Handling Cyclic Reinforcement of Lattice Values in Incremental Dependency-driven Static Analysis

Jens Van der Plas, Quentin Stiévenart, **Coen De Roover**

coen.de.roover@vub.be

SCAM 2025
8-9 September 2025
Auckland, NZ

Credit where credit is due

Quentin

Jens

Static program analysis

```
const onClickHandler = () => {
  const $ = document.querySelector;
  let pass = $("#pass").value;
  console.log(pass);
}
```

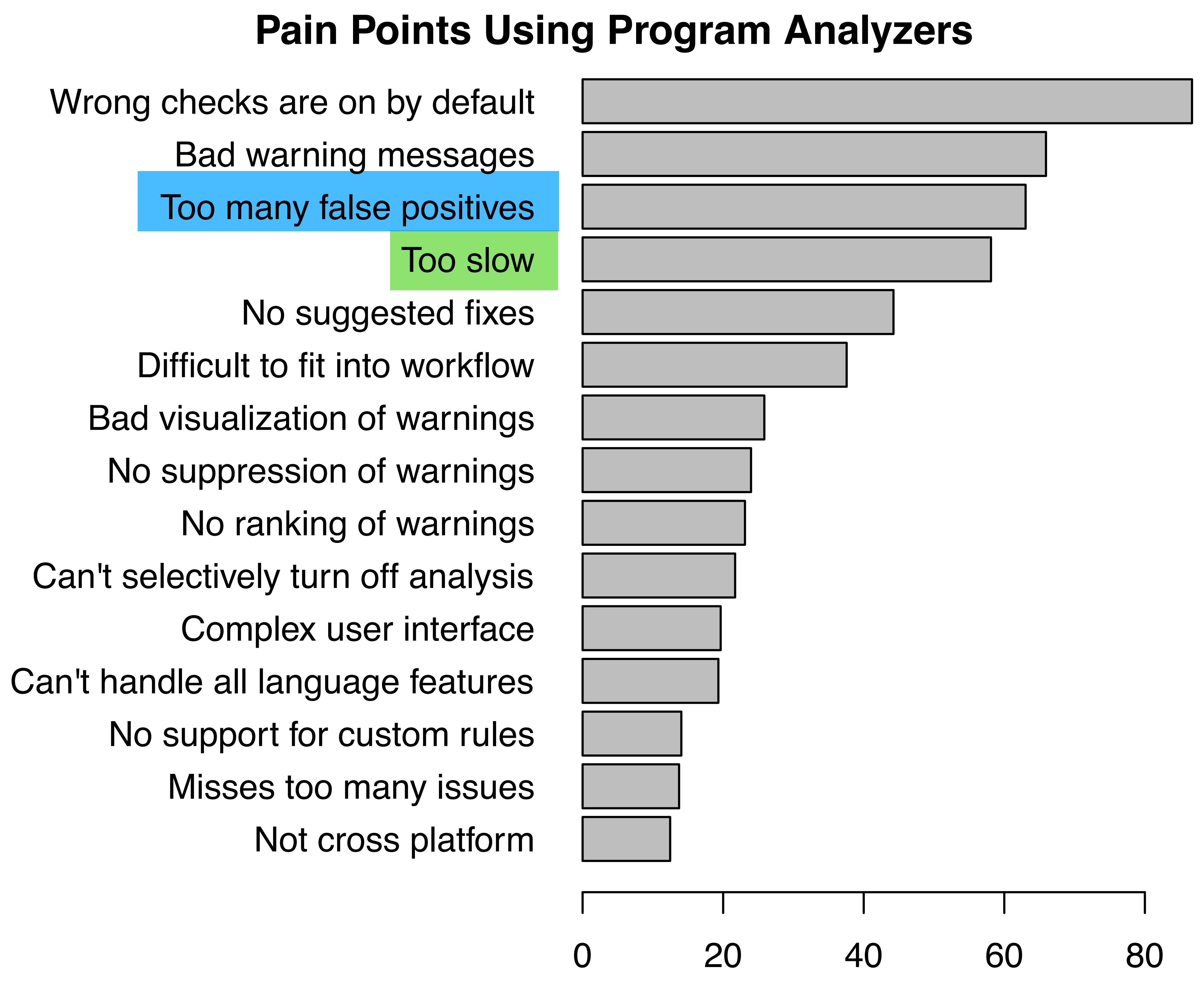
sink to be avoided

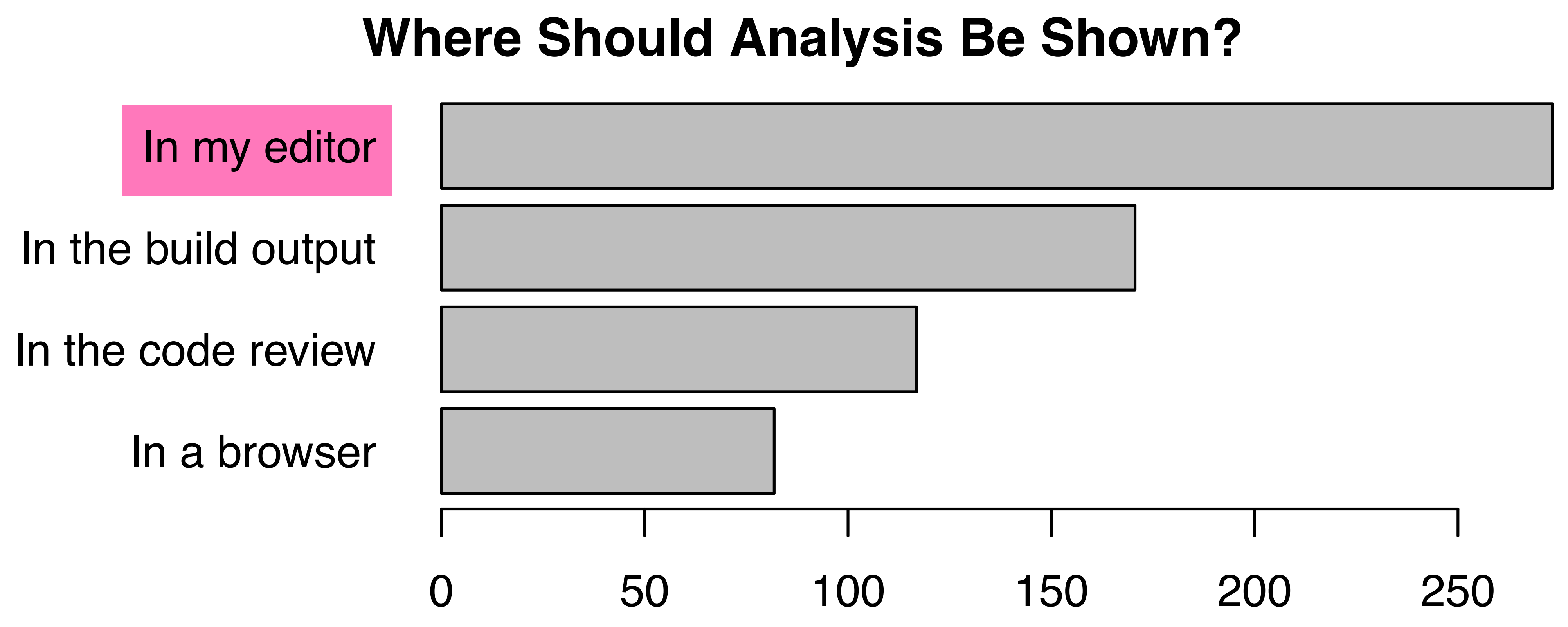
source of sensitive information

- Where is this class instantiated?
- Which code will never be executed?
- Can this acces raise a NullPointerException?
- Can this integer arithmetic overflow?
- **May sensitive information leak outside?**
- …

answer questions about **any execution** of the **program**, **without executing** it

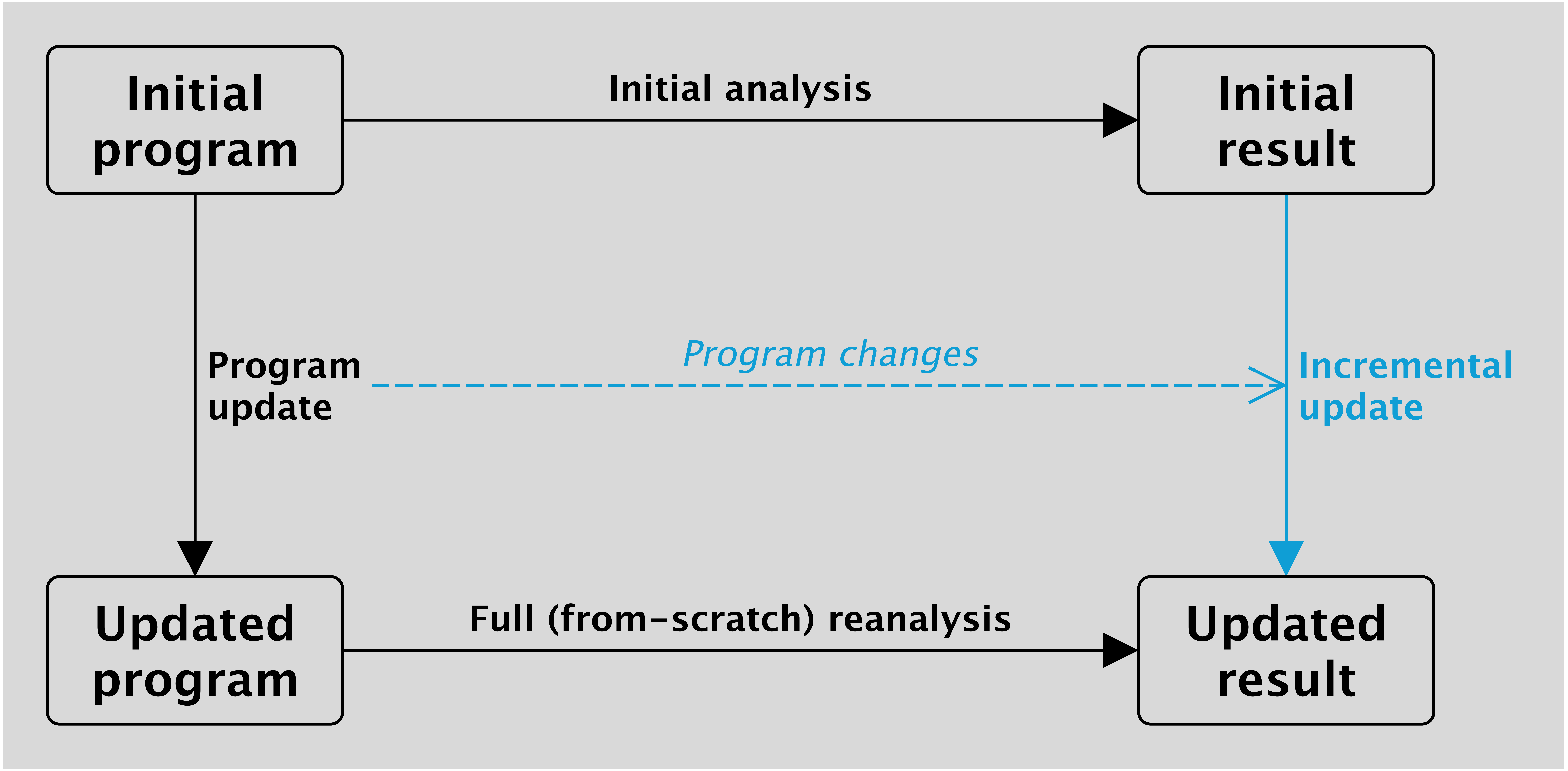
What (375 Microsoft) developers need





Christakis et al. [ASE2016]

Incrementalisation to the rescue



Our approach to incrementalisation

- **perform change impact analysis**
  from AST changes to analysis results to know what can be kept

- **until a new fixed point has been found**
    - **remove and refine outdated results**
    - **add new results**

by rescheduling dependents of changed result:
requires reifying computational dependencies

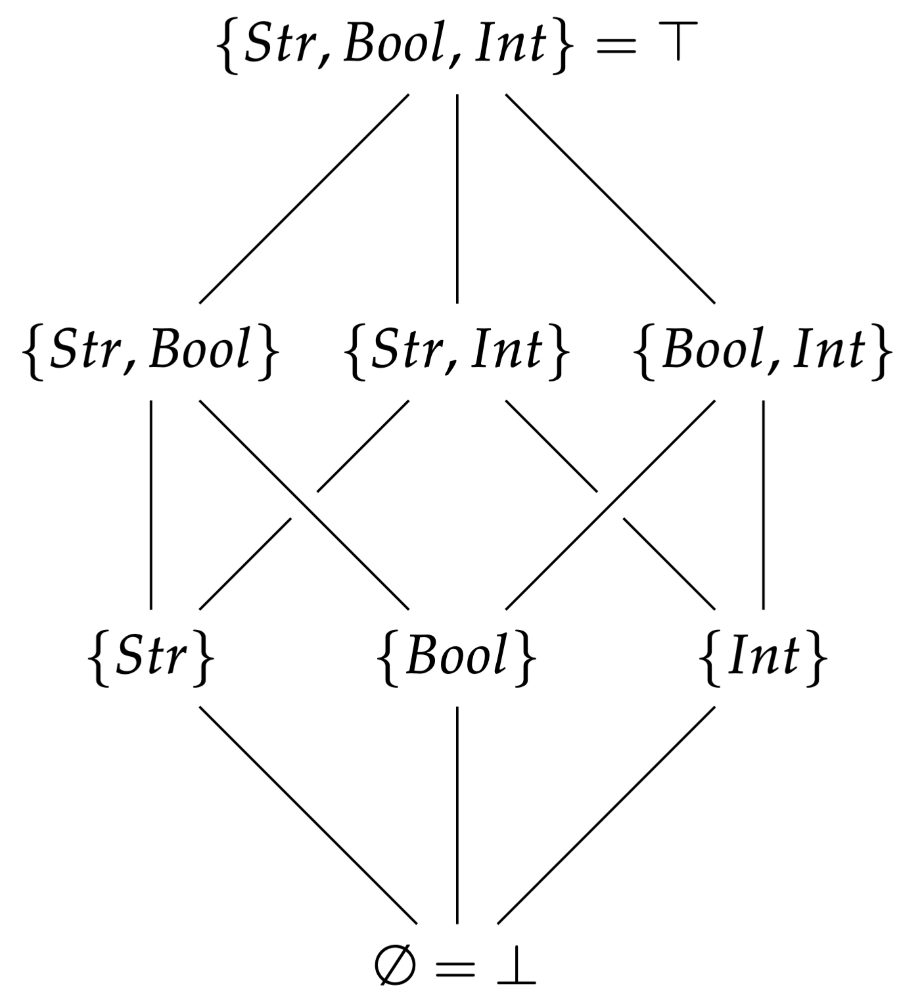
Reified computational dependencies

```
(define x 0)
(define (fun) (inc) x)
(define (inc) (set! x (+ x 1)) #t)
(fun)
```

Start of the analysis

Worklist
Main

Global store
$\ldots \mapsto \perp$



$$\begin{tikzcd}
 & \{Str, Bool, Int\} = \top \arrow[dl, no head] \arrow[d, no head] \arrow[dr, no head] & \\
\{Str, Bool\} \arrow[d, no head] \arrow[dr, no head] & \{Str, Int\} \arrow[dl, no head] \arrow[dr, no head] & \{Bool, Int\} \arrow[dl, no head] \arrow[d, no head] \\
\{Str\} \arrow[dr, no head] & \{Bool\} \arrow[d, no head] & \{Int\} \arrow[dl, no head] \\
 & \emptyset = \perp &
\end{tikzcd}$$

🟧 Component ● Global store address $\xrightarrow{e}$ Effect

Reified computational dependencies

```
(define x 0)
(define (fun) (inc) x)
(define (inc) (set! x (+ x 1)) #t)
(fun)
```

Analysed main

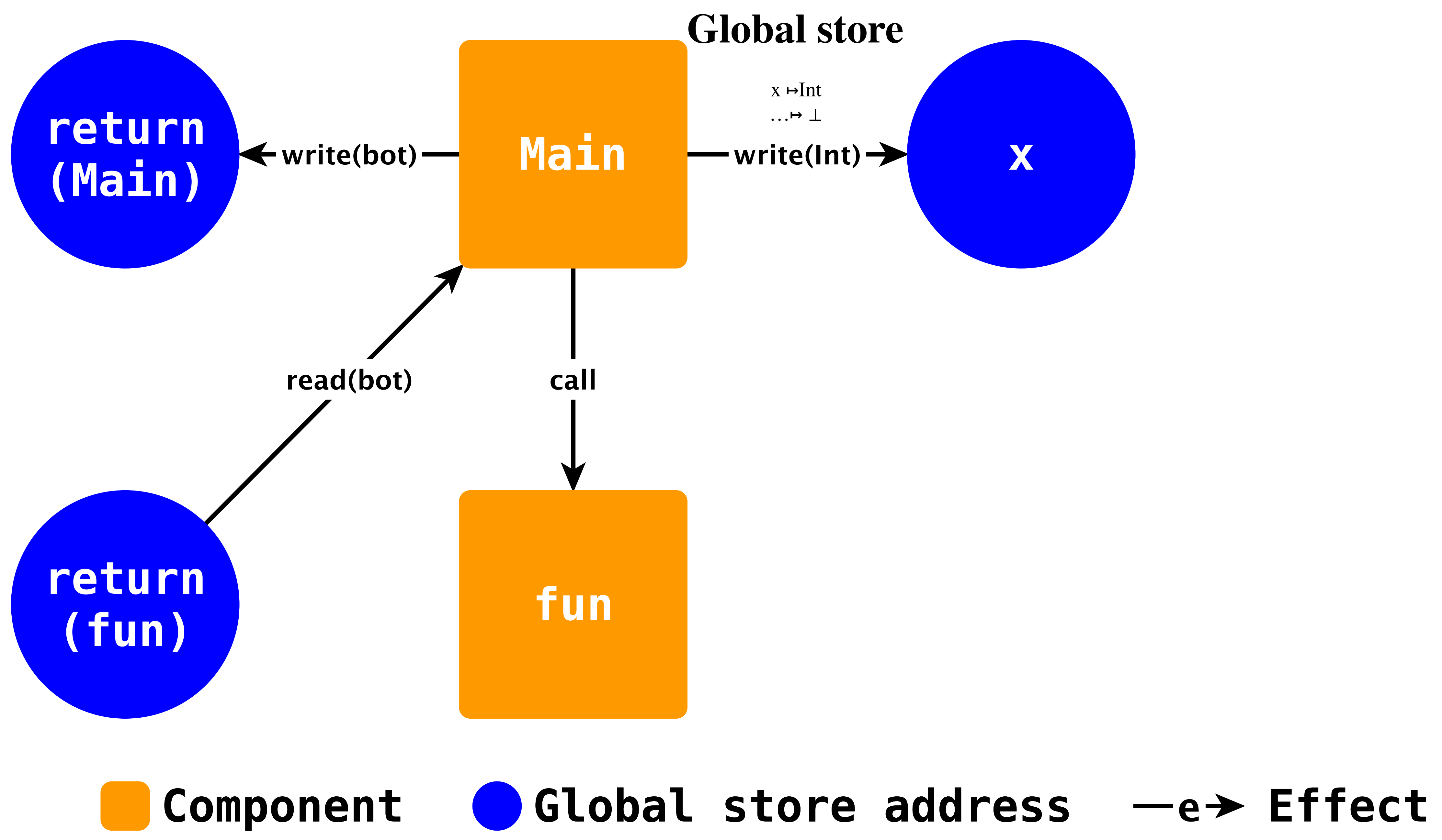
Worklist

fun

Global store

$x \mapsto \mathrm{Int}$

$\ldots \mapsto \bot$



Reified computational dependencies

```
(define x 0)
(define (fun) (inc) x)
(define (inc) (set! x (+ x 1)) #t)
(fun)
```

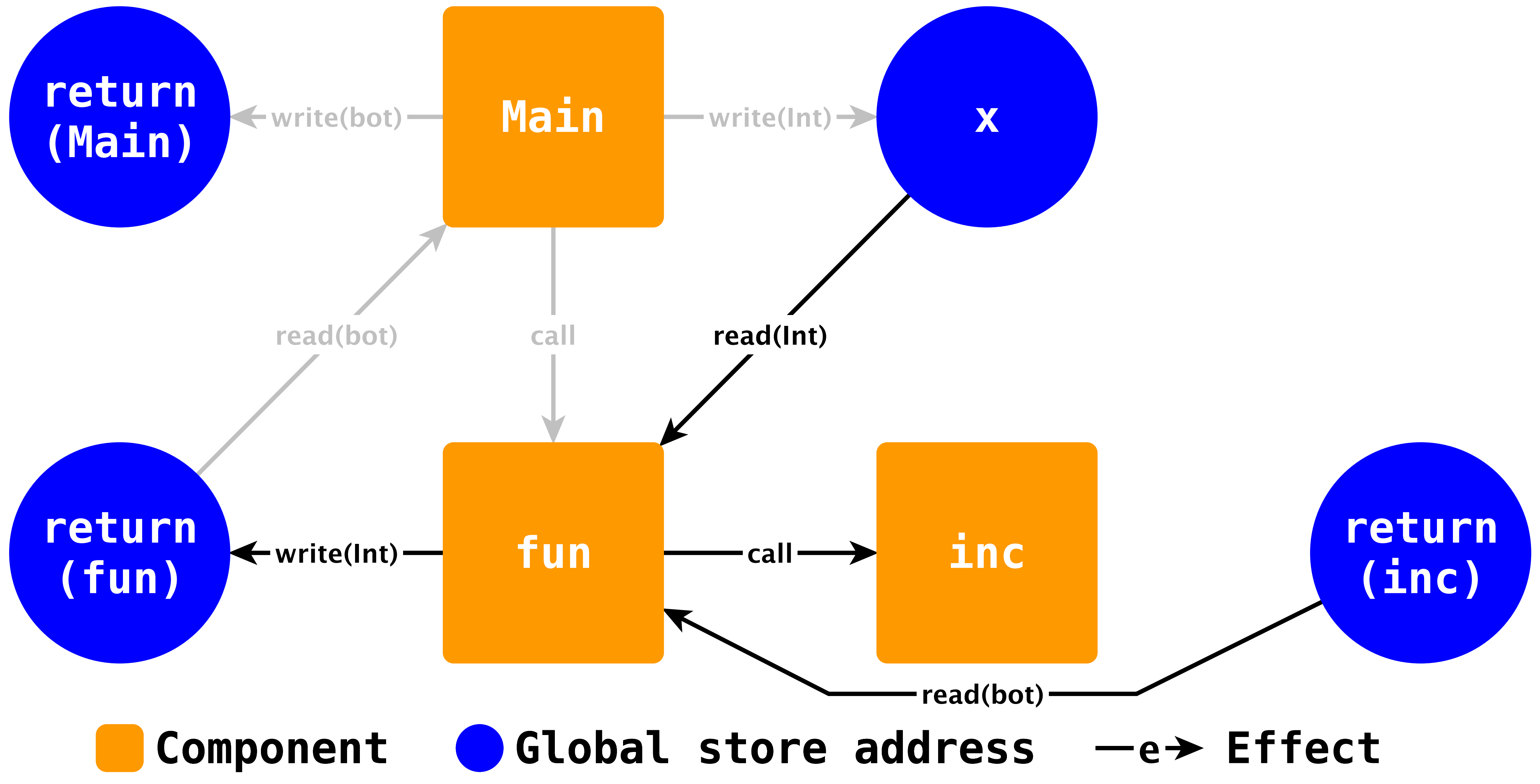
Analysed fun

Worklist

inc
Main

Global store

$$\begin{aligned} x &\mapsto \mathrm{Int} \\ \mathrm{return}(\mathrm{fun}) &\mapsto \mathrm{Int} \\ \ldots &\mapsto \bot \end{aligned}$$



Reified computational dependencies

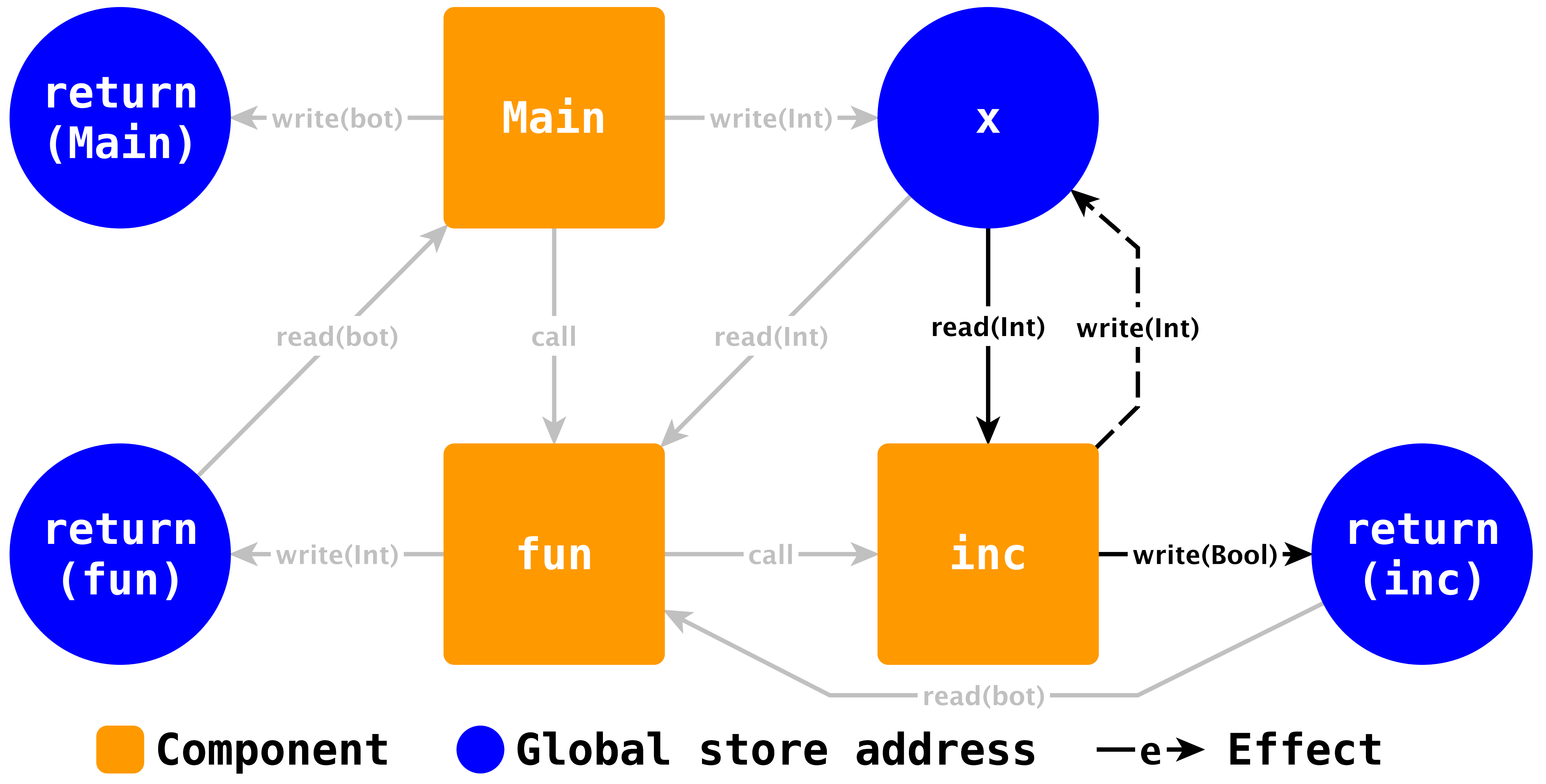
```
(define x 0)
(define (fun) (inc) x)
(define (inc) (set! x (+ x 1)) #t)
(fun)
```

Analysed inc

Worklist
Main
fun

Global store

$$\begin{aligned} x &\mapsto \text{Int} \\ \text{return}(\text{fun}) &\mapsto \text{Int} \\ \text{return}(\text{inc}) &\mapsto \text{Bool} \\ \ldots &\mapsto \bot \end{aligned}$$



Reified computational dependencies

```
(define x 0)
(define (fun) (inc) x)
(define (inc) (set! x (+ x 1)) #t)
(fun)
```

End of analysis

Worklist

$\varnothing$

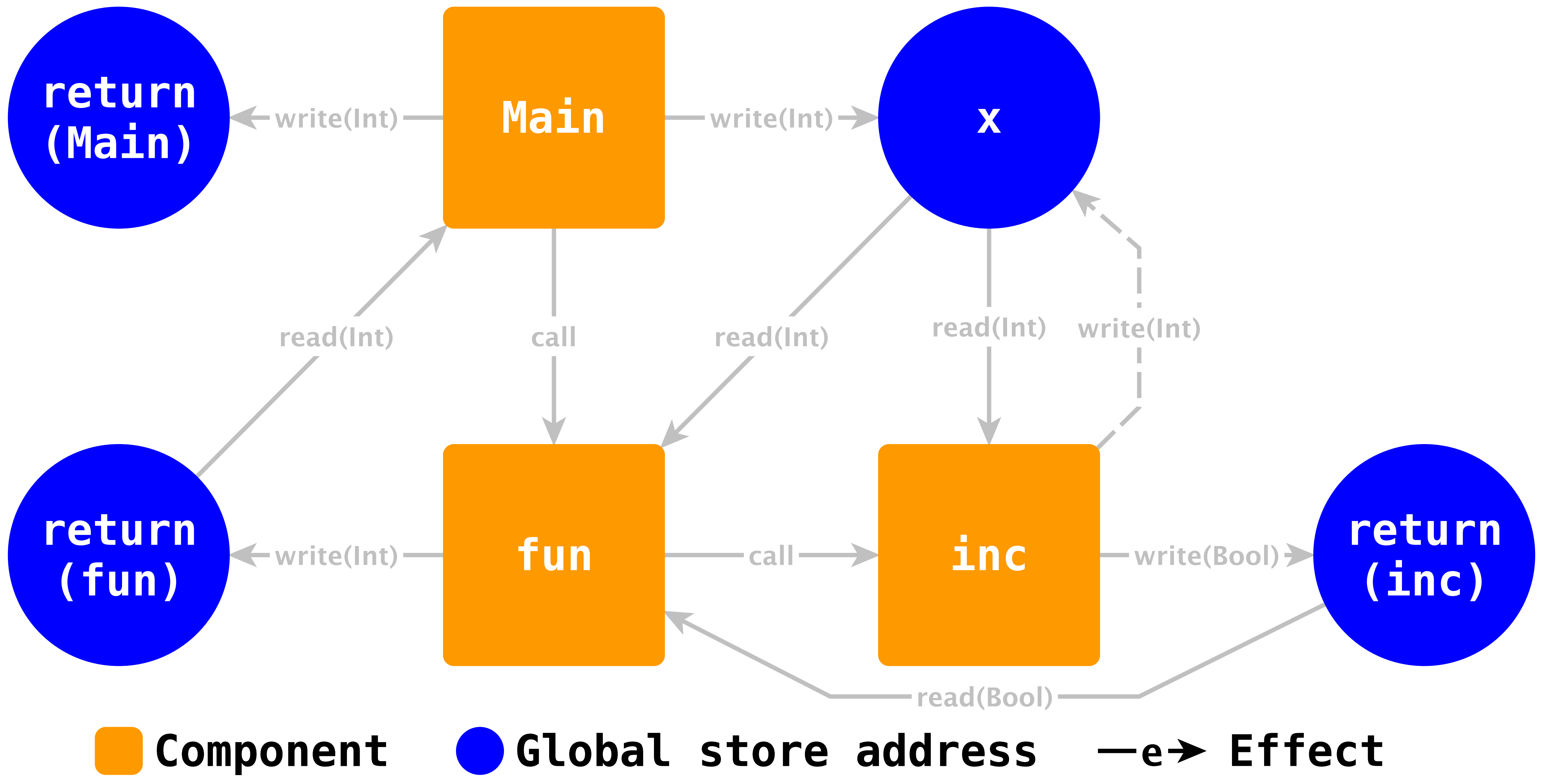
Global store

$\mathrm{x} \mapsto \mathrm{Int}$

$\mathrm{return}(\mathrm{fun}) \mapsto \mathrm{Int}$

$\mathrm{return}(\mathrm{inc}) \mapsto \mathrm{Bool}$

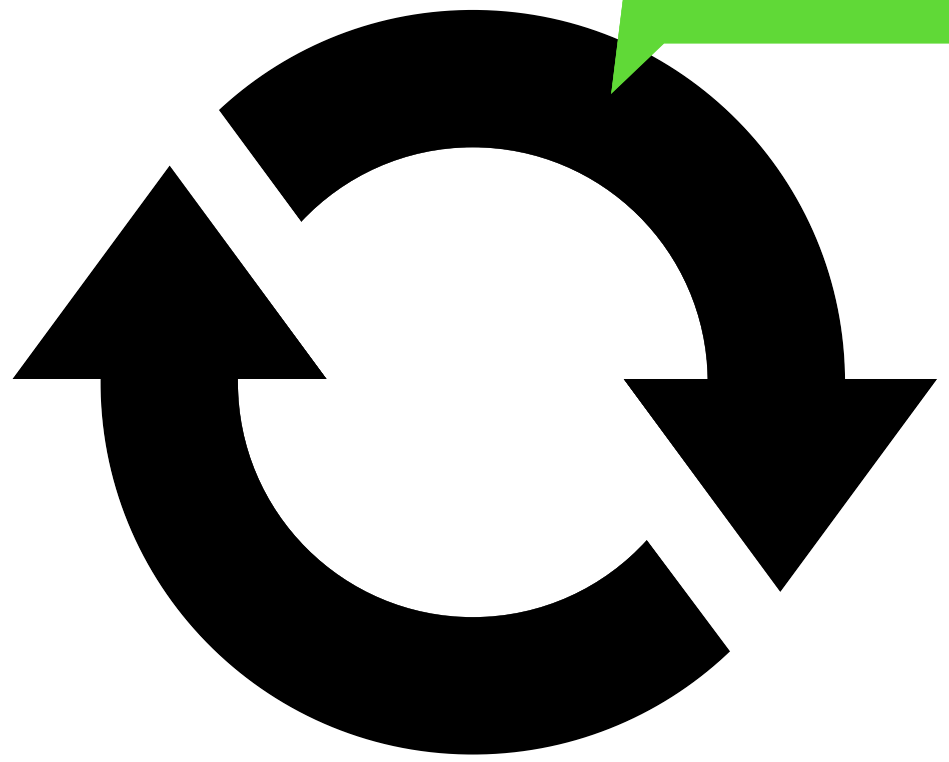
$\mathrm{return}(\mathrm{Main}) \mapsto \mathrm{Int}$



Approach to incrementalisation revisited

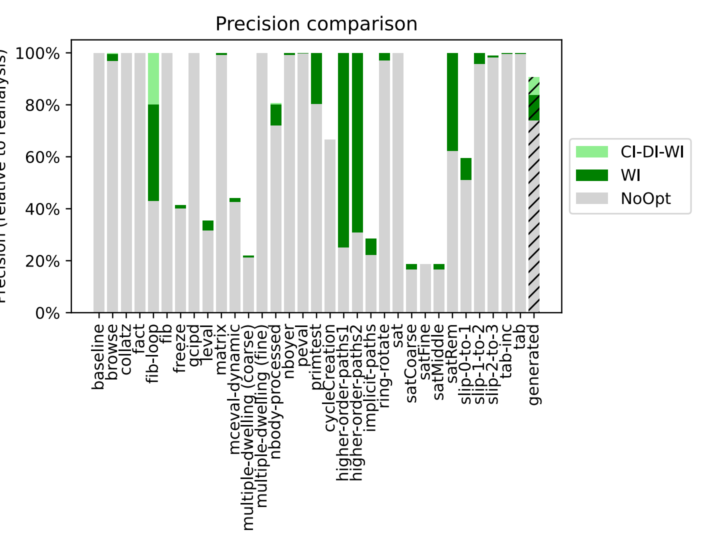
- **perform change impact analysis**
  from changes to source code modules to components in previous analysis results

- **until a new fixed point has been found**


dependency-driven
recompute
impacted component as usual
invalidate
compare results for component to previous version and remove outdated components, dependencies, store writes
can change values

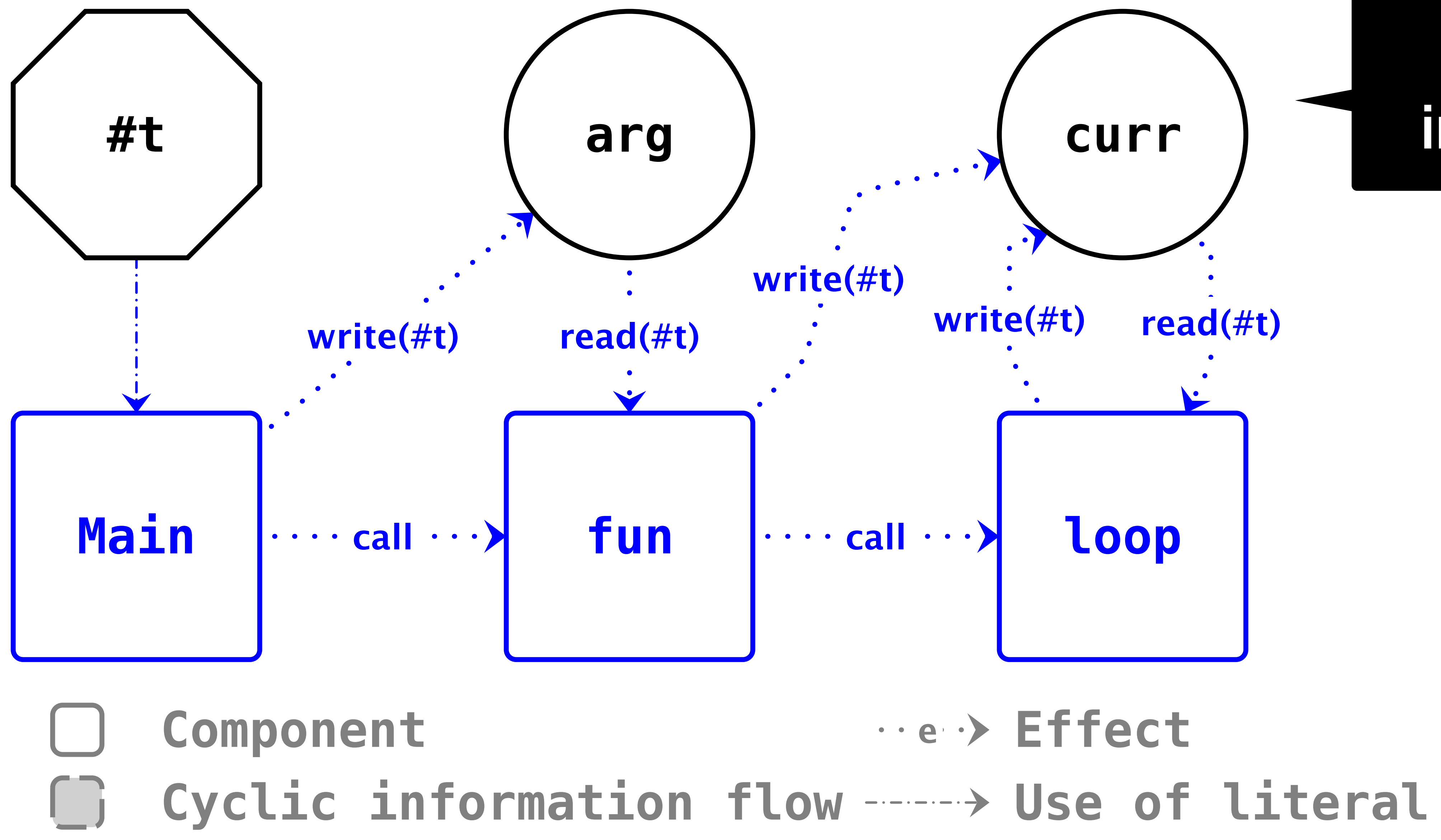
Van der Plas *et al.* (SCAM 2020) Incremental Flow Analysis through Computational Dependency Reification
Van der Plas *et al.* (VMCAI 2023) Result Invalidation for Incremental Modular Analyses

fast, but not yet fully precise



Problem: cyclically reinforced values

```
(letrec
  ((fun (lambda (arg)
          (letrec
            ((loop (lambda (curr)
                     (if curr
                         (loop curr)
                         "stop"))))
             (loop arg)))))
   (fun #t))
```

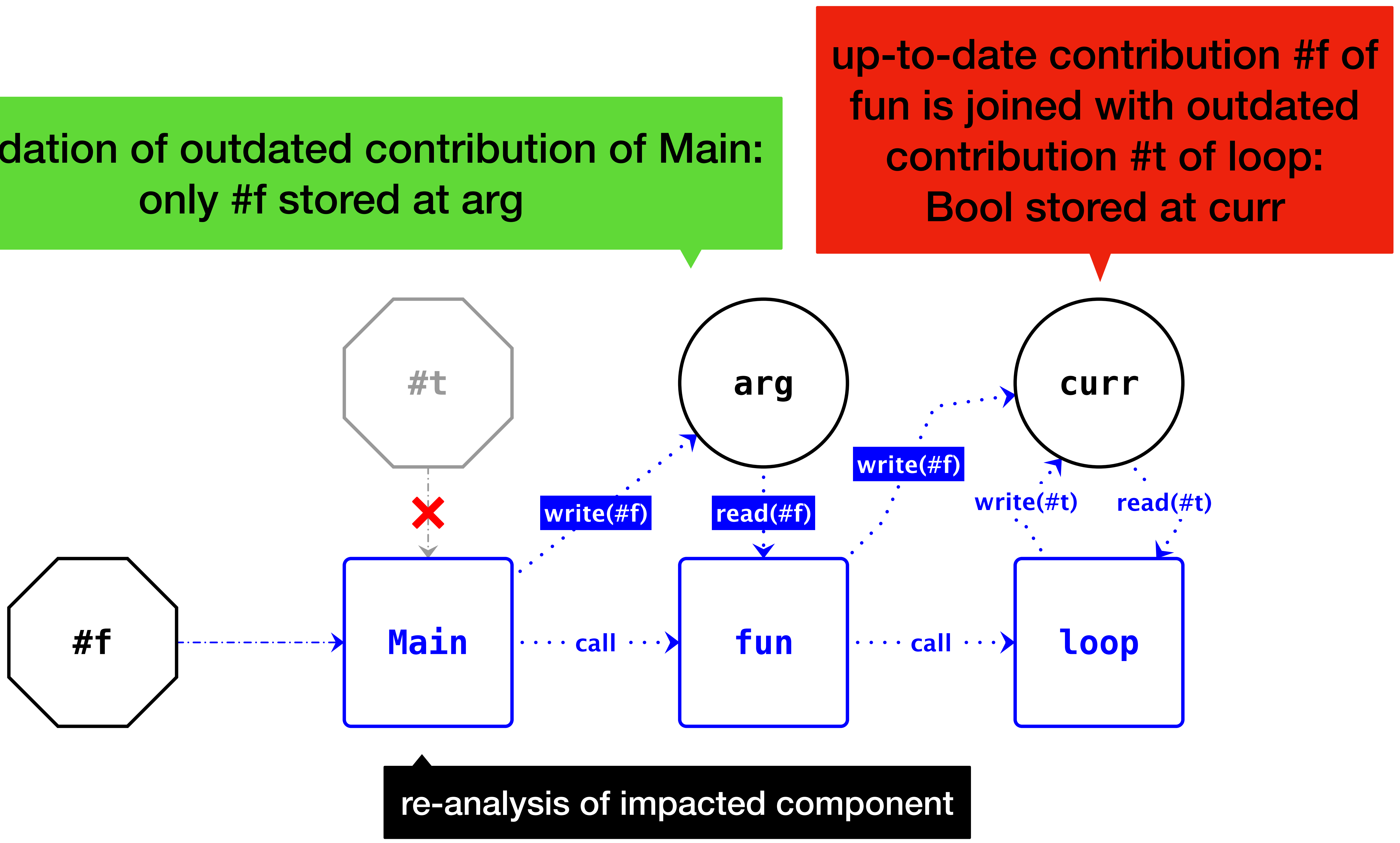


curr computed through data flow equations involving curr

Problem: prevents precise updates

```
(letrec
  ((fun (lambda (arg)
          (letrec
            ((loop (lambda (curr)
                     (if curr
                         (loop curr)
                         "stop"))))
             (loop arg)))))
   (fun (<change> #t #f)))
```

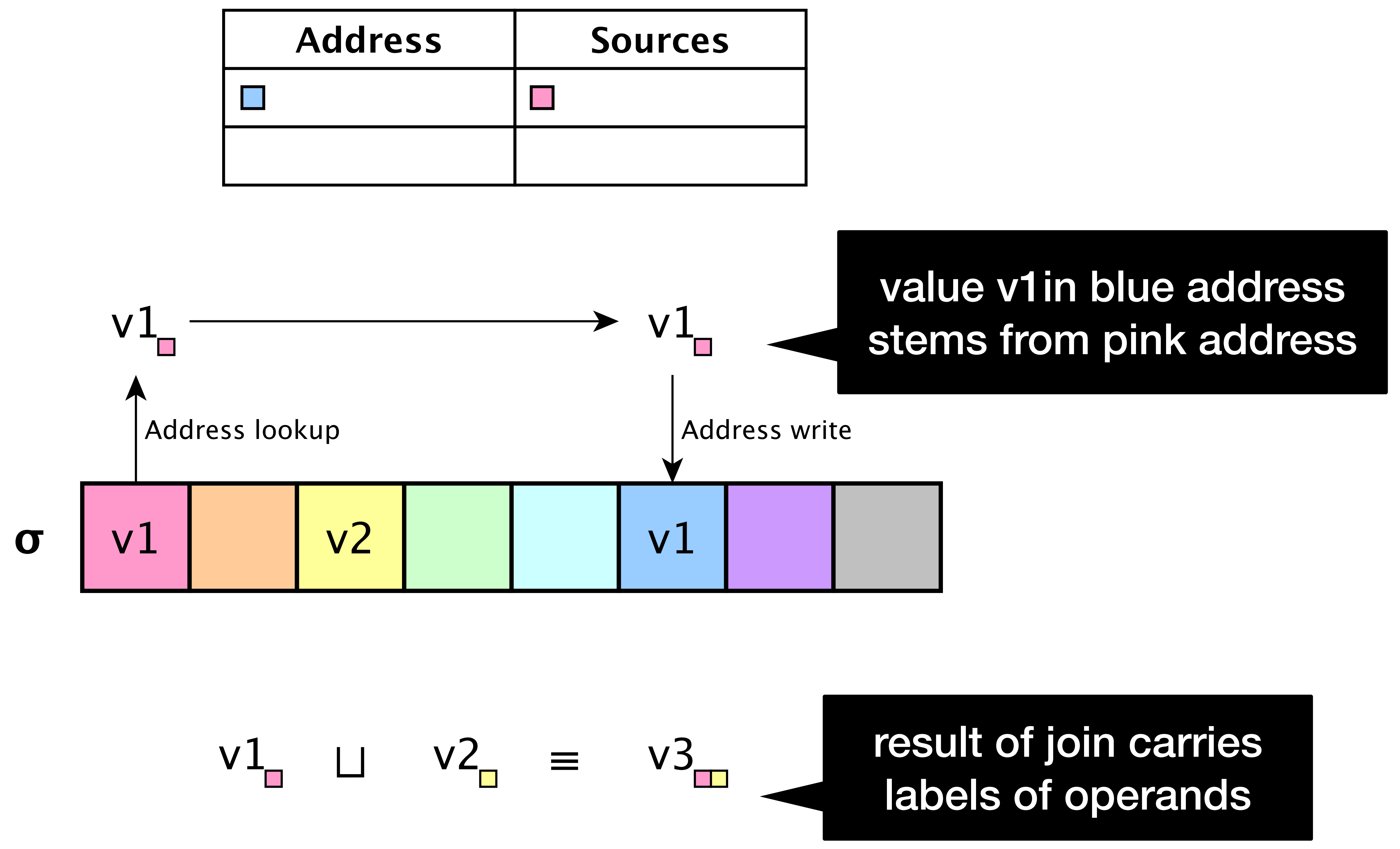
program change


invalidation of outdated contribution of Main: only #f stored at arg
up-to-date contribution #f of fun is joined with outdated contribution #t of loop: Bool stored at curr
#t
arg
curr
#f
Main
fun
loop
call
call
write(#f)
read(#f)
write(#f)
write(#t)
read(#t)
re-analysis of impacted component

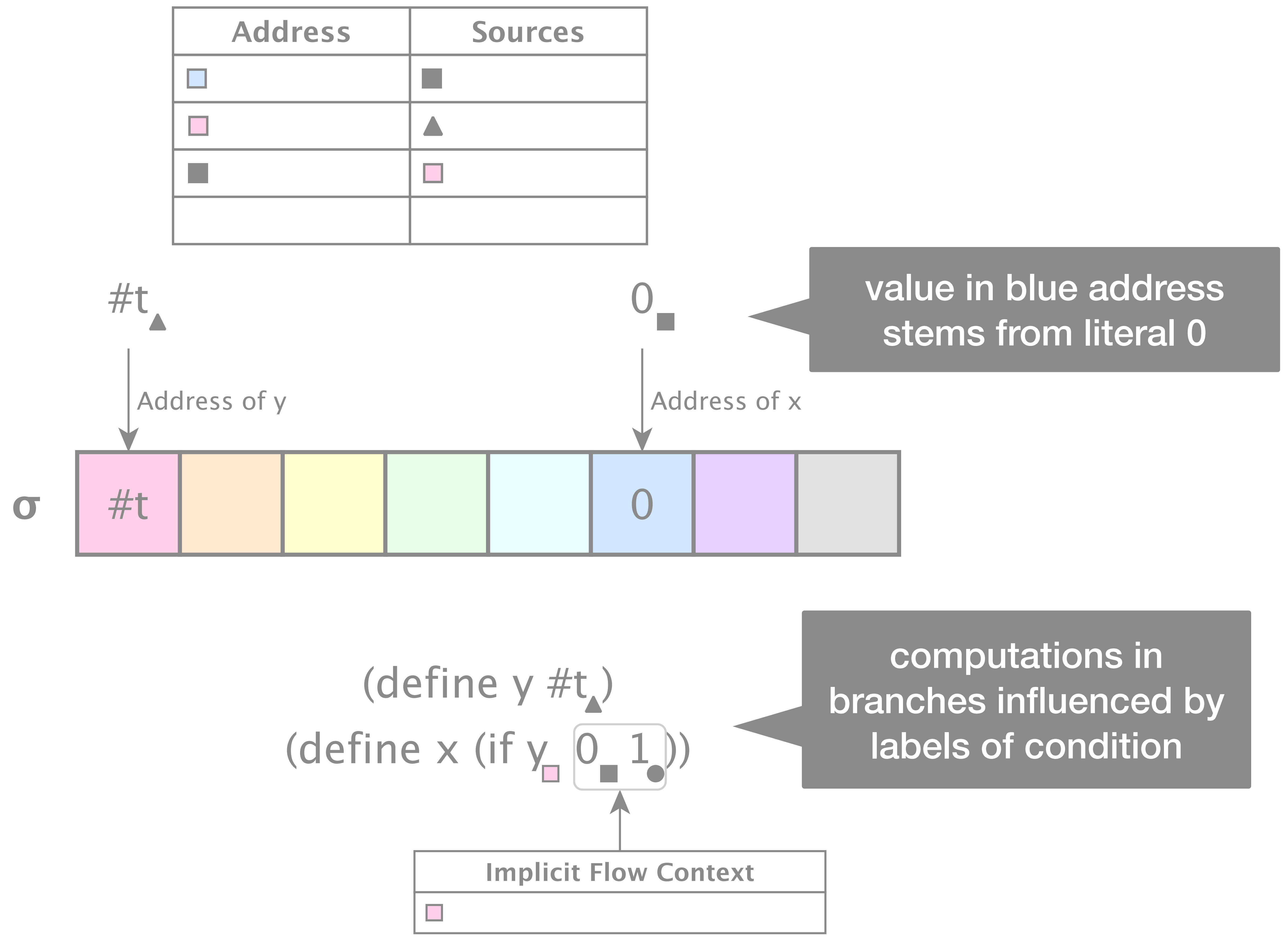
cylic reinforcement of values

the old value of `curr` from the previous program version influences the computation of the new one, leading to precision loss

Solution: track information flow between addresses


Address
Sources
v1
v1
Address lookup
Address write
σ
v1
v2
v1
value v1in blue address stems from pink address
v1 ⊔ v2 ≡ v3
result of join carries labels of operands

explicit flow

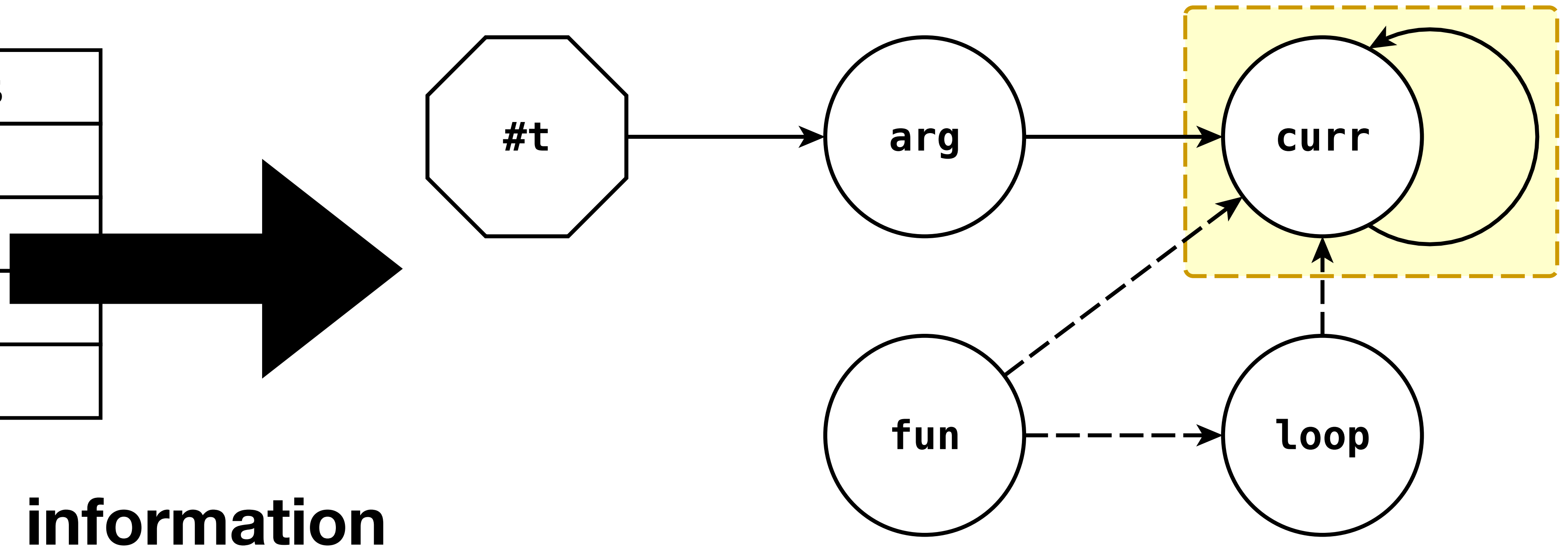

Address
Sources
#t
0
Address of y
Address of x
σ
#t
0
value in blue address stems from literal 0
(define y #t)
(define x (if y 0 1))
computations in branches influenced by labels of condition
Implicit Flow Context

implicit flow and literals

lightweight: label values with source address on store reads, propagate the labels, and extract them upon store writes

Solution: identify cycles and “refine” values within

Address	Sources
🟦	■
🟪	▲
■	🟪



detect SCCs

information flow graph

refinement condition

- external incoming value is updated non-monotonically
- **SCC is partially broken or a value is no longer flowing to it**
- a literal value is no longer used

drastic, but sound

set all addresses to ⊥, and trigger re-analysis

Evaluation

Implemented in MAF framework (Scala implementation, Scheme programs)

• 33 curated refactoring-like changes to benchmark programs	13
• 950 generated versions of benchmark programs	163

# with cycles

RQ1: Precision
Does the result of an incremental update with cycle invalidation always match the result of a full reanalysis?

RQ2: Performance
How does an incremental update with cycle invalidation perform compared to a full reanalysis of the updated program?

Evaluation: precision

Compare all values in the store to the store of a full reanalysis

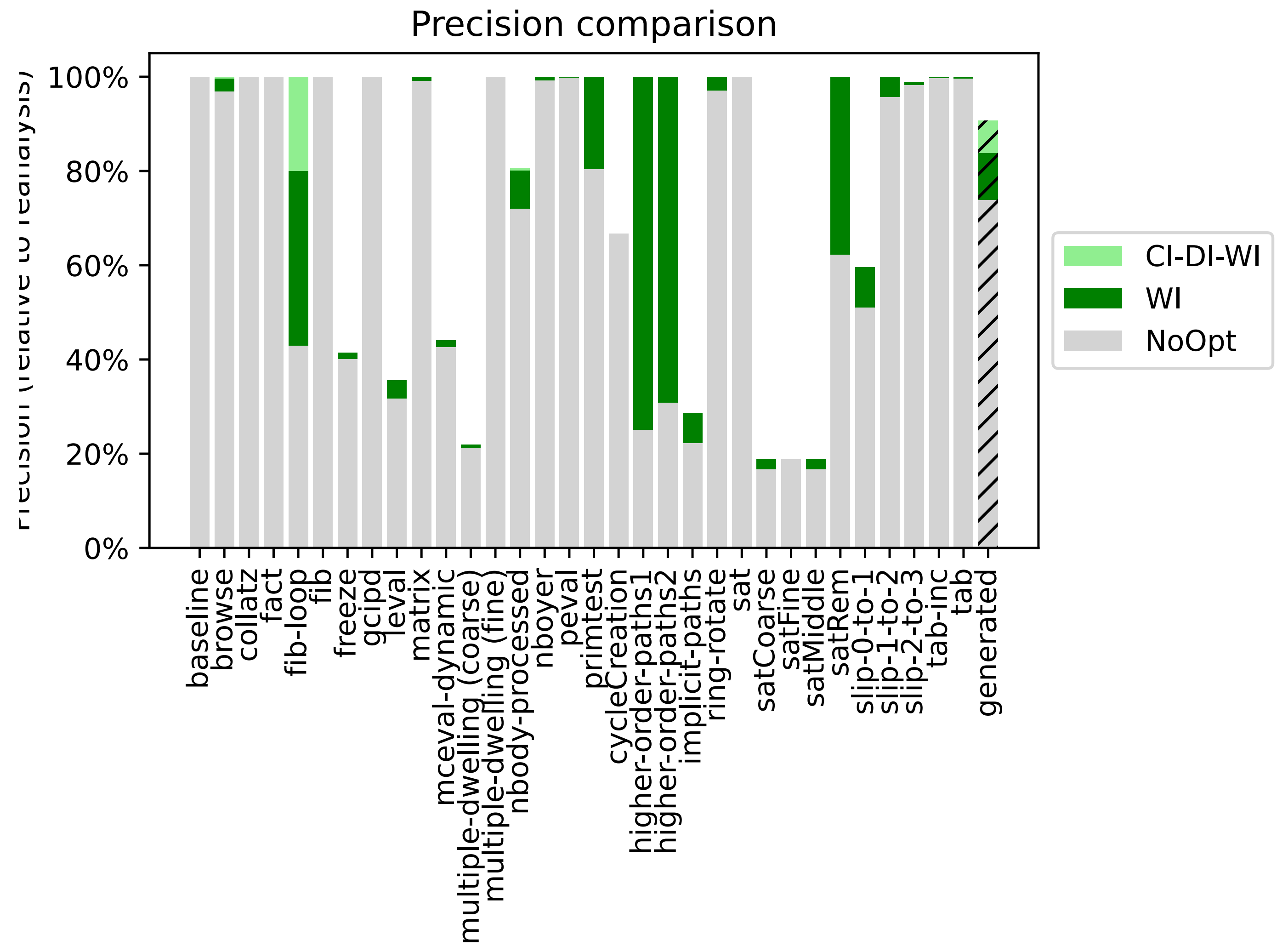
- $v_{inc} \sqsubset v_{rean} \rightarrow$ unsound
- $v_{inc} = v_{rean} \rightarrow$ fully precise
- $v_{rean} \sqsubset v_{inc} \rightarrow$ imprecise

➡ **No unsoundness**

➡ **Precise** (on all but one program,
missed one cycle: edge case)

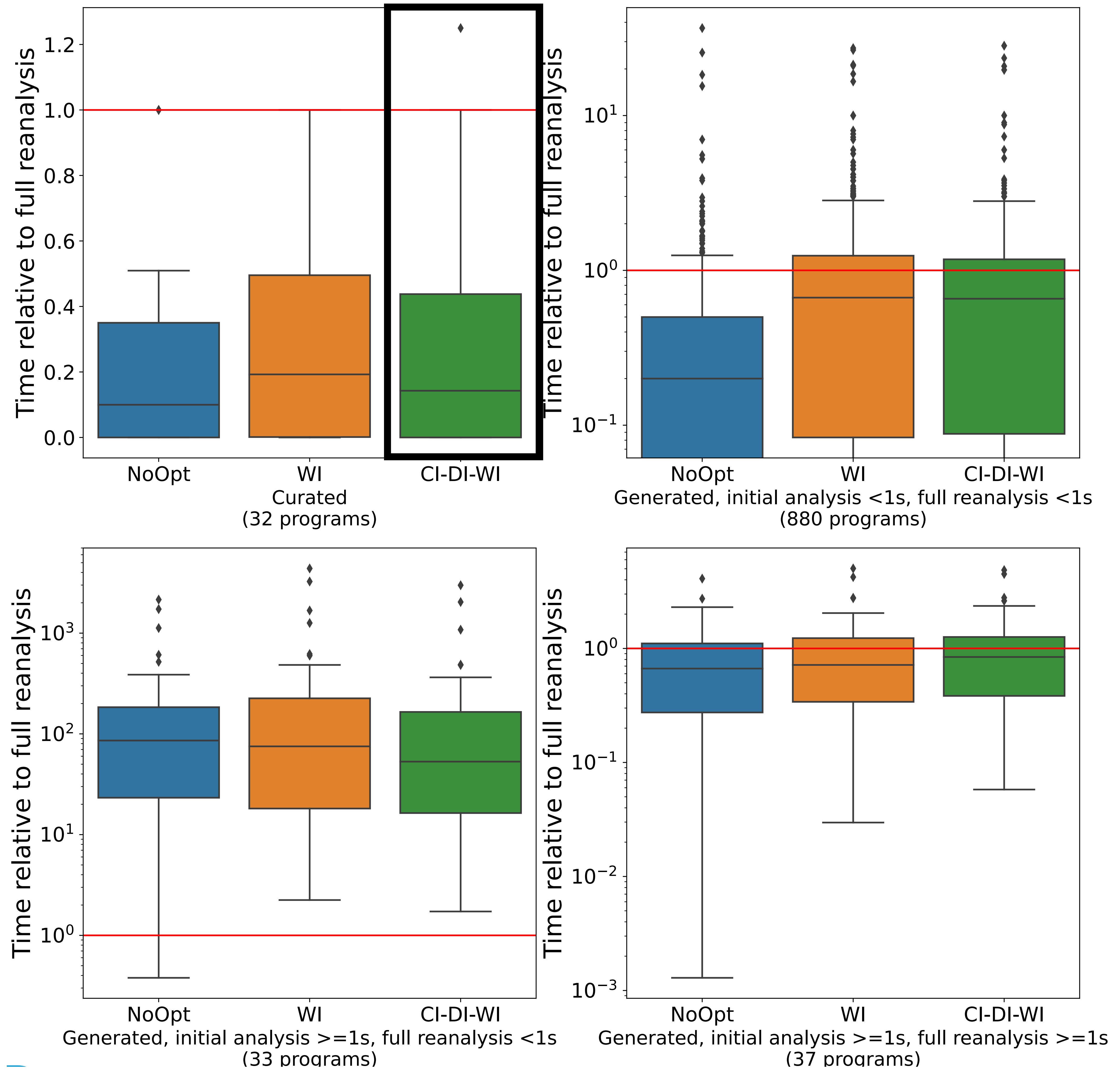
now all but one bar at 100%

VMCAI23



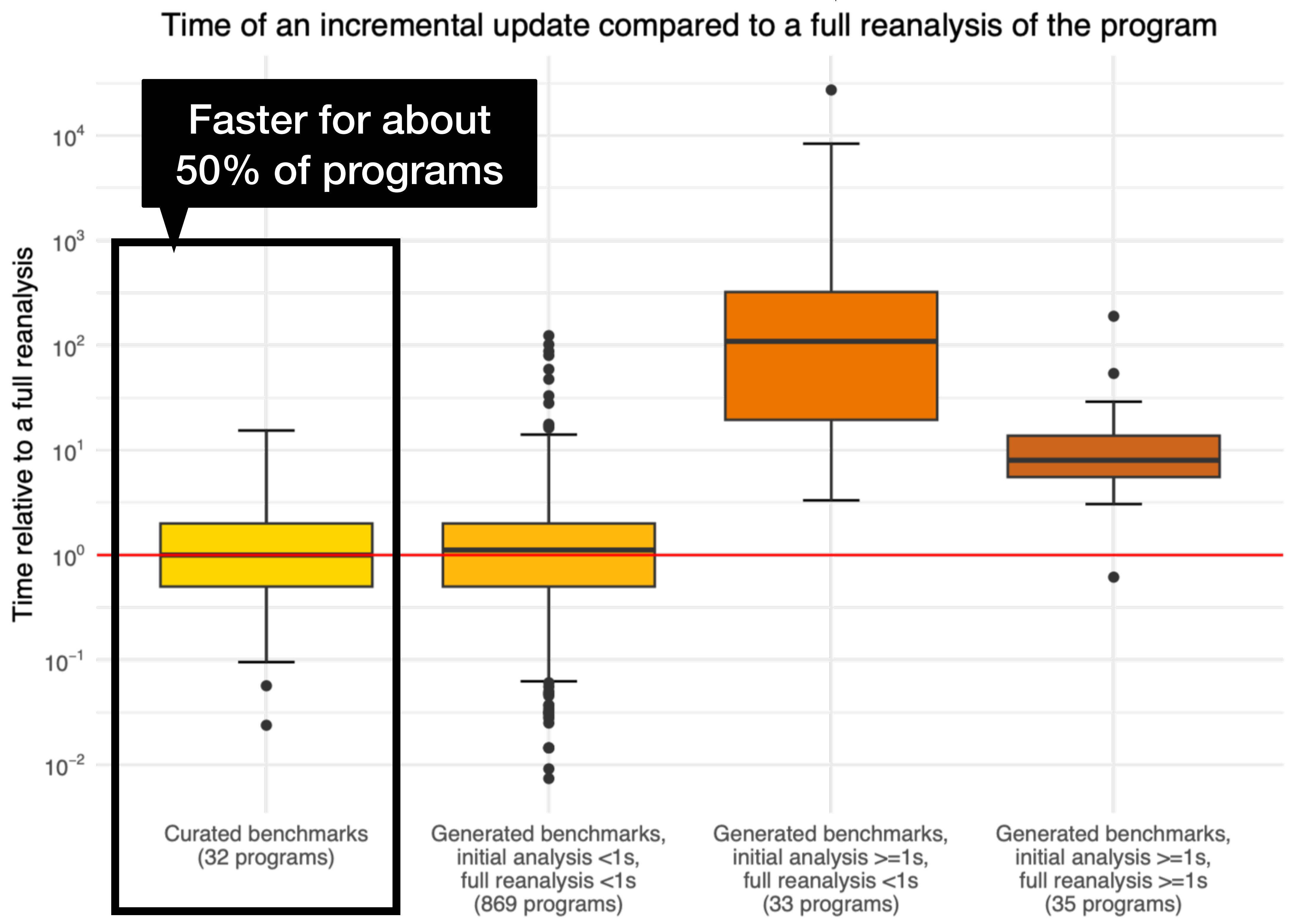
Evaluation: performance

Performance gain possible depends on size of impact of the program changes.



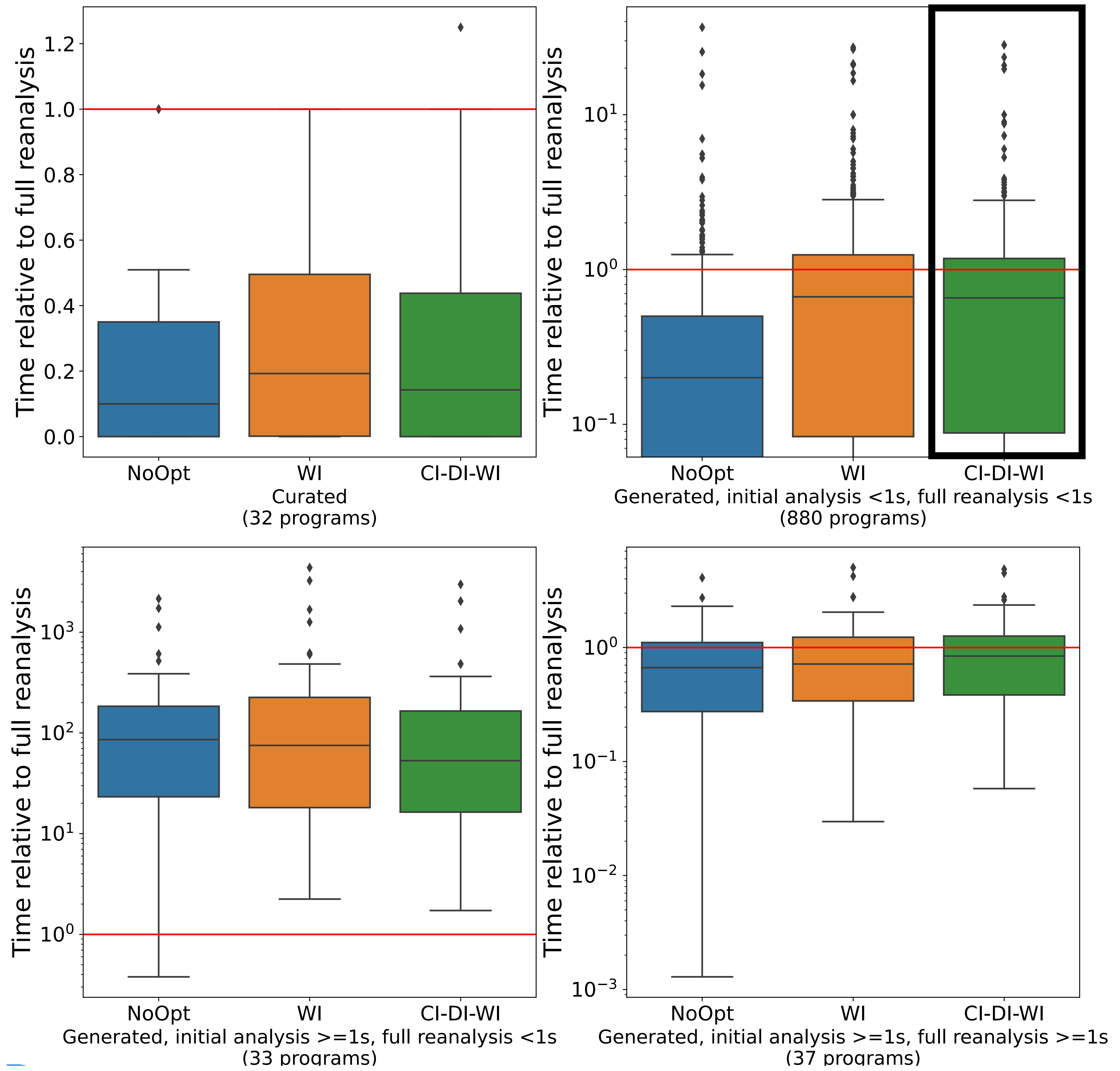
VMCAI23

And some of the generated changes are rather drastic
(e.g., removal of function calls).

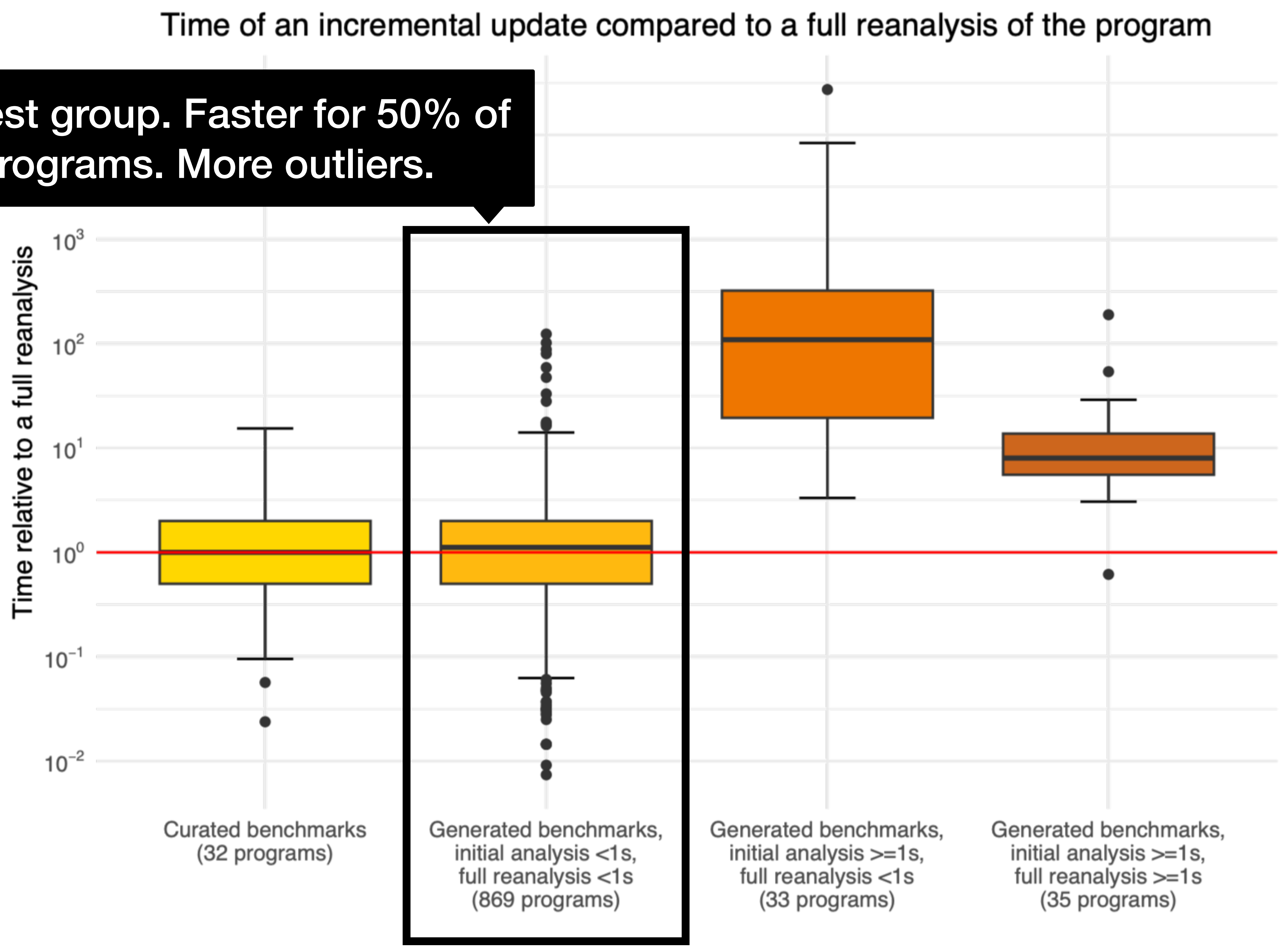


SCAM25

Evaluation: performance



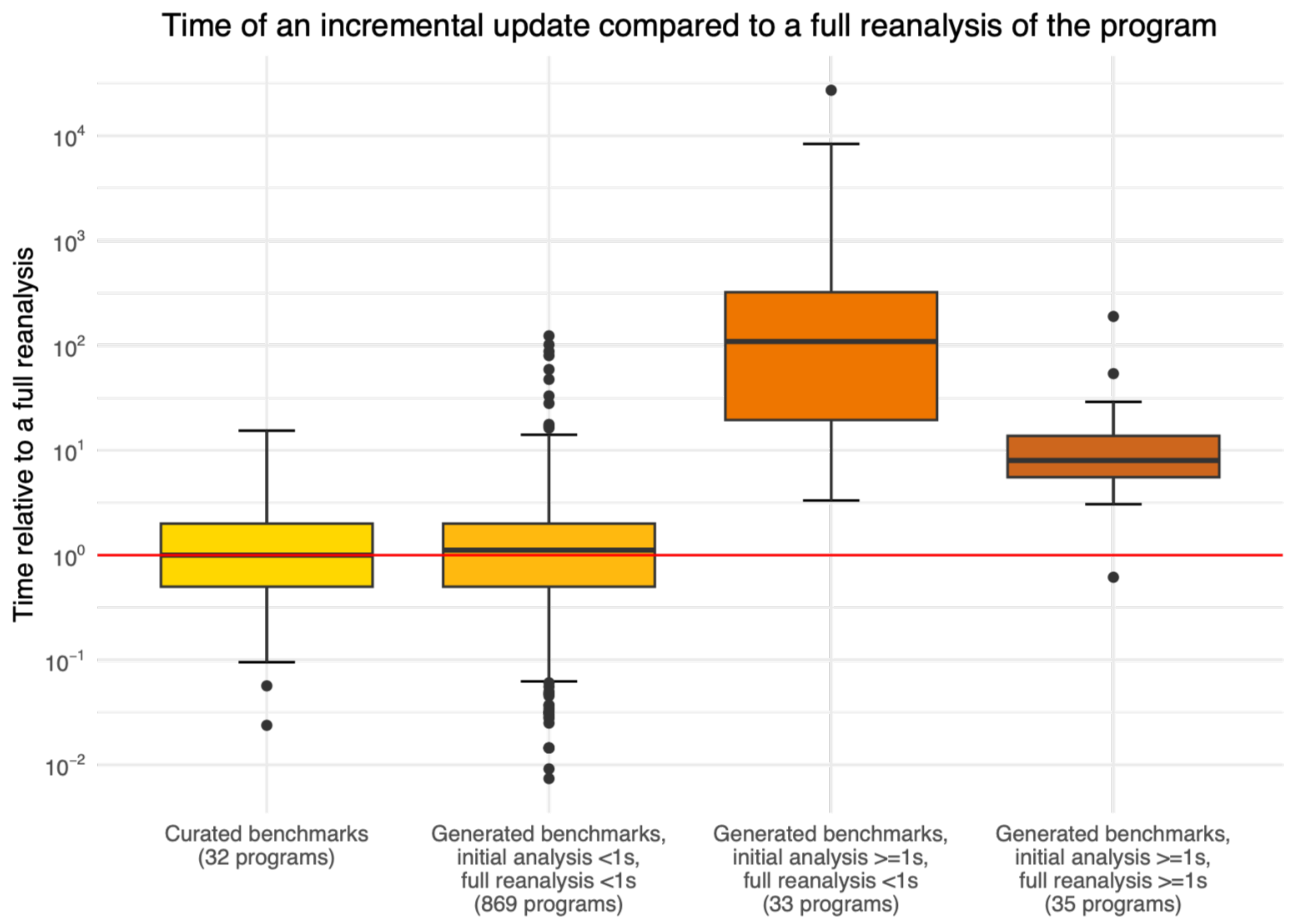
VMCAI23



Largest group. Faster for 50% of programs. More outliers.

SCAM25

Evaluation: performance



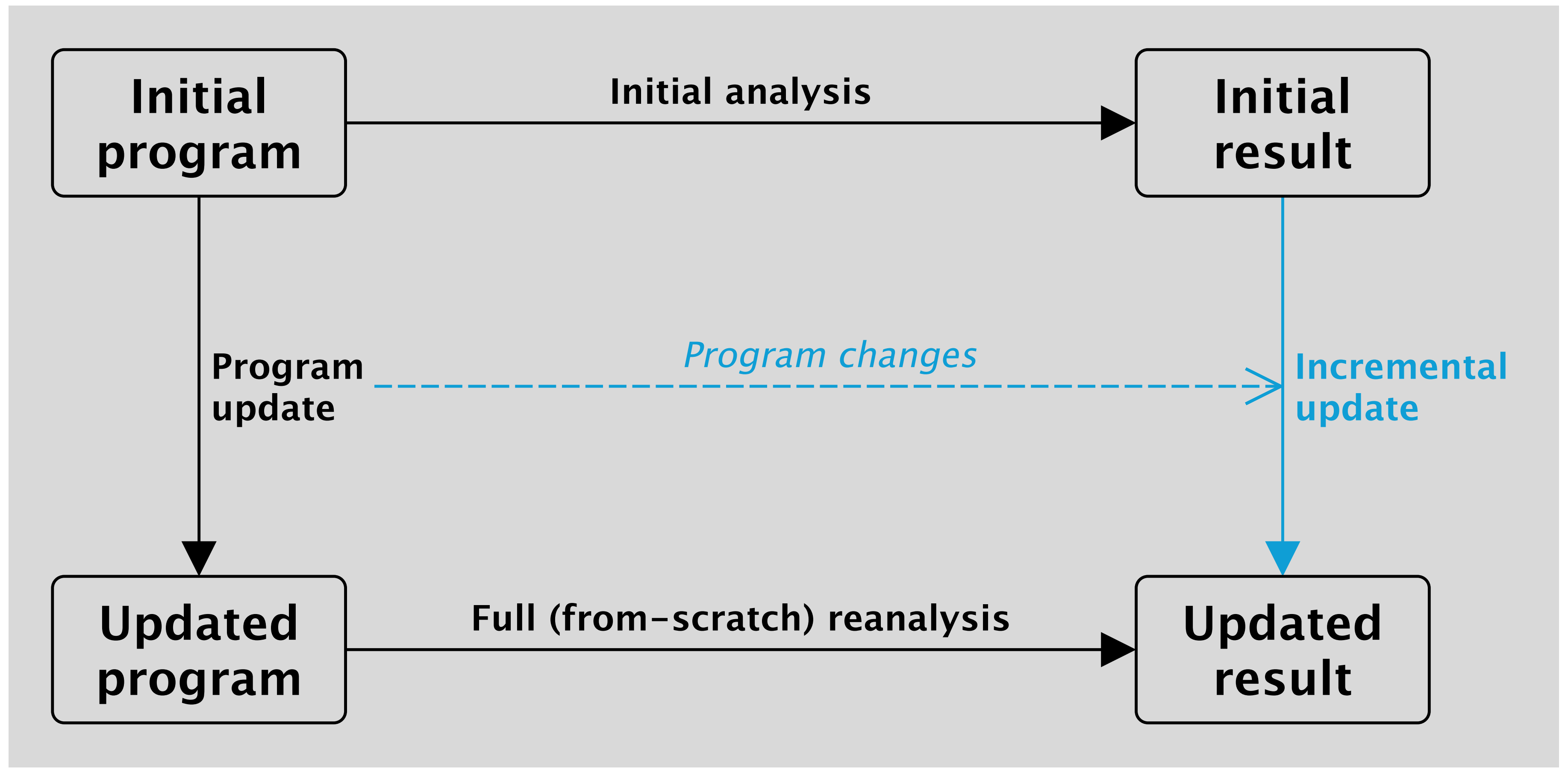
SCAM25

Overall: **performance hit** ↔ VMCAI23

- plenty of optimisation opportunities, as we focused on regaining precision:
  - optimise data structures and algorithms
  - run cycle detection & invalidation less often than after every component analysis
  - invalidate less aggressively so some addresses in cycle retain their value
  - heuristics to determine when to analyse from scratch and when to analyse incrementally
  - …
- but **results** now **as precise** as a full re-analysis

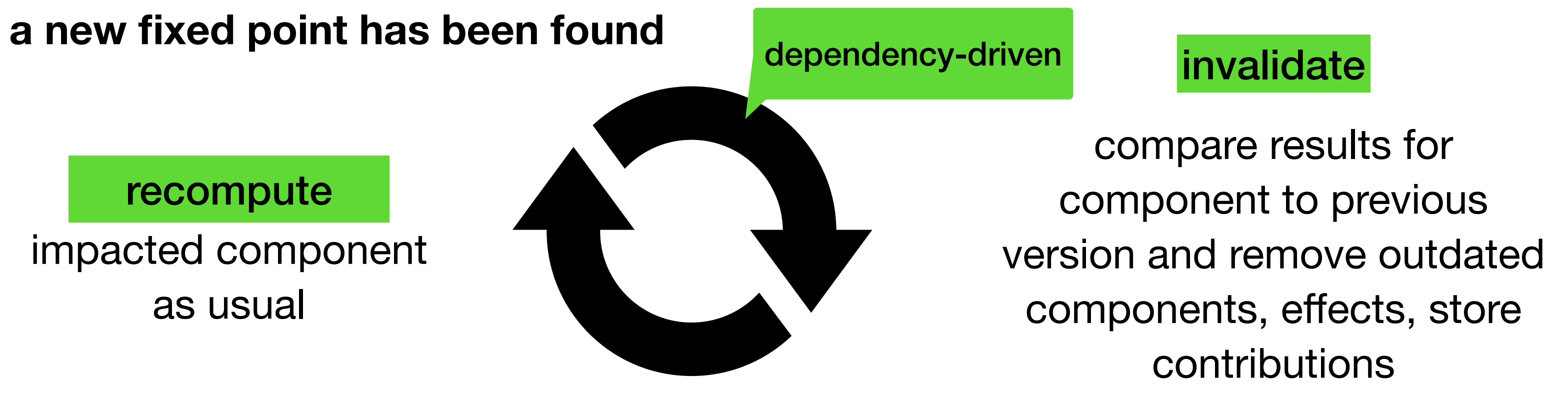
Conclusion

Incrementalisation to the rescue



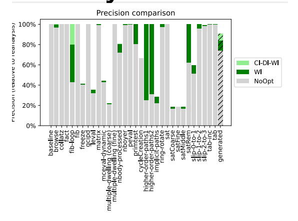
Approach to incrementalisation revisited

- **perform change impact analysis**
  from AST changes to previous analysis results
- **until a new fixed point has been found**


dependency-driven
recompute
impacted component as usual
invalidate
compare results for component to previous version and remove outdated components, effects, store contributions

Van der Plas *et al.* (SCAM 2020) Incremental Flow Analysis through Computational Dependency Reification
Van der Plas *et al.* (VMCAI 2023) Result Invalidation for Incremental Modular Analyses

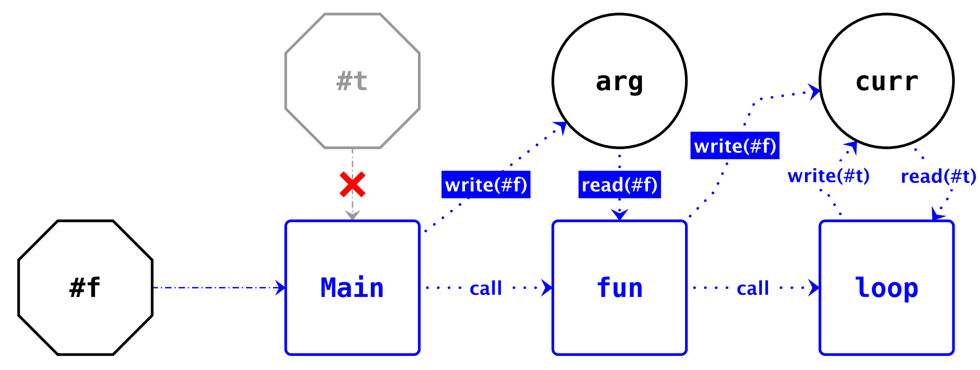
fast, but not yet fully precise



Problem: prevents precise updates

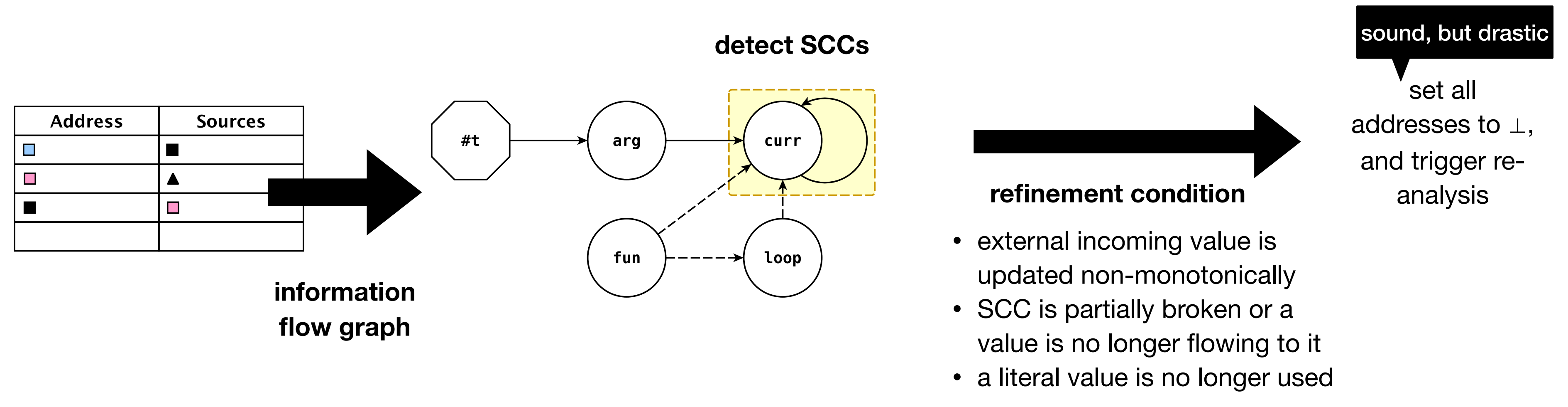
```
(letrec
  ((fun (lambda (arg)
          (letrec
            ((loop (lambda (curr)
                       (if curr
                           (loop curr)
                           "stop"))))
             (loop arg)))))
   (fun (<change> #t #f)))
```

program change


invalidation of outdated contribution of Main: only #f stored at arg
up-to-date contribution of fun is joined with outdated contribution of loop: Bool stored at curr
#t
arg
curr
#f
Main
fun
loop
write(#f)
read(#f)
write(#f)
write(#t)
read(#t)
call
call
re-analysis of impacted component

cylic reinforcement of lattice values
value of `curr` is used in its own computation,
meaning that the old value of `curr` influences the computation of the new one, leading to precision loss

Solution: identify cycles and “refine” values within



- external incoming value is updated non-monotonically
- SCC is partially broken or a value is no longer flowing to it
- a literal value is no longer used

Preprint link