,+∞]
z	[1,+∞]

cheap but imprecise

Selective Flow-Sensitivity



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}) = {~~∅,{x},{y},{z}~~,{x,y},~~{y,z},{x,z}~~,{x,y,z}}

Our Research

- How to efficiently find a good abstraction?
- Two directions:
  - PL approaches [PLDI'14, TOPLAS'16]
  - ML approaches [OOPSLA'15, on-going work]

OOPSLA'15

Learning-based 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

- 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

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

Codebase

- Solve the optimization problem:

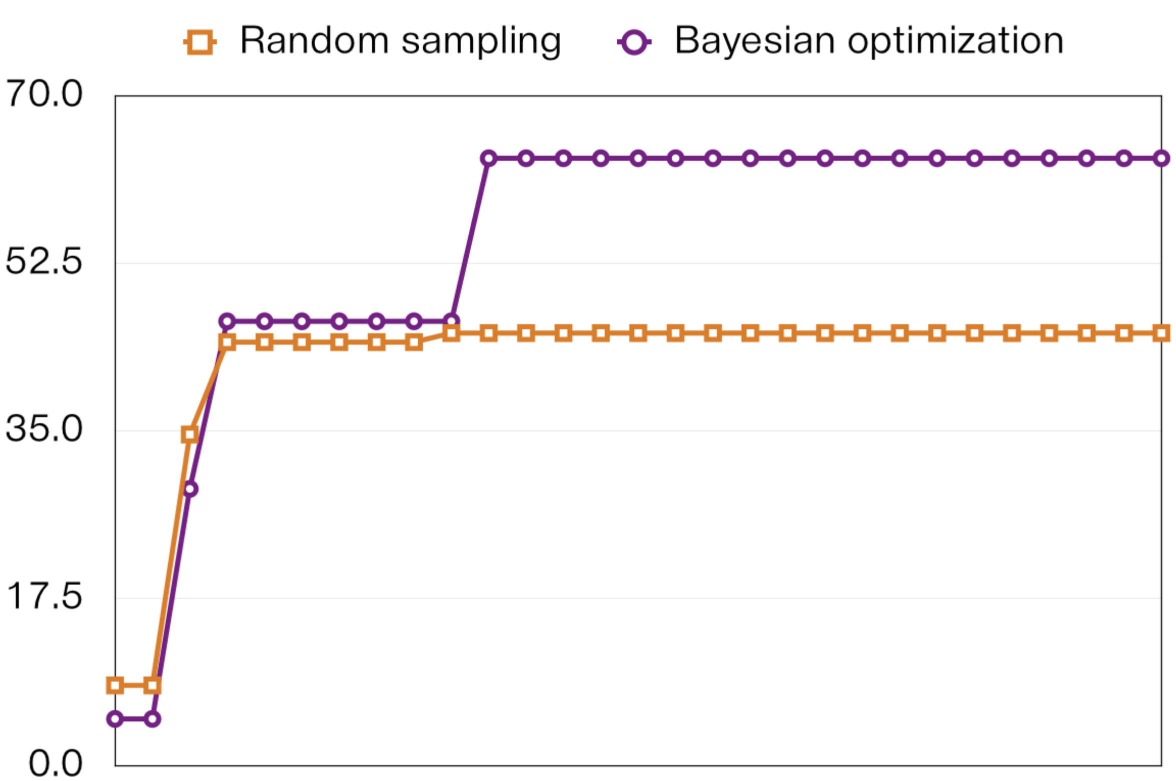
Find **w** that maximizes $\sum_{P_i} F(P_i, S_{\mathbf{w}}(P_i))$

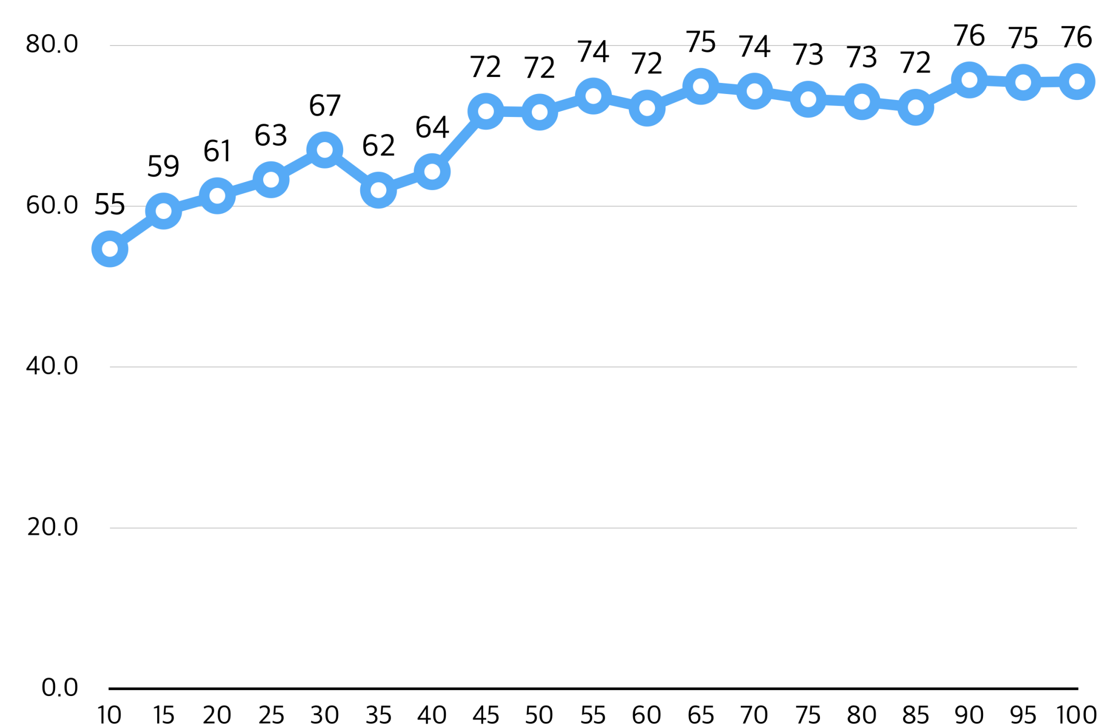
Solving the Opt. Problem

- How to solve the optimization problem efficiently?

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

- Using ideas of Bayesian optimization and ordinal optimization

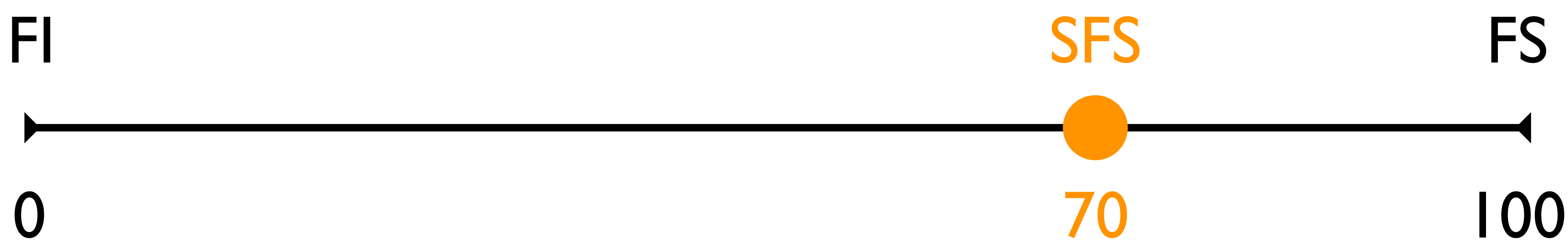




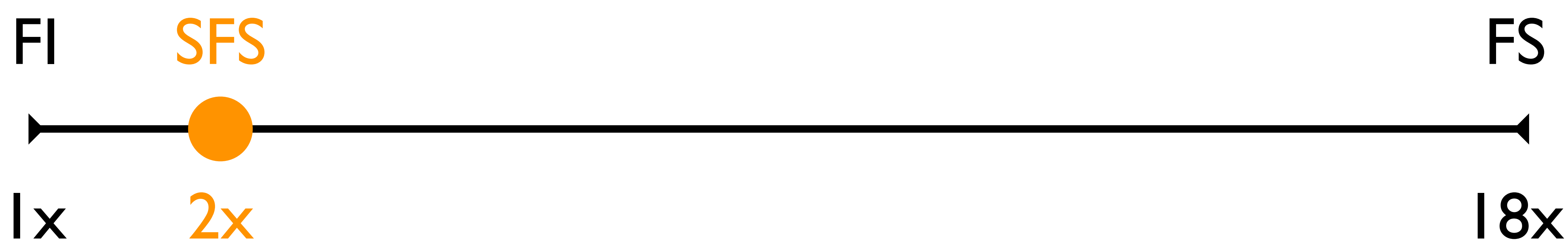
Effectiveness

- Implemented in Sparrow, an interval analyzer for C
- Evaluated on 30 open-source benchmarks

Precision



Cost



Hurdle

- The success crucially depends on the choice of features
- Feature construction is nontrivial and tedious
- |analyzers| x |parameter types| x |query types|

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	$2 \wedge (19 \vee 20) \wedge \neg 33$

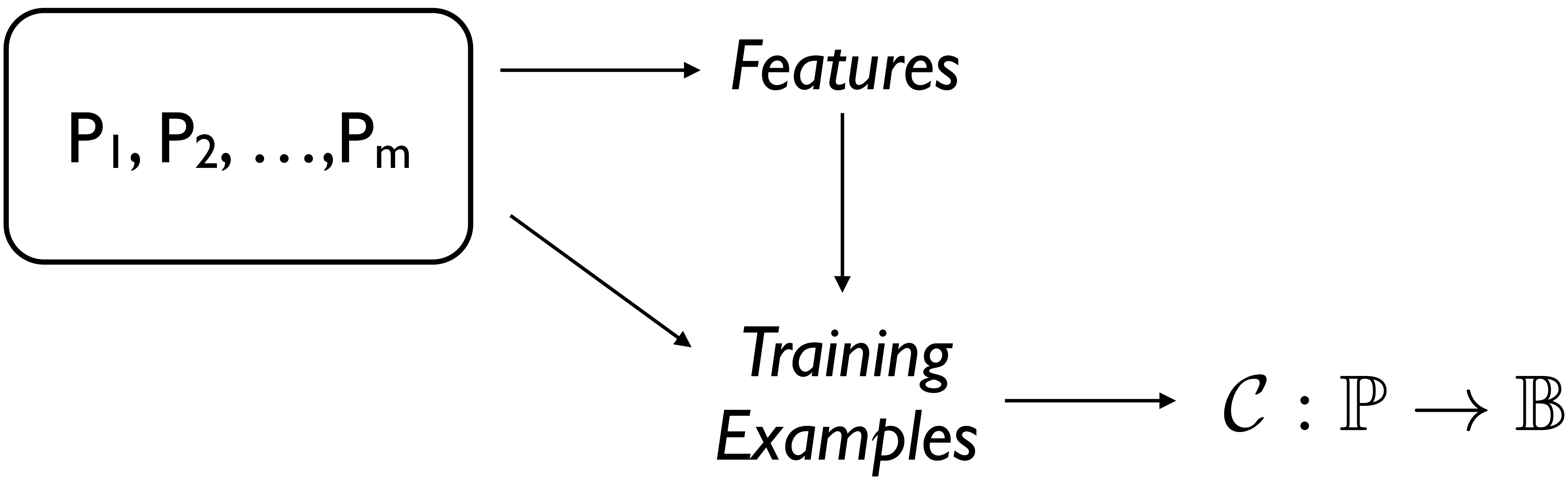
Type	#	Features
A	1	leaf function
	2	function containing malloc
	3	function containing realloc
	4	function containing a loop
	5	function containing an if statement
	6	function containing a switch statement
	7	function using a string-related library function
	8	write to a global variable
	9	read a global 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	$(21 \vee 22) \wedge \neg 23$

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	$13 \wedge 12$

On-going

Learning with Automatic Feature Construction

- Fully automatic learning approach



Key Ideas

- Generate *feature programs* that capture the key reason why FS succeeds but FI fails.

```
int j = 0;
main() {
  char num[5];
  int tmp = j++;
  num[tmp];
}
```

```
double B[309];
main() {
 for (int i=1;i<50; i++) {
    B[i];
  }
}
```

- Apply FS to a new program, if it matches some feature program.

Key Ideas

“C-Reduce” [PLDI’12]

Test-Case Reduction for C Compiler Bugs

John Regehr, University of Utah, regehr@cs.utah.edu
Yang Chen, University of Utah, chenyang@cs.utah.edu
Pascal Cuoq, CEA LIST, pascal.cuoq@cea.fr
Eric Eide, University of Utah, eeide@cs.utah.edu
Chucky Ellison, University of Illinois, celliso2@illinois.edu
Xuejun Yang, University of Utah, jxyang@cs.utah.edu

Abstract

To report a compiler bug, one must often find a small test case that triggers the bug. The existing approach to automated test-case reduction, delta debugging, works by removing substrings of the original input; the result is a concatenation of substrings that delta cannot remove. We have found this approach less than ideal for reducing C programs because it typically yields test cases that are too large or even invalid (relying on undefined behavior). To obtain small and valid test cases consistently, we designed and implemented three new, domain-specific test-case reducers. The best of these is based on a novel framework in which a generic fixpoint computation invokes modular transformations that perform reduction operations. This reducer produces outputs that are, on average, more than 25 times smaller than those produced by our other reducers or by the existing reducer that is most commonly used by compiler developers. We conclude that effective program reduction requires more than straightforward delta debugging.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—testing tools; D.3.2 [*Programming Languages*]: Language Classifications—C; D.3.4 [*Programming Languages*]: Processors—compilers

Keywords compiler testing, compiler defect, automated testing, random testing, bug reporting, test-case minimization

1. Introduction

Although many compiler bugs can be demonstrated by small test cases, bugs in released compilers are more typically discovered while building large projects. Before a bug can be reported, the circumstances leading to it must be narrowed down. The most important part of this process is *test-case reduction*: the construction of a small input that triggers the compiler bug.

The importance of test-case reduction is emphasized in the GCC documentation,[1] which states that:

> Our bug reporting instructions ask for the preprocessed version of the file that triggers the bug. Often this file is very large; there are several reasons for making it as small as possible…

The instructions for submitting bug reports to the LLVM developers also highlight the importance of test-case reduction.[2] Indeed, LLVM ships with the Bugpoint tool[3] that automates reduction at the level of LLVM IR. The importance that compiler developers place on small test cases stems from the simple fact that manual test-case reduction is both difficult and time consuming. With limited time to spend fixing bugs, compiler writers require bug reporters to undertake the effort of reducing large fault-causing inputs to small ones.

Like debugging, distilling a bug-causing compiler input to its essence is often an exercise in trial and error. One must repeatedly experiment by removing or simplifying pieces of the input program, compiling and running the partially reduced program, and backtracking when a change to the input causes the compiler bug to no longer be triggered. In some cases—for example, reducing a deterministic assertion failure in the compiler—manual test-case reduction is tedious but tractable. In other cases—e.g., reducing a miscompilation bug in a large, multi-threaded application—manual test-case reduction may be so difficult as to be infeasible. Our belief is that many compiler bugs go unreported due to the high difficulty of isolating them. When confronted with a compiler bug, a reasonable compiler user might easily decide that the most economic course is to find a workaround, and not to be “sidetracked” by the significant time and effort required to produce a small test case and report the bug.

Our goal in this paper is to automate most or all of the work required to reduce bug-triggering test cases for C compilers. Our work is motivated by two problems that we have encountered in applying state-of-the-art reducers. First, these tools get stuck at local minima that are too large. This necessitates subsequent manual reduction, preventing reportable compiler bugs from being generated in an entirely automated fashion. We have developed new reducers that use domain-specific knowledge to overcome the barriers that trap previous tools. Second, existing test-case reducers often generate test cases that execute *undefined behaviors*. These test cases are useless because the C language standard guarantees

The definitive version was published in *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Beijing, China, Jun. 2012, http://doi.acm.org/10.1145/NNNNNNN.NNNNNNN

[1] http://gcc.gnu.org/bugs/minimize.html
http://gcc.gnu.org/wiki/A_guide_to_testcase_reduction

[2] http://llvm.org/docs/HowToSubmitABug.html

[3] http://llvm.org/docs/Bugpoint.html

- Feature programs are automatically generated by using a general-purpose program reducer.

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

- Keep reducing when FS succeeds but FI fails:

$$\phi(P) = F(P, 0) = 0 \ \wedge \ F(P, 1) = 1$$

cf) Other Applications

- Bug-finding of static analyzers
- Alarm reduction:

```
a = input();
b = a;
10 / b;
```

reduce →

```
a = input();
10 / a;
```

$$a = \alpha \wedge b = a \wedge b \neq 0 \quad \Longleftarrow \quad a = \alpha \wedge a \neq 0$$

Summary

- Key problem in static analysis: automatic adaptation
- Promising approach: use ML [OOPSLA'15]
- Major hurdle: manual feature construction
- Our Solution: generate and match feature programs

Thank you