Automatically Generating Features for Learning Program Analysis Heuristics for C-Like Languages

Kwonsoo Chae
Korea University

Joint work with Hakjoo Oh (Korea University), Kihong Heo (Seoul National University), Hongseok Yang (University of Oxford)

OOPSLA'17 @Vancouver, Canada

Static Analysis

- Diverse engineering decisions in static analysis:
  - Context-sensitivity for which procedures?
  - Relational analysis for which variables?
  - Unsoundness for which part of the program?
  - etc.

Static Analysis

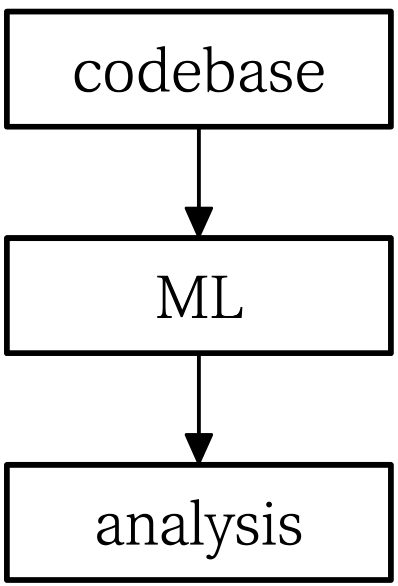
Data-Driven

- Diverse engineering decisions in static analysis:
  - Context-sensitivity for which procedures?
  - Relational analysis for which variables?
  - Unsoundness for which part of the program?
  - etc.
- Data-driven static analysis aims at automatically learning analysis heuristics from the codebase.

- ✓ context-sensitivity heuristics
- ✓ relation tracking heuristics
- ✓ unsoundness heuristics
- ✓ etc.

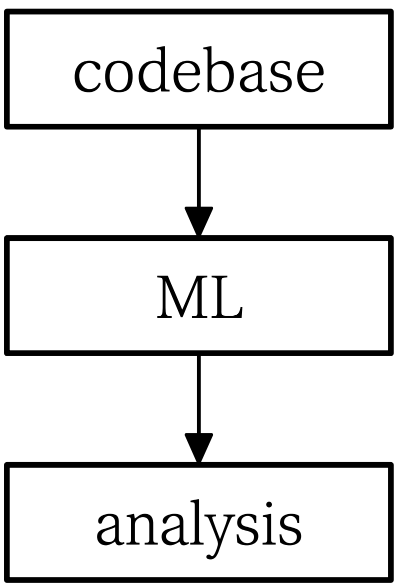
Main Obstacle: Manual Feature Engineering

What people expect

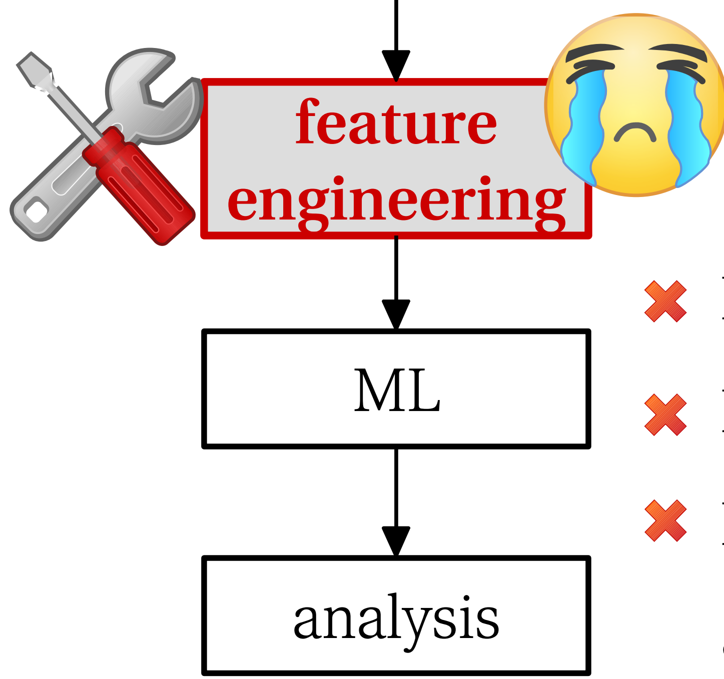


Main Obstacle:
Manual Feature Engineering

What people expect



Reality



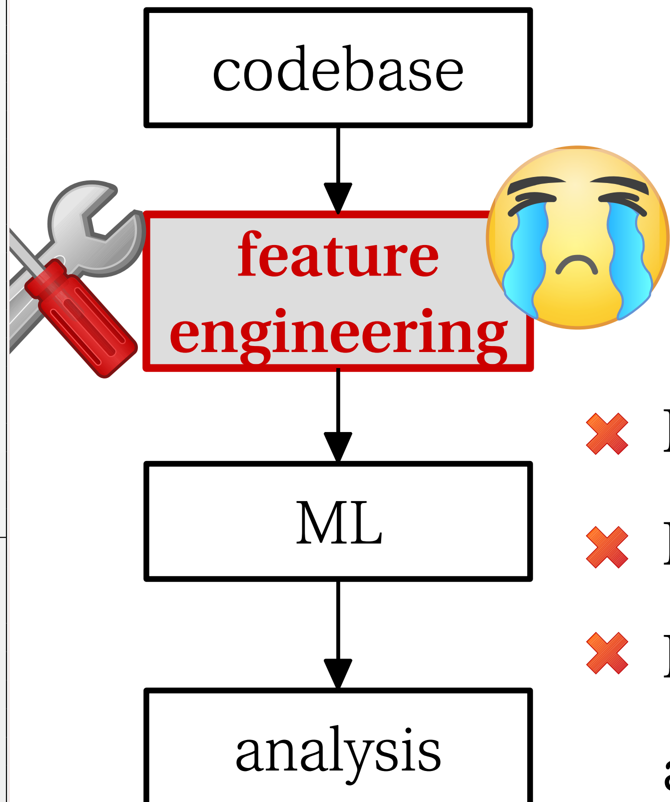
- ✖ Manual, time-consuming
- ✖ Need for domain expertise
- ✖ Not interchangeable among different analyses

Main Obstacle: Feature Engineering

flow-sensitivity (OOPSLA'15)

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

Reality



- ✖ Manual, time-consuming
- ✖ Need for domain expertise
- ✖ Not interchangeable among different analyses

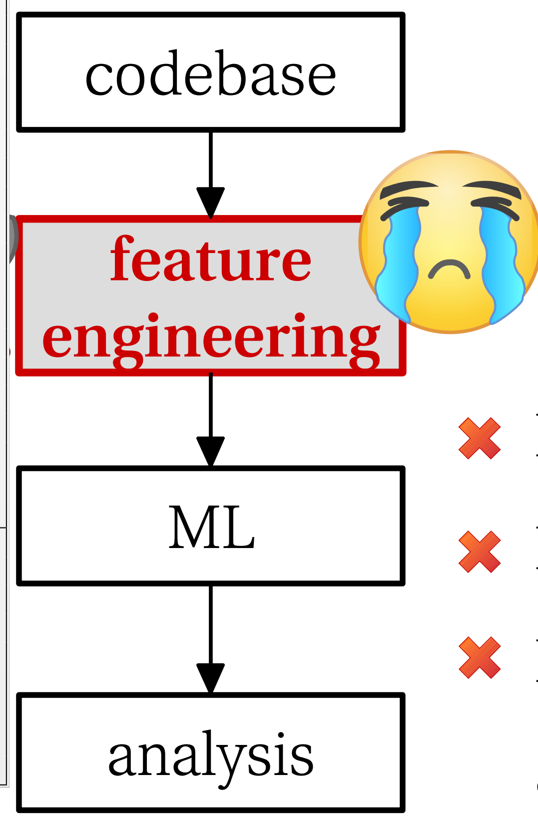
Main Obstacle: Feature Engineering

context-sensitivity (OOPSLA'15)

Type	#	Features
A	1	local variable
	2	global variable
	3	struct
	4	locati
	5	define
	6	locati
	7	assigr
	8	comp
	9	comp
	10	negat
	11	direct
	12	indire
	13	direct
	14	indire
	15	direct
	16	indire
	17	direct
	18	indire
	19	increr
	20	increr
	21	increr
	22	decre
	23	decre
	24	decre
	25	multi
	26	multi
	27	increr
	28	used a
	29	used i
	30	returr
	31	modif
	32	modif
	33	read i
B	34	1 ∧ 8
	35	2 ∧ 8
	36	1 ∧ (
	37	2 ∧ (
	38	1 ∧ (
	39	2 ∧ (
	40	(11 ∨
	41	(15 ∨
	42	1 ∧ (
	43	2 ∧ (19 ∨ 20) ∧ 33
	44	1 ∧ (19 ∨ 20) ∧ ¬33
	45	2 ∧ (19 ∨ 20) ∧ ¬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 ∧ (21 ∨ 22) ∧ (14 ∨ 15)
	31	2 ∧ (21 ∨ 22) ∧ ¬(14 ∨ 15)
	32	2 ∧ 23 ∧ (14 ∨ 15)
	33	2 ∧ 23 ∧ ¬(14 ∨ 15)
	34	2 ∧ (21 ∨ 22) ∧ (16 ∨ 17)
	35	2 ∧ (21 ∨ 22) ∧ ¬(16 ∨ 17)
	36	2 ∧ 23 ∧ (16 ∨ 17)
	37	2 ∧ 23 ∧ ¬(16 ∨ 17)
	38	(21 ∨ 22) ∧ ¬23

<u>Reality</u>



- ✖ Manual, time-consuming
- ✖ Need for domain expertise
- ✖ Not interchangeable among different analyses

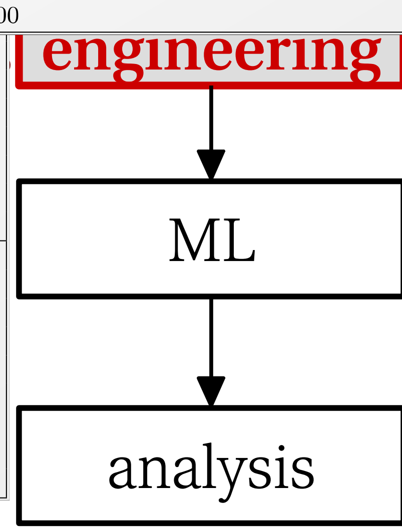
Main Obstacle: …eature Engineering

widening threshold (APLAS'16)

Type	#	Features
A	1	local variable
	2	global variable
	3	struct…
	4	locati…
	5	define…
	6	locati…
	7	assign…
	8	comp…
	9	comp…
	10	negat…
	11	direct…
	12	indire…
	13	direct…
	14	indire…
	15	direct…
	16	indire…
	17	direct…
	18	indire…
	19	incre…
	20	incre…
	21	incre…
	22	decre…
	23	decre…
	24	decre…
	25	multi…
	26	multi…
	27	incre…
	28	used a…
	29	used i…
	30	return…
	31	modif…
	32	modif…
	33	read i…
B	34	$1 \wedge 8$…
	35	$2 \wedge 8$…
	36	$1 \wedge ($…
	37	$2 \wedge ($…
	38	$1 \wedge ($…
	39	$2 \wedge ($…
	40	$(11 \vee$…
	41	$(15 \vee$…
	42	$1 \wedge ($…$) \wedge 33$
	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 c…
	5	function c…
	6	function c…
	7	function u…
	8	write to a…
	9	read a glo…
	10	write to a…
	11	read from…
	12	directly re…
	13	indirectly…
	14	directly re…
	15	indirectly…
	16	directly re…
	17	indirectly…
	18	return exp…
	19	return valu…
	20	return voi…
	21	directly in…
	22	constant i…
	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	$(21 \vee 22) \wedge \neg 23$

#	Description
1	used as the size of a static array
2	the size of a static array $-$ 1
3	returned by a function (e.g. return 1)
4	three successive numbers appear in the program (e.g. $n, n+1, n+2$)
5	most frequently appeared numbers in the program (i.e. top 10%)
6	least frequently appeared numbers in the program (i.e. bottom 10%)
7	passed as the size arguments of memory copy functions (e.g. memcpy)
8	used as the size of the destination arrays in memory copy functions (e.g. memcpy)
9	the null position of a string buffer involved in some loop condition
10	the null position of a static array of primitive types (e.g., arrays of int and char)
11	the null position of a static array of structure fields
12	constants involved in conditional expressions (e.g. if (x == 1))
13	integers of the form 2^n (e.g. 2, 4, 8, 16)
14	integers of the form $2^n - 1$ (e.g., 1, 3, 7, 15)
15	integers in the range $0 < n \leq 50$
16	integers in the range $50 < n \leq 100$
17	integers in the range $n > 1000$



- ✖ Manual, time-consuming
- ✖ Need for domain expertise
- ✖ Not interchangeable among different analyses

Main Obstacle: ...eature Engineering

relational analysis (SAS'16)

Type	#	Features
A	1	local variable
	2	global variable
	3	struct...
	4	locati
	5	define
	6	locati
	7	assigr
	8	comp
	9	comp
	10	negat
	11	direct
	12	indire
	13	direct
	14	indire
	15	direct
	16	indire
	17	direct
	18	indire
	19	increr
	20	increr
	21	increr
	22	decrer
	23	decrer
	24	decrer
	25	multij
	26	multij
	27	increr
	28	used a
	29	used i
	30	return
	31	modif
	32	modif
	33	read i
B	34	$1 \wedge 8$
	35	$2 \wedge 8$
	36	$1 \wedge ($
	37	$2 \wedge ($
	38	$1 \wedge ($
	39	$2 \wedge ($
	40	$(11 \vee$
	41	$(15 \vee$
	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 c
	5	function c
	6	function c
	7	function u
	8	write to a
	9	read a glo
	10	write to a
	11	read from
	12	directly re
	13	indirectly
	14	directly re
	15	indirectly
	16	directly re
	17	indirectly
	18	return exp
	19	return valu
	20	return voi
	21	directly in
	22	constant is
	23	invoked with an unknown
	24	functions having no argu
	25	functions having one argu
	26	functions having more tha
	27	functions having an intege
	28	functions having a pointer
	29	functions having a structu
B	30	$2 \wedge (21 \vee 22) \wedge (14 \vee 15$
	31	$2 \wedge (21 \vee 22) \wedge \neg(14 \vee 1$
	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	$2 \wedge (21 \vee 22) \wedge \neg(16 \vee 1$
	36	$2 \wedge 23 \wedge (16 \vee 17)$
	37	$2 \wedge 23 \wedge \neg(16 \vee 17)$
	38	$(21 \vee 22) \wedge \neg 23$

#	Description
1	used as the size of a static array
2	the size of a static array − 1
3	returned by a function (e.g. return 1)
4	three succ
5	most frequ
6	least frequ
7	passed as
8	used as th
9	the null p
10	the null p
11	the null p
12	constants
13	integers o
14	integers o
15	integers in
16	integers in
17	integers in

i	Description of feature $f_i(P, (x, y))$. k represents a constant.
1	P contains an assignment $x = y + k$ or $y = x + k$.
2	P contains a guard $x \leq y + k$ or $y \leq x + k$.
3	P contains a malloc of the form $x = \texttt{malloc}(y)$ or $y = \texttt{malloc}(x)$.
4	P contains a command $x = \texttt{strlen}(y)$ or $y = \texttt{strlen}(x)$.
5	P sets x to $\texttt{strlen}(y)$ or y to $\texttt{strlen}(x)$ indirectly, as in $t = \texttt{strlen}(y); x = t$.
6	P contains an expression of the form $x[y]$ or $y[x]$.
7	P contains an expression that multiplies x or y by a constant different from 1.
8	P contains an expression that multiplies x or y by a variable.
9	P contains an expression that divides x or y by a variable.
10	P contains an expression that has x or y as an operand of bitwise operations.
11	P contains an assignment that updates x or y using non-Octagonal expressions.
12	x and y are has the same name in different scopes.
13	x and y are both global variables in P.
14	x or y is a global variable in P.
15	x or y is a field of a structure in P.
16	x and y represent sizes of some arrays in P.
17	x and y are temporary variables in P.
18	x or y is a local variable of a recursive function in P.
19	x or y is tested for the equality with ± 1 in P.
20	x and y represent sizes of some global arrays in P.
21	x or y stores the result of a library call in P.
22	x and y are local variables of different functions in P.
23	$\{x, y\}$ consists of a local var. and the size of a local array in different fun. in P.
24	$\{x, y\}$ consists of a local var. and a temporary var. in different functions in P.
25	$\{x, y\}$ consists of a global var. and the size of a local array in P.
26	$\{x, y\}$ contains a temporary var. and the size of a local array in P.
27	$\{x, y\}$ consists of local and global variables not accessed by the same fun. in P.
28	x or y is a self-updating global var. in P.
29	The flow-insensitive analysis of P results in a finite interval for x or y.
30	x or y is the size of a constant string in P.

..., time-consuming

...r domain expertise

...erchangeable

... different analyses

Main Obstacle: Feature Engineering

unsoundness (ICSE'17)

Type	#	Features
A	1	local variable
	2	global variable
	3	struct
	4	locati
	5	define
	6	locati
	7	assigr
	8	comp
	9	comp
	10	negat
	11	direct
	12	indire
	13	direct
	14	indire
	15	direct
	16	indire
	17	direct
	18	indire
	19	increr
	20	increr
	21	increr
	22	decre
	23	decre
	24	decre
	25	multi
	26	multi
	27	increr
	28	used
	29	used i
	30	return
	31	modif
	32	modif
	33	read i
B	34	$1 \wedge 8$
	35	$2 \wedge 8$
	36	$1 \wedge ($
	37	$2 \wedge ($
	38	$1 \wedge ($
	39	$2 \wedge ($
	40	$(11 \vee$
	41	$(15 \vee$
	42	$1 \wedge (1$ ... $\vee 20) \wedge 33$
	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 c
	5	function c
	6	function c
	7	function u
	8	write to a
	9	read a glo
	10	write to a
	11	read from
	12	directly re
	13	indirectly
	14	directly re
	15	indirectly
	16	directly re
	17	indirectly
	18	return exp
	19	return valu
	20	return voi
	21	directly in
	22	constant is
	23	invoked with an unknown
	24	functions having no argu
	25	functions having one argu
	26	functions having more tha
	27	functions having an intege
	28	functions having a pointe
	29	functions having a structu
B	30	$2 \wedge (21 \vee 22) \wedge (14 \vee 15$
	31	$2 \wedge (21 \vee 22) \wedge \neg(14 \vee 1$
	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	$2 \wedge (21 \vee 22) \wedge \neg(16 \vee 1$
	36	$2 \wedge 23 \wedge (16 \vee 17)$
	37	$2 \wedge 23 \wedge \neg(16 \vee 17)$
	38	$(21 \vee 22) \wedge \neg 23$

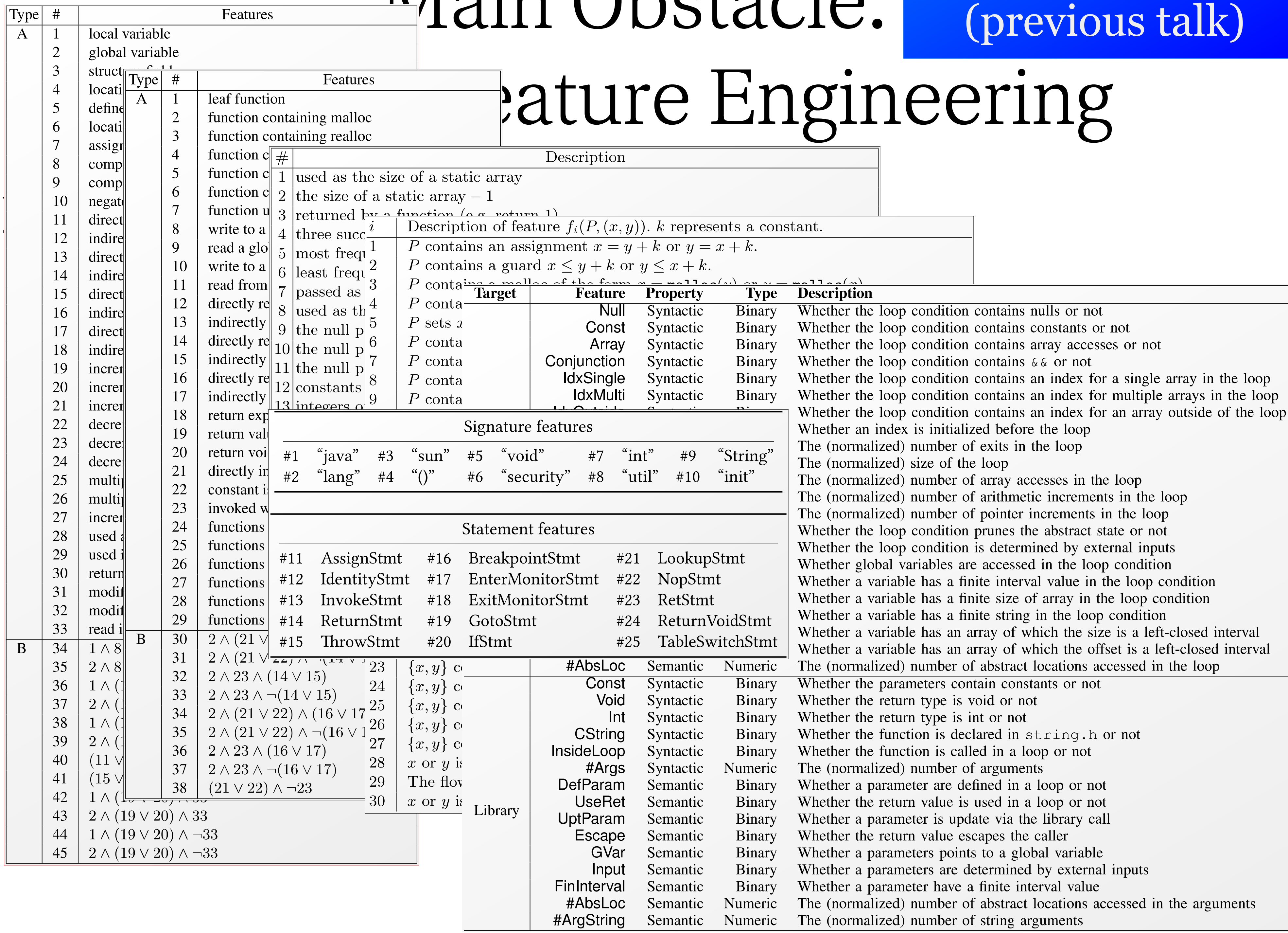
#	Description
1	used as the size of a static array
2	the size of a static array − 1
3	returned by a function (e.g. return 1)
4	three succ
5	most frequ
6	least frequ
7	passed as
8	used as th
9	the null p
10	the null p
11	the null p
12	constants
13	integers o
14	integers o
15	integers in
16	integers in
17	integers in

i	Description of feature $f_i(P,(x,y))$. k represents a constant.
1	P contains an assignment $x = y + k$ or $y = x + k$.
2	P contains a guard $x \leq y + k$ or $y \leq x + k$.
3	P contains a malloc of the form x = malloc(y) or y = malloc(x)
4	P conta
5	P sets x
6	P conta
7	P conta
8	P conta
9	P conta
10	P conta
11	P conta
12	x and y
13	x and y
14	x or y is
15	x or y is
16	x and y
17	x and y
18	x or y is
19	x or y is
20	x and y
21	x or y s
22	x and y
23	$\{x, y\}$ c
24	$\{x, y\}$ c
25	$\{x, y\}$ c
26	$\{x, y\}$ c
27	$\{x, y\}$ c
28	x or y is
29	The flov
30	x or y is

Target	Feature	Property	Type	Description
Loop	Null	Syntactic	Binary	Whether the loop condition contains nulls or not
	Const	Syntactic	Binary	Whether the loop condition contains constants or not
	Array	Syntactic	Binary	Whether the loop condition contains array accesses or not
	Conjunction	Syntactic	Binary	Whether the loop condition contains `&&` or not
	IdxSingle	Syntactic	Binary	Whether the loop condition contains an index for a single array in the loop
	IdxMulti	Syntactic	Binary	Whether the loop condition contains an index for multiple arrays in the loop
	IdxOutside	Syntactic	Binary	Whether the loop condition contains an index for an array outside of the loop
	InitIdx	Syntactic	Binary	Whether an index is initialized before the loop
	Exit	Syntactic	Numeric	The (normalized) number of exits in the loop
	Size	Syntactic	Numeric	The (normalized) size of the loop
	ArrayAccess	Syntactic	Numeric	The (normalized) number of array accesses in the loop
	ArithInc	Syntactic	Numeric	The (normalized) number of arithmetic increments in the loop
	PointerInc	Syntactic	Numeric	The (normalized) number of pointer increments in the loop
	Prune	Semantic	Binary	Whether the loop condition prunes the abstract state or not
	Input	Semantic	Binary	Whether the loop condition is determined by external inputs
	GVar	Semantic	Binary	Whether global variables are accessed in the loop condition
	FinInterval	Semantic	Binary	Whether a variable has a finite interval value in the loop condition
	FinArray	Semantic	Binary	Whether a variable has a finite size of array in the loop condition
	FinString	Semantic	Binary	Whether a variable has a finite string in the loop condition
	LCSize	Semantic	Binary	Whether a variable has an array of which the size is a left-closed interval
	LCOffset	Semantic	Binary	Whether a variable has an array of which the offset is a left-closed interval
	#AbsLoc	Semantic	Numeric	The (normalized) number of abstract locations accessed in the loop
Library	Const	Syntactic	Binary	Whether the parameters contain constants or not
	Void	Syntactic	Binary	Whether the return type is void or not
	Int	Syntactic	Binary	Whether the return type is int or not
	CString	Syntactic	Binary	Whether the function is declared in `string.h` or not
	InsideLoop	Syntactic	Binary	Whether the function is called in a loop or not
	#Args	Syntactic	Numeric	The (normalized) number of arguments
	DefParam	Semantic	Binary	Whether a parameter are defined in a loop or not
	UseRet	Semantic	Binary	Whether the return value is used in a loop or not
	UptParam	Semantic	Binary	Whether a parameter is update via the library call
	Escape	Semantic	Binary	Whether the return value escapes the caller
	GVar	Semantic	Binary	Whether a parameters points to a global variable
	Input	Semantic	Binary	Whether a parameters are determined by external inputs
	FinInterval	Semantic	Binary	Whether a parameter have a finite interval value
	#AbsLoc	Semantic	Numeric	The (normalized) number of abstract locations accessed in the arguments
	#ArgString	Semantic	Numeric	The (normalized) number of string arguments

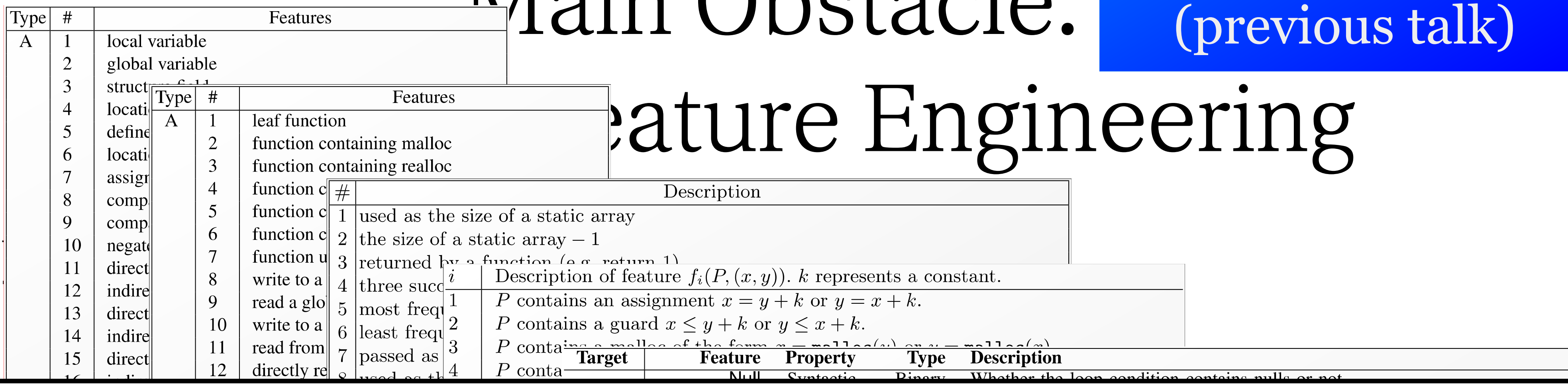
Main Obstacle: Feature Engineering

context-sensitivity (previous talk)

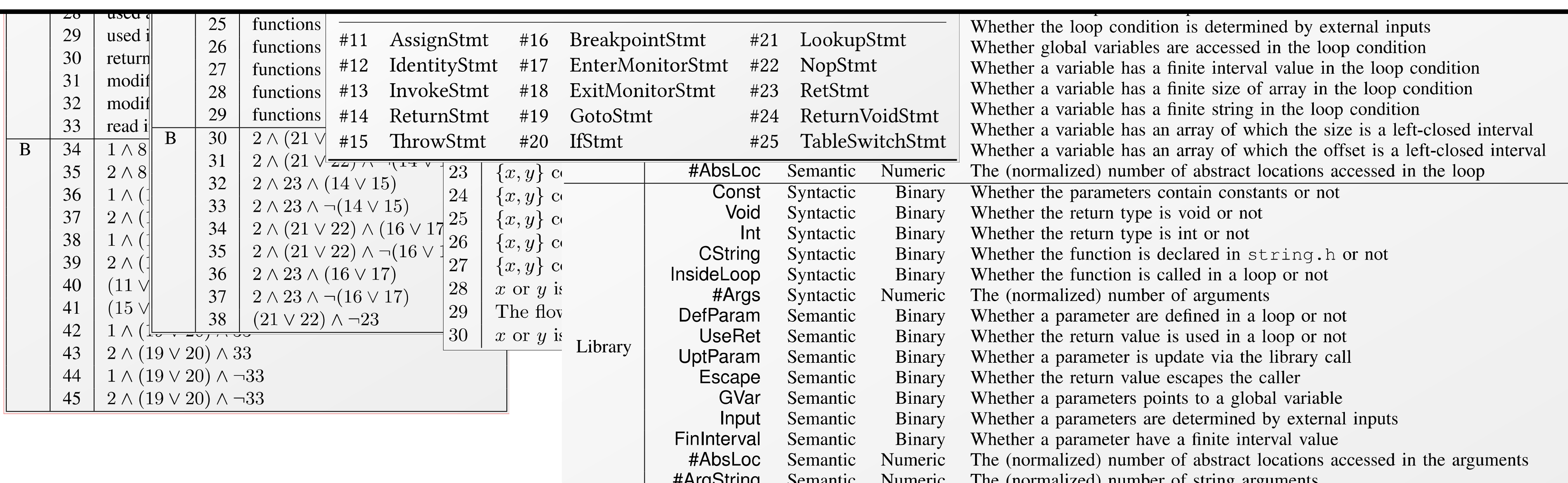


Main Obstacle: Feature Engineering

context-sensitivity (previous talk)

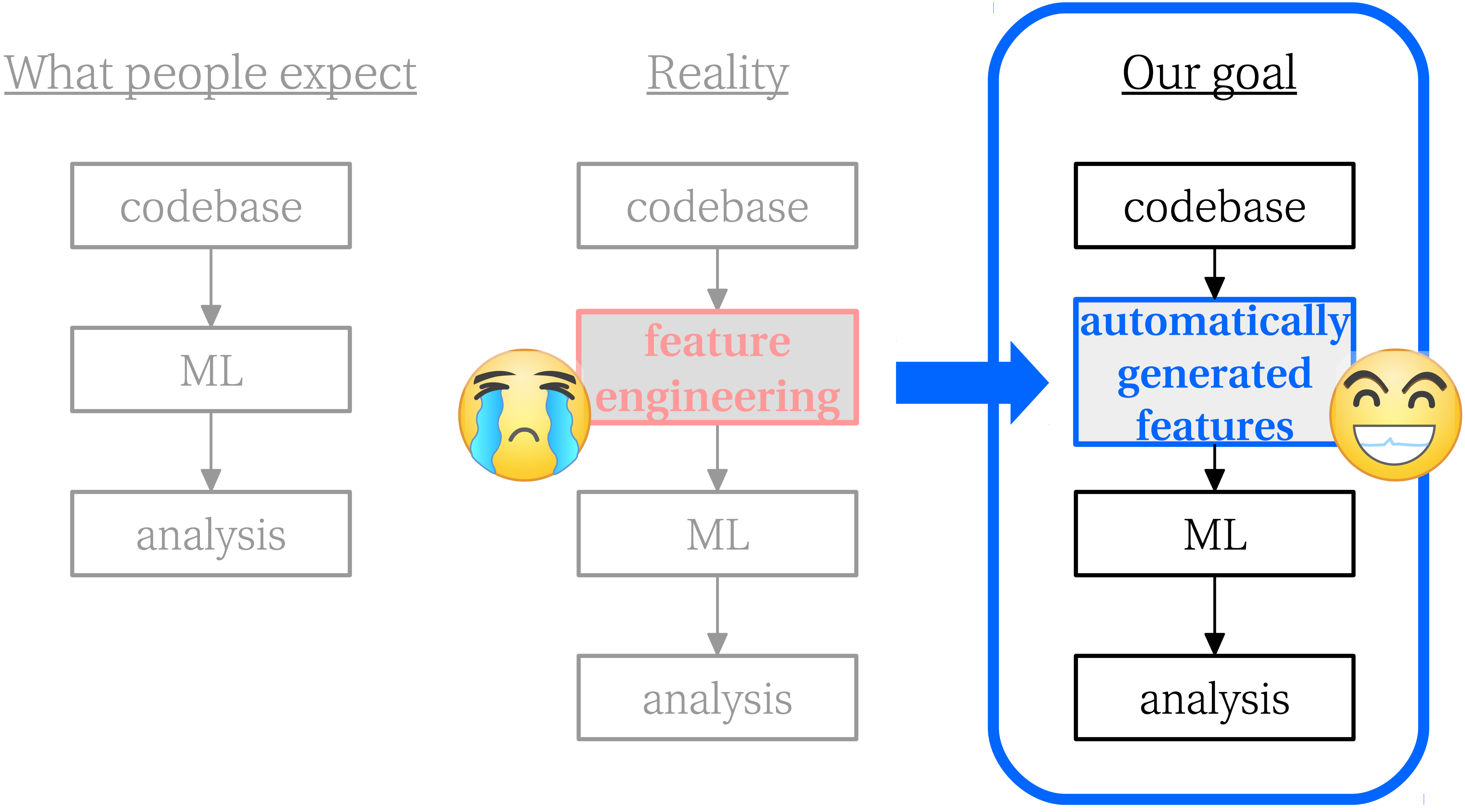


Feature engineering: major bottleneck in data-driven static analysis

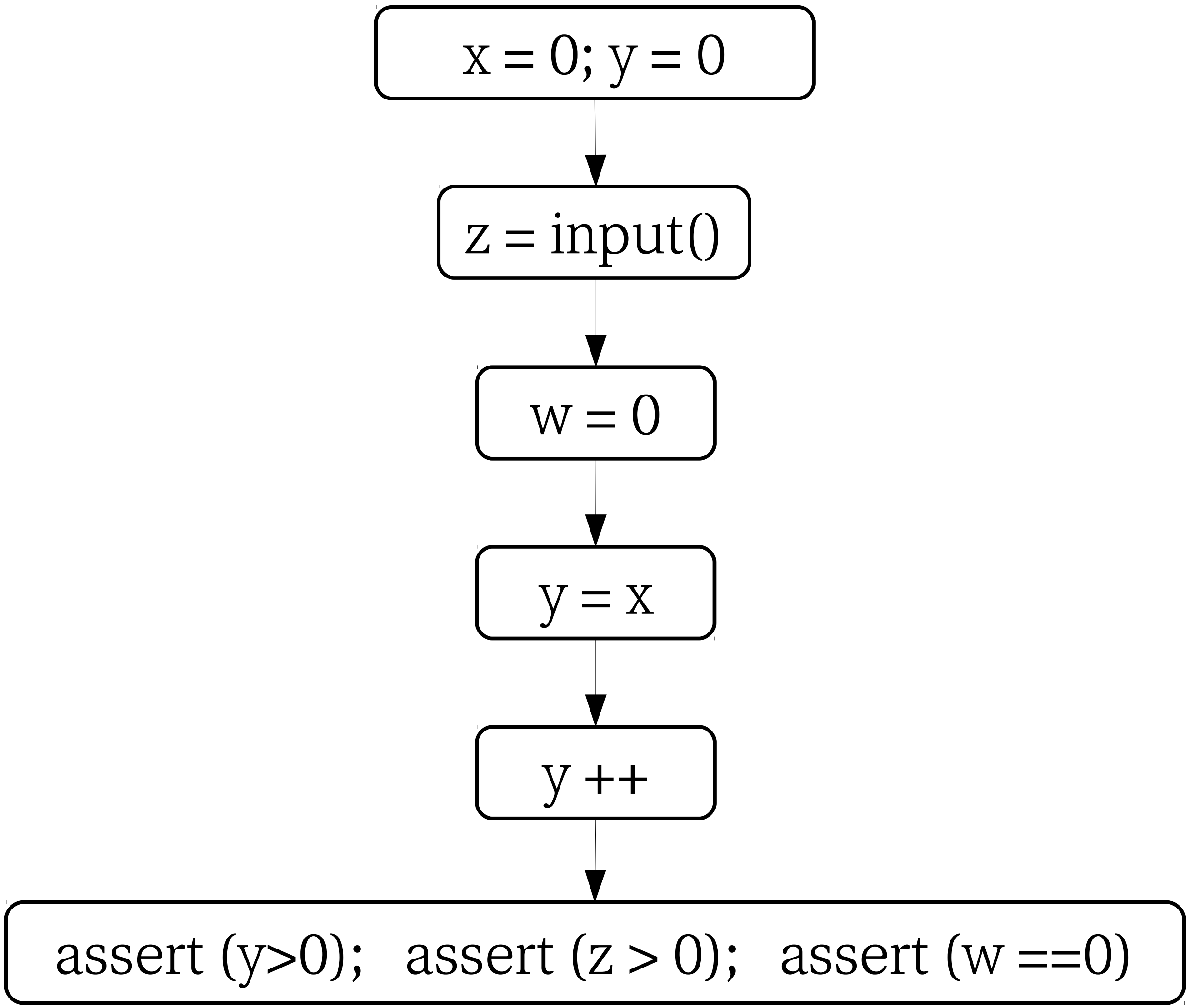


This talk

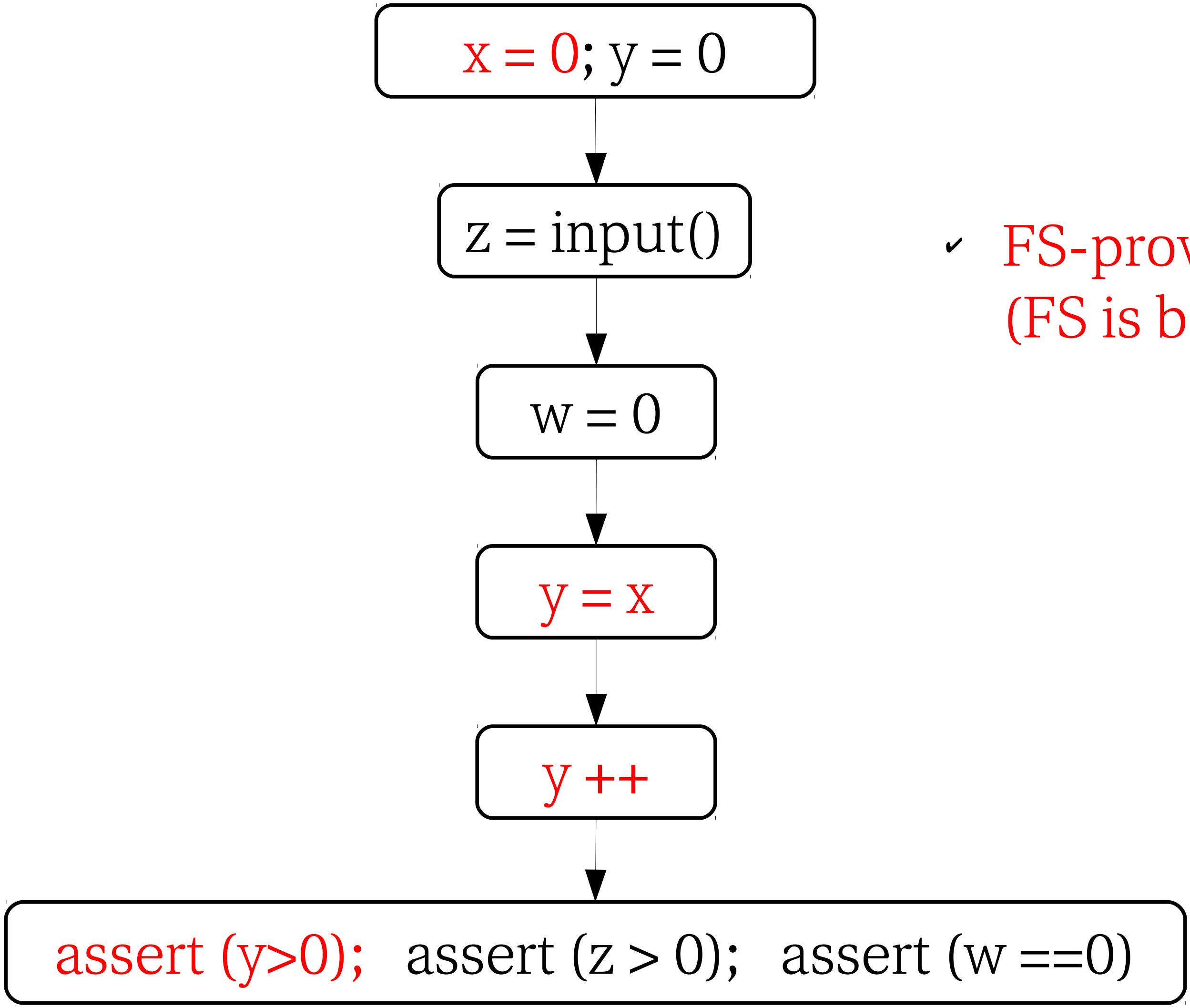
We Aim At Generating Features Automatically



Example: Partially Flow-Sensitive Interval Analysis

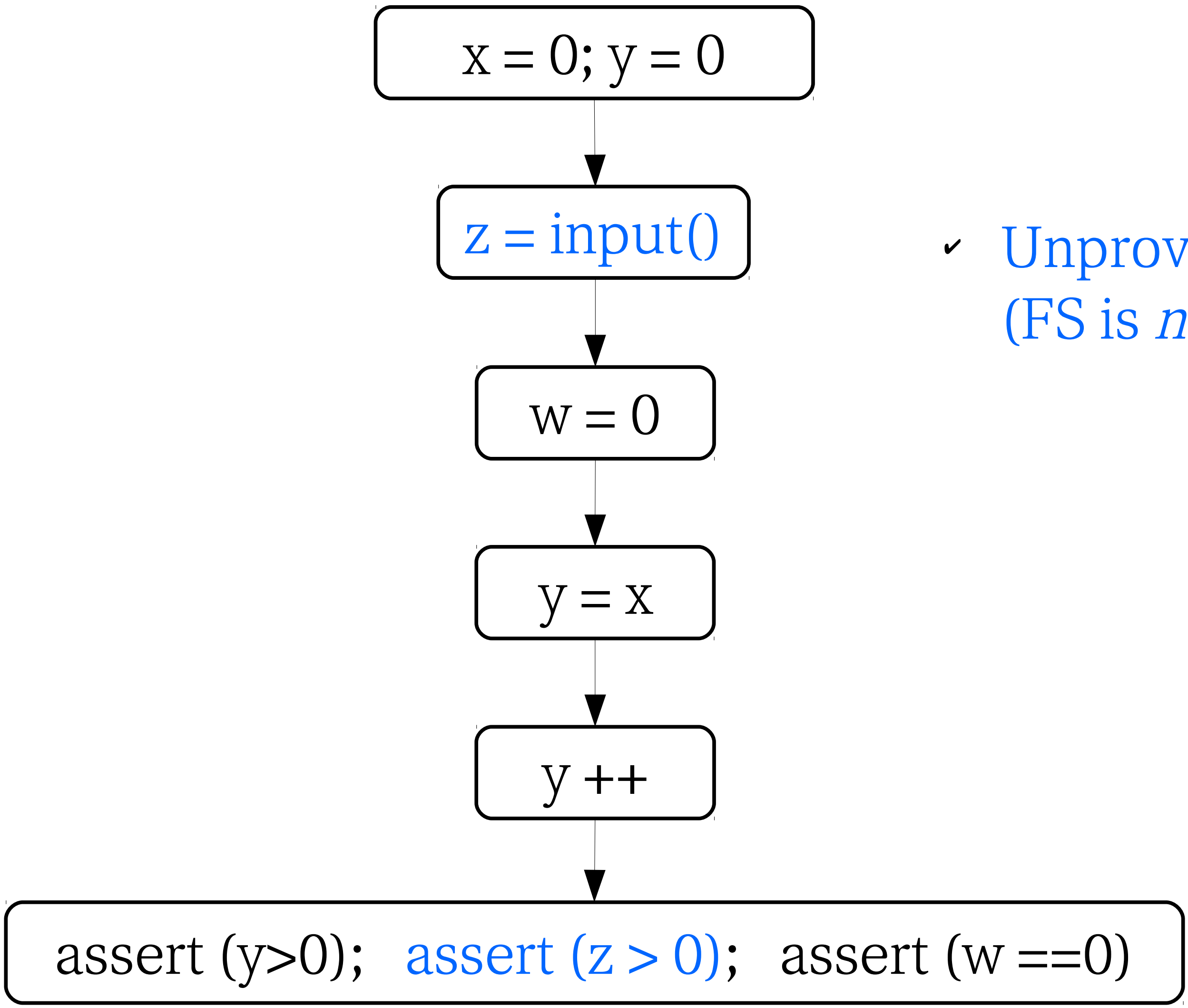


Example: Partially Flow-Sensitive Interval Analysis



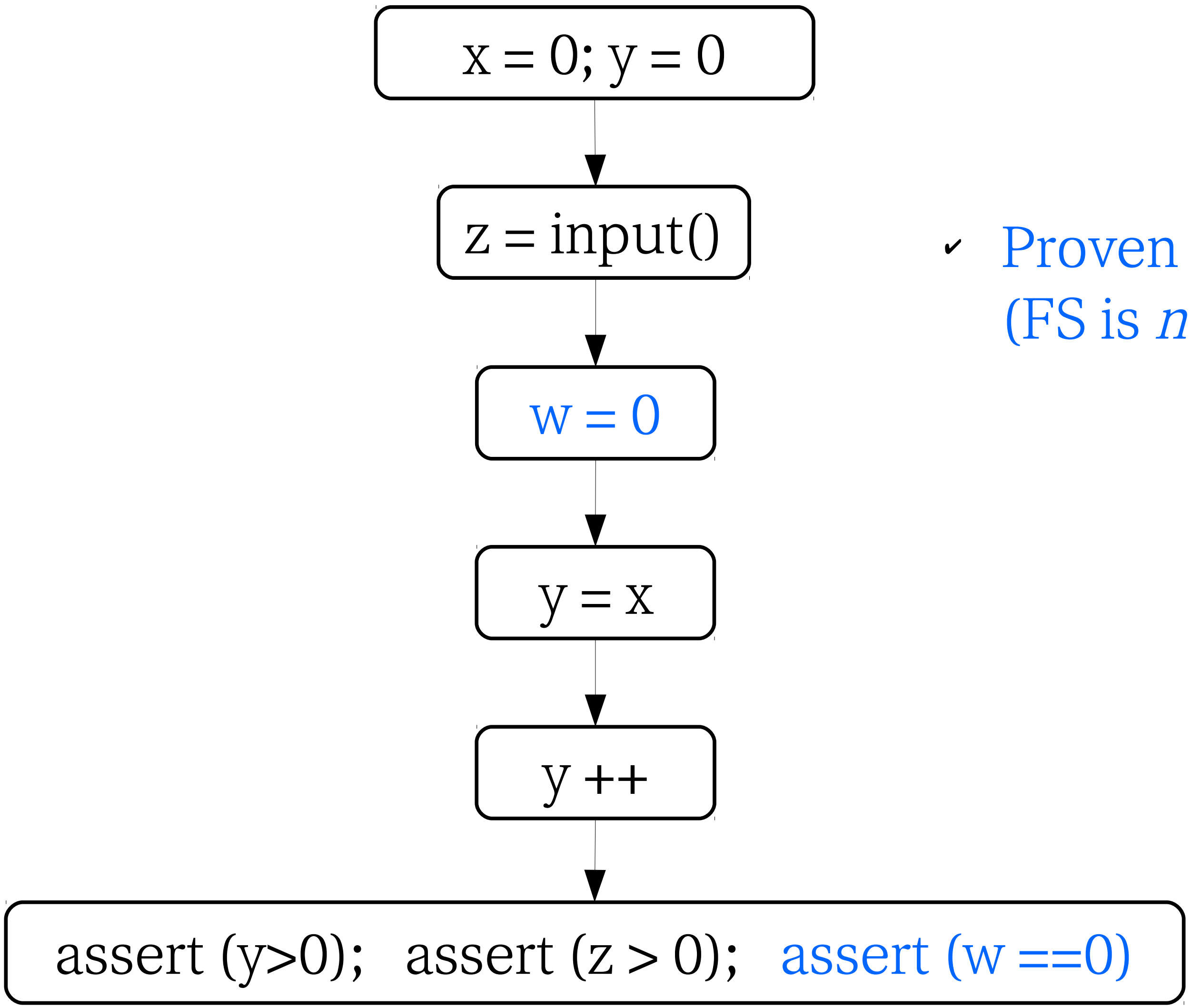
- ✓ FS-proven but FI-unproven (FS is beneficial)

Example: Partially Flow-Sensitive Interval Analysis



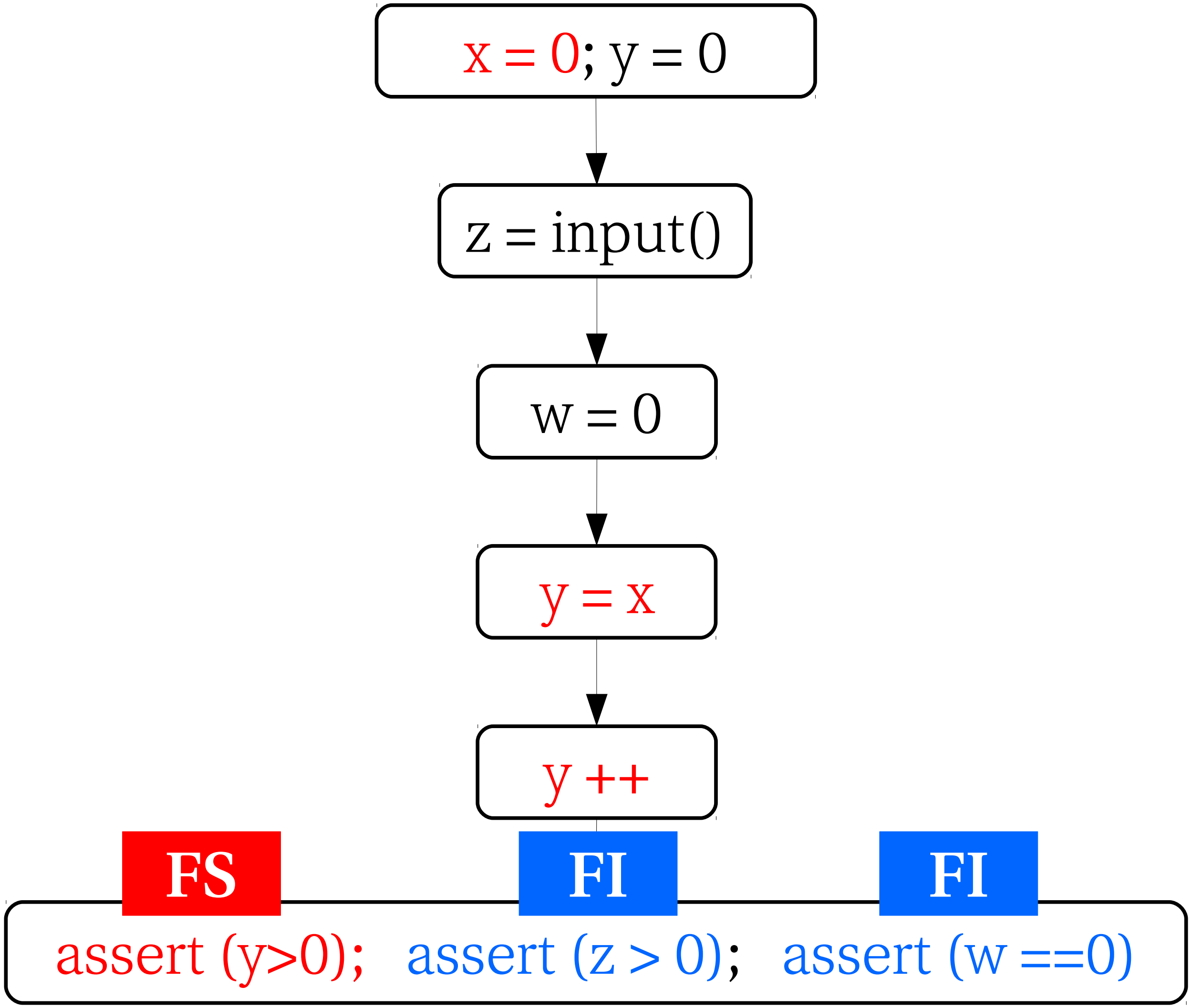
- Unproven even by FS (FS is *not* beneficial)

Example: Partially Flow-Sensitive Interval Analysis

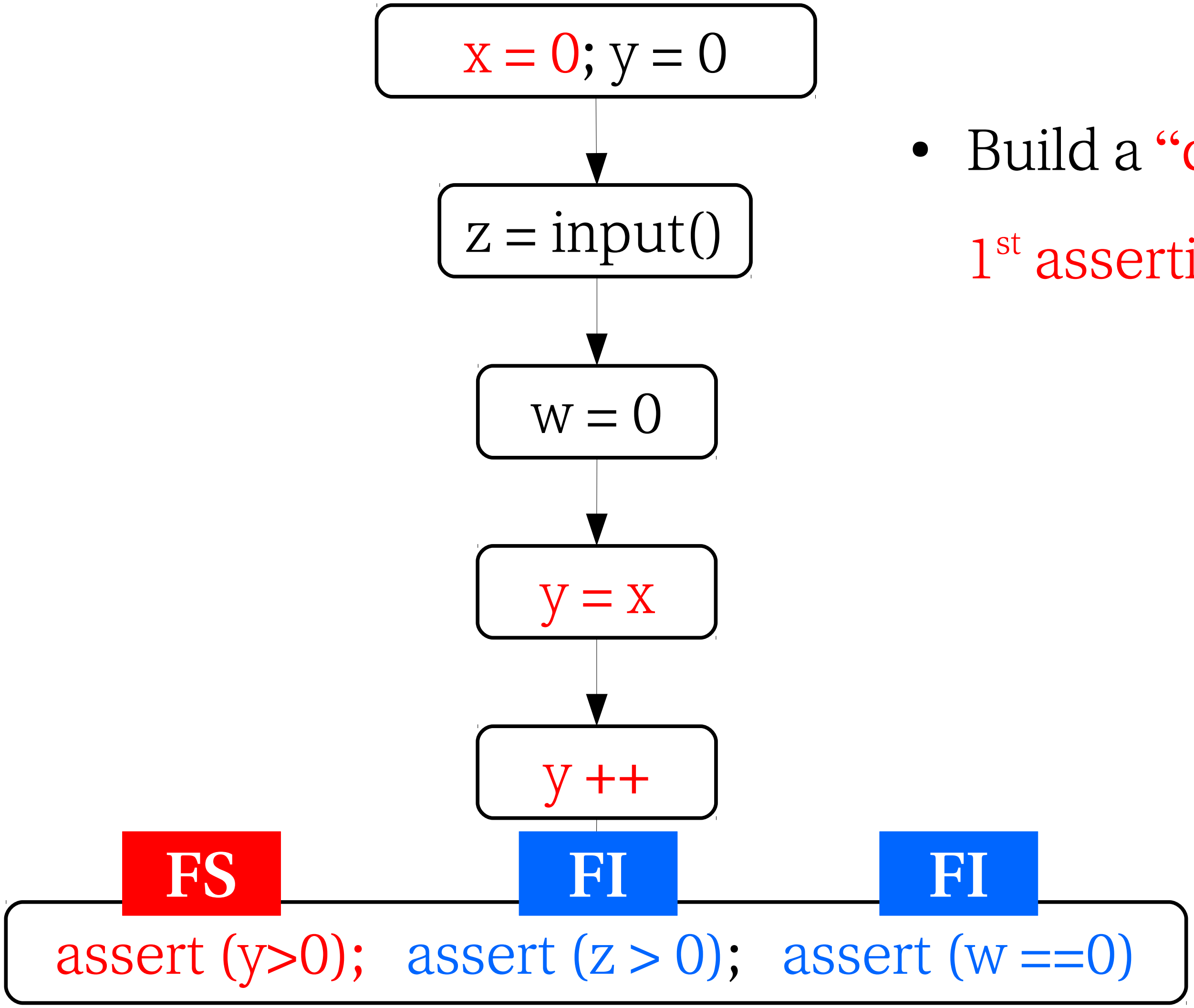


- ✔ Proven even by FI
  (FS is *not* beneficial)

Example: Partially Flow-Sensitive Interval Analysis



Example: Partially Flow-Sensitive Interval Analysis



- Build a "classifier" that selects the 1st assertion only.

Building a Classifier

- Usual procedure
  (1) Design a good set of features manually.

$$F = \{f_1, \ldots, f_k\}$$

Building a Classifier

- Usual procedure
  (1) Design a good set of features manually.
  (2) Generate labeled data.

$$F = \{f_1, \ldots, f_k\} \Rightarrow \begin{array}{c} [f_1(Q_1), \ldots, f_k(Q_1)] : 1 \\ [f_1(Q_2), \ldots, f_k(Q_2)] : 0 \\ \ldots \end{array}$$

Building a Classifier

- Usual procedure
  (1) Design a good set of features manually.
  (2) Generate labeled data.
  (3) Run an off-the-shelf classification algorithm.

$$F = \{f_1, \ldots, f_k\} \Rightarrow \begin{array}{l} [f_1(Q_1), \ldots, f_k(Q_1)] : 1 \\ [f_1(Q_2), \ldots, f_k(Q_2)] : 0 \\ \qquad\qquad \cdots \end{array} \Rightarrow \boxed{\text{classification algorithm}} \Rightarrow \text{classifier}$$

Building a Classifier automatically

- Our ~~Usual~~ procedure

  ~~(1) Design a good set of features manually.~~

  (2) Generate labeled data.

  (3) Run an off-the-shelf classification algorithm.

$F = \{f_1, \ldots, f_k\}$ ⇒ $[f_1(Q_1), \ldots, f_k(Q_1)] : 1$ ⇒ classification algorithm

"automatically"

$[f_1(Q_2), \ldots, f_k(Q_2)] : 0$

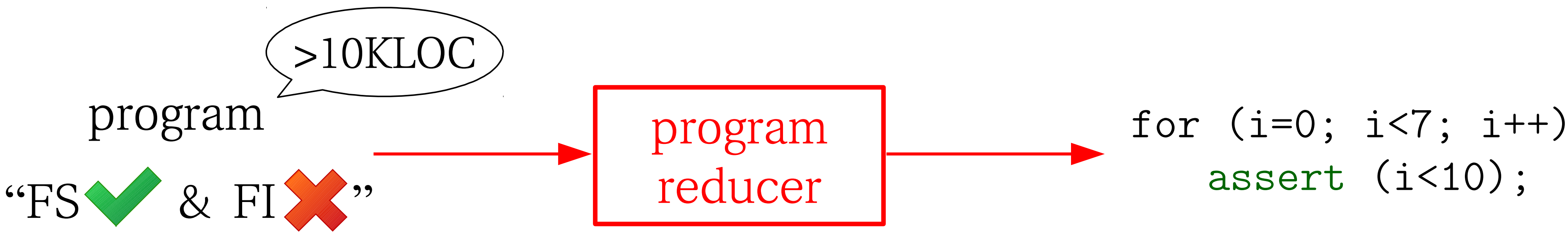
$\cdots$

⇓

classifier

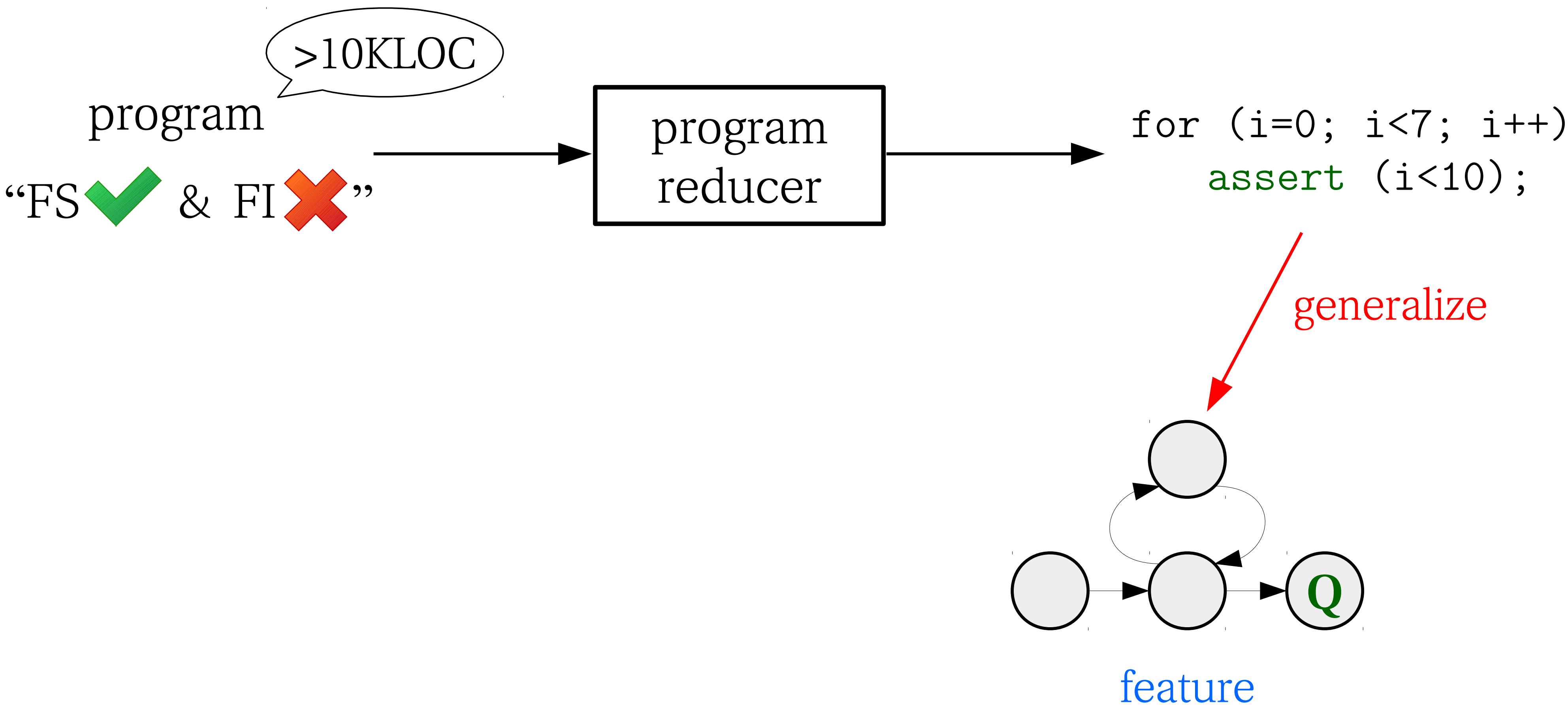
Highlight: Key Ideas

1. Capture the key reason why FS is beneficial using a program reducer.



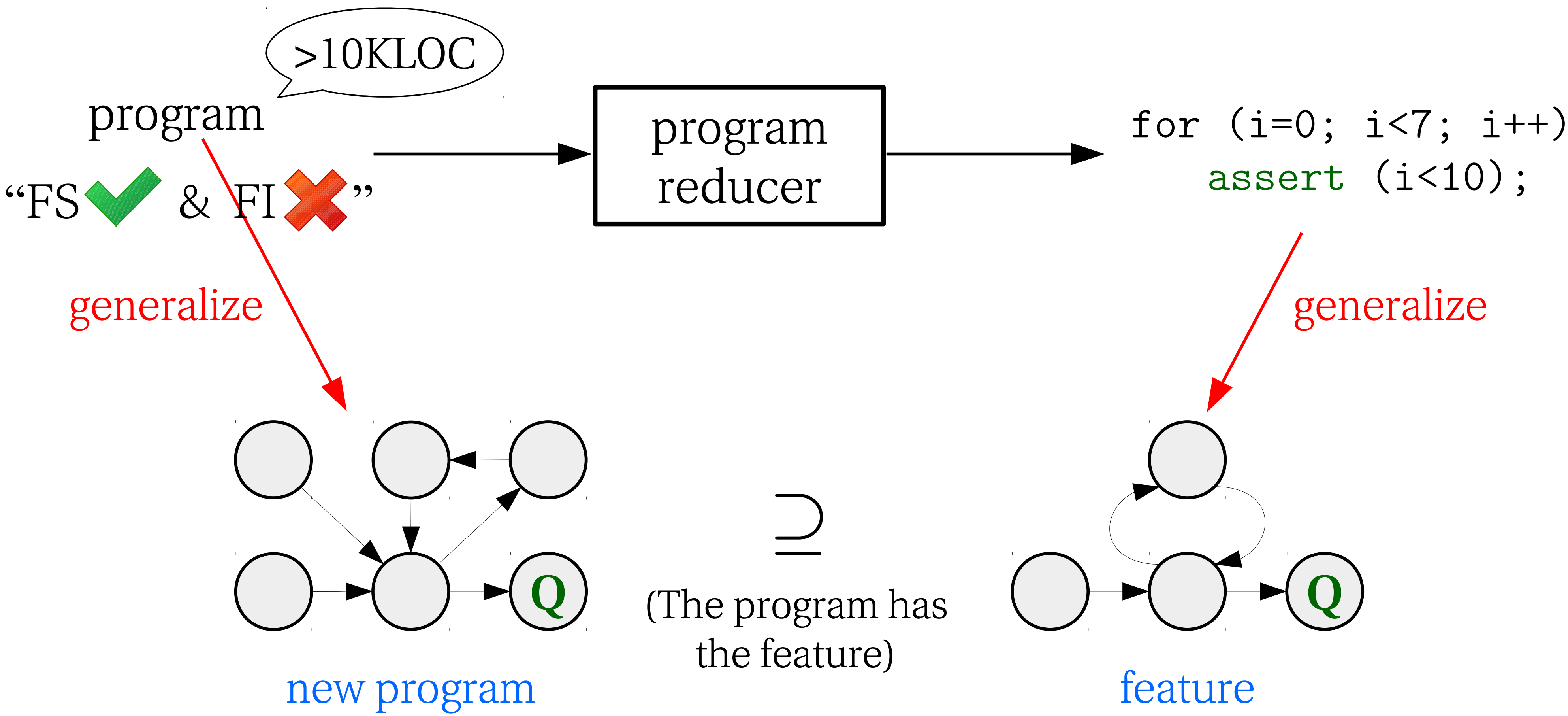
Highlight: Key Ideas

1. Capture the key reason why FS is beneficial using a program reducer.
2. Generalize the reduced program.



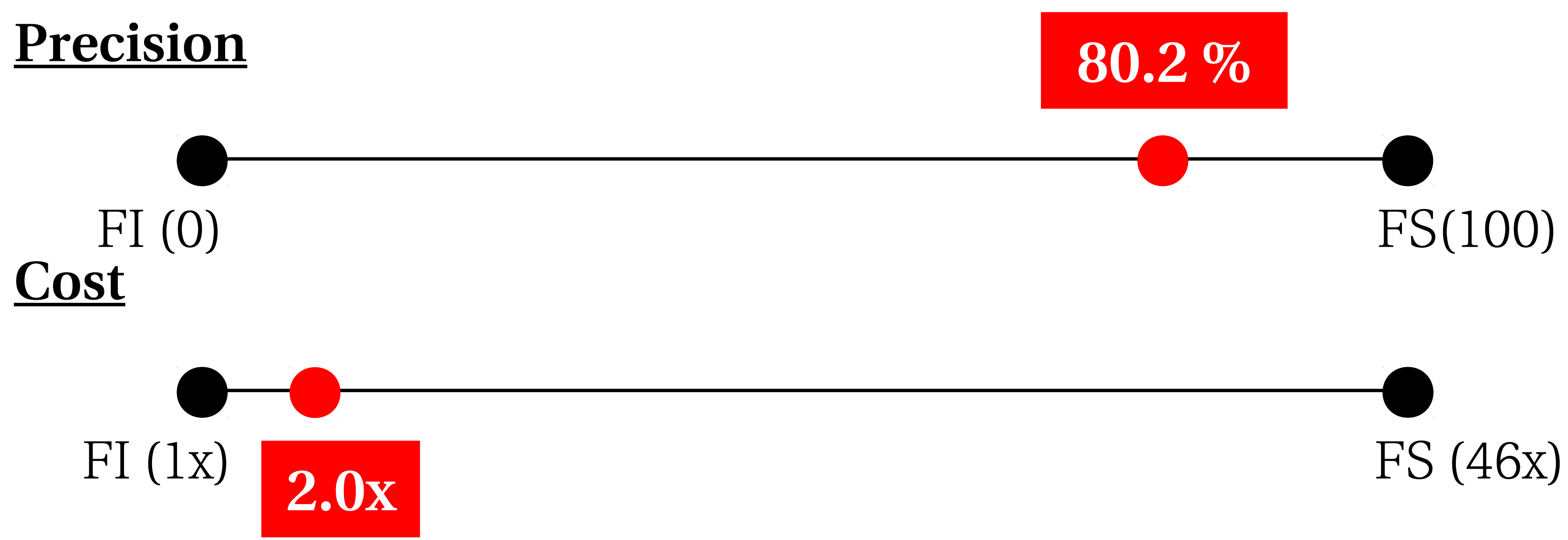
Highlight: Key Ideas

1. Capture the key reason why FS is beneficial using a program reducer.
2. Generalize the reduced program.



Highlight: Results

- Generated 38 (interval), 45 (pointer), 44 (Octagon) features.
- Analysis heuristics built on top of automatically generated features
- Excellent balance between cost and precision, e.g.,
  - Partially flow-sensitive interval analysis:



Automatic Feature Generation

Recipe

(1) Capture the key reasons from the codebase

(using a program reducer).

(2) Properly generalize the key reasons

(to build generic features).

The end.

(1) Capture The Key Reasons

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

(1) Capture The Key Reasons

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

reduce →

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

(**FS:** "a<3 before assertion in the loop")

e.g., C-Reduce

- Use a program reducer to generate a feature program.
- The reduction preserves an invariant ϕ :

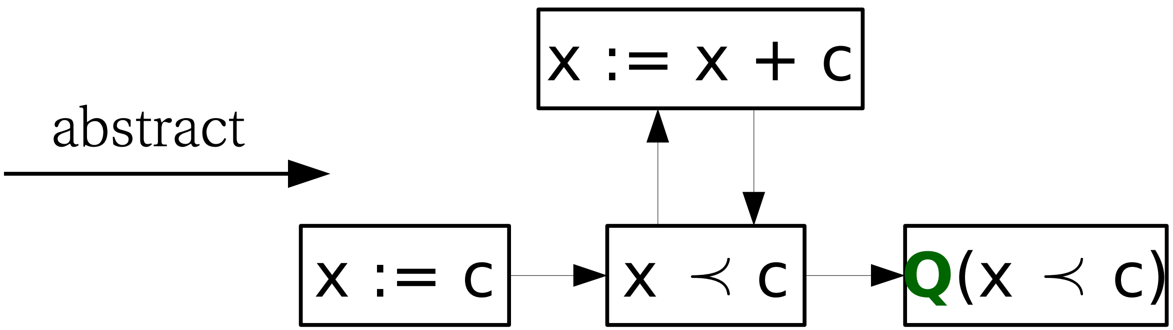
$$\phi(p, q) \equiv FI(p, q) = \texttt{unproven} \ \wedge \ FS(p, q) = \texttt{proven}$$

(2) Generalize The Key Reasons

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

(2) Generalize The Key Reasons

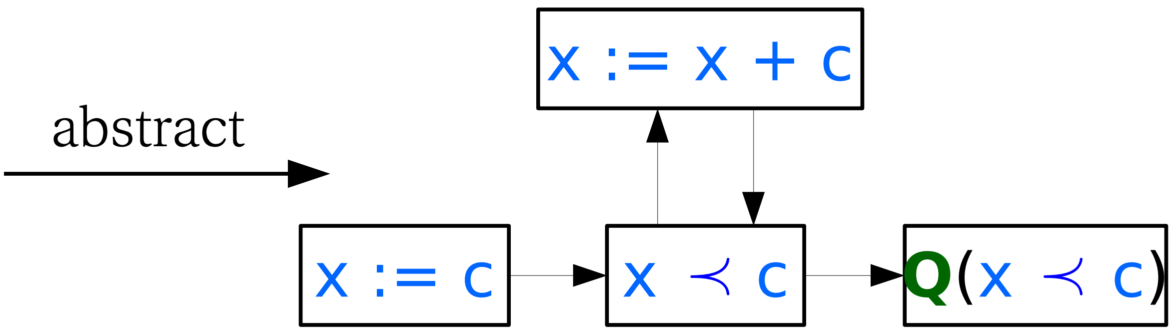
```
a = 0;
while (1) {
    if (a < 3)
        assert (a < 5);
    a++;
}
```



- Properly generalize the feature program to an abstract data-flow graph (= feature).

(2) Generalize The Key Reasons

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

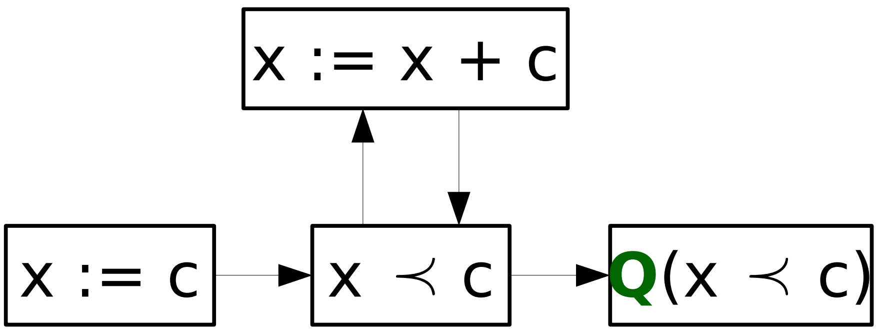


- Properly generalize the feature program to an abstract data-flow graph (= feature).
- The right level of abstraction is automatically identified by an iterative search and cross validation.

Generalization vs. Preservation

Feature Check = Graph Inclusion Check

feature:



original program:

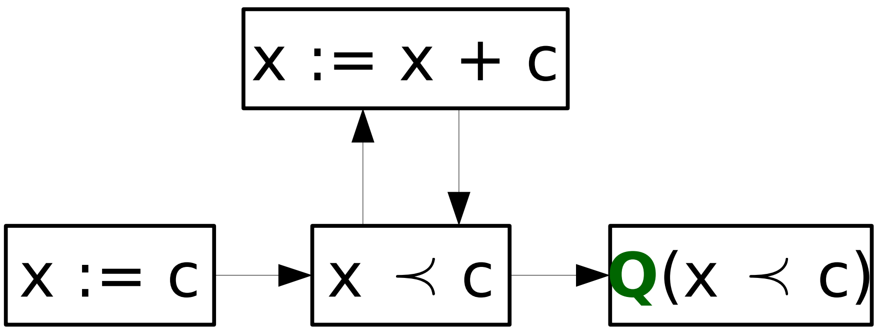
```
a = 0; b = 0;
while (1) {
    b = unknown();
    if (a > b)
        if (a < 3)
            assert (a < 5);
    a++;
}
```

Feature Check = Graph Inclusion Check

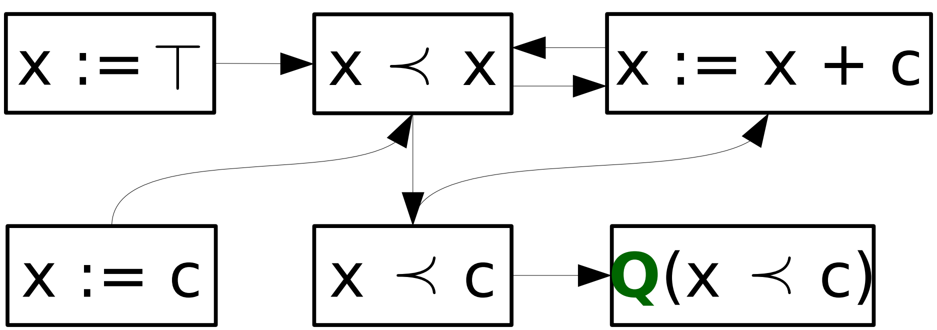
original program:

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

feature:

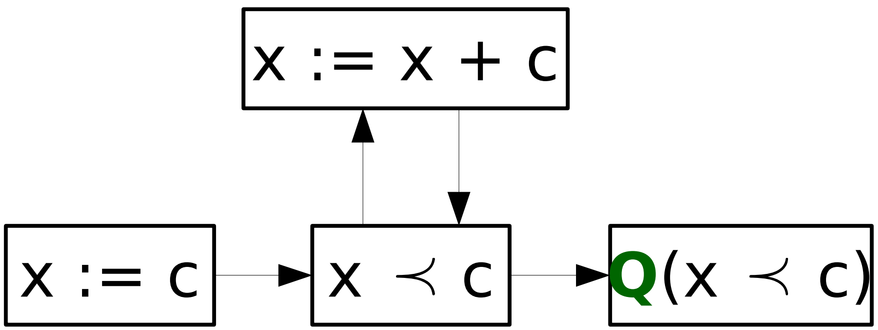


abstract data-flow graph:



Feature Check = Graph Inclusion Check

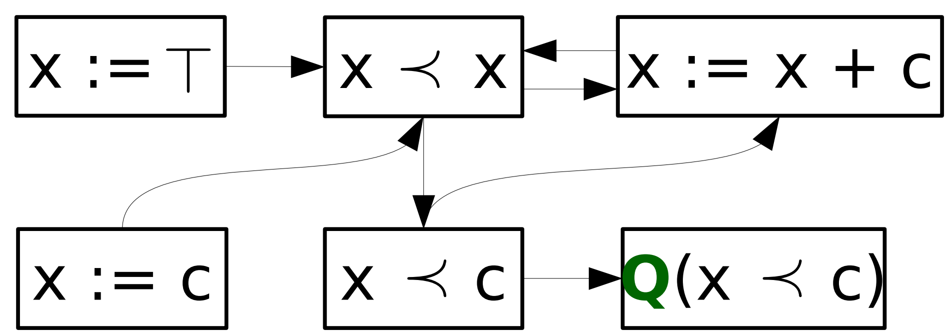
feature:



original program:

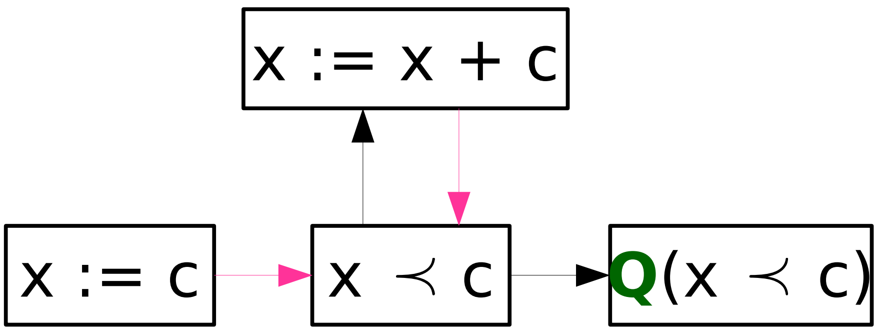
```
a = 0; b = 0;
while (1) {
    b = unknown();
    if (a > b)
        if (a < 3)
            assert (a < 5);
    a++;
}
```

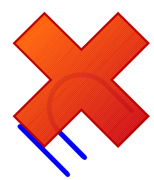
abstract data-flow graph:



Feature Check = Graph Inclusion Check

feature:

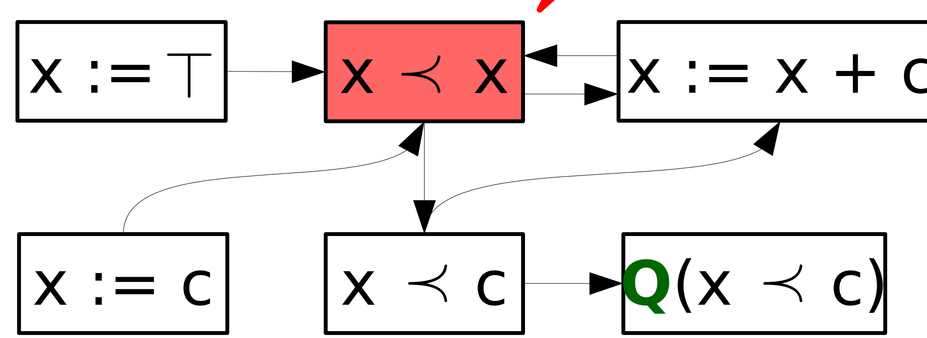




original program:

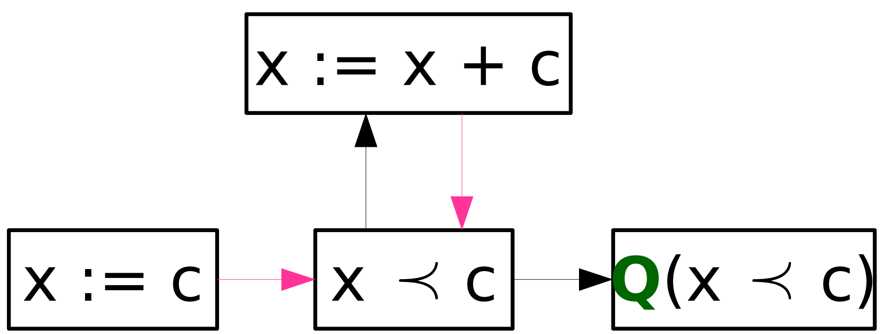
```
a = 0; b = 0;
while (1) {
    b = unknown();
    if (a > b)
        if (a < 3)
            assert (a < 5);
    a++;
}
```

abstract data-flow graph:



Feature Check = Graph Inclusion Check

feature:



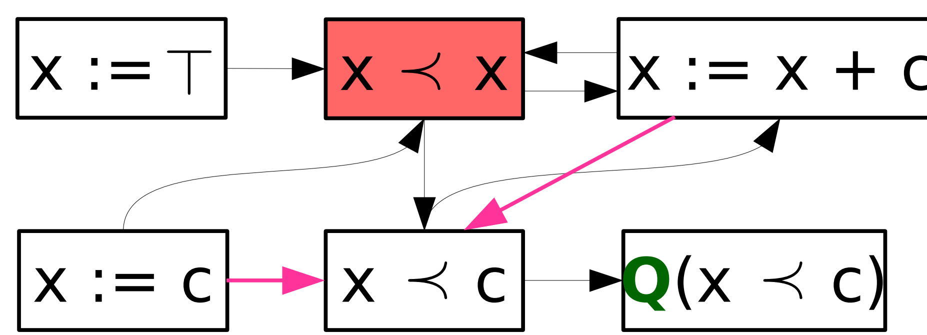
⊆

original program:

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

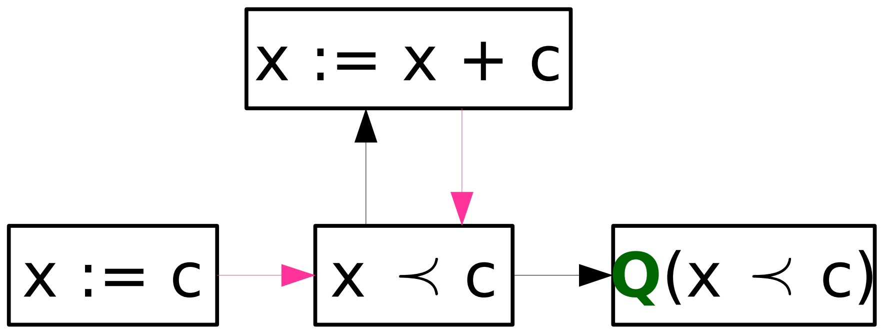
transitive closure

abstract data-flow graph:



Feature Check = Graph Inclusion Check

feature:



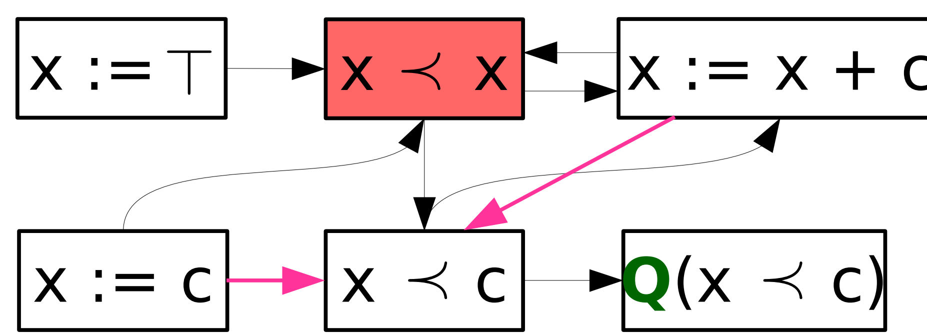
$\subseteq$

original program:

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

transitive closure

abstract data-flow graph:

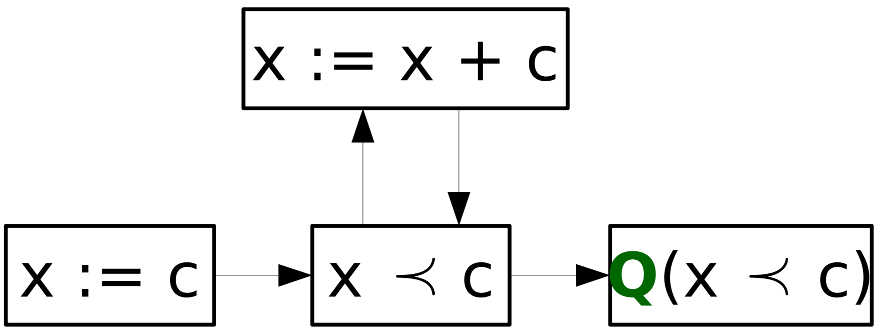


Removing noise:

- ✓ reducer (offline, feature)
- ✓ transitive closure (online, new pgm)

Feature Check = Graph Inclusion Check

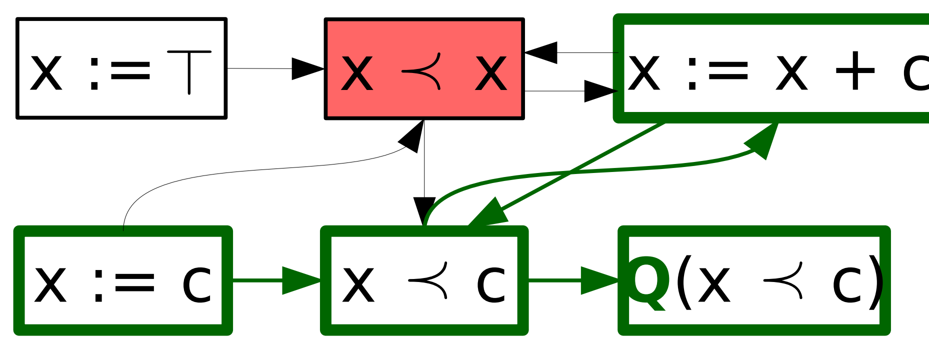
feature:



original program:

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

abstract data-flow graph:



Evaluation

- Static analyzer: Sparrow (https://github.com/ropas/sparrow)
- Reducer: C-Reduce [PLDI'12] (https://embed.cs.utah.edu/creduce)
- Three instance analyses for C
  - Partially flow-sensitive interval analysis
  - Partially flow-sensitive pointer analysis
  - Partial Octagon analysis
- 60 benchmark programs from Linux and GNU packages

Results: Effectiveness (Classifier)

- Partially flow-sensitive interval analysis

Trial	Query Prediction		#Proved Queries			Analysis Cost (sec)			Quality		(Oh et al. 2015)	
	Precision	Recall	FII (*a*)	FSI (*b*)	Ours (*c*)	FII (*d*)	FSI	Ours (*e*)	Prove	Cost	Prove	Cost
1	71.5 %	78.9 %	6,537	7,126	7,019	26.7	569.0	52.0	81.8 %	1.9x	56.6 %	2.0x
2	60.9 %	75.1 %	4,127	4,544	4,487	58.3	654.2	79.9	86.3 %	1.4x	49.2 %	2.4x
3	78.3 %	74.0 %	6,701	7,532	7,337	50.9	6,175.2	167.5	76.5 %	3.3x	51.1 %	3.4x
4	73.0 %	76.2 %	4,399	4,956	4,859	36.9	385.1	44.9	82.6 %	1.2x	54.8 %	1.2x
5	83.2 %	75.4 %	5,676	6,277	6,140	31.7	1,740.3	61.6	77.2 %	1.9x	65.6 %	1.8x
TOTAL	74.5 %	75.8 %	27,440	30,435	29,842	204.9	9,523.9	406.1	**80.2** %	**2.0x**	55.1 %	2.3x

- Partially flow-sensitive pointer analysis

Trial	Query Prediction		#Proved Queries			Analysis Cost (sec)			Quality	
	Precision	Recall	FIP	FSP	Ours	FIP	FSP	Ours	Prove	Cost
1	79.2 %	76.8 %	4,399	6,346	6,032	48.3	3,705.0	150.0	83.9 %	3.1x
2	78.3 %	77.2 %	7,029	8,650	8,436	48.9	651.4	74.0	86.8 %	1.5x
3	74.6 %	75.0 %	8,781	10,352	10,000	41.5	707.0	59.4	77.6 %	1.4x
4	73.9 %	76.0 %	10,559	12,914	12,326	51.1	4,107.0	164.3	75.0 %	3.2x
5	78.0 %	82.5 %	4,205	5,705	5,482	23.0	847.2	56.7	85.1 %	2.5x
TOTAL	76.7 %	77.4 %	34,973	43,967	42,276	212.9	10,017.8	504.6	**81.2** %	**2.4x**

- Partial Octagon analysis

Trial	Query Prediction		#Proved Queries			Analysis Cost (sec)			Quality		(Heo et al. 2016)	
	Precision	Recall	FSI	IMPCT	Ours	FSI	IMPCT	Ours	Prove	Cost	Prove	Cost
1	74.8 %	81.3 %	3,678	3,806	3,789	140.7	389.8	230.5	86.7 %	1.6 x	100.0 %	3.0 x
2	84.1 %	82.6 %	5,845	6,004	5,977	613.5	18,022.9	782.9	83.0 %	1.3 x	94.3 %	1.8 x
3	82.8 %	73.0 %	1,926	2,079	2,036	315.2	2,396.9	416.0	71.9 %	1.3 x	92.2 %	1.1 x
4	77.6 %	85.2 %	2,221	2,335	2,313	72.7	495.1	119.9	80.7 %	1.6 x	100.0 %	2.0 x
5	71.6 %	78.4 %	2,886	2,962	2,944	148.9	557.2	209.7	76.3 %	1.4 x	96.1 %	2.3 x
TOTAL	79.0 %	79.9 %	16,556	17,186	17,067	1,291.0	21,861.9	1,759.0	**81.1** %	**1.4** x	96.2 %	1.8 x

Results: Effectiveness (Analysis)

- Partially flow-sensitive interval analysis

Trial	Query Prediction		#Proved Queries			Analysis Cost (sec)			Quality		(Oh et al. 2015)	
	Precision	Recall	FSI (a)	FSI (b)	Ours (c)	FII (d)	FSI	Ours (e)	Prove	Cost	Prove	Cost
1	71.5 %	78.9 %	6,537	7,126	7,019	26.7	569.0	52.0	81.8 %	1.9x	56.6 %	2.0x
2	60.9 %	75.1 %	4,127	4,544	4,487	58.3	654.2	79.9	86.3 %	1.4x	49.2 %	2.4x
3	78.3 %	74.0 %	6,701	7,532	7,337	50.9	6,175.2	167.5	76.5 %	3.3x	51.1 %	3.4x
4	73.0 %	76.2 %	4,399	4,956	4,859	36.9	385.1	44.9	82.6 %	1.2x	54.8 %	1.2x
5	83.2 %	75.4 %	5,676	6,277	6,140	31.7	1,740.3	61.6	77.2 %	1.9x	65.6 %	1.8x
TOTAL	74.5 %	75.8 %	27,440	30,435	29,842	204.9	9,523.9	406.1	**80.2 %**	**2.0x**	55.1 %	2.3x

- Partially flow-sensitive pointer analysis

Trial	Query Prediction		#Proved Queries			Analysis Cost (sec)			Quality	
	Precision	Recall	FIP	FSP	Ours	FIP	FSP	Ours	Prove	Cost
1	79.2 %	76.8 %	4,399	6,346	6,032	48.3	3,705.0	150.0	83.9 %	3.1x
2	78.3 %	77.2 %	7,029	8,650	8,436	48.9	651.4	74.0	86.8 %	1.5x
3	74.6 %	75.0 %	8,781	10,352	10,000	41.5	707.0	59.4	77.6 %	1.4x
4	73.9 %	76.0 %	10,559	12,914	12,326	51.1	4,107.0	164.3	75.0 %	3.2x
5	78.0 %	82.5 %	4,205	5,705	5,482	23.0	847.2	56.7	85.1 %	2.5x
TOTAL	76.7 %	77.4 %	34,973	43,967	42,276	212.9	10,017.8	504.6	**81.2 %**	**2.4x**

- Partial Octagon analysis

Trial	Query Prediction		#Proved Queries			Analysis Cost (sec)			Quality		(Heo et al. 2016)	
	Precision	Recall	FSI	IMPCT	Ours	FSI	IMPCT	Ours	Prove	Cost	Prove	Cost
1	74.8 %	81.3 %	3,678	3,806	3,789	140.7	389.8	230.5	86.7 %	1.6 x	100.0 %	3.0 x
2	84.1 %	82.6 %	5,845	6,004	5,977	613.5	18,022.9	782.9	83.0 %	1.3 x	94.3 %	1.8 x
3	82.8 %	73.0 %	1,926	2,079	2,036	315.2	2,396.9	416.0	71.9 %	1.3 x	92.2 %	1.1 x
4	77.6 %	85.2 %	2,221	2,335	2,313	72.7	495.1	119.9	80.7 %	1.6 x	100.0 %	2.0 x
5	71.6 %	78.4 %	2,886	2,962	2,944	148.9	557.2	209.7	76.3 %	1.4 x	96.1 %	2.3 x
TOTAL	79.0 %	79.9 %	16,556	17,186	17,067	1,291.0	21,861.9	1,759.0	**81.1 %**	**1.4 x**	96.2 %	1.8 x

Results: Comparison

- Partially flow-sensitive interval analysis

Trial	Query Prediction		#Proved Queries			Analysis Cost (sec)			Quality		(Oh et al. 2015)	
	Precision	Recall	FII (*a*)	FSI (*b*)	Ours (*c*)	FII (*d*)	FSI	Ours (*e*)	Prove	Cost	Prove	Cost
1	71.5 %	78.9 %	6,537	7,126	7,019	26.7	569.0	52.0	81.8 %	1.9x	56.6 %	2.0x
2	60.9 %	75.1 %	4,127	4,544	4,487	58.3	654.2	79.9	86.3 %	1.4x	49.2 %	2.4x
3	78.3 %	74.0 %	6,701	7,532	7,337	50.9	6,175.2	167.5	76.5 %	3.3x	51.1 %	3.4x
4	73.0 %	76.2 %	4,399	4,956	4,859	36.9	385.1	44.9	82.6 %	1.2x	54.8 %	1.2x
5	83.2 %	75.4 %	5,676	6,277	6,140	31.7	1,740.3	61.6	77.2 %	1.9x	65.6 %	1.8x
TOTAL	74.5 %	75.8 %	27,440	30,435	29,842	204.9	9,523.9	406.1	**80.2** %	**2.0x**	55.1 %	2.3x

- Partial Octagon analysis

Trial	Query Prediction		#Proved Queries			Analysis Cos[illegible]			[illegible]lity		(He[illegible]	
	Precision	Recall	FSI	IMPCT	Ours	FSI	IMPCT	[illegible]	[illegible]	Cost	Pr[illegible]	[illegible]
1	74.8 %	81.3 %	3,678	3,806	3,789	140.7	389.8	[illegible]	[illegible]	1.6 x	100.0 %	[illegible]
2	84.1 %	82.6 %	5,845	6,004	5,977	613.5	18,022.9	782.9	[illegible]	1.3 x	94.3 %	[illegible]
3	82.8 %	73.0 %	1,926	2,079	2,036	315.2	2,396.9	416.0	[illegible]	1.3 x	92.2 %	[illegible]
4	77.6 %	85.2 %	2,221	2,335	2,313	72.7	495.1	119.9	[illegible]	1.6 x	100.0 %	2.0 x
5	71.6 %	78.4 %	2,886	2,962	2,944	148.9	557.2	209.7	76.3 %	1.4 x	96.1 %	2.3 x
TOTAL	79.0 %	79.9 %	16,556	17,186	17,067	1,291.0	21,861.9	1,759.0	**81.1** %	**1.4** x	96.2 %	1.8 x

automatic

manual

- Consistently perform well on a wide range of programs. (↔ wide variation)
- No clear conclusion (different approaches and learning algorithms)

Results: Generated Features (Top 2)

- Partially flow-sensitive interval analysis

feature program 1:

```
int buf [10];
for (i = 0; i < 7; i++) {
    buf[i] = 0;  // Query
}
```

feature program 2:

```
k = 255; p = malloc (k);
while (k > 0) {
    *(p + k) = 0; // Query
    k-- ;
}
```

- Access to a consecutive memory region in a loop
- Bounded indice by a constant

Results: Generated Features (Top 2)

- Partially flow-sensitive pointer analysis

feature program 1:

```
int j = 16; q = malloc(j)
if (q == 0)
   return;
else *q = 0;  // Query
```

feature program 2:

```
r = malloc(v);
r = &a;
*r = 0;  // Query
```

- Null-check before buffer access
- Strong update by the address of another variable

Results: Generated Features (Top 2)

- Partial Octagon analysis

feature program 1:

```
size = POS_NUM;
arr = malloc(size);
arr[size-1] = 0; // Query
```

feature program 2:

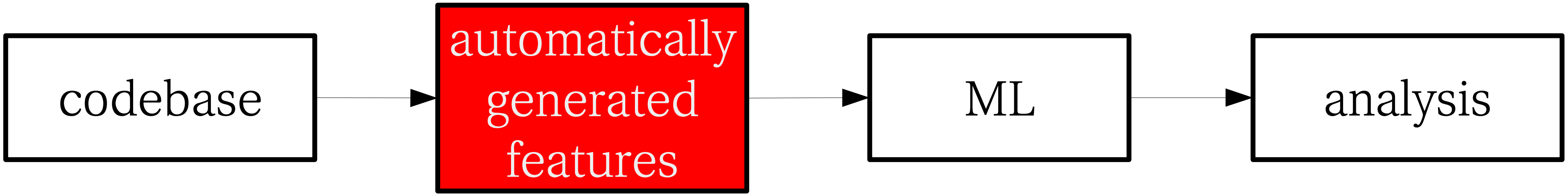
```
idx = POS_NUM;
buf = malloc(idx);
for (n = 0; n < idx; n++) {
    buf[n] = 0;  // Query
}
```

- Array of a positive size
  - e.g., when `POS_NUM = [1,+oo]` in the flow-sensitive interval analysis
- Index related to the size in a simple linear way

Caveats: Expressiveness of Features

- Our feature representation is expressive enough, but not perfect, e.g.,
  - "x and y results in finite intervals after analysis." ✘
  - "2^k type of integers are important constants." ✘

Summary



- “Features” in data-driven static analysis
  - By reducing programs
  - As generalized graphs ($\leftrightarrow$ program text)