Lecture 6

Static Analysis Engineering Techniques

Hakjoo Oh
2016 Fall

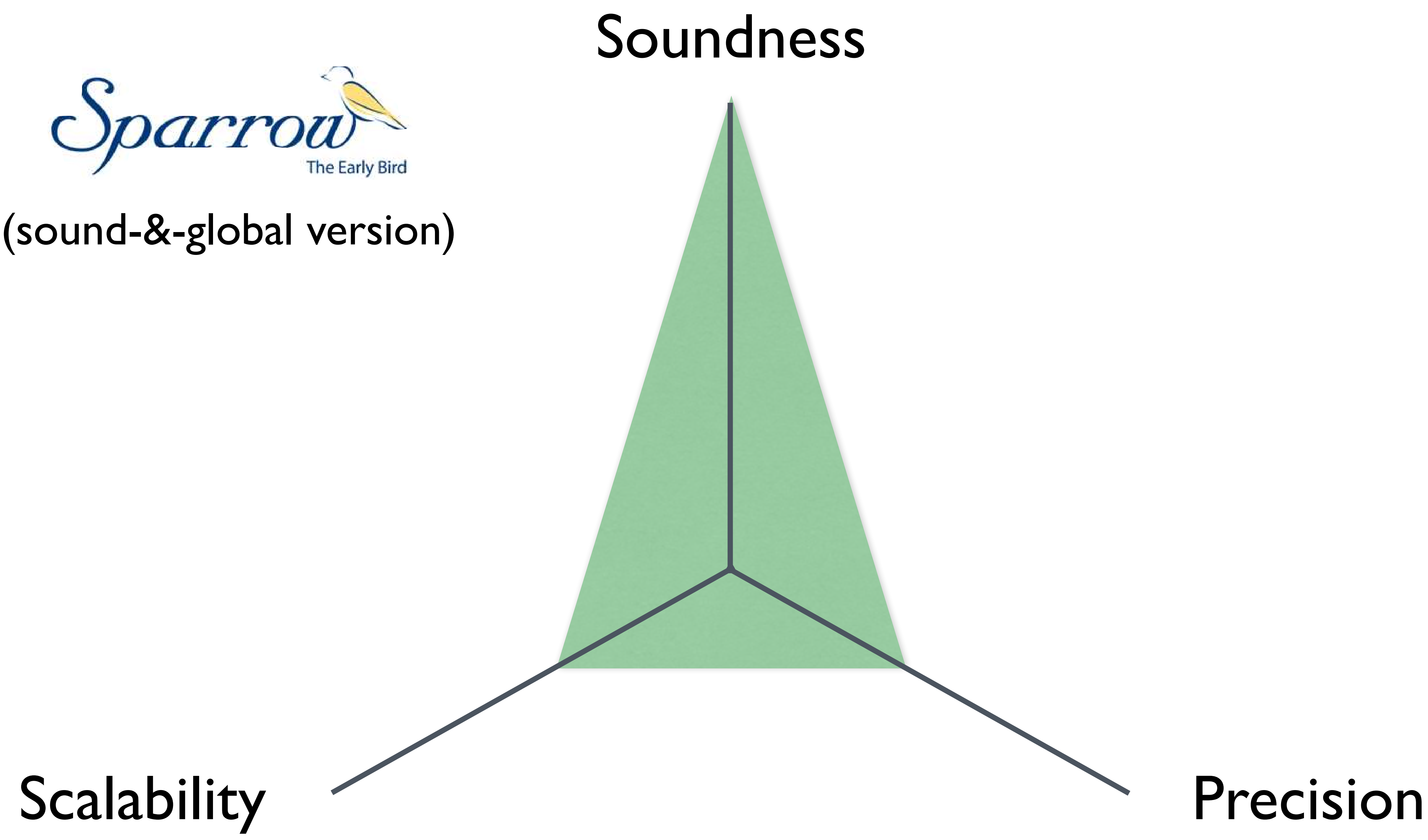
A Story

- In 2007, I participated commercializing Sparrow The Early Bird
  - memory-bug-finding tool for full C, non domain-specific
  - designed in abstract interpretation framework
  - sound in design, unsound yet scalable in reality
- Realistic workbench available
  - “let’s try to scale-up its sound & global analysis version”

The Elusive Three in Static Analysis

Sparrow
The Early Bird

(sound-&-global version)



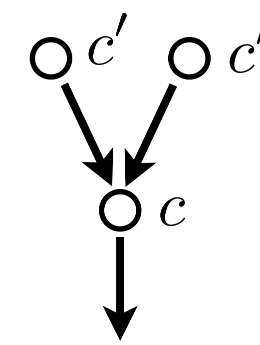
Typical Static Analyzer for C

- One abstract state $\in \hat{\mathbb{S}}$ that subsumes all reachable states at each program point

$$\begin{aligned} [\![\hat{P}]\!] \in \mathbb{C} \to \hat{\mathbb{S}} &= \mathit{fix} \hat{F} \\ \hat{\mathbb{S}} &= \hat{\mathbb{L}} \to \hat{\mathbb{V}} \end{aligned}$$

- Abstract semantic function

$$\begin{aligned} \hat{F} &\in (\mathbb{C} \to \hat{\mathbb{S}}) \to (\mathbb{C} \to \hat{\mathbb{S}}) \\ \hat{F}(\hat{X}) &= \lambda c \in \mathbb{C}. \hat{f}_c(\bigsqcup_{c' \hookrightarrow c} \hat{X}(c')) \end{aligned}$$



$\hat{f}_c \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$: abstract semantics at point c

Computing $\bigsqcup_{i \in \mathbb{N}} \hat{F}^i(\hat{\bot})$

$$\hat{F}(\hat{X}) = \lambda c \in \mathbb{C}.\hat{f}_c(\bigsqcup_{c' \hookrightarrow c} \hat{X}(c')).$$

$\hat{X}, \hat{X}' \in \mathbb{C} \to \hat{\mathbb{S}}$
$\hat{f}_c \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$

$\hat{X} := \hat{X}' := \lambda c.\bot$

repeat

  $\hat{X}' := \hat{X}$

  for all $c \in \mathbb{C}$ **do**

    $\hat{X}(c) := \hat{f}_c(\bigsqcup_{c' \hookrightarrow c} X(c'))$

until $\hat{X} \sqsubseteq \hat{X}'$

Naive fixpoint algorithm

$W \in \mathit{Worklist} = 2^{\mathbb{C}}$
$\hat{X} \in \mathbb{C} \to \hat{\mathbb{S}}$
$\hat{f}_c \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$

$W := \mathbb{C}$
$\hat{X} := \lambda c.\bot$

repeat

  $c :=$ choose(W)

  $\hat{s} := \hat{f}_c(\bigsqcup_{c' \hookrightarrow c} X(c'))$

  if $\hat{s} \not\sqsubseteq \hat{X}(c)$

    $W := W \cup \{c' \in \mathbb{C} \mid c \hookrightarrow c'\}$

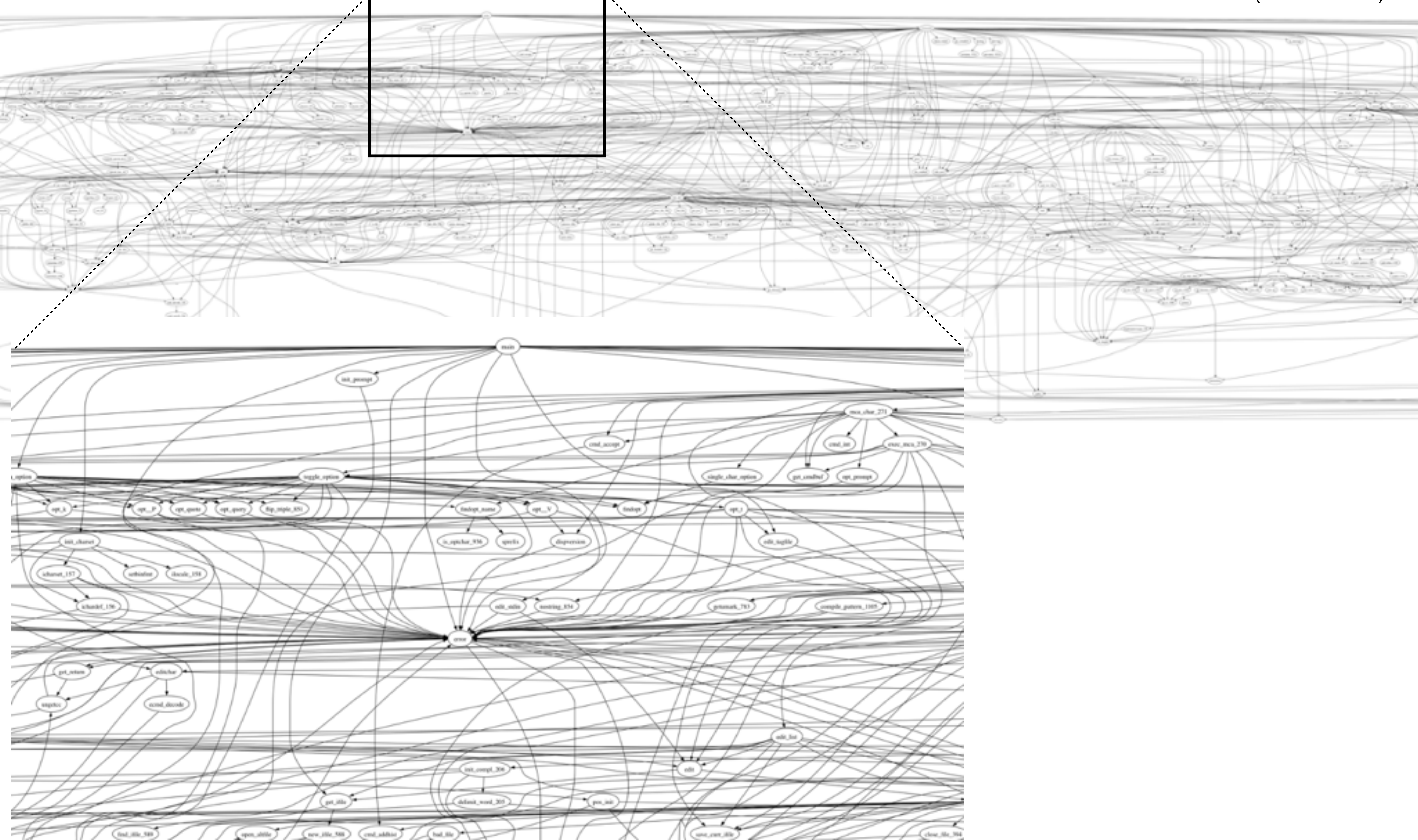
    $\hat{X}(c) := \hat{X}(c) \sqcup \hat{s}$

until $W = \emptyset$

Worklist algorithm

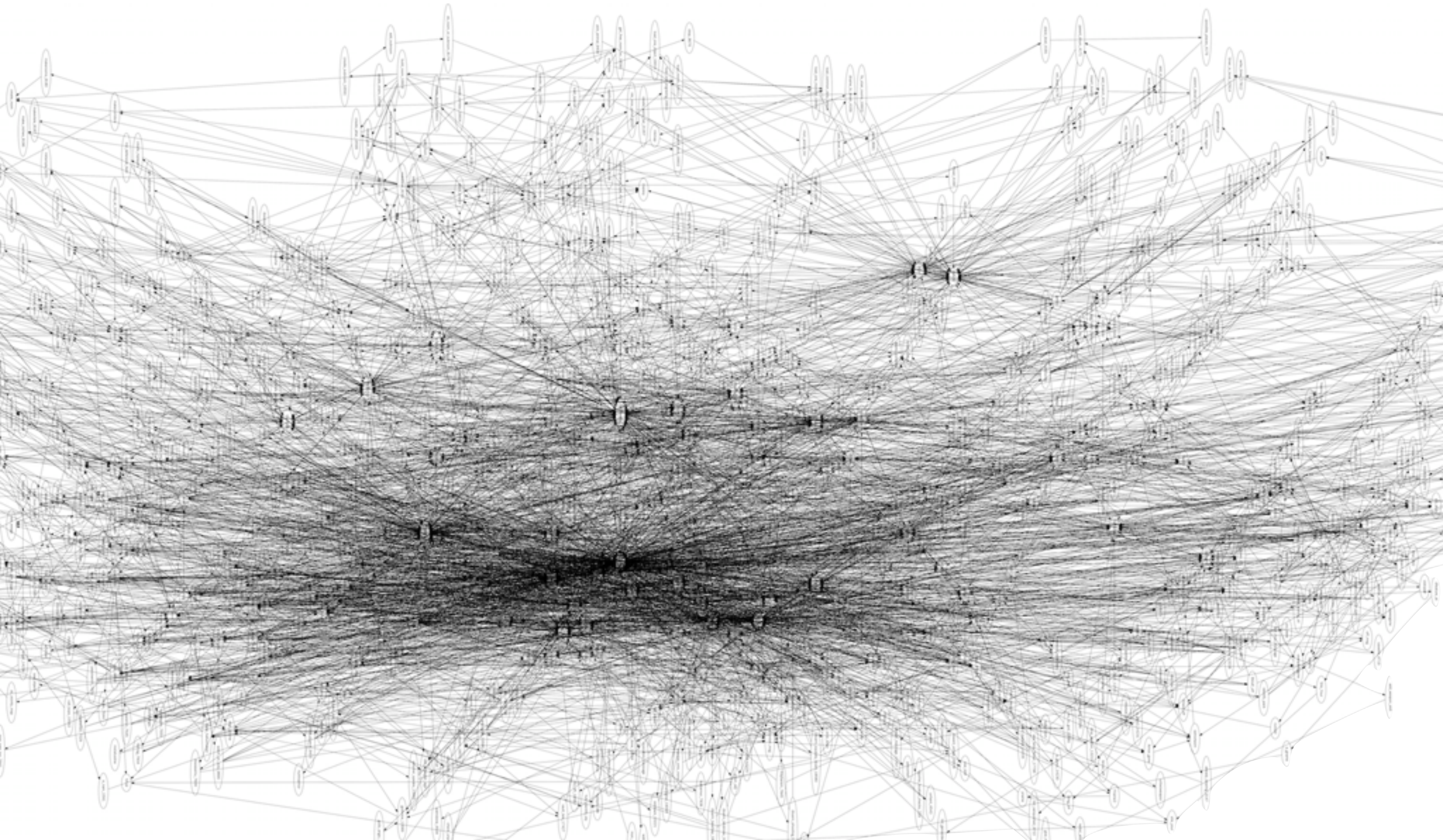
Real C Programs

less-382 (23,822 LoC)



Real C Programs

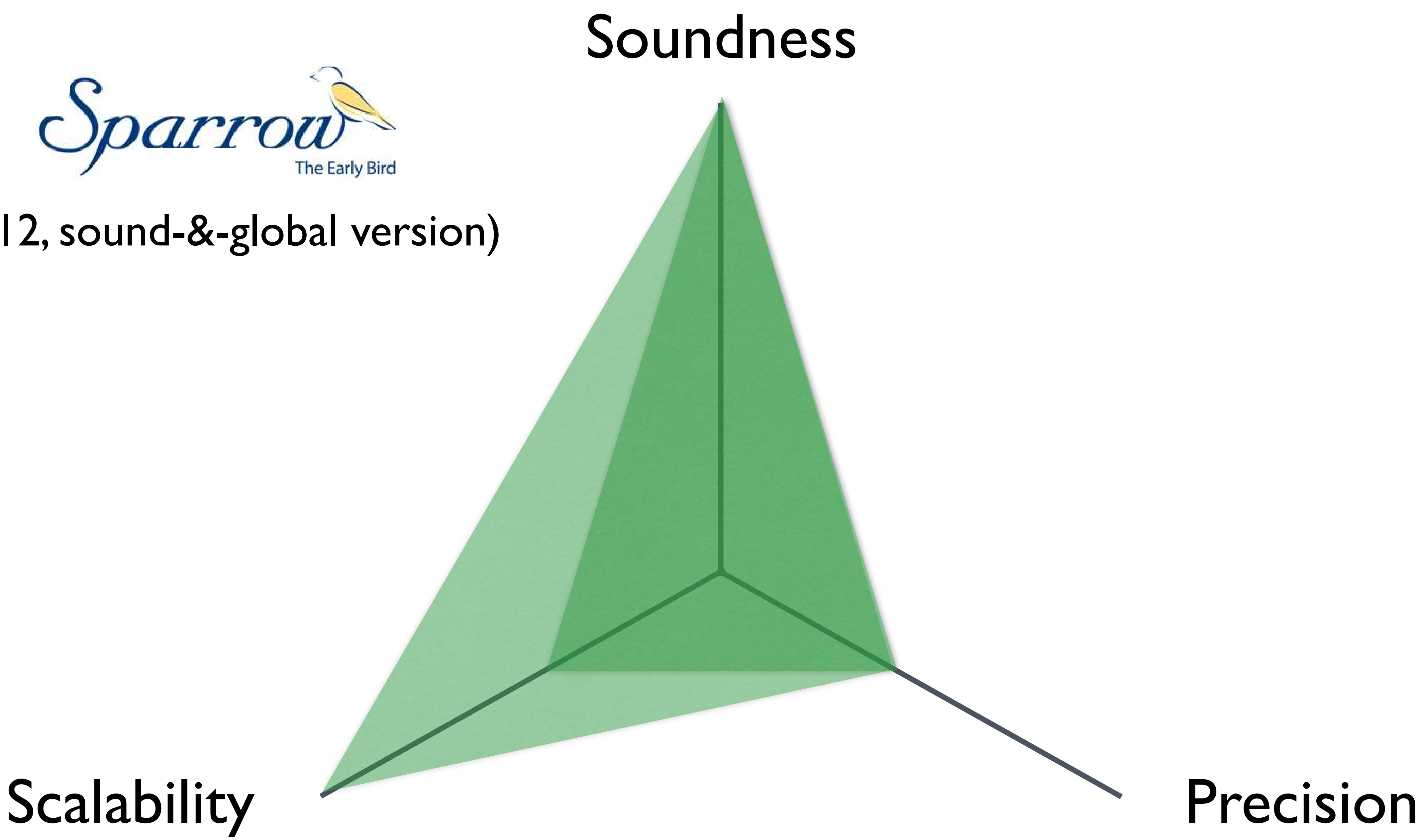
nethack-3.3.0 (211KLoC)



The First Goal: Scalability

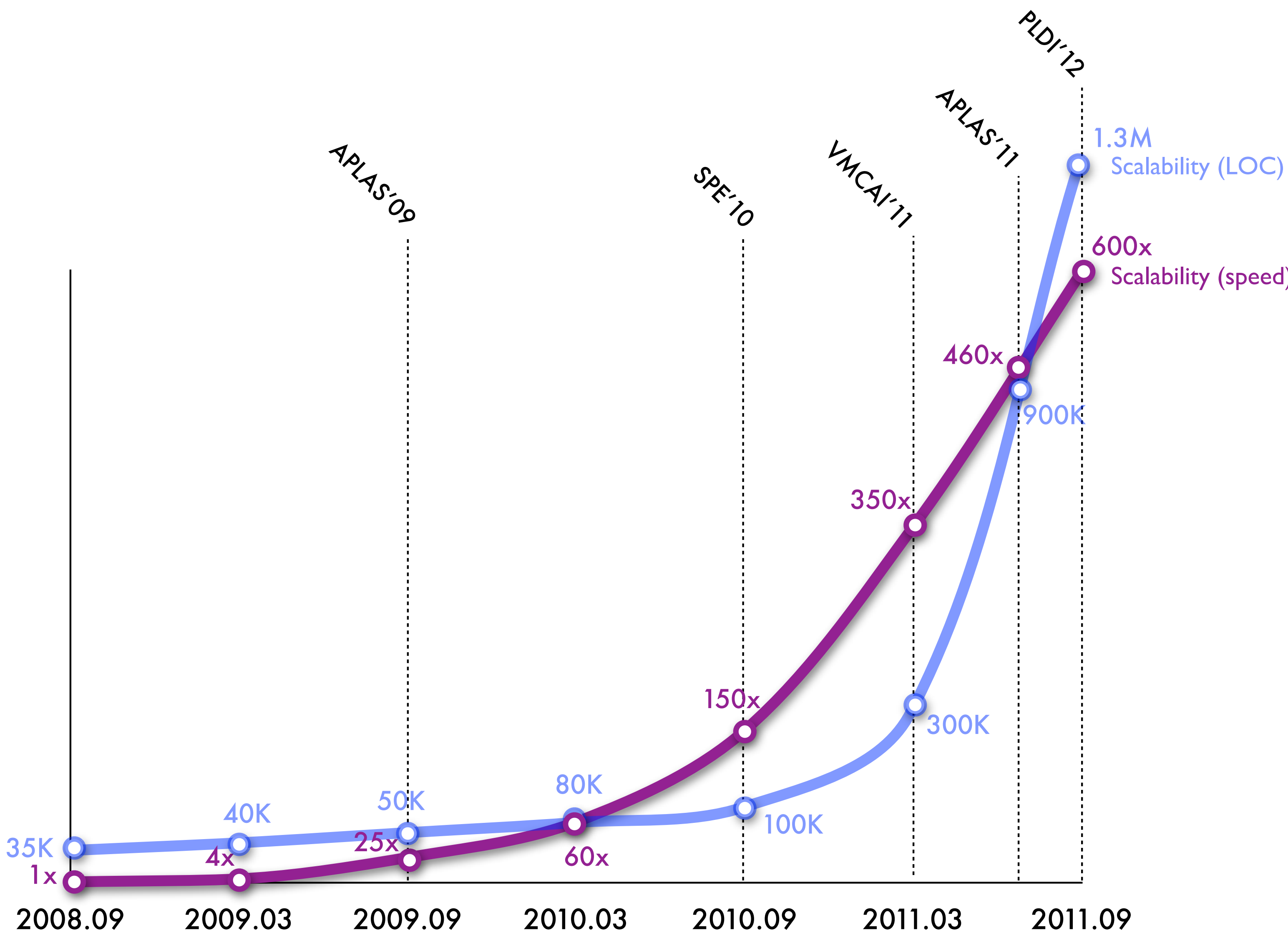
Sparrow
The Early Bird

(2012, sound-&-global version)



Scalability Improvement

Sparrow The Early Bird sound & global analysis version



- < 1.4M in 10hr with intervals
- < 0.14M in 20hrs with octagons

Key Idea: Localization

“Right Part at Right Moment”

“framing”
“abstract garbage collection”

- Spatial localization [VMCAI’11,APLAS’11,SCP’13]
- Temporal localization [PLDI’12,TOPLAS’14]

Scalability Improvement

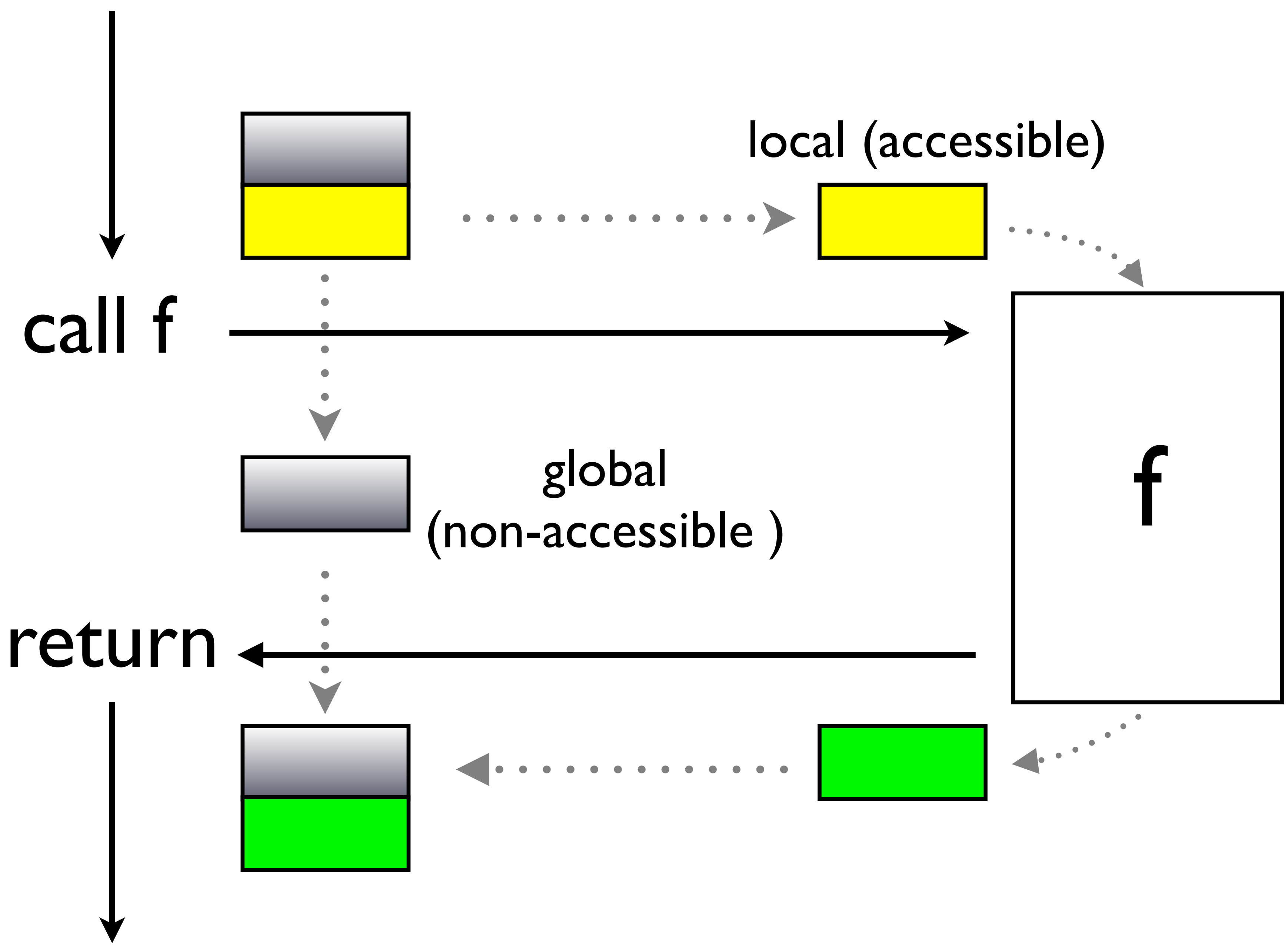
Programs	LOC	Interval$_{vanilla}$		Interval$_{base}$		Spd$\uparrow_1$	Mem$\downarrow_1$	Interval$_{sparse}$						Spd$\uparrow_2$	Mem$\downarrow_2$
		Time	**Mem**	**Time**	**Mem**			**Dep**	**Fix**	**Total**	**Mem**	$\hat{D}(c)$	$\hat{U}(c)$		
gzip-1.2.4a	7K	772	240	14	65	55 x	73 %	2	1	3	63	2.4	2.5	5 x	3 %
bc-1.06	13K	1,270	276	96	126	13 x	54 %	4	3	7	75	4.6	4.9	14 x	40 %
tar-1.13	20K	12,947	881	338	177	38 x	80 %	6	2	8	93	2.9	2.9	42 x	47 %
less-382	23K	9,561	1,113	1,211	378	8 x	66 %	27	6	33	127	11.9	11.9	37 x	66 %
make-3.76.1	27K	24,240	1,391	1,893	443	13 x	68 %	16	5	21	114	5.8	5.8	90 x	74 %
wget-1.9	35K	44,092	2,546	1,214	378	36 x	85 %	8	3	11	85	2.4	2.4	110 x	78 %
screen-4.0.2	45K	∞	N/A	31,324	3,996	N/A	N/A	724	43	767	303	53.0	54.0	41 x	92 %
a2ps-4.14	64K	∞	N/A	3,200	1,392	N/A	N/A	31	9	40	353	2.6	2.8	80 x	75 %
bash-2.05a	105K	∞	N/A	1,683	1,386	N/A	N/A	45	22	67	220	3.0	3.0	25 x	84 %
lsh-2.0.4	111K	∞	N/A	45,522	5,266	N/A	N/A	391	80	471	577	21.1	21.2	97 x	89 %
sendmail-8.13.6	130K	∞	N/A	∞	N/A	N/A	N/A	517	227	744	678	20.7	20.7	N/A	N/A
nethack-3.3.0	211K	∞	N/A	∞	N/A	N/A	N/A	14,126	2,247	16,373	5,298	72.4	72.4	N/A	N/A
vim60	227K	∞	N/A	∞	N/A	N/A	N/A	17,518	6,280	23,798	5,190	180.2	180.3	N/A	N/A
emacs-22.1	399K	∞	N/A	∞	N/A	N/A	N/A	29,552	8,278	37,830	7,795	285.3	285.5	N/A	N/A
python-2.5.1	435K	∞	N/A	∞	N/A	N/A	N/A	9,677	1,362	11,039	5,535	108.1	108.1	N/A	N/A
linux-3.0	710K	∞	N/A	∞	N/A	N/A	N/A	26,669	6,949	33,618	20,529	76.2	74.8	N/A	N/A
gimp-2.6	959K	∞	N/A	∞	N/A	N/A	N/A	3,751	123	3,874	3,602	4.1	3.9	N/A	N/A
ghostscript-9.00	1,363K	∞	N/A	∞	N/A	N/A	N/A	14,116	698	14,814	6,384	9.7	9.7	N/A	N/A

none

spatial localization

spatial+temporal localization

Spatial Localization



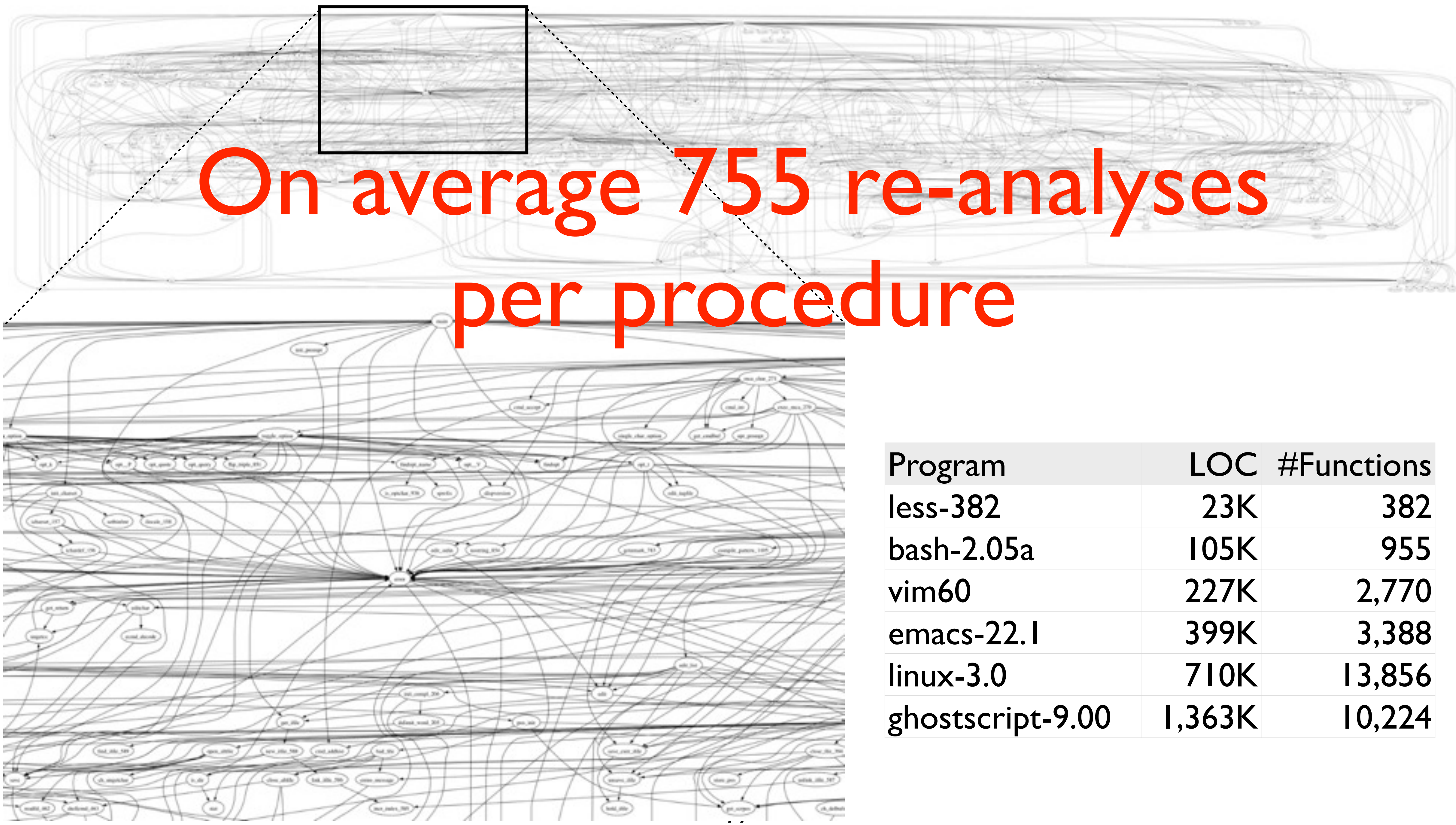
Benefits

```
int g;

int f() {...}

int main() {
  g = 0;
  f();

  g = 1;
  f();
}
```

f does not access g

Vital in Practice

less-382 (23,822 LOC)

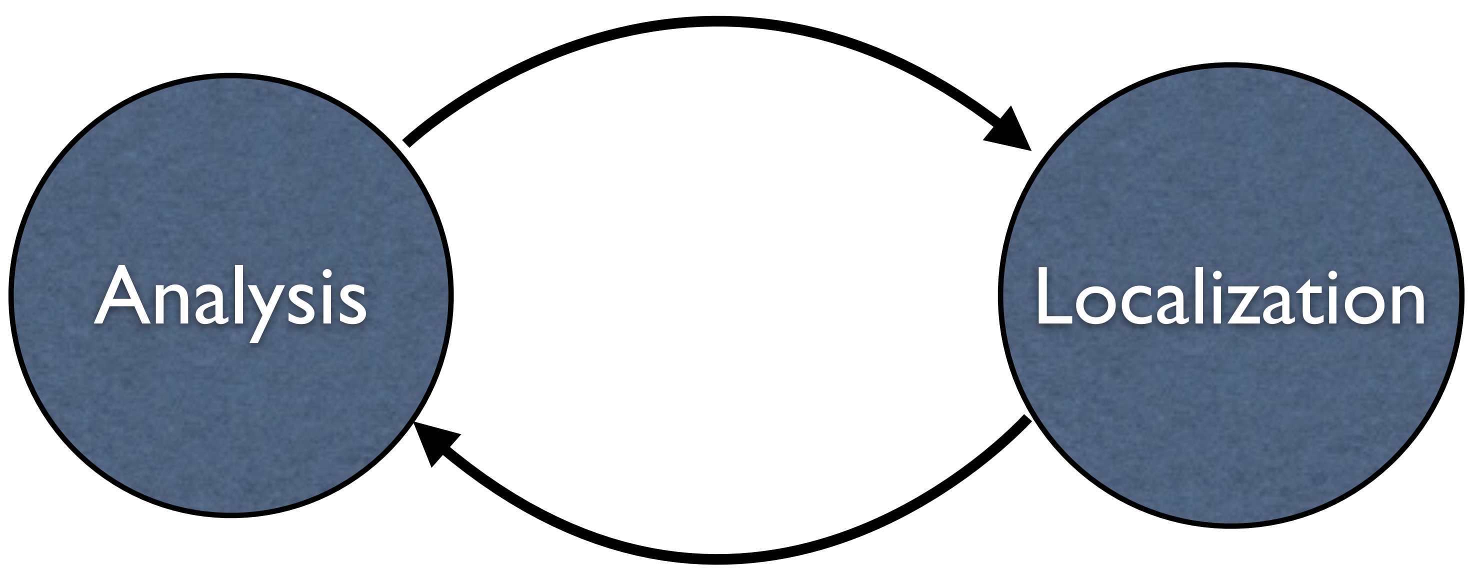


On average 755 re-analyses per procedure

Program	LOC	#Functions
less-382	23K	382
bash-2.05a	105K	955
vim60	227K	2,770
emacs-22.1	399K	3,388
linux-3.0	710K	13,856
ghostscript-9.00	1,363K	10,224

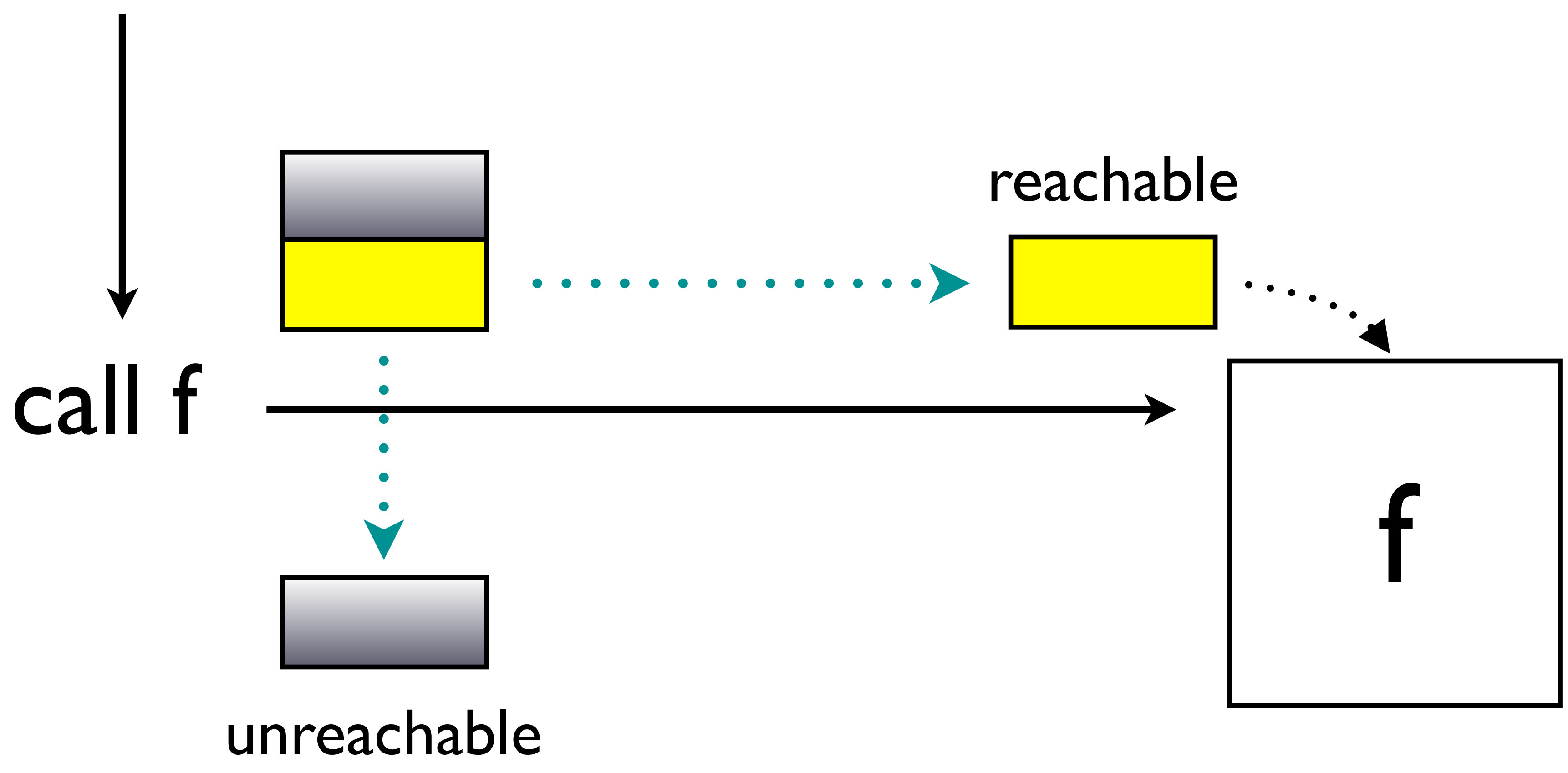
A Catch-22 Situation

The optimal localization is impossible



Reachability-based Localization

- Remove the unreachable from params and globals



Reachability-based Localization

$$\mathcal{R}(f_x, \hat{m}) = \mathsf{Reach}(\mathsf{Globals}, \hat{m}) \cup \mathsf{Reach}(\{x\}, \hat{m})$$

$$\mathsf{Reach}(X, \hat{m}) = \mathsf{lfp}(\lambda Y.X \cup \mathsf{OneHop}(Y, \hat{m}))$$

$$\mathsf{OneHop}(X, \hat{m}) = \bigcup_{x \in X} \hat{m}(x).2 \cup \{l \mid \langle l, o, s\rangle \in \hat{m}(x).3\} \cup \{\langle l, f\rangle \mid \langle l, \{f\}\rangle \in \hat{m}(x).4\}$$

$$\hat{f}\ \texttt{call(}f_x\texttt{,}e\texttt{)}\ \hat{m} \quad = \quad \hat{m}'|_{\mathcal{R}(f_x, \hat{m}')} \text{ where } \hat{m}' = \hat{m}\{\hat{\mathcal{V}}(e)(\hat{m}) /\!\!/ \{x\}\}$$

Key Observation

Reachability is too conservative

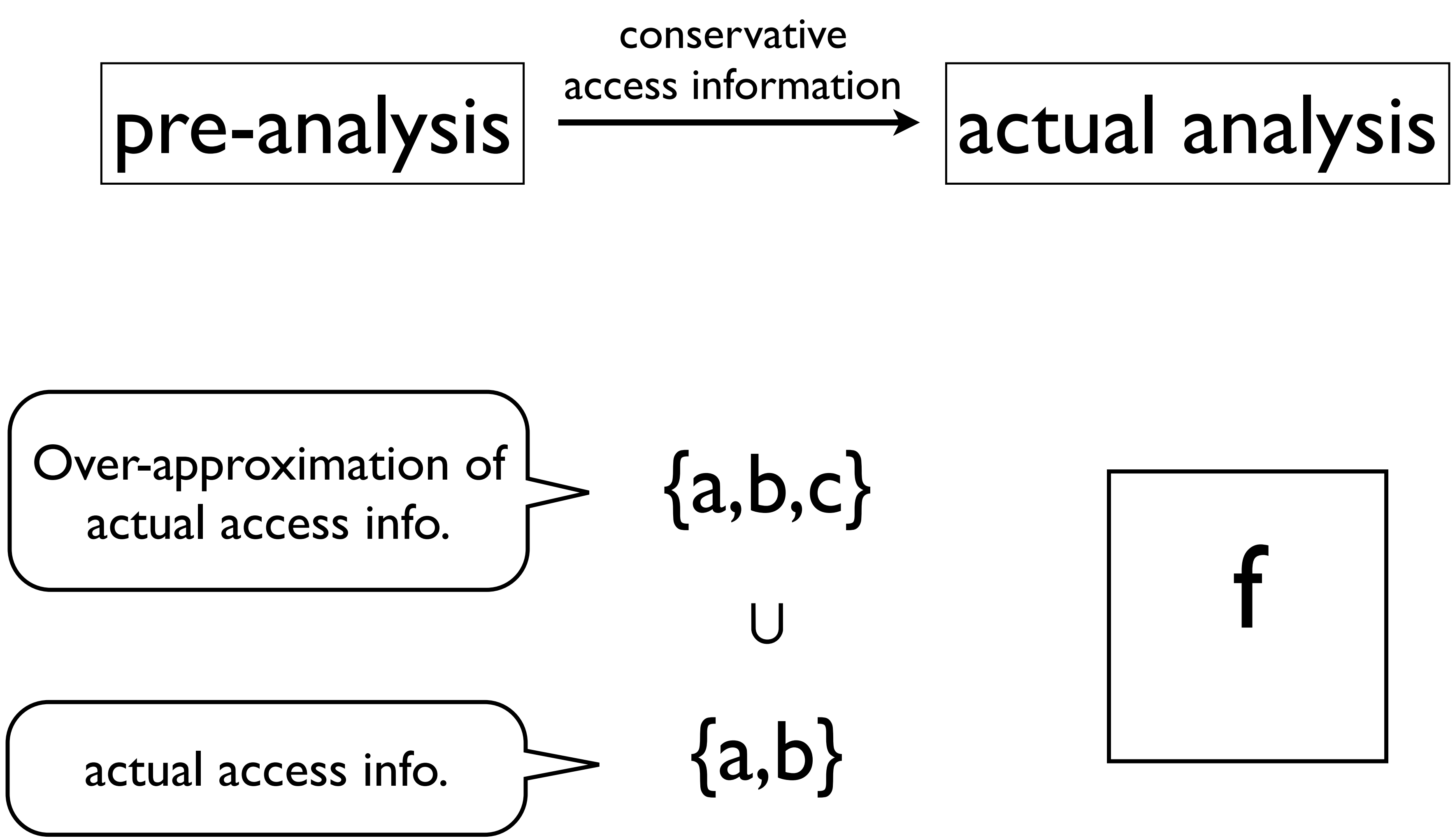
Program	LOC	accessed memory / reachable memory	
spell-1.0	2,213	5 / 453	(1.1%)
barcode-0.96	4,460	19 / 1175	(1.6%)
httptunnel-3.3	6,174	10 / 673	(1.5%)
gzip-1.2.4a	7,327	22 / 1002	(2.2%)
jwhois-3.0.1	9,344	28 / 830	(3.4%)
parser	10,900	75 / 1787	(4.2%)
bc-1.06	13,093	24 / 824	(2.9%)
less-290	18,449	86 / 1546	(5.6%)

average : 4%

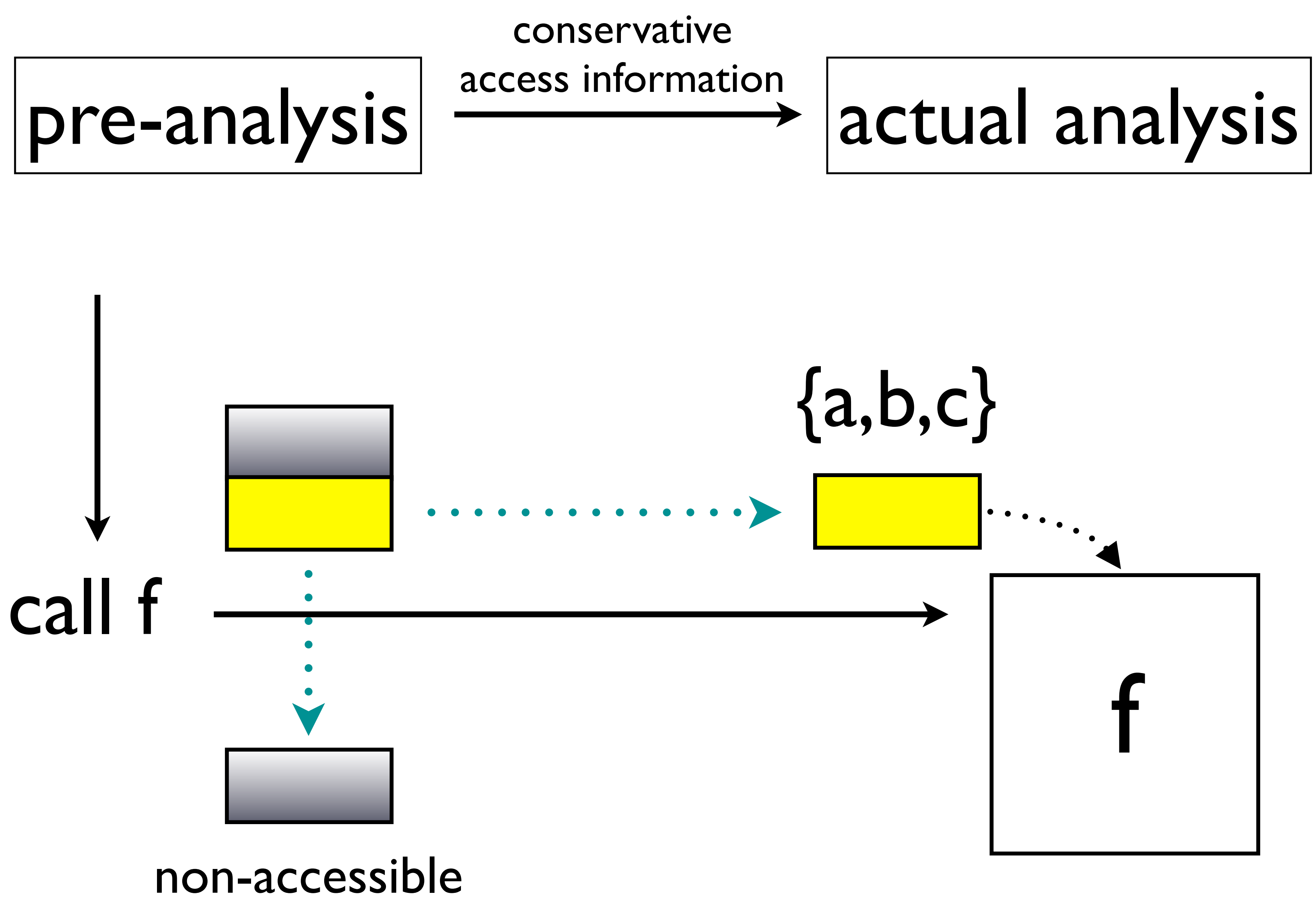
VMCAI'11

Access-based Localization

- Staging the analysis into two phases



Access-based Localization



Our Pre-analysis

- abstract domain

$$\mathbb{C} \to \hat{\mathbb{S}} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{\mathbb{S}}$$

$$\alpha = \lambda \hat{X}. \bigsqcup_{c \in \mathbb{C}} \hat{X}(c)$$

$$\gamma = \lambda \hat{s}. \lambda c \in \mathbb{C}. \hat{s}$$

- abstract semantic function

$$\hat{F}_p = \lambda \hat{s}. (\bigsqcup_{c \in \mathbb{C}} \hat{f}_c(\hat{s}))$$

$$\hat{s}_{pre} = \mathbf{fix} \hat{F}_p$$

Implementation

$$\hat{f}_c \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$$

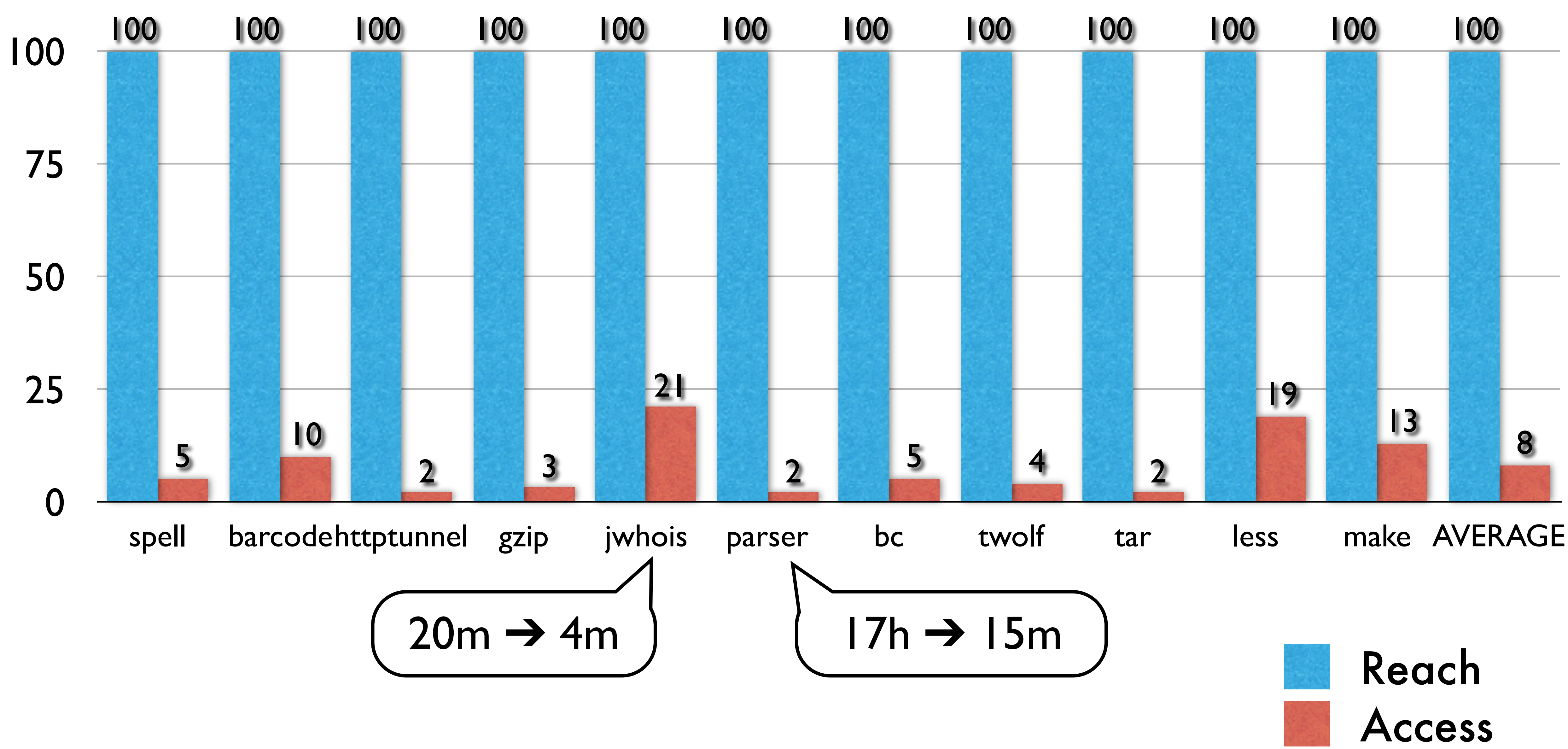
$$\hat{f}_c(\hat{s}) = \begin{cases} \hat{s}[\hat{\mathcal{L}}(lv)(\hat{s}) \overset{w}{\mapsto} \hat{\mathcal{V}}(e)(\hat{s})] & \mathsf{cmd}(c) = lv := e \\ \hat{s}[\hat{\mathcal{L}}(lv)(\hat{s}) \overset{w}{\mapsto} \langle \bot, \bot, \{\langle l, [0,0], \hat{\mathcal{V}}(e)(\hat{s}).1\rangle\}, \bot\rangle] & \mathsf{cmd}(c) = lv := \mathsf{alloc}(\texttt{[}e\texttt{]}_l) \\ \hat{s}[\hat{\mathcal{L}}(lv)(\hat{s}) \overset{w}{\mapsto} \langle \bot, \bot, \bot, \{\langle l, \{x\}\rangle\}\rangle] & \mathsf{cmd}(c) = lv := \mathsf{alloc}(\texttt{\{}x\texttt{\}}_l) \\ \hat{s}[x \mapsto \langle \hat{s}(x).1 \sqcap [-\infty, \mathsf{u}(\hat{\mathcal{V}}(e)(\hat{s}).1)], \hat{s}(x).2, \hat{s}(x).3, \hat{s}(x).4\rangle] & \mathsf{cmd}(c) = \mathsf{assume}(x < e) \\ \boxed{\hat{s}[x \mapsto \hat{\mathcal{V}}(e)(\hat{s})]} & \boxed{\mathsf{cmd}(c) = \mathsf{call}(f_x, e)} \\ \hat{s} & \mathsf{cmd}(c) = \mathsf{return}_f \end{cases}$$

$$\hat{f}_c(\hat{s}) = \hat{s}'|_{\mathsf{access}(f)} \text{ where } \hat{s}' = \hat{s}[x \mapsto \hat{\mathcal{V}}(e)(\hat{s})]$$

$$\mathsf{access}(f) = \bigcup_{g \in \mathsf{callees}(f)} (\bigcup_{c \in \mathsf{control}(g)} \mathcal{A}(c)\boxed{(\hat{s}_{pre})})$$

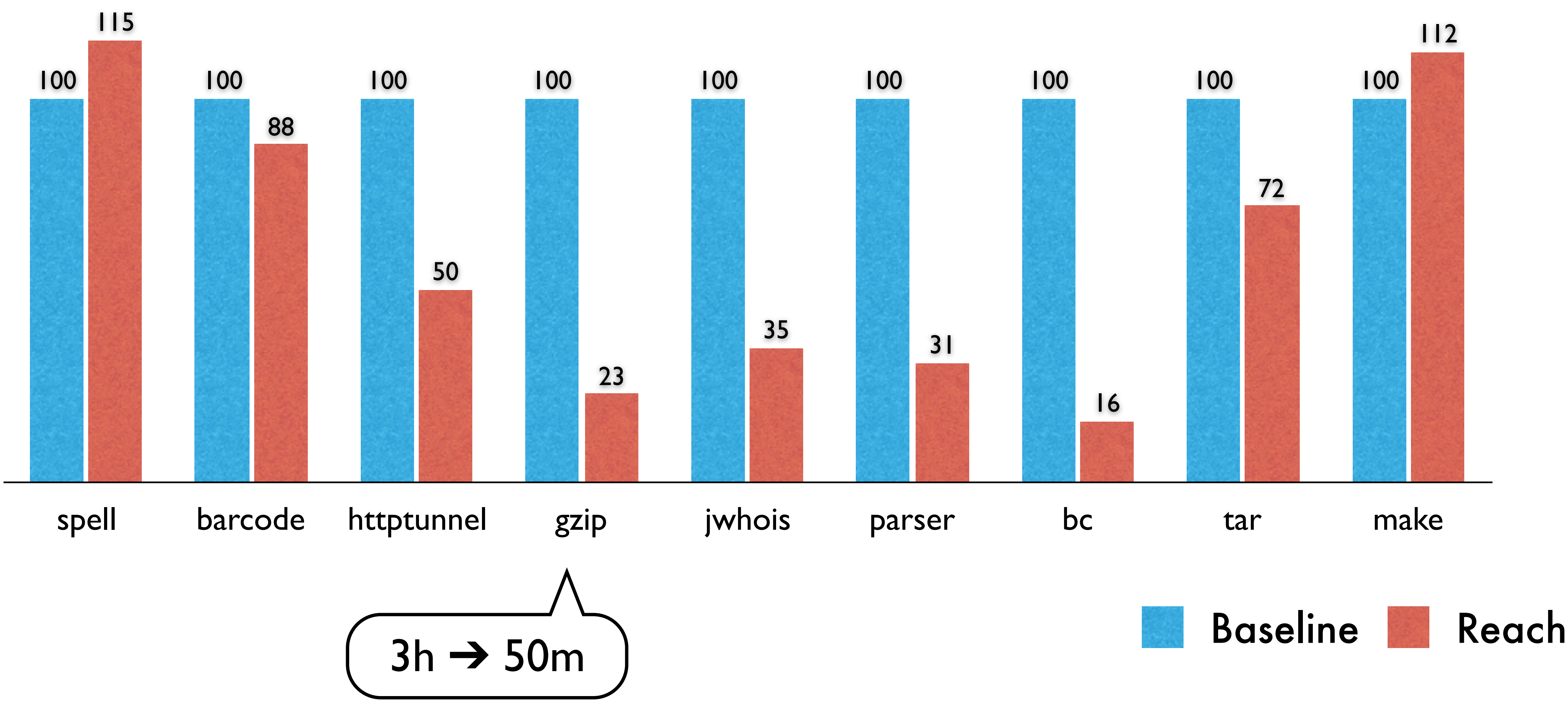
Reach vs. Access

78.5%-98.5% reduction
92.1% in average



Baseline vs. Reach

~6x speed-up



Pre-analysis Overhead

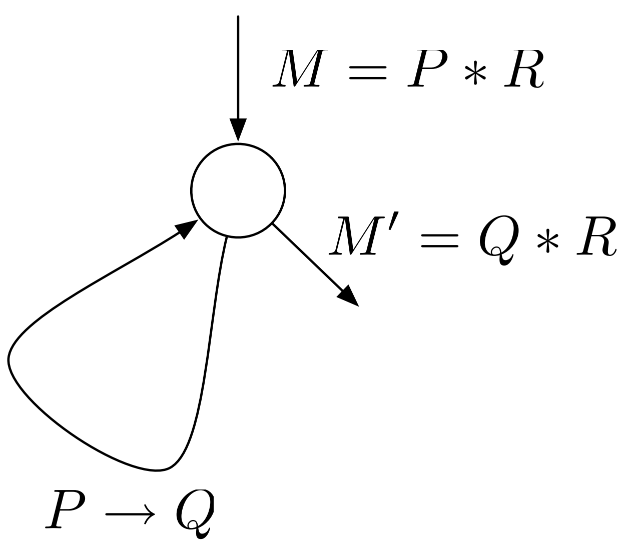
- Small overhead compared to the total analysis time
  - 0.1 ~ 8%

Program	LOC	Time		Overhead
		Total	Pre	
gzip	7,327	95s	1.3s	1.4%
bc	13,093	730s	4.1s	0.6%
bash	105,174	2011s	20.2s	1%

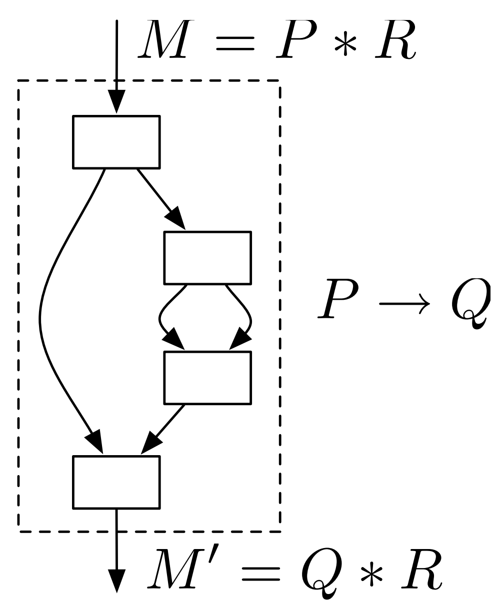
Block-level Localization

Access-based localization at any level

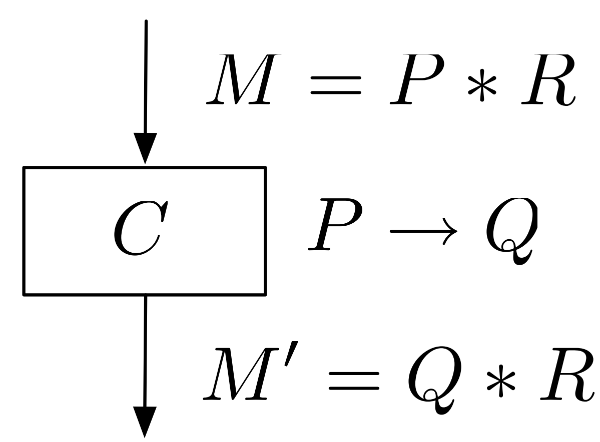
loops



branches

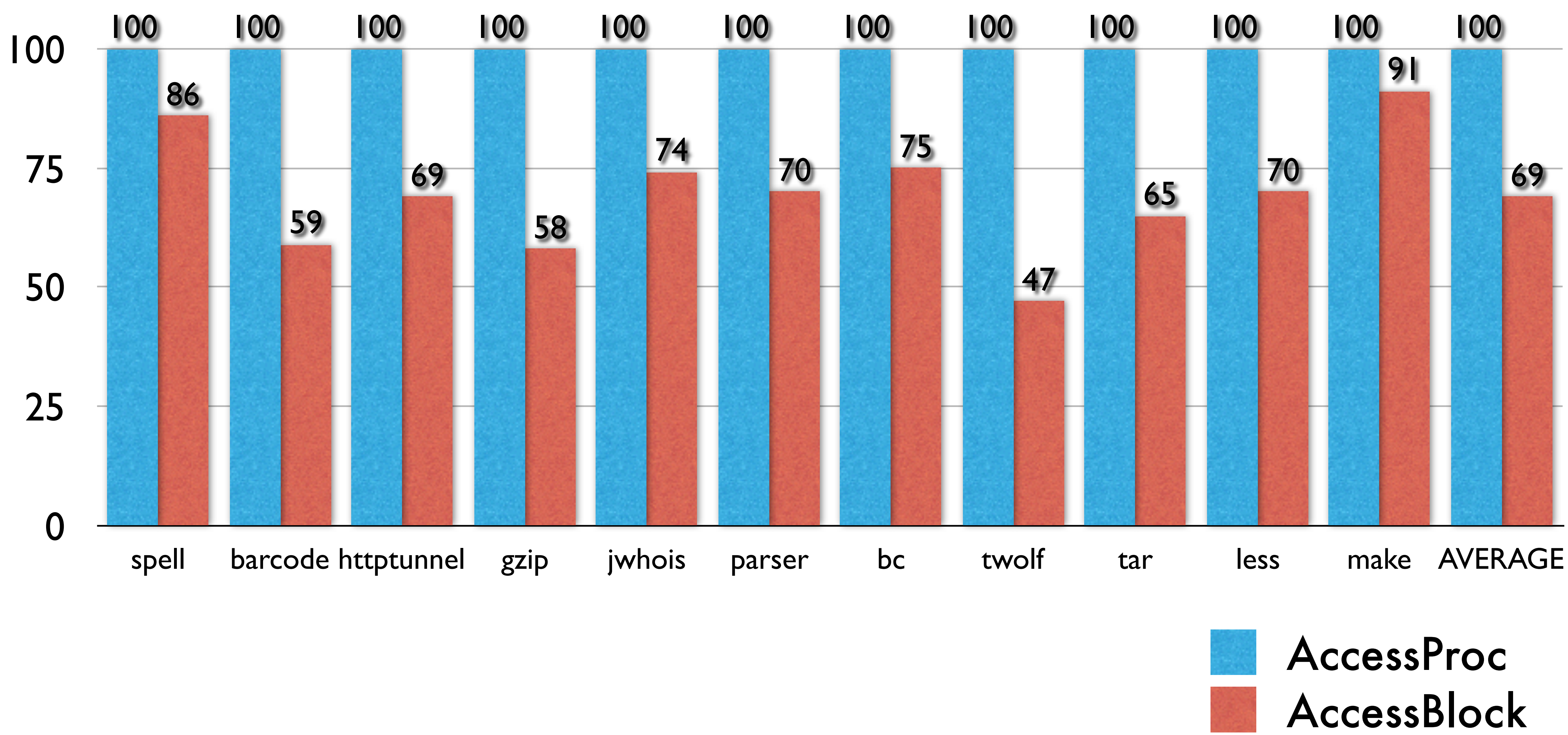


basic blocks



Block-level Localization

On average 31% reduction in time (k=6)



Precision

- No precision loss
- Sometimes, even improved

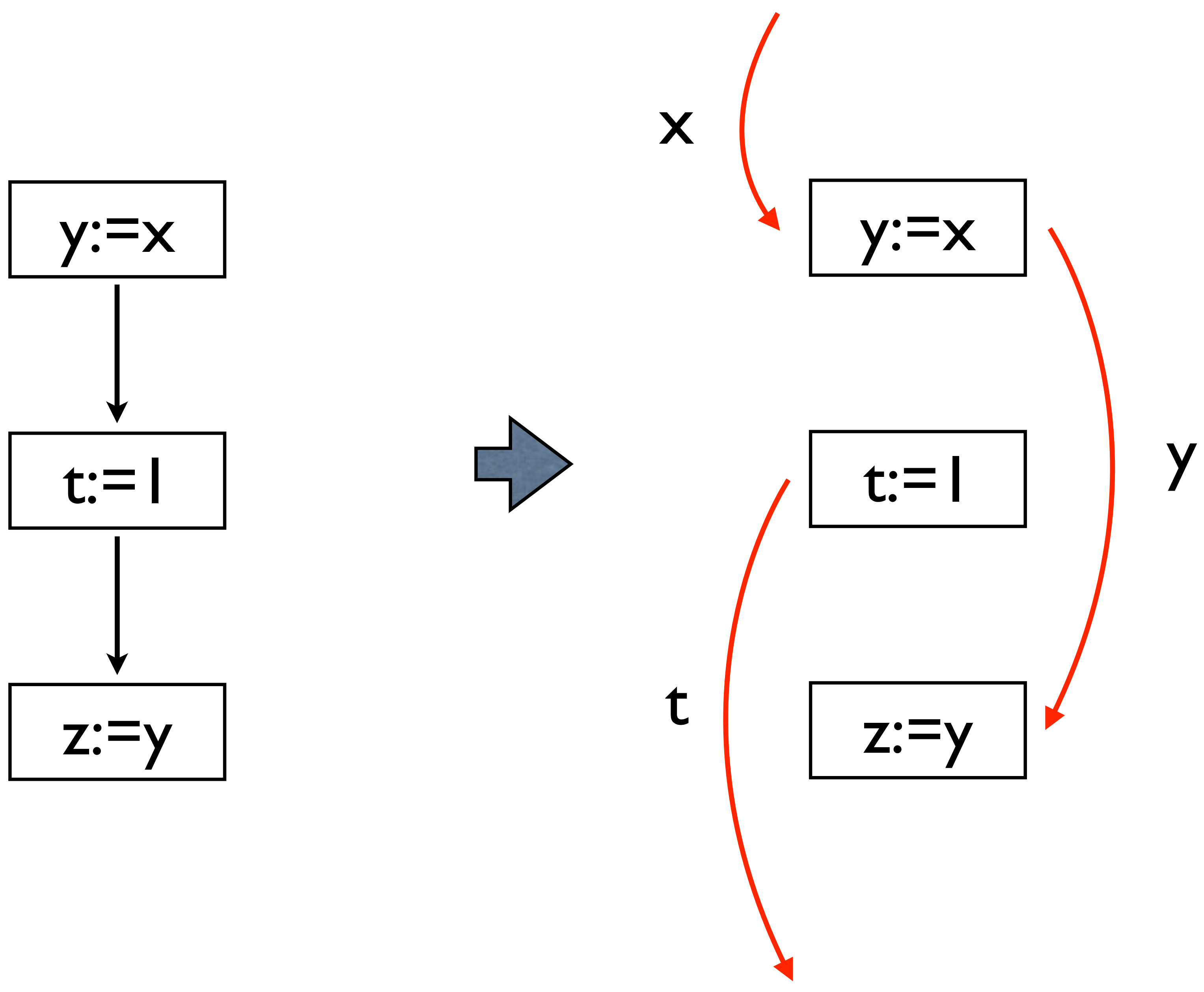
f does not access g

```
int g;

void f () {
  while (...) { ... }
}

void main () {
  g = 0; f ();
  g = 1; f ();
}
```

g : [0,0] ∇ [1,1] = [0,+oo]

g : [0,+oo] vs. [1,1]

Temporal Localization

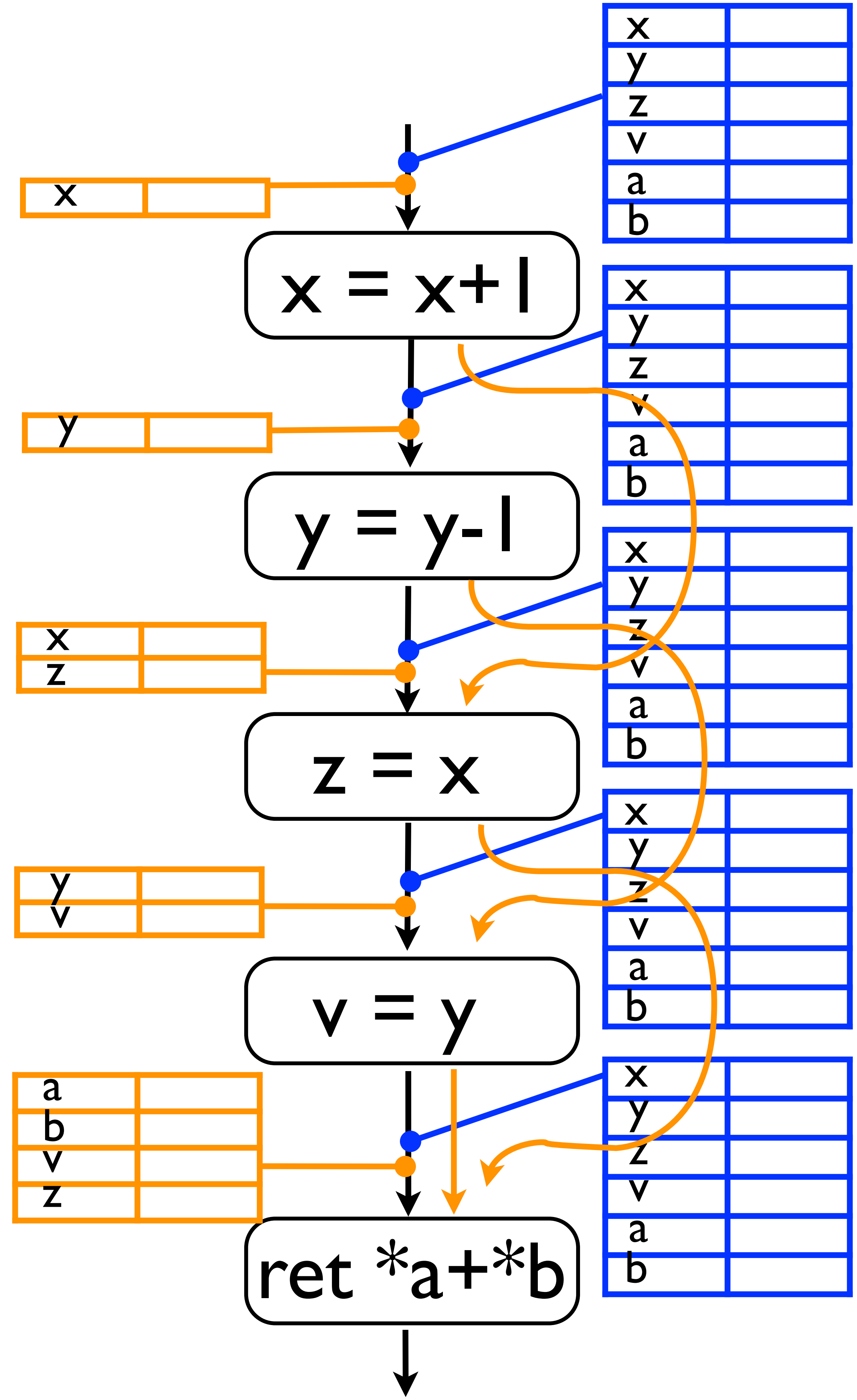


Sparse Analysis

"Right Part at Right Moment"

$$\hat{F}(\hat{X}) = \lambda c \in \mathbb{C}.\hat{f}_c(\bigsqcup_{\boxed{c' \hookrightarrow c}} \hat{X}(c')).$$

replace syntactic dependency
by semantic dependency
(data dependency)



Precision-Preserving Sparse Analysis Framework

Theorem. (preservation of soundness and precision)

$$\hat{F} : \hat{D} \to \hat{D} \quad \stackrel{\text{sparsify}}{\Longrightarrow} \quad \hat{F}_s : \hat{D} \to \hat{D}$$

$$\mathit{fix}\,\hat{F} \quad = \quad \mathit{fix}\,\hat{F}_s$$

"An important strength is that the ***theoretical result*** *is* ***very general*** *... The result should be* ***highly influential*** *on future work in sparse analysis."* *(from PLDI reviews)*

Towards Sparse Version

Analyzer computes the fixpoint of $\hat{F}_{\_} \in (\mathbb{C} \to \hat{\mathbb{S}}) \to (\mathbb{C} \to \hat{\mathbb{S}})$

- baseline non-sparse one

$$\hat{F}(\hat{X}) = \lambda c \in \mathbb{C}.\hat{f}_c(\bigsqcup_{\boxed{c' \hookrightarrow c}} \hat{X}(c')).$$

- unrealizable sparse version

$$\hat{F}_s(\hat{X}) = \lambda c \in \mathbb{C}.\hat{f}_c(\bigsqcup_{\boxed{c' \overset{l}{\rightsquigarrow} c}} \hat{X}(c')|_l).$$

- realizable sparse version

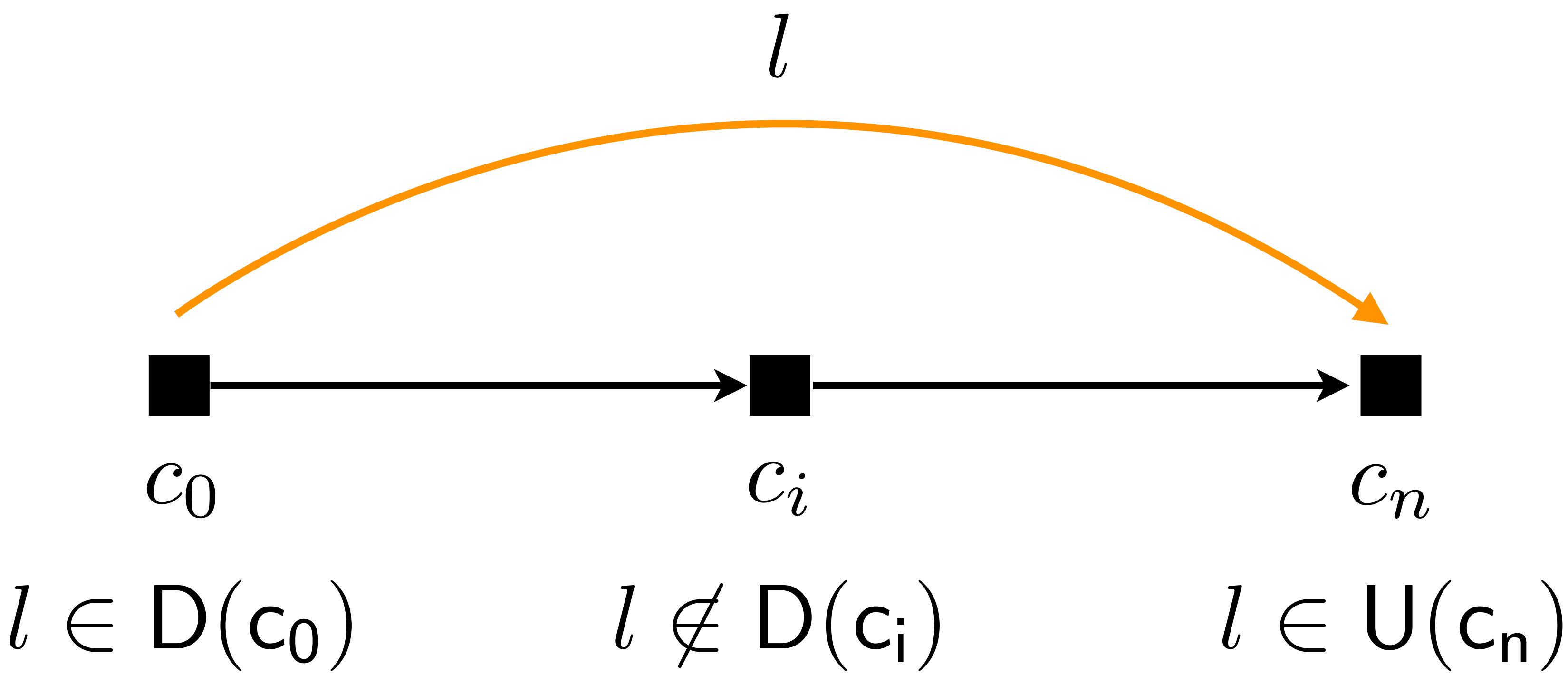
$$\hat{F}_a(\hat{X}) = \lambda c \in \mathbb{C}.\hat{f}_c(\bigsqcup_{\boxed{c' \overset{l}{\rightsquigarrow}_a c}} \hat{X}(c')|_l).$$

Unrealizable Sparse One

$$\hat{F}_s(\hat{X}) = \lambda c \in \mathbb{C}. \hat{f}_c\Big(\bigsqcup_{\boxed{c' \overset{l}{\leadsto} c}} \hat{X}(c')|_l\Big).$$

Data Dependency

$$\begin{aligned} c_0 \overset{l}{\leadsto} c_n \quad \triangleq \quad & \exists c_0 \ldots c_n \in \mathsf{Paths}, l \in \hat{\mathbb{L}}. \\ & l \in \mathsf{D}(c_0) \cap \mathsf{U}(c_n) \wedge \forall i \in (0, n). l \notin \mathsf{D}(c_i) \end{aligned}$$



Unrealizable Sparse One

$$\hat{F}_s(\hat{X}) = \lambda c \in \mathbb{C}.\hat{f}_c(\bigsqcup_{\boxed{c' \overset{l}{\rightsquigarrow} c}} \hat{X}(c')|_l).$$

Data Dependency

$$\begin{aligned} c_0 \overset{l}{\rightsquigarrow} c_n \quad &\triangleq \quad \exists c_0 \ldots c_n \in \mathsf{Paths}, l \in \hat{\mathbb{L}}. \\ & \quad\;\; l \in \mathsf{D}(c_0) \cap \mathsf{U}(c_n) \wedge \forall i \in (0, n).l \notin \mathsf{D}(c_i) \end{aligned}$$

Def-Use Sets

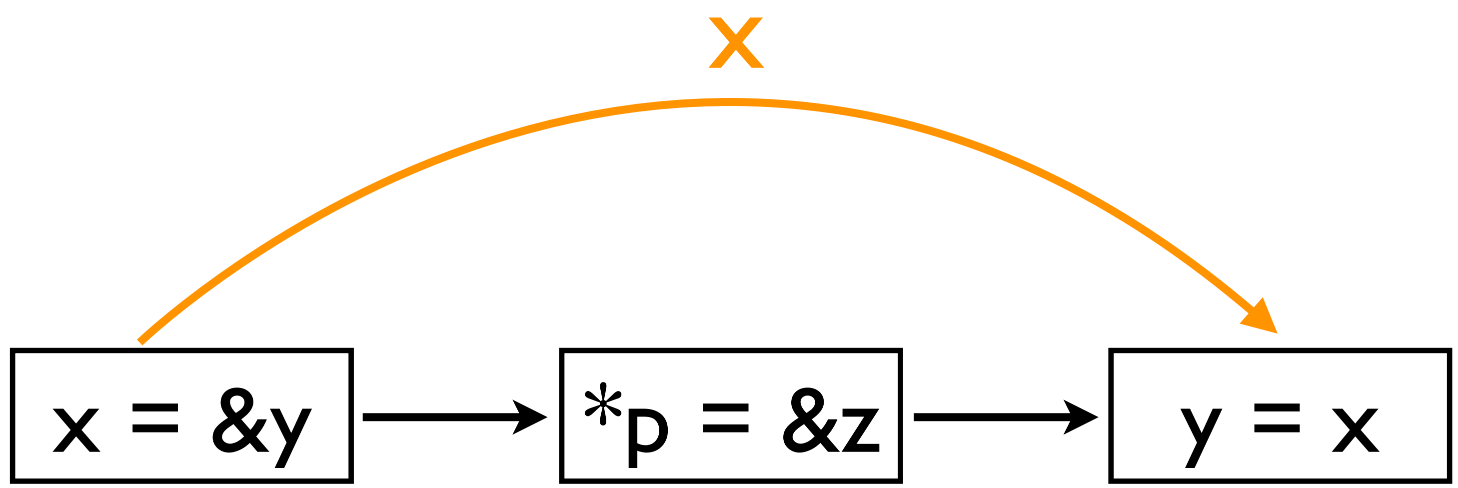
$$\mathsf{D}(c) \triangleq \{l \in \hat{\mathbb{L}} \mid \exists \hat{s} \sqsubseteq \bigsqcup_{c' \hookrightarrow c} (\mathit{fix}\hat{F})(c').\hat{f}_c(\hat{s})(l) \neq \hat{s}(l)\}.$$

$$\mathsf{U}(c) \triangleq \{l \in \hat{\mathbb{L}} \mid \exists \hat{s} \sqsubseteq \bigsqcup_{c' \hookrightarrow c} (\mathit{fix}\hat{F})(c').\hat{f}_c(\hat{s})|_{\mathsf{D}(c)} \neq \hat{f}_c(\hat{s}\backslash_l)|_{\mathsf{D}(c)}\}.$$

Preserving

$$\mathit{fix}\hat{F} = \mathit{fix}\hat{F}_s \quad \text{modulo } \mathsf{D}$$

Data Dependency Example



Def	{x}	{a, b}	{y}
Use	ϕ	{p, a, b}	{x}

Realizable Sparse One

$$\hat{F}_a(\hat{X}) = \lambda c \in \mathbb{C}.\hat{f}_c(\bigsqcup_{\boxed{c' \overset{l}{\leadsto}_a c}} \hat{X}(c')|_l).$$

Realizable Data Dependency

$$c_0 \overset{l}{\leadsto}_a c_n \quad \triangleq \quad \exists c_0 \ldots c_n \in \mathsf{Paths}, l \in \hat{\mathbb{L}}.$$
$$l \in \underline{\hat{\mathsf{D}}}(c_0) \cap \underline{\hat{\mathsf{U}}}(c_n) \wedge \forall i \in (0, n).l \notin \hat{\mathsf{D}}(c_i)$$

Preserving

$$\mathit{fix}\hat{F} \overset{\text{still}}{=} \mathit{fix}\hat{F}_a \quad \text{modulo } \hat{\mathsf{D}}$$

If the following two conditions hold

Conditions of $\hat{\mathsf{D}}$ & $\hat{\mathsf{U}}$

- over-approximation

$$\hat{\mathsf{D}}(c) \supseteq \mathsf{D}(c) \wedge \hat{\mathsf{U}}(c) \supseteq \mathsf{U}(c)$$

- spurious definitions should be also included in uses

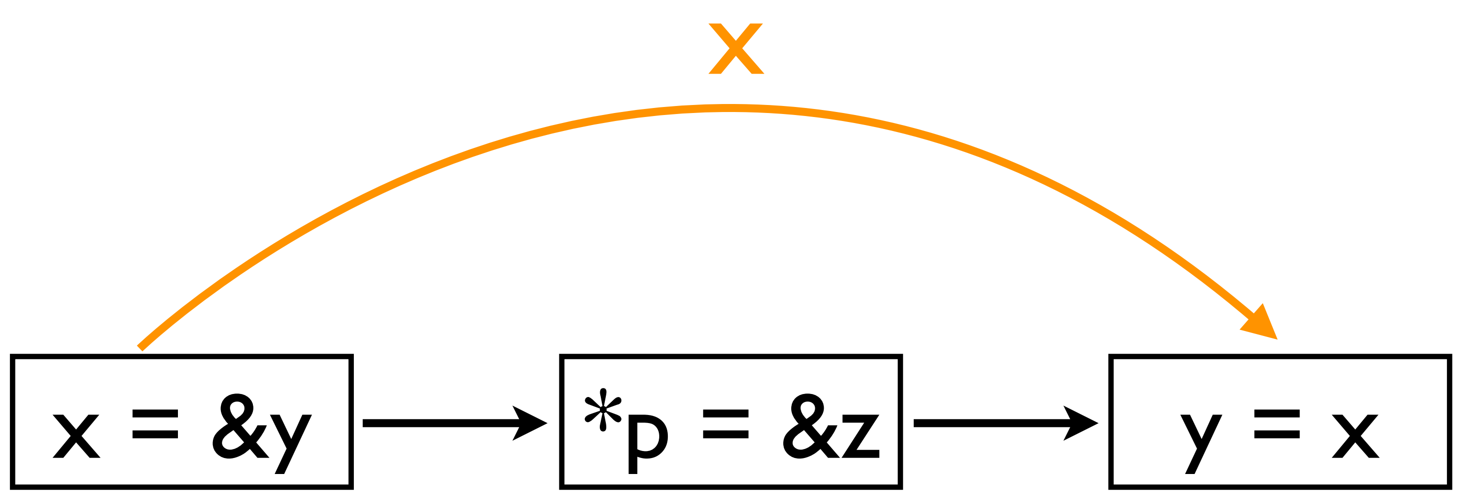
$$\underbrace{\hat{\mathsf{D}}(c) - \mathsf{D}(c)}_{\text{spurious definitions}} \subseteq \hat{\mathsf{U}}(c)$$

Why the Conditions of $\hat{D}$ & $\hat{U}$

$$\begin{tikzcd} \boxed{x = \&y} \arrow[r] \arrow[rr, bend left, "x"] & \boxed{*p = \&z} \arrow[r] & \boxed{y = x} \end{tikzcd}$$

Def	{x}	{a, b}	{y}
Use	ϕ	{p, a, b}	{x}

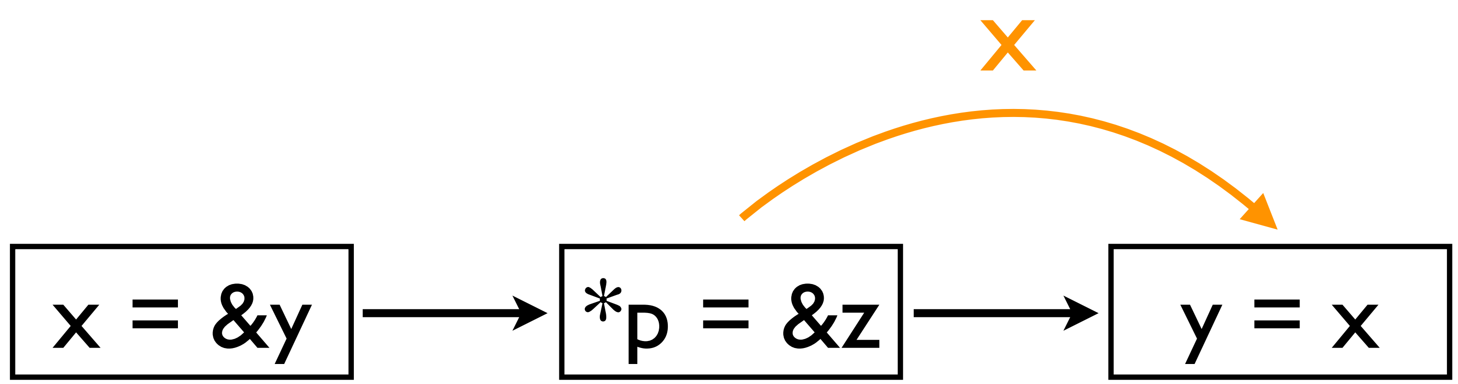
Why the Conditions of $\hat{\mathrm{D}}$ & $\hat{\mathrm{U}}$



Approx. Def	{x}	{a, b, x}	{y}
Approx. Use	ϕ	{p, a, b}	{x}

$$\underbrace{\hat{\mathrm{D}}(c) - \mathrm{D}(c)}_{\{x\}} \not\subseteq \hat{\mathrm{U}}(c)$$

Why the Conditions of $\hat{D}$ & $\hat{U}$



Approx. Def	{x}	{a, b, x}	{y}
Approx. Use	ϕ	{p, a, b}	{x}

$$\underbrace{\hat{D}(c) - D(c)}_{\{x\}} \not\subseteq \hat{U}(c)$$

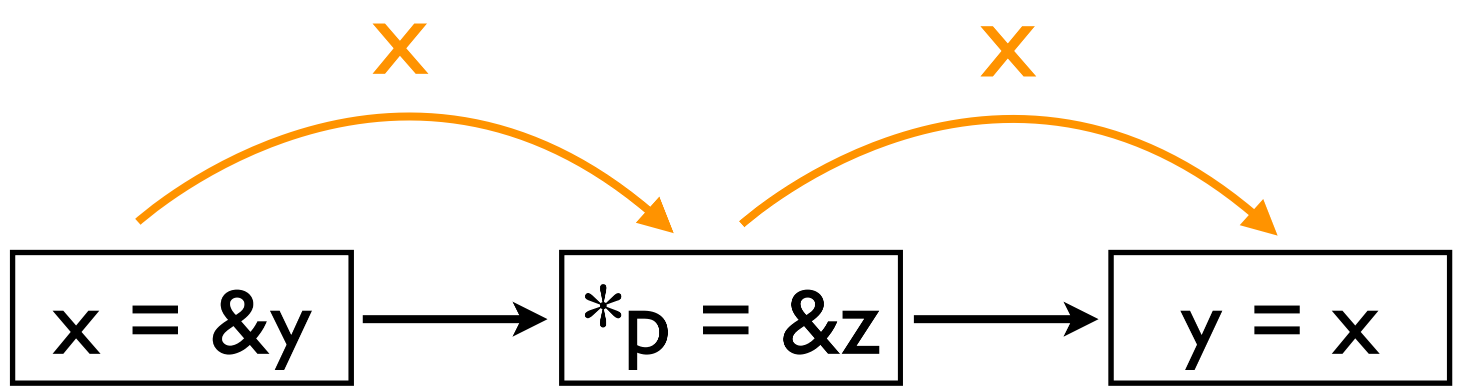
Why the Conditions of $\hat{D}$ & $\hat{U}$

$$\begin{tikzcd} \boxed{x = \&y} \arrow[r] & \boxed{*p = \&z} \arrow[r] \arrow[r, bend left, "x"] & \boxed{y = x} \end{tikzcd}$$

Approx. Def	$\{x\}$	$\{a, b, x\}$	$\{y\}$
Approx. Use	ϕ	$\{p, a, b, x\}$	$\{x\}$

$$\underbrace{\hat{D}(c) - D(c)}_{\{x\}} \subseteq \hat{U}(c)$$

Why the Conditions of $\hat{\mathsf{D}}$ & $\hat{\mathsf{U}}$



Approx. Def	{x}	{a, b, x}	{y}
Approx. Use	ϕ	{p, a, b, x}	{x}

$$\underbrace{\hat{\mathsf{D}}(c) - \mathsf{D}(c)}_{\{x\}} \subseteq \hat{\mathsf{U}}(c)$$

Hurdle: $\hat{\mathsf{D}}$ & $\hat{\mathsf{U}}$ Before Analysis?

- Yes, by yet another analysis with further abstraction
  - e.g., flow-insensitive abstraction

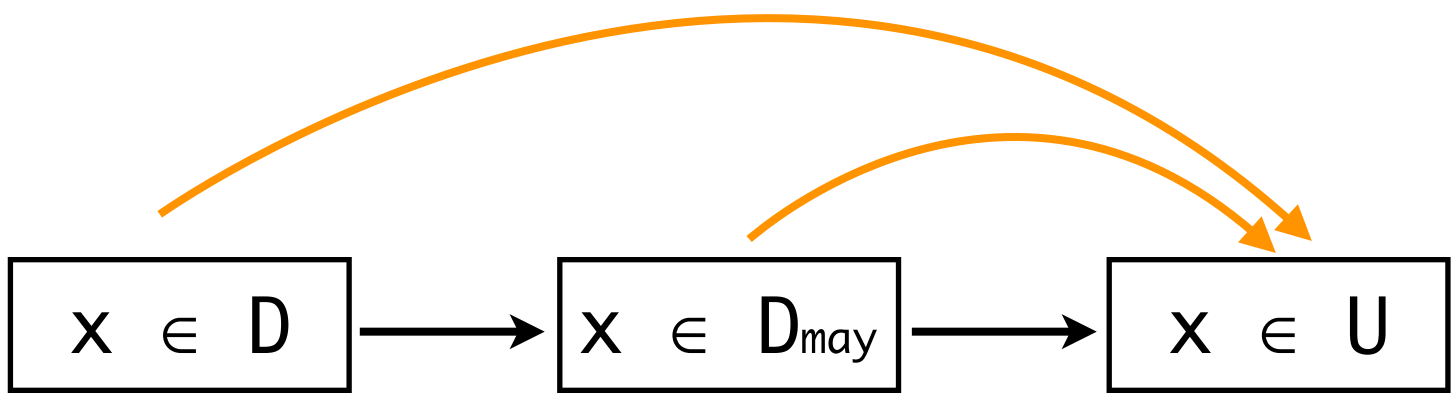
$$\mathbb{C} \to \hat{\mathbb{S}} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{\mathbb{S}} \qquad \hat{F}_p = \lambda \hat{s}.(\bigsqcup_{c \in \mathbb{C}} \hat{f}_c(\hat{s}))$$

- In implementation, $\hat{\mathsf{U}}$ includes $\hat{\mathsf{D}}$

$$\hat{\mathsf{D}}(c) - \mathsf{D}(c) \subseteq \hat{\mathsf{U}}(c)$$

Existing Sparse Techniques
(developed mostly in dfa community)

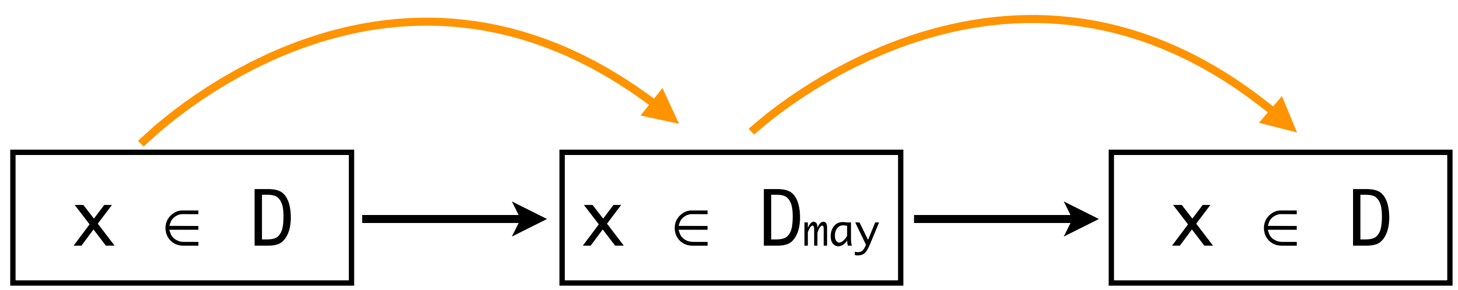
- Different notion of data dependency



def-use chains fail to preserve original precision

Existing Sparse Techniques
(developed mostly in dfa community)

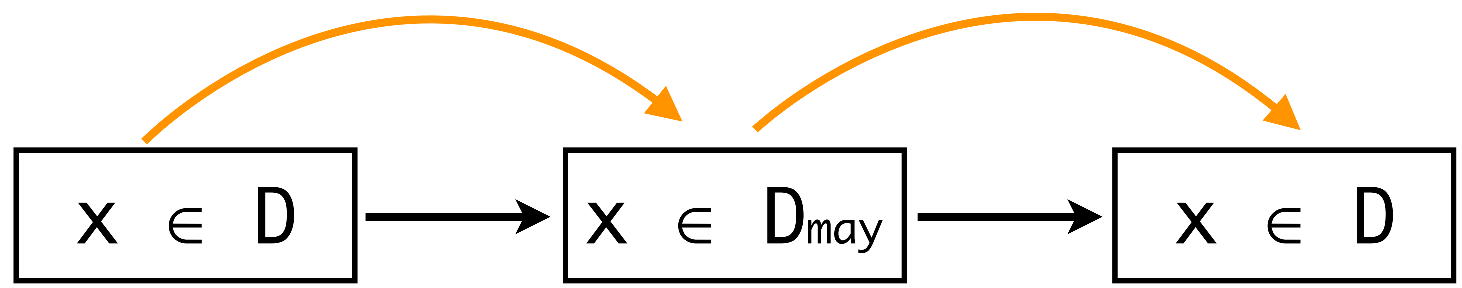
- Different notion of data dependency



our data dependency preserves original precision

Existing Sparse Techniques
(developed mostly in dfa community)

- Different notion of data dependency



- Existing sparse analyses are not general
  - tightly coupled with particular analysis, or
  - limited to a particular target language

Design and Implementation of Sparse Global Analyses for C-like Languages

Hakjoo Oh   Kihong Heo   Wonchan Lee   Woosuk Lee   Kwangkeun Yi

Seoul National University

{pronto,khheo,wclee,wslee,kwang}@ropas.snu.ac.kr

Abstract

In this article we present a general method for achieving global static analyzers that are precise, sound, yet also scalable. Our method generalizes the sparse analysis techniques on top of the abstract interpretation framework to support relational as well as non-relational semantics properties for C-like languages. We first use the abstract interpretation framework to have a global static analyzer whose scalability is unattended. Upon this underlying sound static analyzer, we add our generalized sparse analysis techniques to improve its scalability while preserving the precision of the underlying analysis. Our framework determines what to prove to guarantee that the resulting sparse version should preserve the precision of the underlying analyzer.

We formally present our framework; we present that existing sparse analyses are all restricted instances of our framework; we show more semantically elaborate design examples of sparse non-relational and relational static analyses; we present their implementation results that scale to analyze up to one million lines of C programs. We also show a set of implementation techniques that turn out to be critical to economically support the sparse analysis process.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis

interpretation framework. Since the abstract interpretation framework [9, 11] guides us to design sound yet arbitrarily precise static analyzers for any target language, we first use the framework to have a global static analyzer whose scalability is unattended. Upon this underlying sound static analyzer, we add our generalized sparse analysis techniques to improve its scalability while preserving the precision of the underlying analysis. Our framework determines what to prove to guarantee that the resulting sparse version should preserve the precision of the underlying analyzer.

Our framework bridges the gap between the two existing technologies – abstract interpretation and sparse analysis – towards the design of sound, yet scalable global static analyzers. Note that while abstract interpretation framework provides a theoretical knob to control the analysis precision without violating its correctness, the framework does not provide a knob to control the resulting analyzer's scalability preserving its precision. On the other hand, existing sparse analysis techniques [6, 14, 15, 19, 20, 24, 40, 42, 44] achieve scalability, but they are mostly algorithmic and tightly coupled with particular analyses.[1] The sparse techniques are not general enough to be used for an arbitrarily complicated semantic analysis.

Contributions Our contributions are as follows.

General Sparse Analysis Framework

A

Global Sparse Analysis Framework

Hakjoo Oh, Seoul National University
Kihong Heo, Seoul National University
Wonchan Lee, Seoul National University
Woosuk Lee, Seoul National University
Daejun Park, Seoul National University
Jeehoon Kang, Seoul National University
Kwangkeun Yi, Seoul National University

In this article we present a general method for achieving global static analyzers that are precise, sound, yet also scalable. Our method, on top of the abstract interpretation framework, is a general sparse analysis technique that supports relational as well as non-relational semantics properties for various programming languages. Analysis designers first use the abstract interpretation framework to have a global and correct static analyzer whose scalability is unattended. Upon this underlying sound static analyzer, analysis designers add our generalized sparse analysis techniques to improve its scalability while preserving the precision of the underlying analysis. Our method prescribes what to prove to guarantee that the resulting sparse version should preserve the precision of the underlying analyzer.

We formally present our framework; we show that existing sparse analyses are all restricted instances of our framework; we show more semantically elaborate design examples of sparse non-relational and relational static analyses; we present their implementation results that scale to globally analyze up to one million lines of C programs. We also show a set of implementation techniques that turn out to be critical to economically support the sparse analysis process.

Categories and Subject Descriptors: F.3.2 [**Semantics of Programming Languages**]: Program Analysis

General Terms: Programming Languages, Program Analysis

Additional Key Words and Phrases: Static analysis, abstract interpretation, sparse analysis

ACM Reference Format:
ACM Trans. Program. Lang. Syst. V, N, Article A (January YYYY), 44 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Precise, sound, scalable yet global static analyzers have been unachievable in general. Other than almost syntactic properties, once the target property becomes slightly deep in semantics it's been a daunting challenge to achieve the four goals in a single static analyzer. This situation explains why, for example, in the static error detection tools for full C, there exists a clear dichotomy: either "bug-finders" that risk being unsound yet scalable or "verifiers" that risk being unscalable yet sound. No such tools are scalable to globally analyze million lines of C code while being sound and precise enough for practical use.

In this article we present a general method for achieving global static analyzers that are precise, sound, yet also scalable. Our approach generalizes the sparse analysis ideas on top of the abstract interpretation framework. Since the abstract interpreta-

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

- General for
  - programming languages, e.g., imperative, functional, oop, etc
  - trace partitioning, context-sensitivity, path-sensitivity, loop-unrolling, etc

Trace Semantics

3.3. Collecting Semantics

The collecting semantics $\llbracket P \rrbracket \in \mathcal{P}(\mathbb{S}^+)$ of program P is the set of all finite traces of P:

$$\llbracket P \rrbracket = \{\sigma \in \mathbb{S}^+ \mid \sigma_0 \in \mathbb{S}_\iota \wedge \forall k. \sigma_k \to \sigma_{k+1}\}$$

Note that the semantics $\llbracket P \rrbracket$ is the least fixpoint of the semantic function $F \in \mathcal{P}(\mathbb{S}^+) \to \mathcal{P}(\mathbb{S}^+)$, i.e., $\llbracket P \rrbracket = \mathbf{lfp} F$, defined as follows:

$$F(\Sigma) = \mathbb{S}_\iota \cup \{\sigma \cdot s \mid \sigma \in \Sigma \wedge \sigma_\dashv \to s\}.$$

Abstract Semantics

3.4. Baseline Abstraction

We abstract the collecting semantics of program P by the following Galois connections:

$$\mathcal{P}(\mathbb{S}^+) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \Delta \to \mathcal{P}(\mathbb{S}^+) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \Delta \to \hat{\mathbb{S}}$$

The abstraction consists of two steps:

(1) *Partitioning abstraction* (α_1, γ_1): we abstract the set of traces $(\mathcal{P}(\mathbb{S}^+))$ into partitioned sets of traces ($\Delta \to \mathcal{P}(\mathbb{S}^+)$, where Δ is the set of partitioning indices).

(2) *State abstraction* (α_2, γ_2): for each partition, the associated set of traces is abstracted into an abstract state ($\hat{\mathbb{S}}$) that over-approximates the reachable states of the traces.

$$\hat{F}(\hat{\phi}) = \lambda i \in \Delta.\, \hat{f}_i\Big(\bigsqcup_{i' \hookrightarrow_{\hat{\phi}} i} \hat{\phi}(i')\Big)$$

Towards Sparse Analysis

Definition 3.10 (*Definition Set*). Definition set $\mathsf{D}(i)$ at partitioning index i is a set of abstract locations whose abstract values are ever changed by $\hat{f}_i$ during the analysis, i.e., (let $\mathcal{S} = \mathbf{lfp}\hat{F}$)

$$\mathsf{D}(i) = \{l \in \hat{\mathbb{L}} \mid \exists \hat{s} \sqsubseteq \bigsqcup_{i' \hookrightarrow_{\mathcal{S}} i} \mathcal{S}(i').\, \hat{f}_i(\hat{s})(l) \neq \hat{s}(l)\}.$$

Definition 3.12 (*Use Set*). Use set $\mathsf{U}(i)$ at partitioning index i consists of two parts:

$$\mathsf{U}(i) = \mathsf{U}_d(i) \cup \mathsf{U}_c(i).$$

The first part ($\mathsf{U}_d(i)$) is the set of abstract locations without which some values in $\mathsf{D}(i)$ are not properly generated, i.e., (let $\mathcal{S} = \mathbf{lfp}\hat{F}$)

$$\mathsf{U}_d(i) = \{l \in \hat{\mathbb{L}} \mid \exists \hat{s} \sqsubseteq \bigsqcup_{i' \hookrightarrow_{\mathcal{S}} i} \mathcal{S}(i').\, \hat{f}_i(\hat{s})|_{\mathsf{D}(i)} \neq \hat{f}_i(\hat{s}\backslash_l)|_{\mathsf{D}(i)}\}.$$

In addition, we collect abstract locations that are necessary to generate transition flows ($\hookrightarrow$): $\mathsf{U}_c(i)$ representing the set of abstract locations without which some flows in $\hookrightarrow_{\mathcal{S}}$ are not properly generated, i.e.,

$$\mathsf{U}_c(i) = \{l \in \hat{\mathbb{L}} \mid \exists i' \in \Delta.\, (i, i') \in (\hookrightarrow_{\mathcal{S}}) \wedge (i, i') \notin (\hookrightarrow_{\mathcal{S}[i \mapsto \mathcal{S}(i)\backslash_l]})\}.$$

□

Towards Sparse Analysis

Definition 3.17 (*Data dependency*). Data dependency is quadruple relation $(\rightsquigarrow) \subseteq \Delta \times \hat{\mathbb{L}} \times \Delta \times (\Delta \to \hat{\mathbb{S}})$ defined as follows:

$$\begin{array}{lll} i_0 \overset{l}{\rightsquigarrow}_{\hat{\phi}} i_n & \textit{iff} & \exists i_0 \ldots i_n \in \mathsf{Paths}(\hat{\phi}), l \in \hat{\mathbb{L}}. \\ & & l \in \mathsf{D}(i_0) \cap \mathsf{U}(i_n) \wedge \forall k \in (0, n). l \notin \mathsf{D}(i_k) \end{array} \tag{3}$$

where $\mathsf{Paths}(\hat{\phi})$ is the set of all paths created by transition relation $\hookrightarrow_{\hat{\phi}}$: a path $p = p_0 p_1 \cdots p_n$ is a sequence of partitioning indices such that $p_0 \hookrightarrow_{\hat{\phi}} p_1 \hookrightarrow_{\hat{\phi}} \cdots \hookrightarrow_{\hat{\phi}} p_n$, then,

$$\mathsf{Paths}(\hat{\phi}) = \mathbf{lfp} \lambda P. \{i_0 i_1 \mid i_0 \hookrightarrow_{\hat{\phi}} i_1\} \cup \{p_0 p_1 \cdots p_n i \mid p \in P \ \wedge \ p_n \hookrightarrow_{\hat{\phi}} i\}.$$

□

$$\hat{F}_s(\hat{\phi}) = \lambda i \in \Delta. \hat{f}_i\Big(\bigsqcup_{i' \overset{l}{\rightsquigarrow}_{\hat{\phi}} i} \hat{\phi}(i')|_l \Big).$$

THEOREM 3.19 (CORRECTNESS).

$$\forall i \in \Delta. \forall l \in \mathsf{D}(i). (\mathbf{lfp} \hat{F}_s)(i)(l) = (\mathbf{lfp} \hat{F})(i)(l).$$

PROOF. Shortly, we will notice that this theorem is a corollary of Theorem 3.23, where $\hat{F}_s$ is an instance of $\hat{F}_a$ such that $\hat{\mathsf{D}}(i) = \mathsf{D}(i)$ and $\hat{\mathsf{U}}(i) = \mathsf{U}(i)$. □

Towards Sparse Analysis

Definition 3.21 (*Safe Approximations of* D *and* U). We say that $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ are safe approximations of D and U, respectively, if and only if

(1) $\hat{\mathsf{D}}(i) \supseteq \mathsf{D}(i) \;\wedge\; \hat{\mathsf{U}}(i) \supseteq \mathsf{U}(i)$
(2) $\hat{\mathsf{U}}(i) \supseteq \mathbb{U}_{(\hat{\mathsf{D}}(i) \setminus \mathsf{D}(i))}(i)$

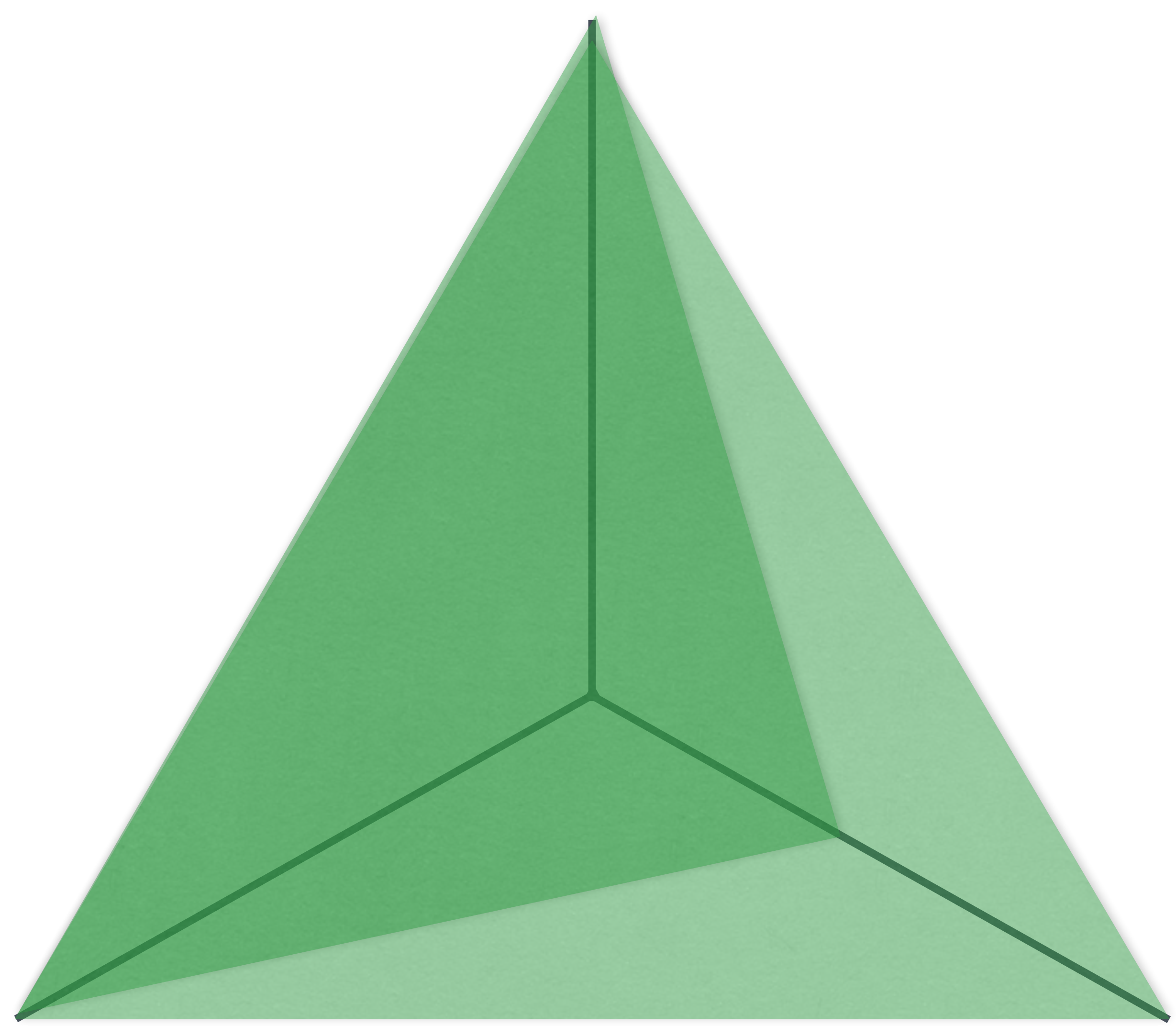
$\square$

THEOREM 3.23 (CORRECTNESS). *Suppose sparse abstract semantic function* $\hat{F}_a$ *is derived by safe approximations* $\hat{\mathsf{D}}$ *and* $\hat{\mathsf{U}}$. *Then,*

$$\forall i \in \Delta.\forall l \in \hat{\mathsf{D}}(i).(\mathbf{lfp}\hat{F}_a)(i)(l) = (\mathbf{lfp}\hat{F})(i)(l).$$

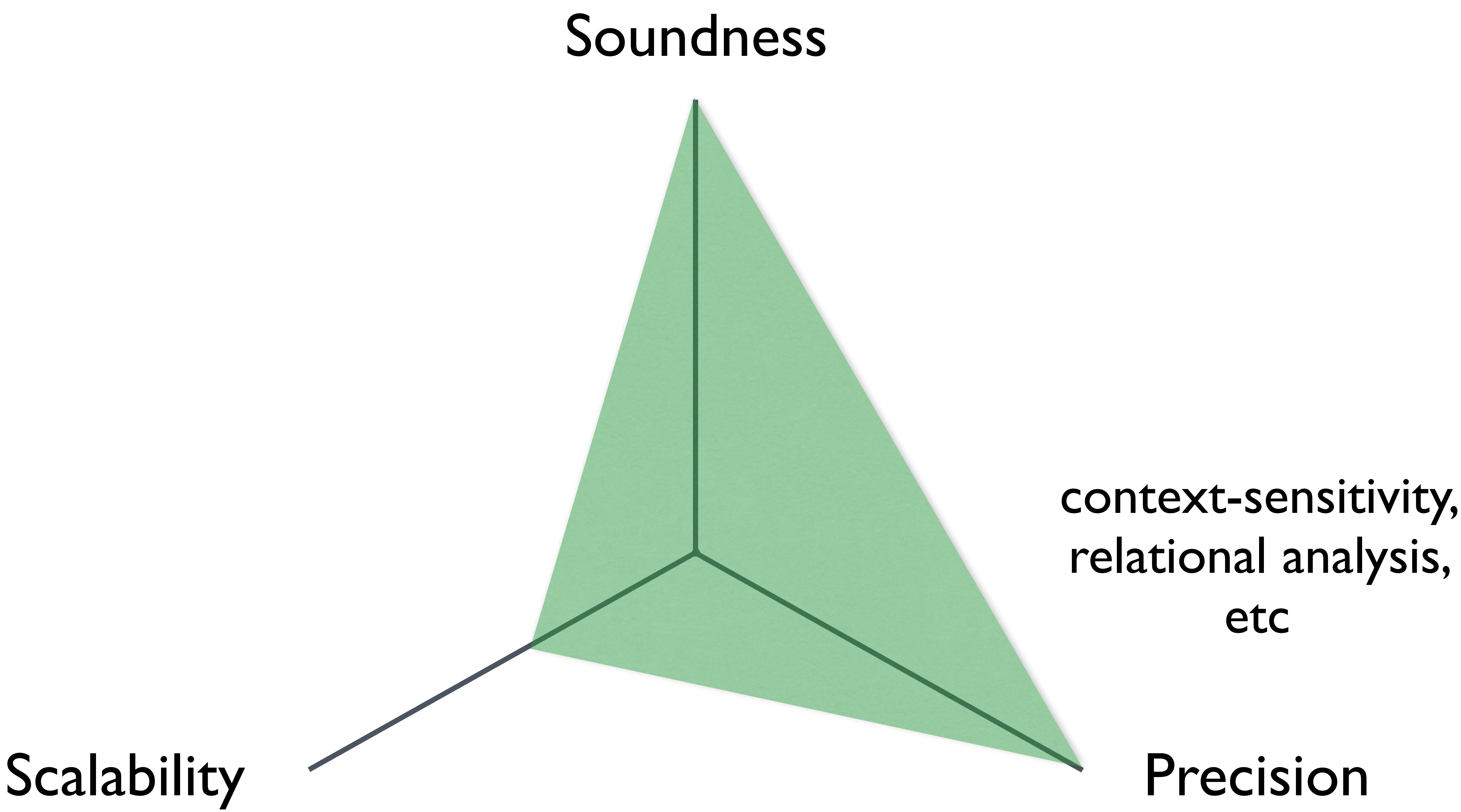
PROOF. See Appendix A. $\square$

The Second Goal: Precision

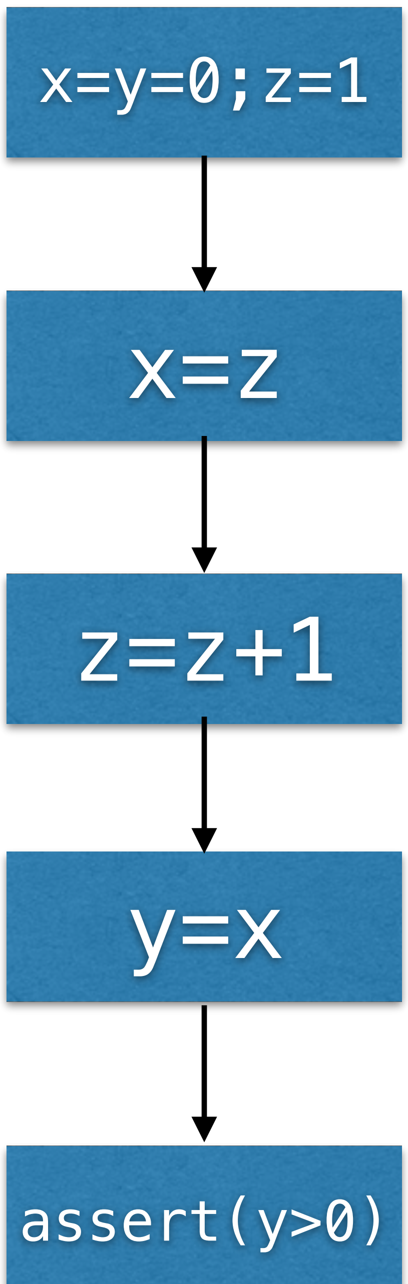


Challenge: Can we achieve it without scalability loss?

Naive Approaches



Flow-Sensitivity



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

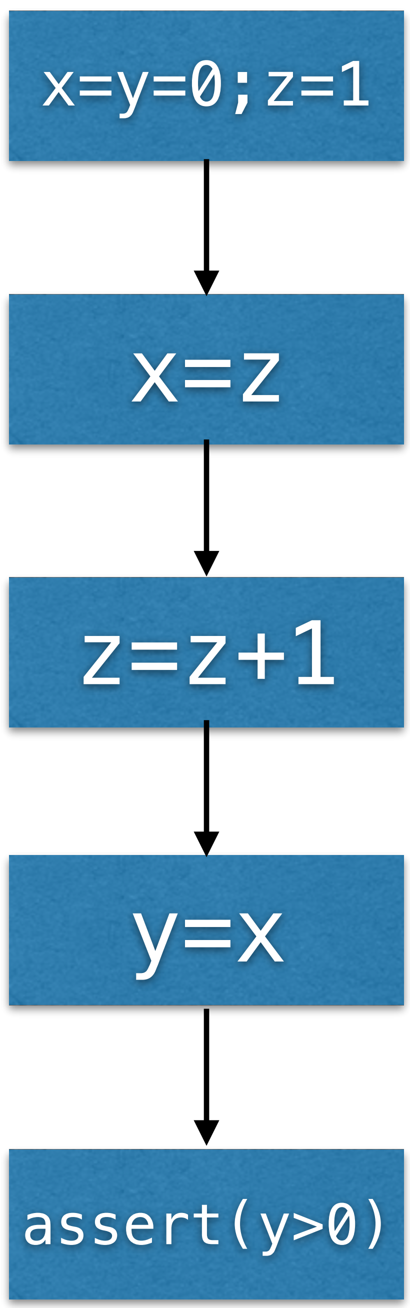
x	[1,1]
y	[0,0]
z	[1,1]

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

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

precise but costly

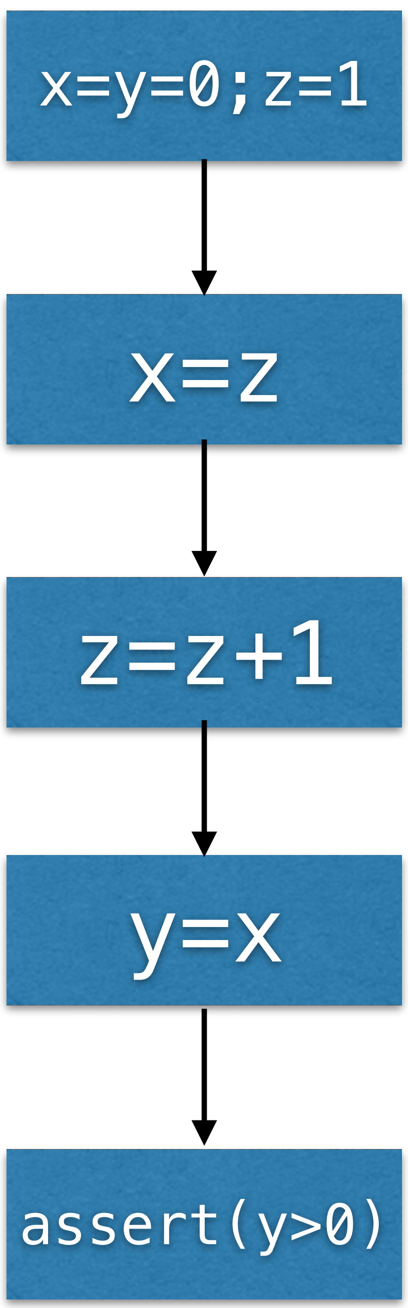
Flow-Insensitivity



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]

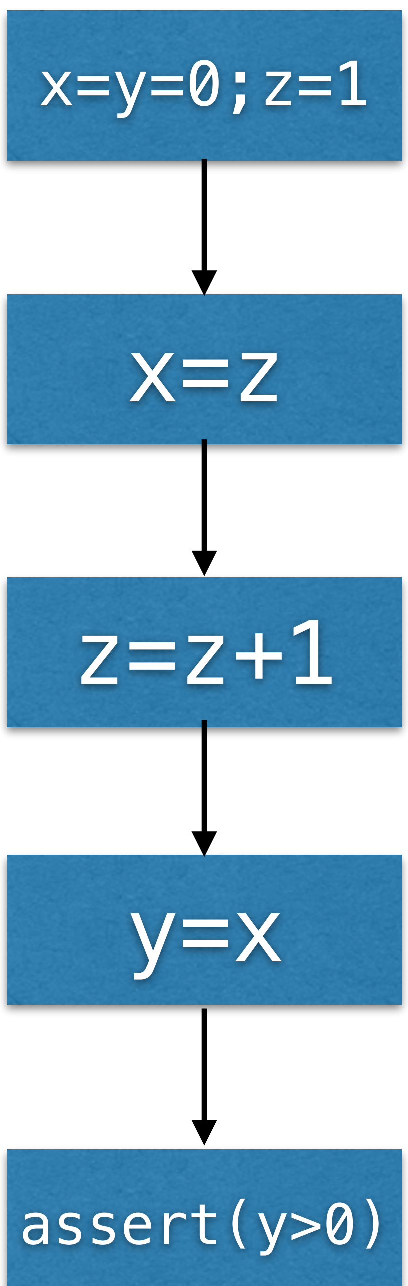
x	[1,+∞]
y	[0,0]

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

fail to prove

FI : {x}

x	$[0,+\infty]$

Hard Search Problem

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

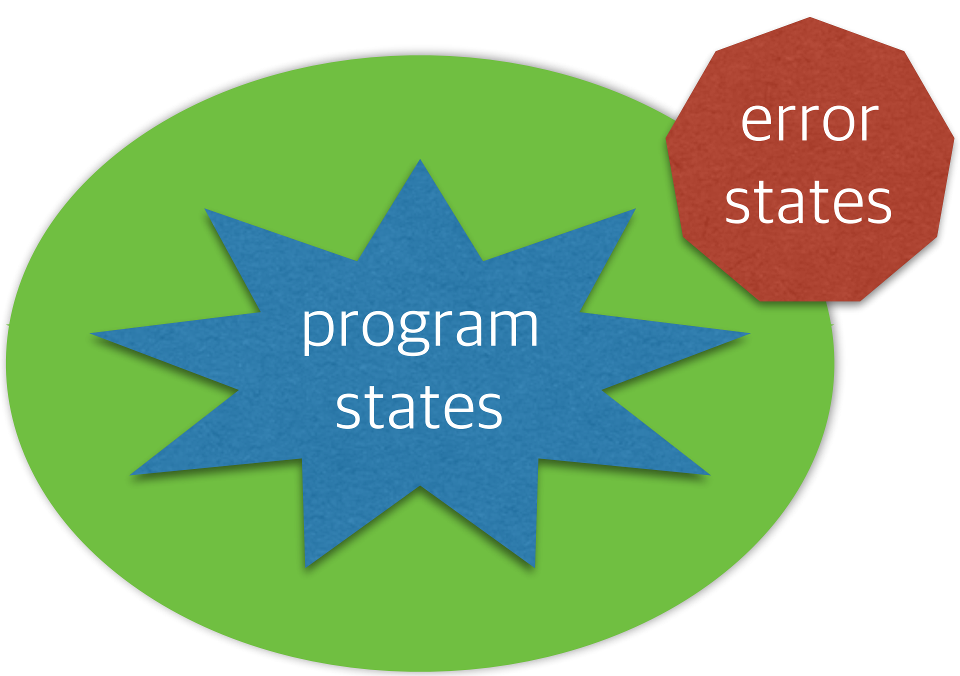
Our Approaches

- Two approaches:
  - Finding a good fixed heuristic [PLDI'14, TOPLAS'16]
  - Finding a heuristic automatically [OOPSLA'15, SAS'16, APLAS,16, …]

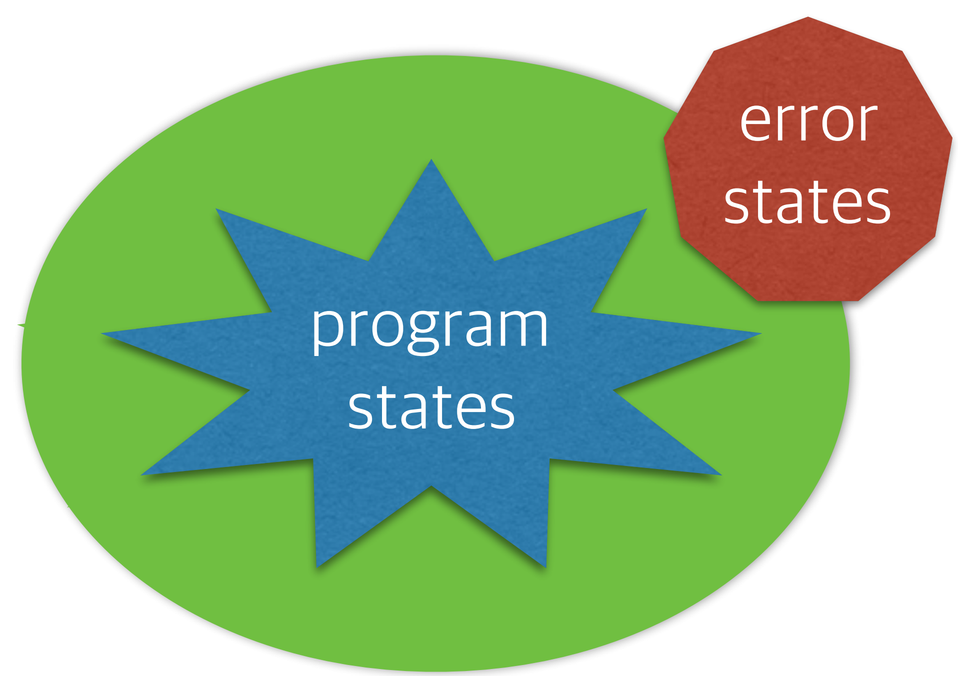
PLDI'14

Selective Context-Sensitivity Guided by Impact Pre-Analysis

- Apply context-sensitivity only when/where it matters
- General for context-sensitivity, relational analysis, etc



vs.



our method: **24%** / **28%**

3-CFA: **24%** / **1300%**

Example Program

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

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

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

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

always holds

does not always hold

Context-Insensitivity

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

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

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

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

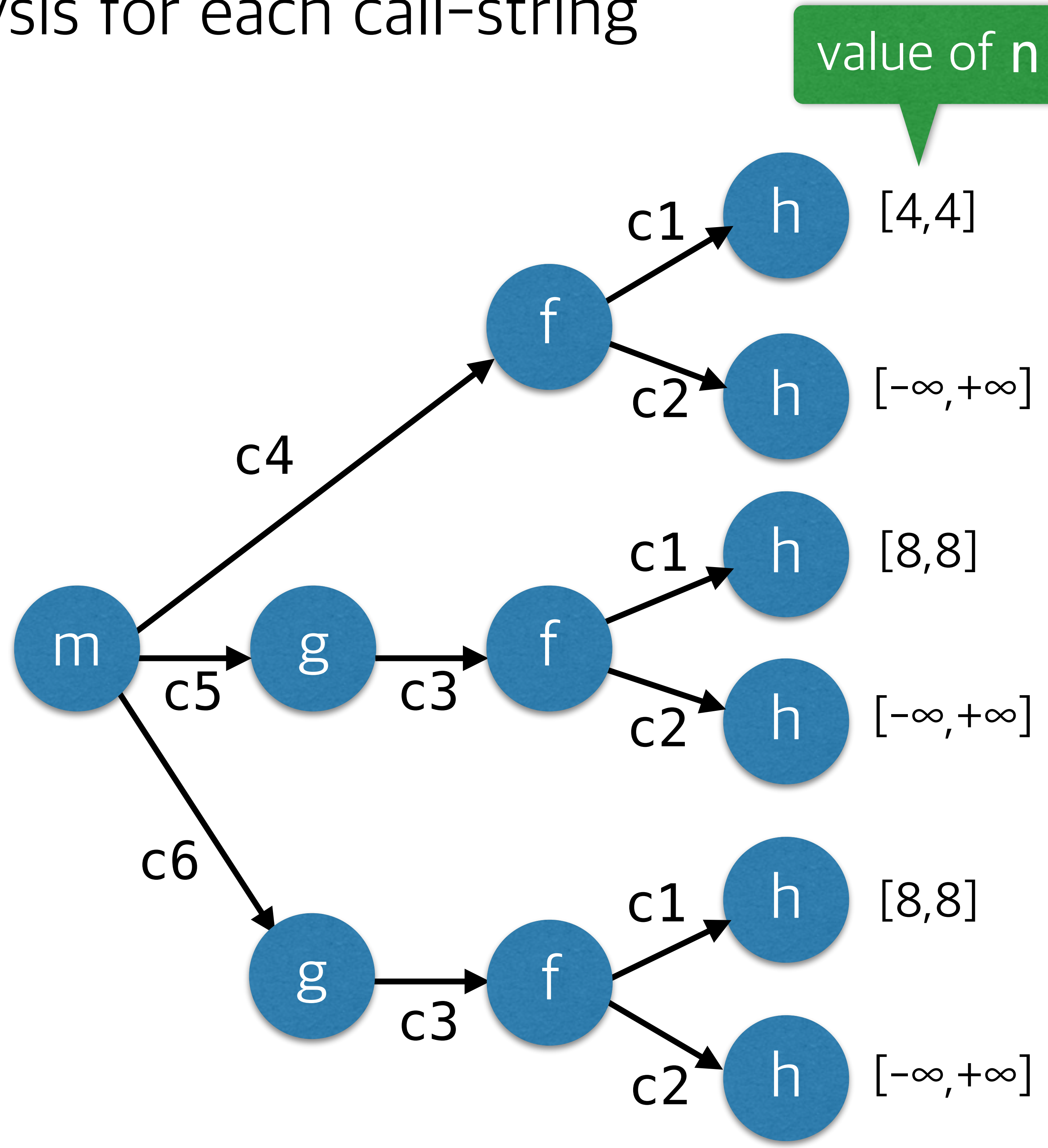
[-∞,+∞]

Context-insensitive interval analysis cannot prove Q1

Context-Sensitivity: 3-CFA

Separate analysis for each call-string

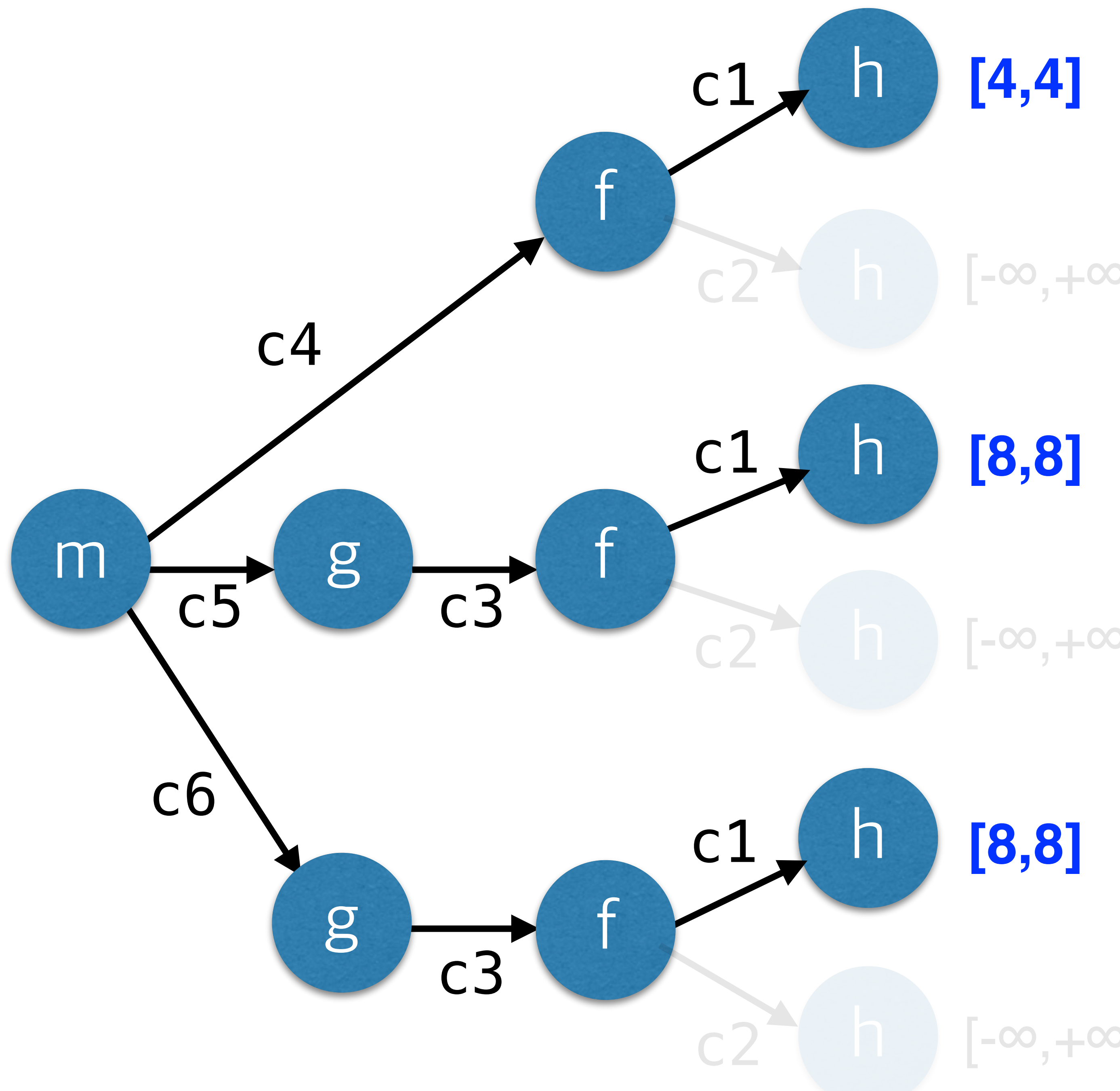
```
    int h(n) {ret n;}

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

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

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



Context-Sensitivity: 3-CFA

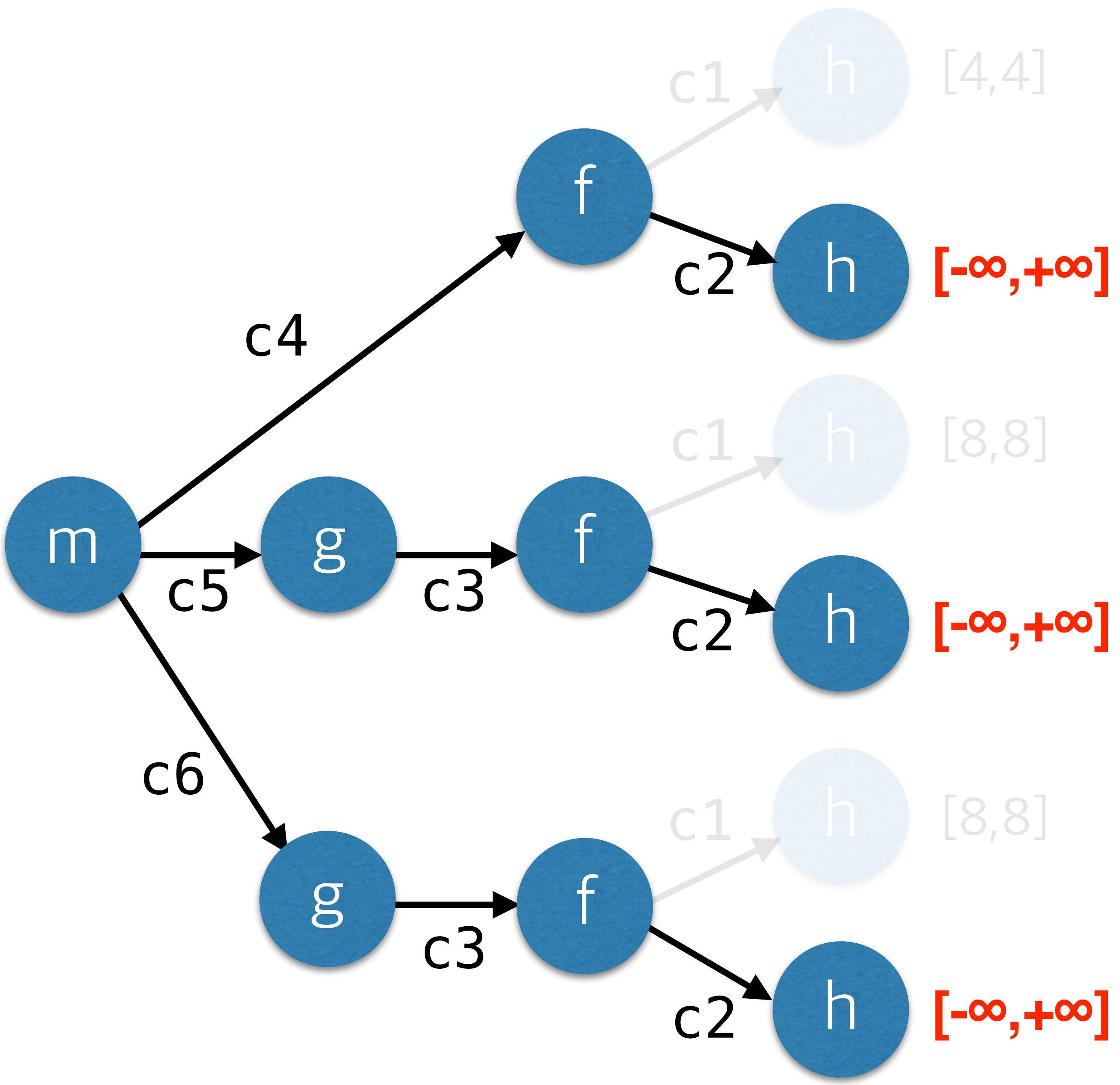
Separate analysis for each call-string

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

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

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

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



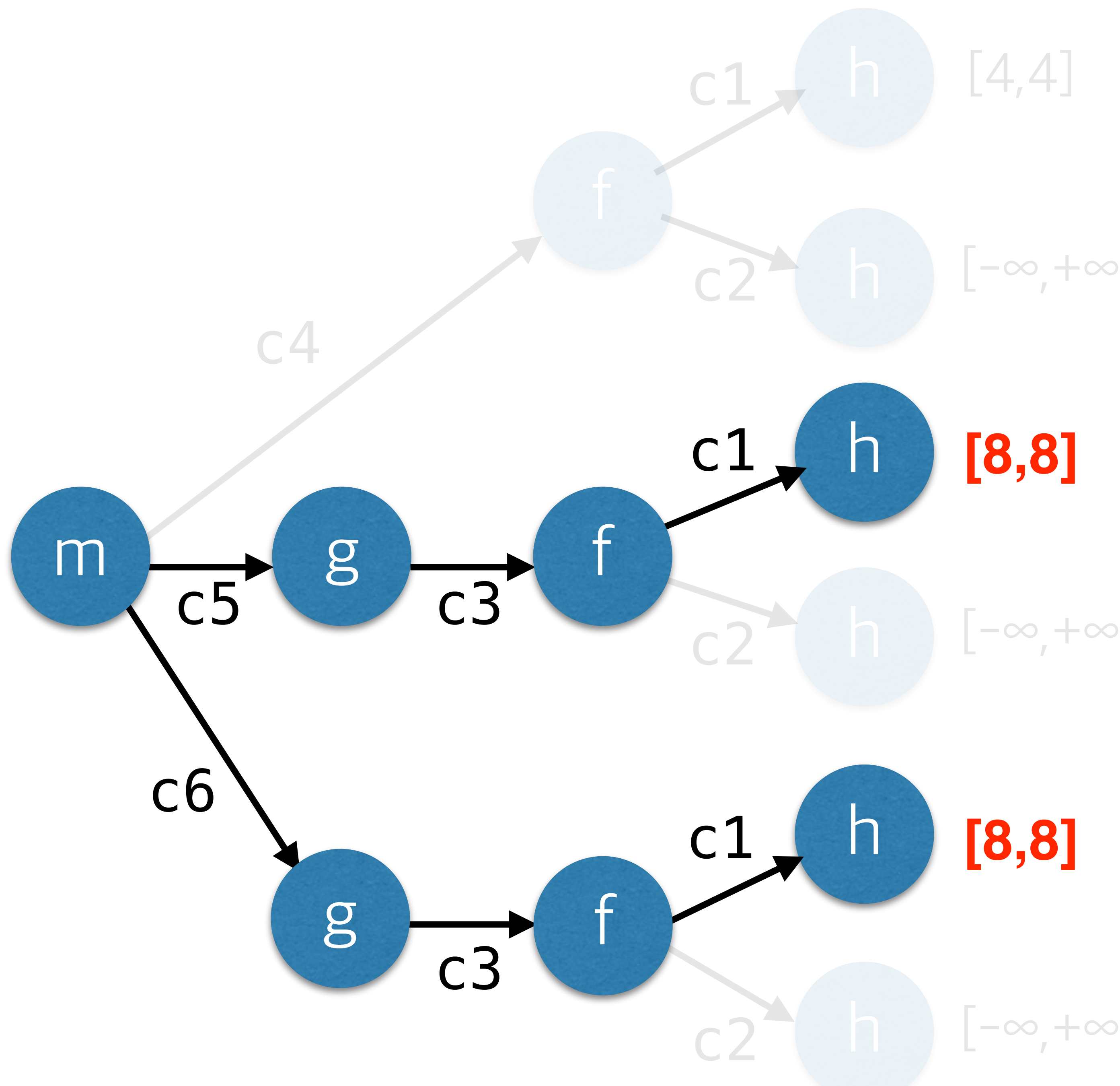
Problems of k-CFA

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

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

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

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



Problems of k-CFA

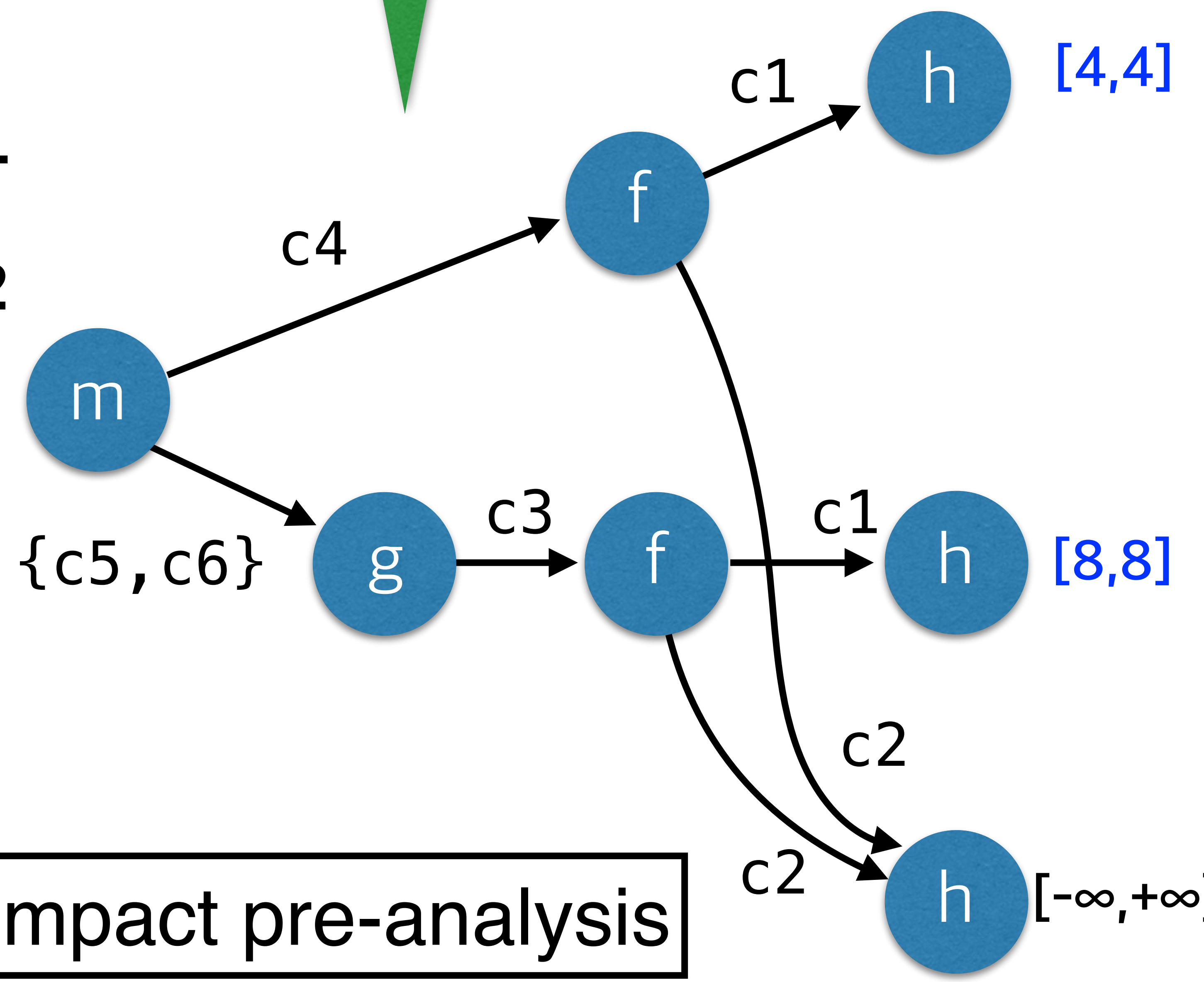
```
    int h(n) {ret n;}

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

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

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



Our Selective Context-Sensitivity

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

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

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

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

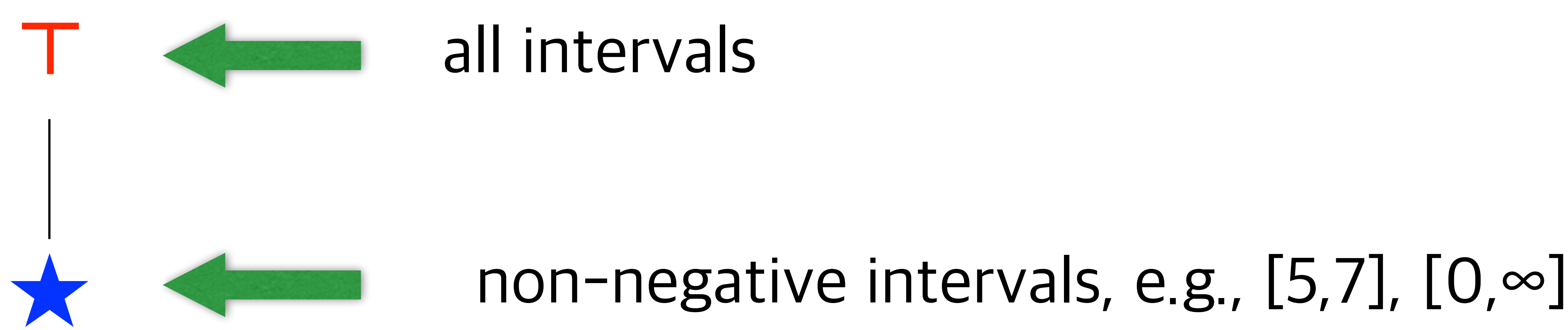
Challenge: How to infer this selective context-sensitivity?



Our solution: Impact pre-analysis

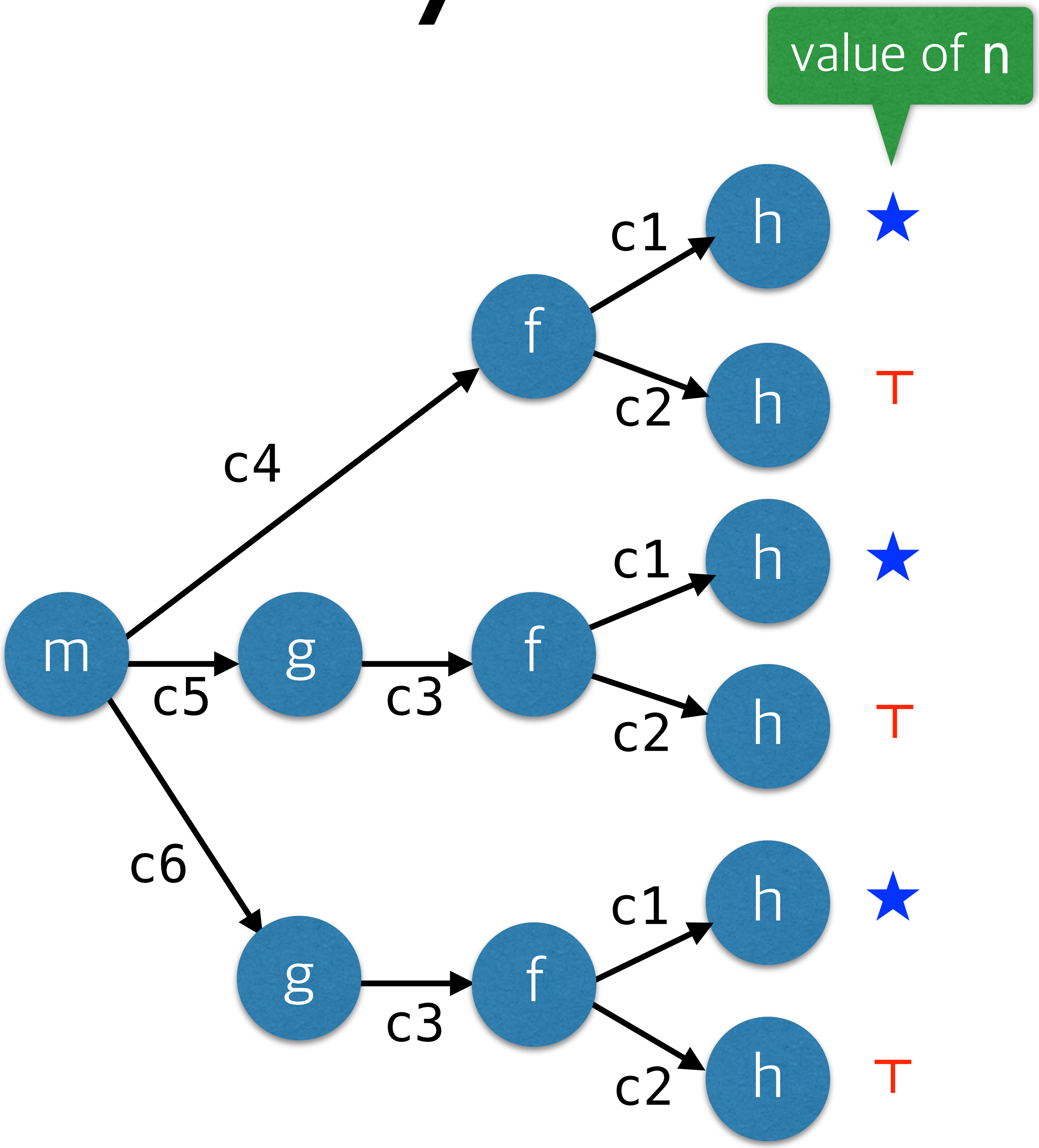
Impact Pre-Analysis

- Full context-sensitivity
- Approximate the interval domain



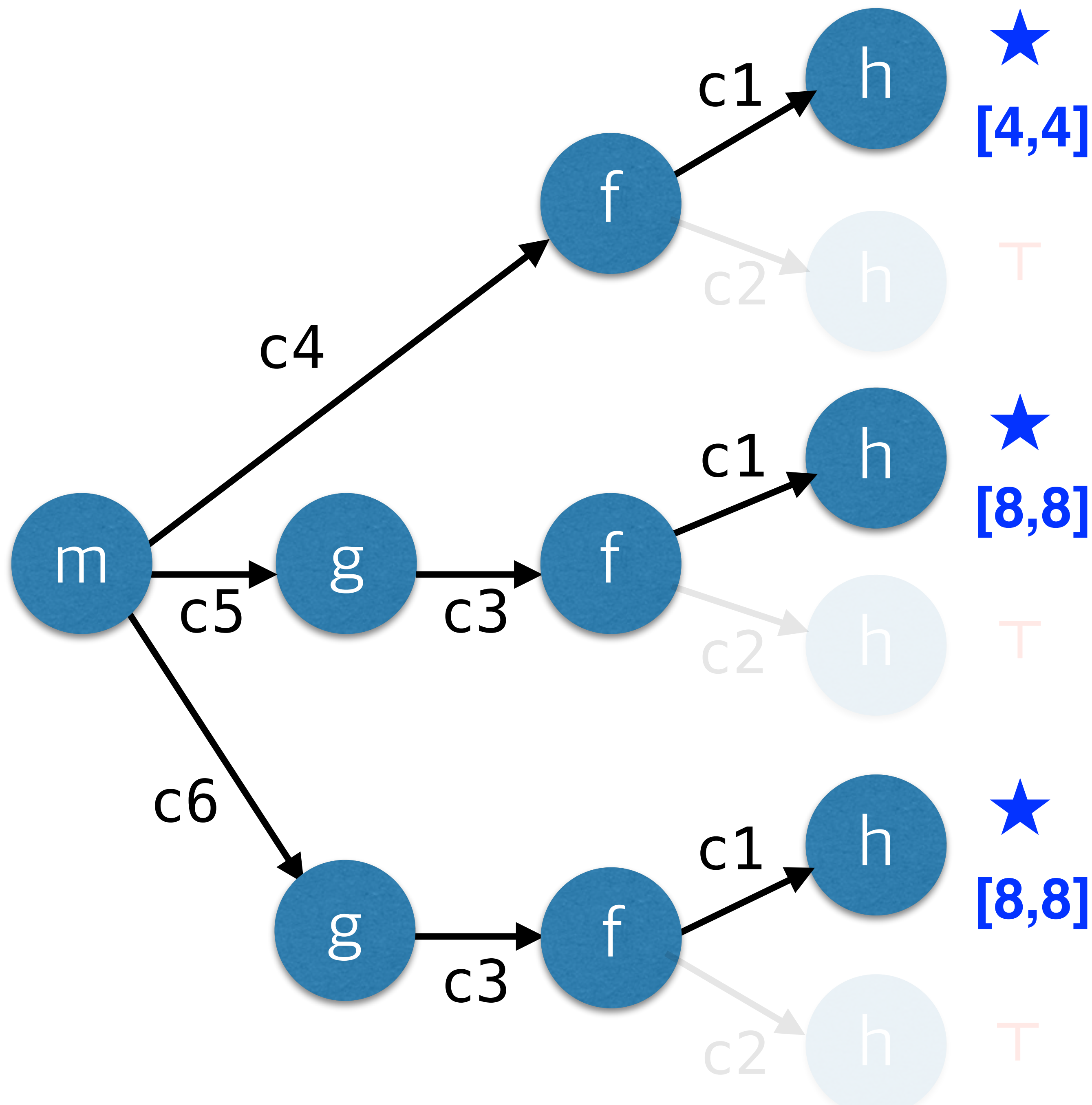
Impact Pre-Analysis

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

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

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

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



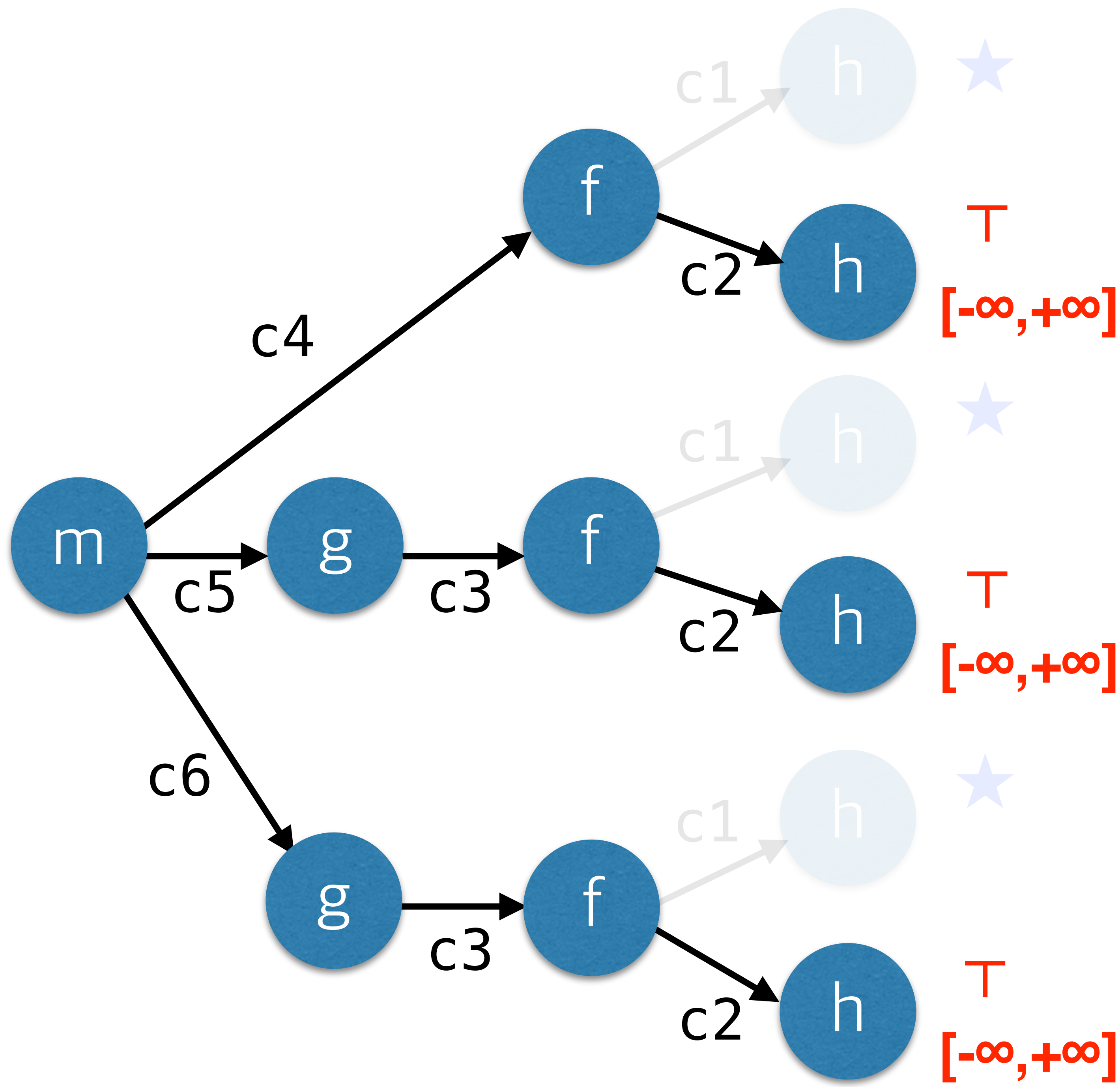
Impact Pre-Analysis

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

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

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

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



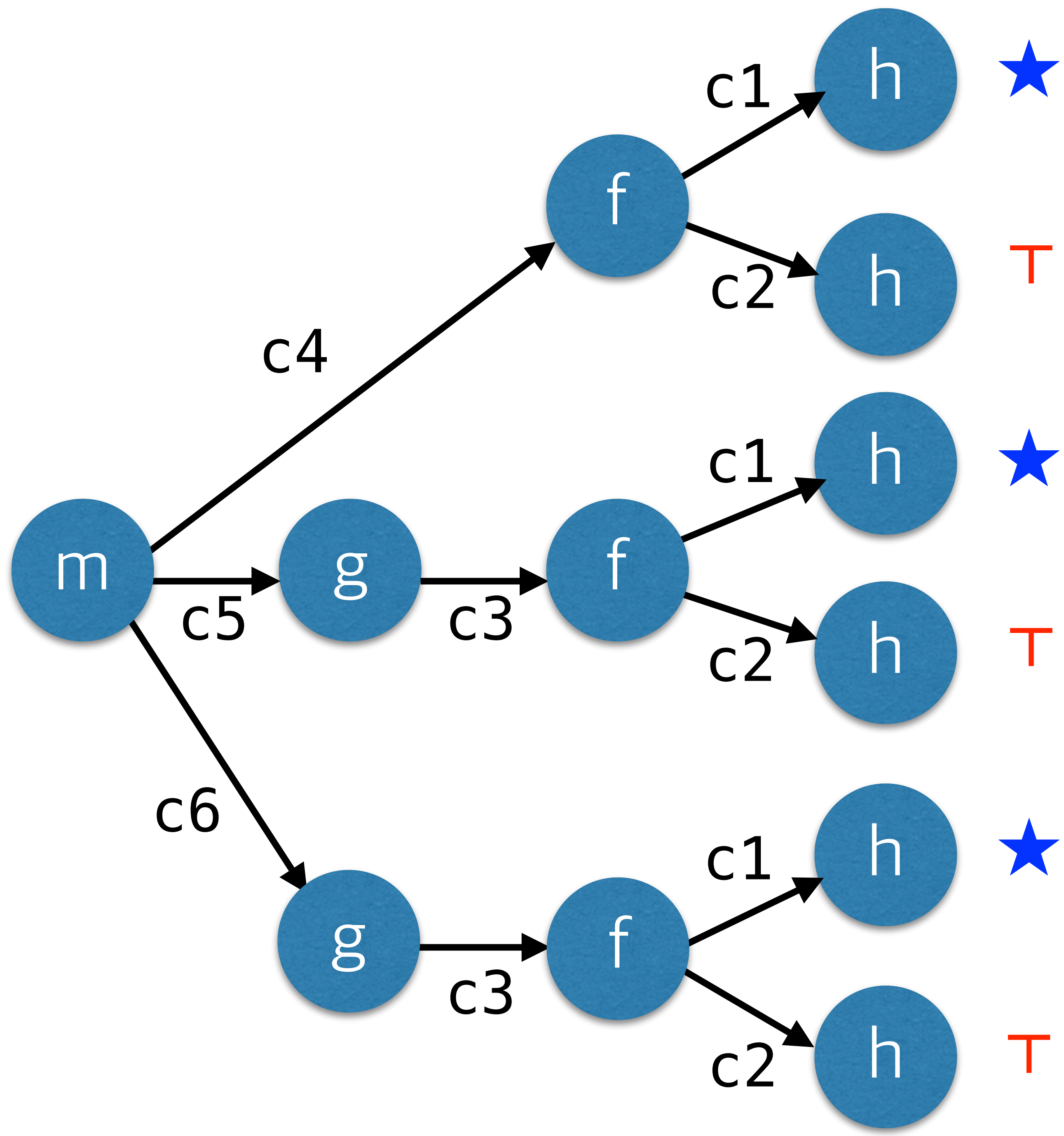
Impact Pre-Analysis

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

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

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

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



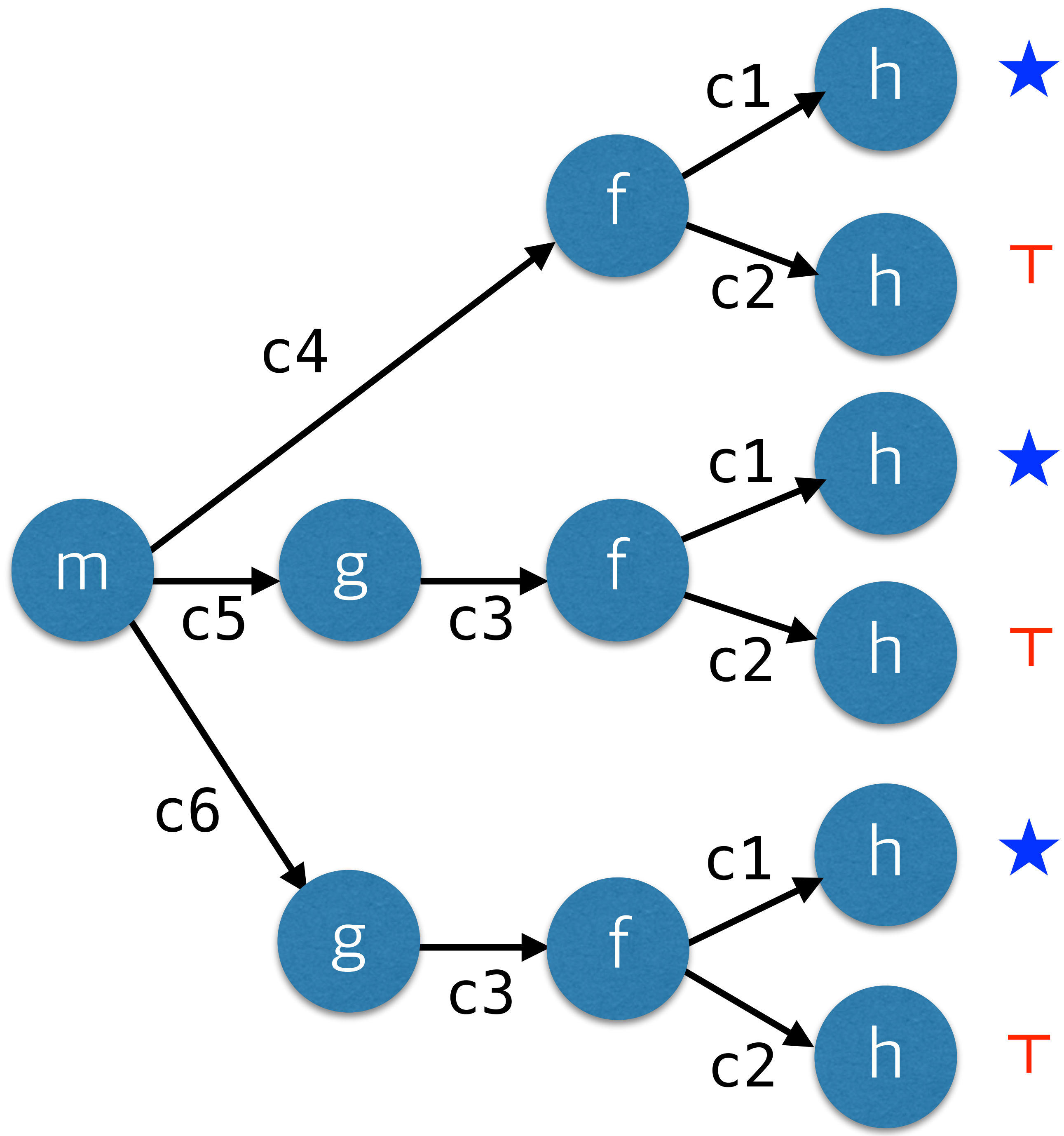
1. Collect queries whose expressions are assigned with ★

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

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

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

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



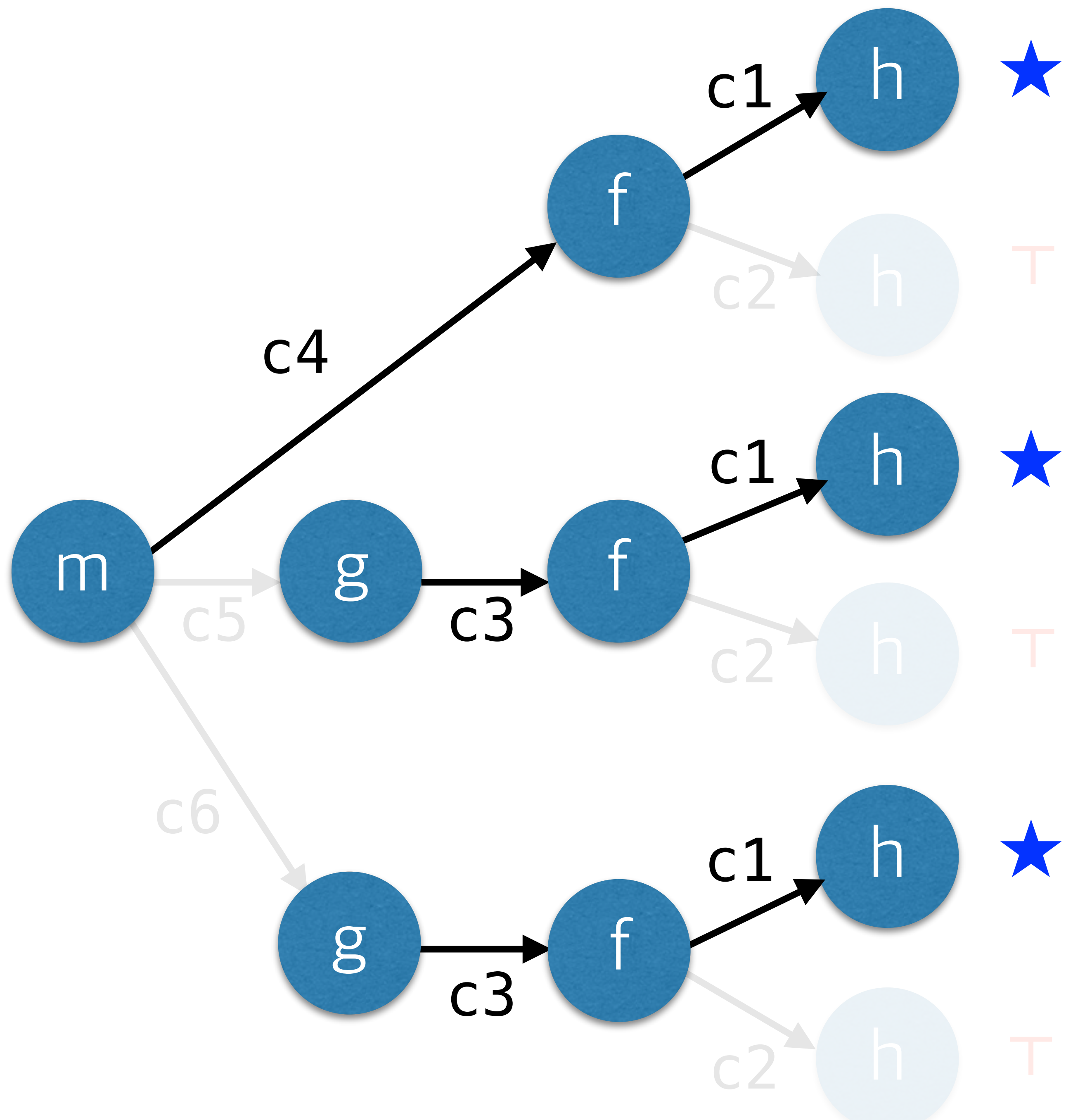
2. Find the program slice that contributes to the selected query

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

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

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

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



3. Collect contexts in the slice

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

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

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

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



=> Contexts for h: {c3 · c1, c4 · c1}

cf) Relational Analysis

a = b

i < b

```
int a = b;
int c = input();
for (i = 0; i < b; i++) {
  assert (i < a);  // Q1
  assert (i < c);  // Q2
}
```

	a	b	c	i
a	0	0	∞	-1
b	0	0	∞	-1
c	∞	∞	0	∞
i	∞	∞	∞	0

$i - a \leq -1$

$i - c \leq \infty$

vs.

	a	b	i
a	0	0	-1
b	0	0	-1
i	∞	∞	0

non-selective analysis

our selective analysis

Impact Pre-Analysis

- Fully relational
- Approximated in other precision aspects

	a	b	c	i
a	0	0	∞	-1
b	0	0	∞	-1
c	∞	∞	0	∞
i	∞	∞	∞	0

octagon analysis

vs.

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

impact pre-analysis

Selective Context-Sensitivity

		Context-Insensitve		Ours	
Pgm	LOC	#alarms	time(s)	#alarms	time(s)
spell	2K	58	0.6	30	0.9
bc	13K	606	14.0	483	16.2
tar	20K	940	42.1	799	47.2
less	23K	654	123.0	562	166.4
sed	27K	1,325	107.5	1,238	117.6
make	27K	1,500	88.4	1,028	106.2
grep	32K	735	12.1	653	15.9
wget	35K	1,307	69.0	942	82.1
a2ps	65K	3,682	118.1	2,121	177.7
bison	102K	1,894	136.3	1,742	173.4
TOTAL	346K	12,701	707.1	9,598	903.6

24.4%

Selective Context-Sensitivity

		Context-Insensitve		Ours	
Pgm	LOC	#alarms	time(s)	#alarms	time(s)
spell	2K	58	0.6	30	0.9
bc	13K	606	14.0	483	16.2
tar	20K	940	42.1	799	47.2
less	23K	654	123.0	562	166.4
sed	27K	1,325	107.5	1,238	117.6
make	27K	1,500	88.4	1,028	106.2
grep	32K	735	12.1	653	15.9
wget	35K	1,307	69.0	942	82.1
a2ps	65K	3,682	118.1	2,121	177.7
bison	102K	1,894	136.3	1,742	173.4
TOTAL	346K	12,701	707.1	9,598	903.6

pre-analysis : 14.7%
main analysis: 13.1%

27.8%

k-CFA did not scale

- 2 or 3-CFA did not scale over 10KLoC
  - e.g., for spell (2KLoC):
    - 3-CFA reported 30 alarms in 11.9s
    - cf) ours: 30 alarms in 0.9s
- 1-CFA did not scale over 40KLoC

Selective Octagon Analysis

			Existing Approach [Miné06]		Ours	
Pgm	LOC	#queries	proven	time(s)	proven	time(s)
calc	298	10	2	0.3	10	0.2
spell	2,213	16	1	4.8	16	2.4
barcode	4,460	37	16	11.8	37	30.5
httptunnel	6,174	28	16	26.0	26	15.3
bc	13,093	10	2	247.1	9	117.3
tar	20,258	17	7	1043.2	17	661.8
less	23,822	13	0	3031.5	13	2849.4
a2ps	64,590	11	0	29473.3	11	2741.7
TOTAL	135,008	142	44	33840.3	139	6418.6

+95

Selective Octagon Analysis

			Existing Approach [Miné06]		Ours	
Pgm	LOC	#queries	proven	time(s)	proven	time(s)
calc	298	10	2	0.3	10	0.2
spell	2,213	16	1	4.8	16	2.4
barcode	4,460	37	16	11.8	37	30.5
httptunnel	6,174	28	16	26.0	26	15.3
bc	13,093	10	2	247.1	9	117.3
tar	20,258	17	7	1043.2	17	661.8
less	23,822	13	0	3031.5	13	2849.4
a2ps	64,590	11	0	29473.3	11	2741.7
TOTAL	135,008	142	44	33840.3	139	6418.6

reduce time by -81%

Learning Automatically

- Develop techniques for automatically finding the selection strategies
- Use machine learning techniques to learn a good strategy from freely available data.

Static Analyzer

number of
proved assertions

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

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

Overall Approach

- Parameterized adaptation strategy

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

- Learn a good parameter W from existing codebase

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

Codebase

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

1. Parameterized Strategy

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

(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

$\Rightarrow$ {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

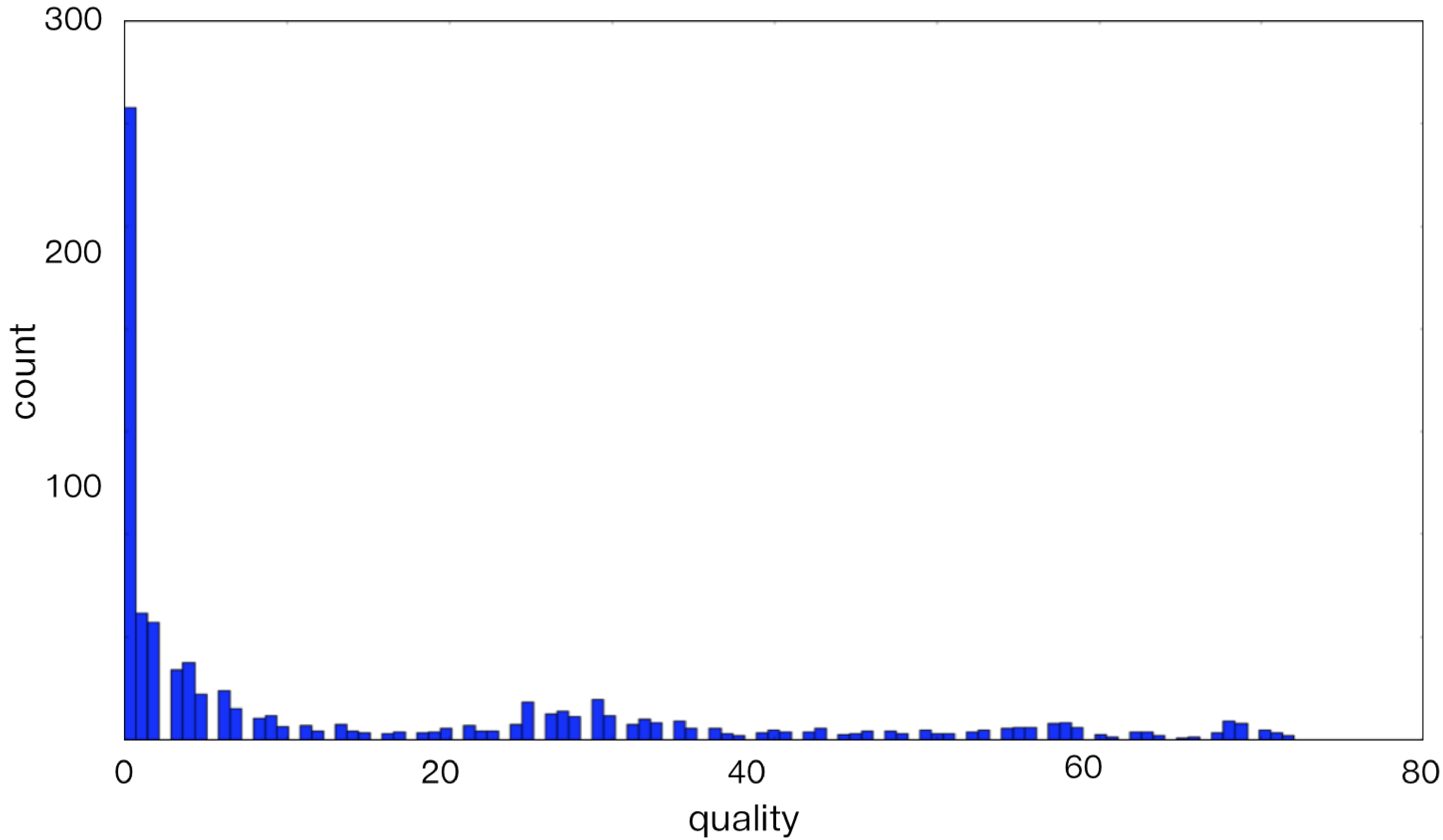
repeat N times

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

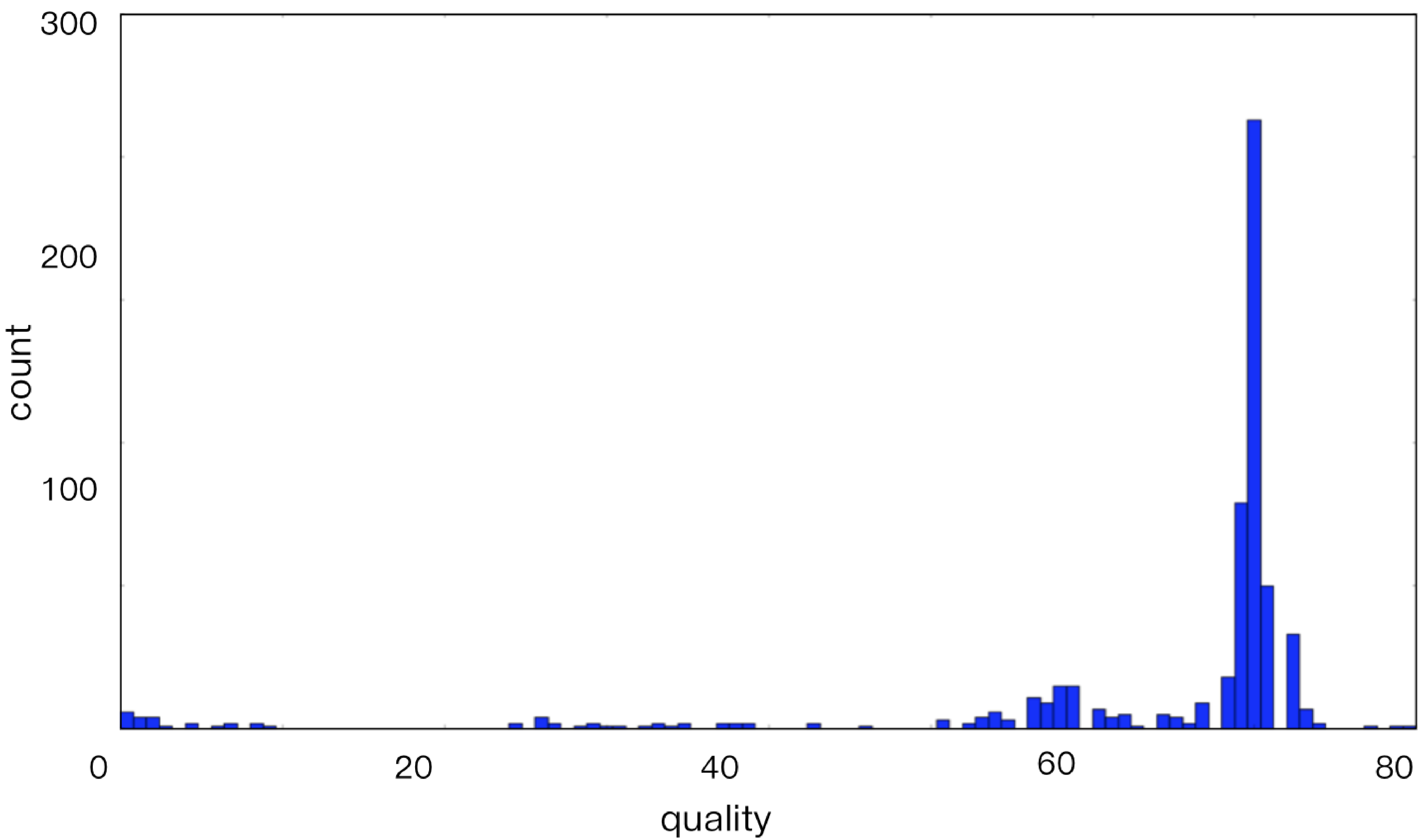
  evaluate $\sum_{P_i} F(P_i, S_{\mathbf{w}}(P_i))$

  update the probabilistic model

return best w found

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

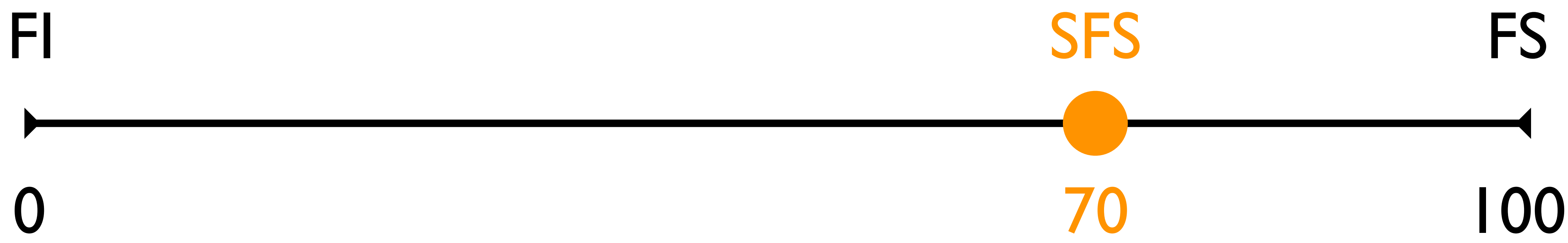
Learning via Bayesian Optimization



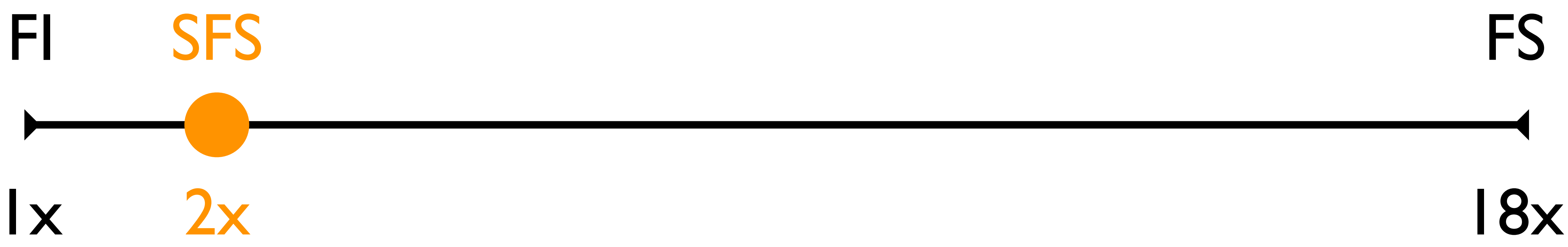
Effectiveness

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

Precision



Cost



Learning via White-box Optimization [APLAS'16]

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

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

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

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

Learning from Automatically Labelled Data [SAS'16]

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

	a	−a	b	−b	c	−c	i	−i
a	★	$\top$	★	$\top$	$\top$	$\top$	★	$\top$
−a	$\top$	★	$\top$	★	$\top$	$\top$	$\top$	$\top$
b	★	$\top$	★	$\top$	$\top$	$\top$	★	$\top$
−b	$\top$	★	$\top$	★	$\top$	$\top$	$\top$	$\top$
c	$\top$	$\top$	$\top$	$\top$	★	$\top$	$\top$	$\top$
−c	$\top$	$\top$	$\top$	$\top$	$\top$	★	$\top$	$\top$
i	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	★	$\top$
−i	$\top$	★	$\top$	★	$\top$	$\top$	$\top$	★

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

Automatically Generating Features (In Progress)

Limitation: Feature Engineering

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

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

flow-sensitivity

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

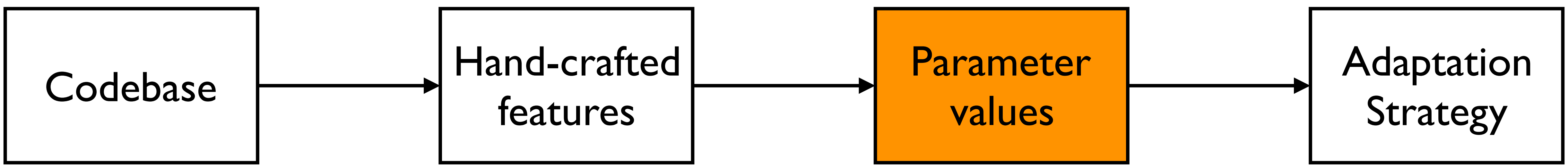
context-sensitivity

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

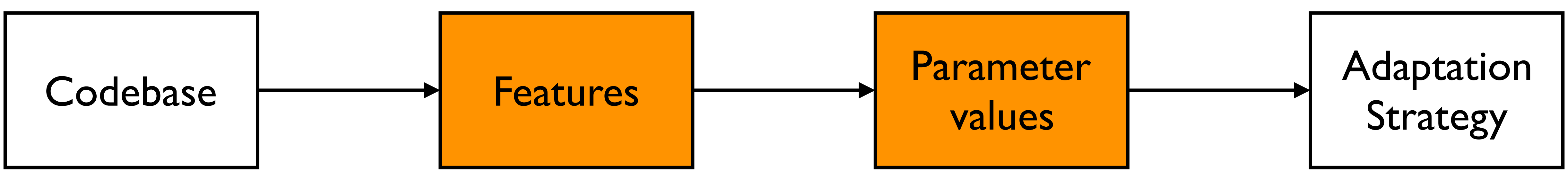
widening thresholds

Automatic Feature Generation

Before



New method



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

Example: Flow-Sensitive Analysis

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

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

flow-sensitive result		flow-insensitive result
line	abstract state	abstract state
1	$\{x \mapsto [0,0], y \mapsto [0,0]\}$	
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:

$$\{(q_i, b_i)\}_{i=1}^{n} \xrightarrow{\text{feature extraction}} \{(v_i, b_i)\}_{i=1}^{n} \ (v_i \in \mathbb{B}^k) \xrightarrow{\text{standard learning algorithms}} \mathcal{C} : \mathbb{B}^k \to \mathbb{B}$$

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

$\boxed{\text{We aim to automatically find the right representation}}$

Feature Extraction

A feature describes a property of queries

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

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

Generating Features

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

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

Generating Features

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

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

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

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

- Reducing programs while preserving the condition

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

  generates feature programs.

Generating Features

- Reduce programs while preserving the condition

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

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

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

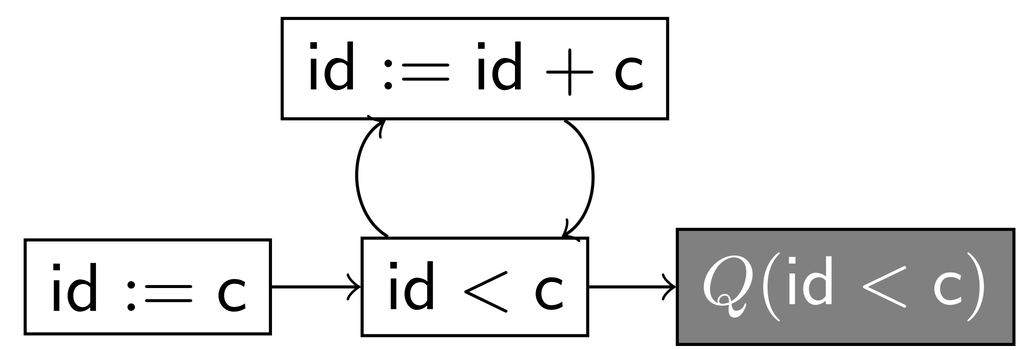
```
a = 0;
while (1) {
 if (a < 3)
   assert (a < 5);
 a++;
}
```

Generating Features

- Represent the features by abstract data flow graphs

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

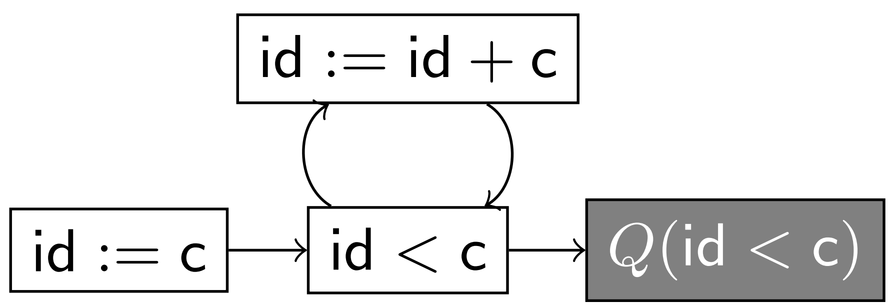
$\overset{\alpha}{\Longrightarrow}$



- The right level of abstraction is learned from codebase

Matching Algorithm

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



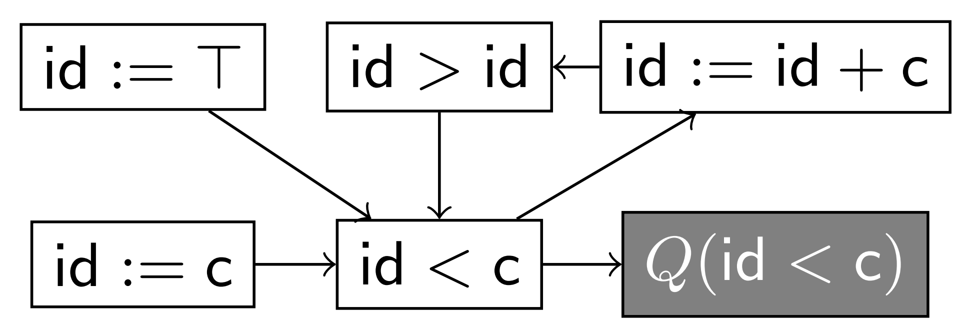
$\overset{?}{\subseteq}$

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

$\subseteq$

Subgraph inclusion:

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



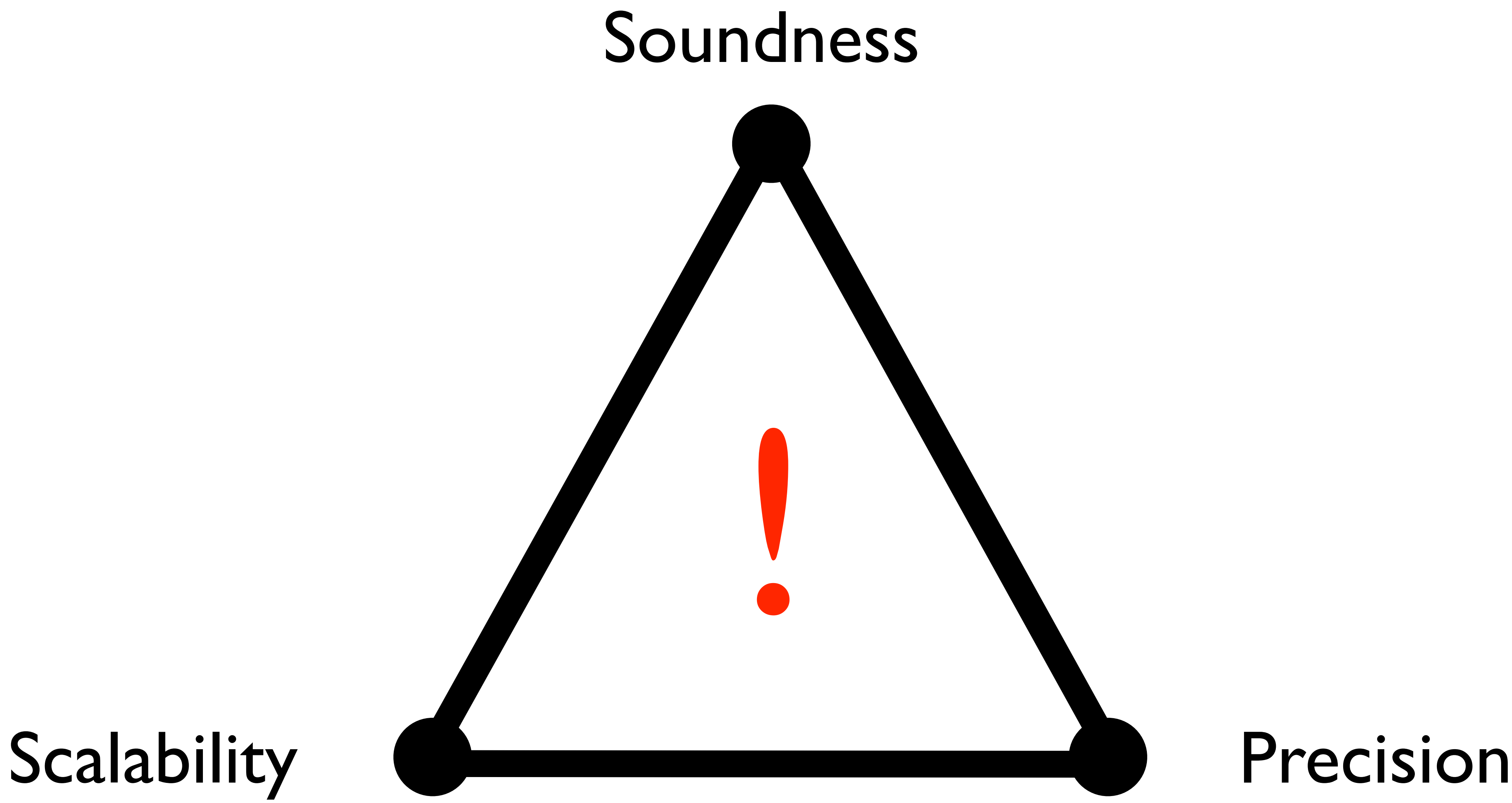
Performance

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

Trial	Query Prediction		Analysis								
			Prove			Sec					
	Precision	Recall	FI	FS	SFS	FI	FS	SFS	Quality	Cost	QualityTR
1	92.6 %	77.9 %	5,340	6,053	5,973	38.2	564.0	55.3	88.7 %	1.4x	88.7 %
2	78.8 %	73.3 %	2,972	3,373	3,262	16.3	460.5	25.7	72.3 %	1.5x	72.0 %
3	66.7 %	73.3 %	3,984	4,668	4,559	27.3	1,635.6	176.2	84.0 %	6.4x	82.7 %
4	88.7 %	68.8 %	4,600	5,450	5,307	38.1	688.2	59.6	83.1 %	1.5x	83.5 %
5	89.9 %	79.4 %	2,517	2,971	2,945	10.9	325.9	18.9	94.2 %	1.7x	94.0 %
TOTAL	81.5 %	73.9 %	19,413	22,515	22,046	131.1	3,674.4	336.0	**84.8** %	**2.5x**	84.6 %

Table 1. Effectiveness of partially flow-sensitive interval analysis

Summary



General Sparse Analysis Framework

Selective X-sensitivity (pre-analysis, ML, etc)