Incremental whole-program analysis in Datalog with lattices

Tamás Szabó, Sebastian Erdweg, Gábor Bergmann

Jaehoon Jang

2024.04.30

IS661

Background

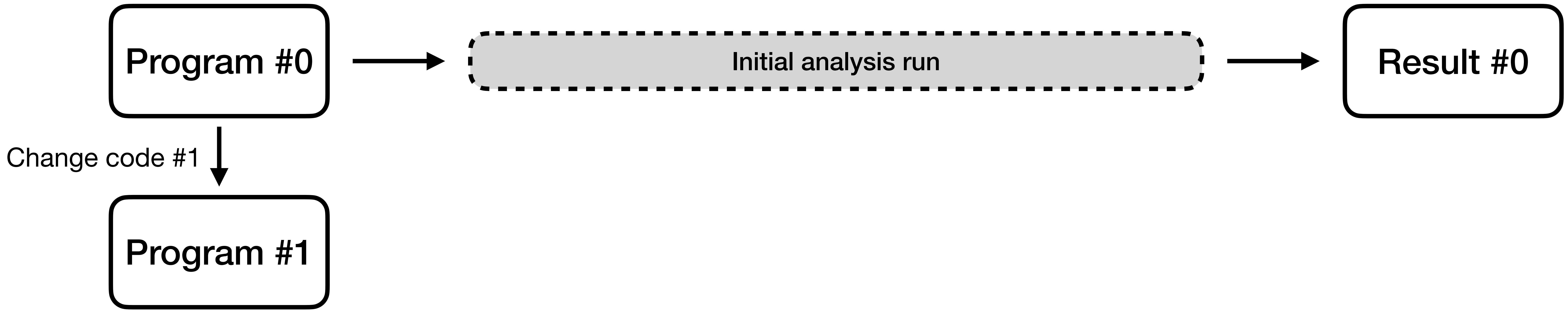
- Incremental analysis provides analysis results for the modified code part
  - not the entire code.
- Whenever the code is modified
  - It is inefficient to perform an analysis for the whole program
- Incremental analysis is efficient by performing analysis only on the corrected part
- This quick feedback allows developers to check the analysis results immediately

Background

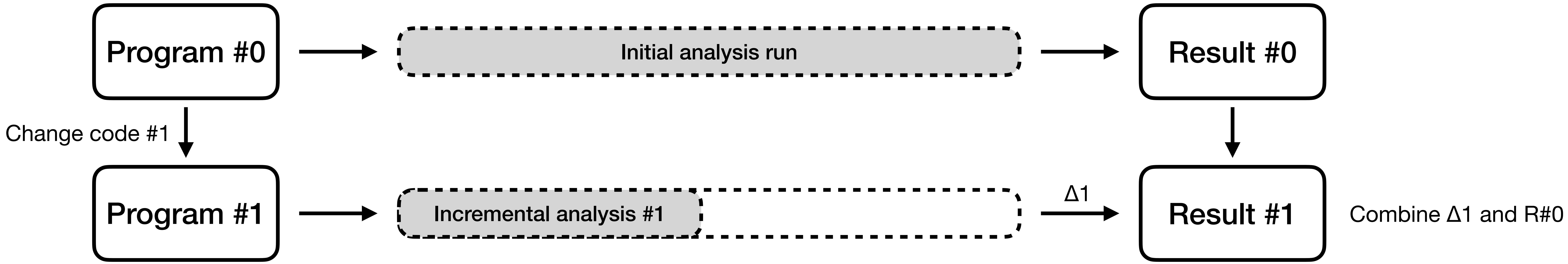
Program #0

Background

Background

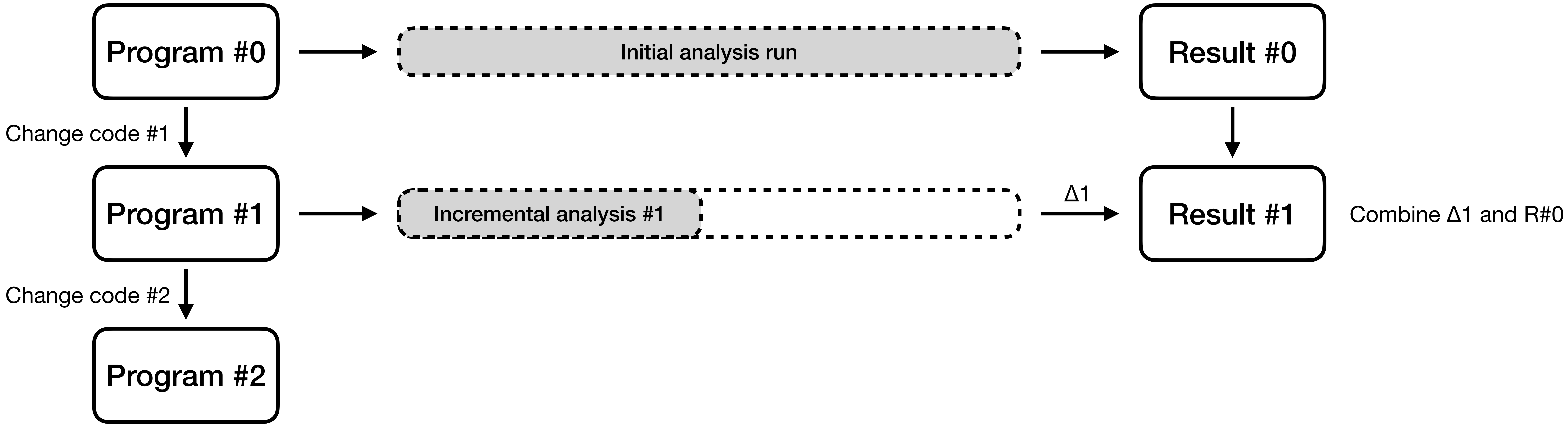

Program #0
Initial analysis run
Result #0
Change code #1
Program #1

Background



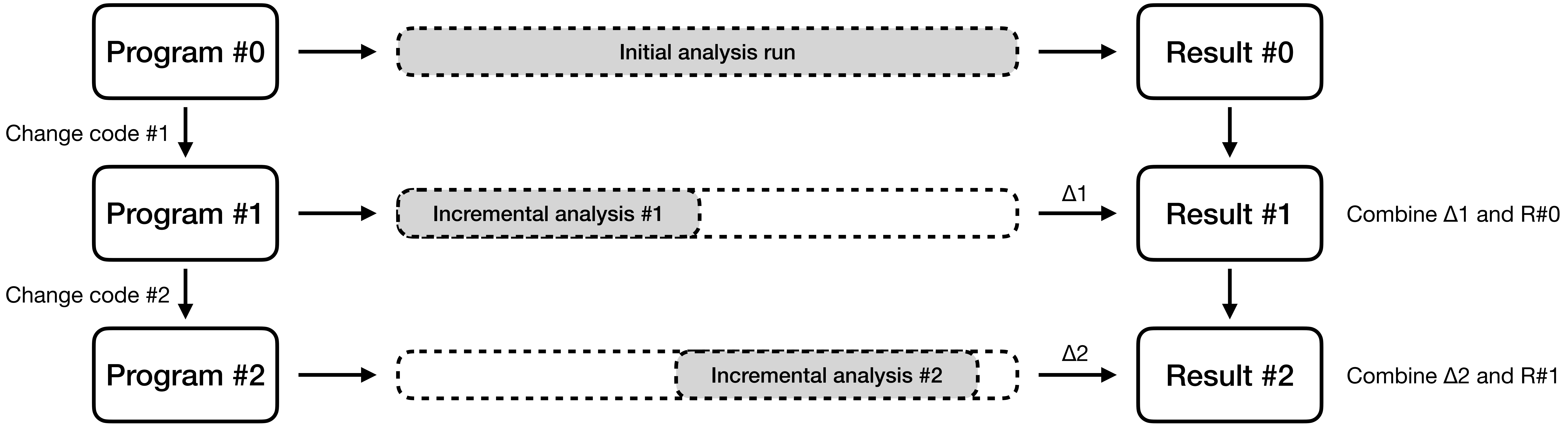
Program #0
Initial analysis run
Result #0
Change code #1
Program #1
Incremental analysis #1
Δ1
Result #1
Combine Δ1 and R#0

Background



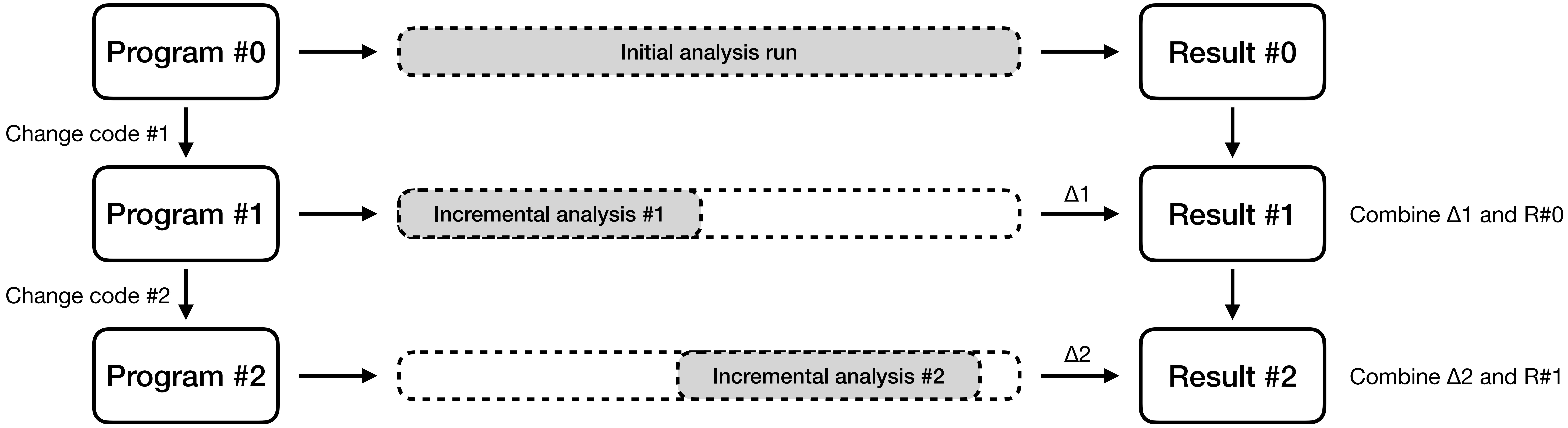
Program #0
Initial analysis run
Result #0
Change code #1
Program #1
Incremental analysis #1
Δ1
Result #1
Combine Δ1 and R#0
Change code #2
Program #2

Background



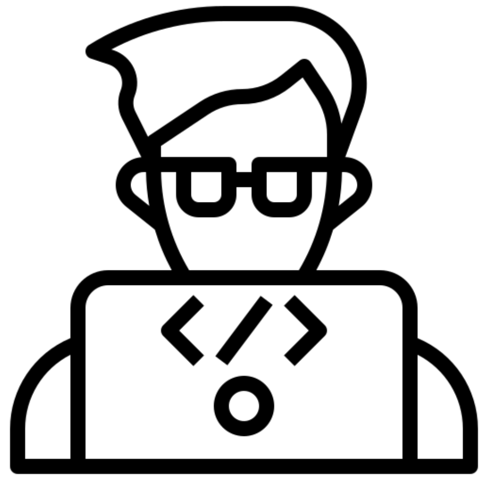
Program #0
Initial analysis run
Result #0
Change code #1
Program #1
Incremental analysis #1
Δ1
Result #1
Combine Δ1 and R#0
Change code #2
Program #2
Incremental analysis #2
Δ2
Result #2
Combine Δ2 and R#1

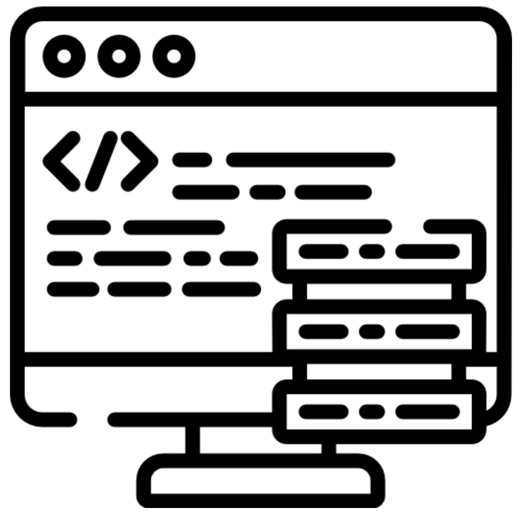
Background

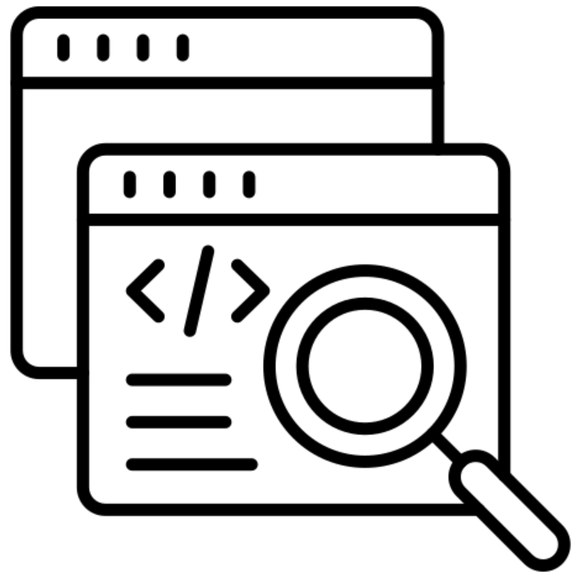


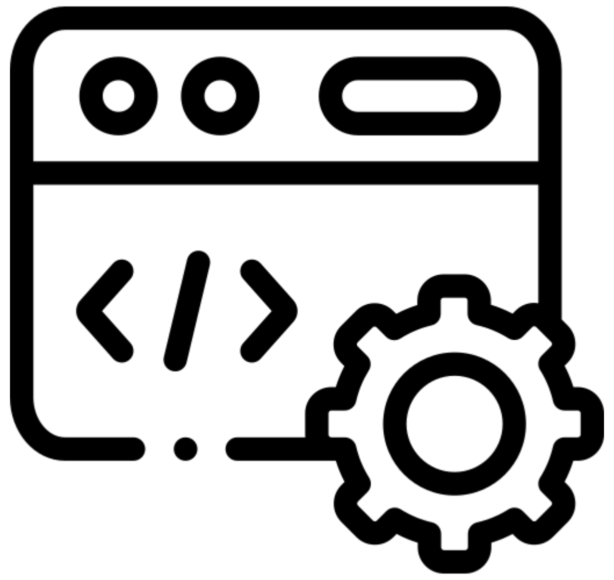
It is faster than when analyzing the whole program.

Motivation

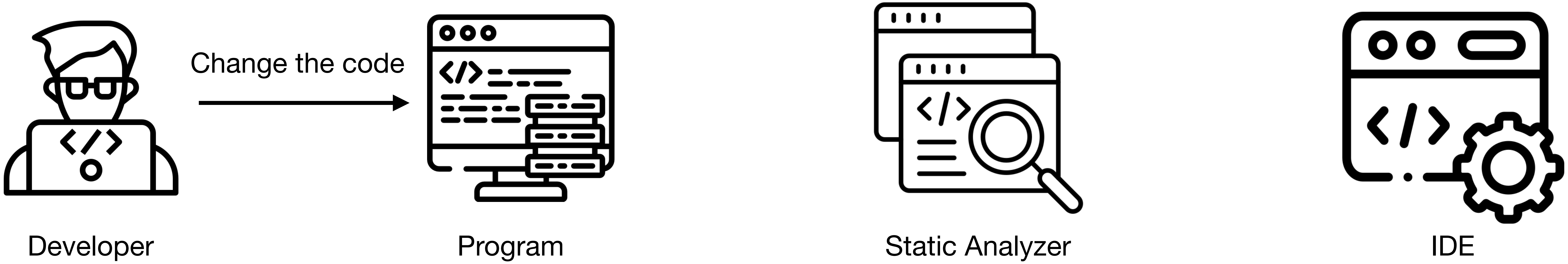

Developer


Program

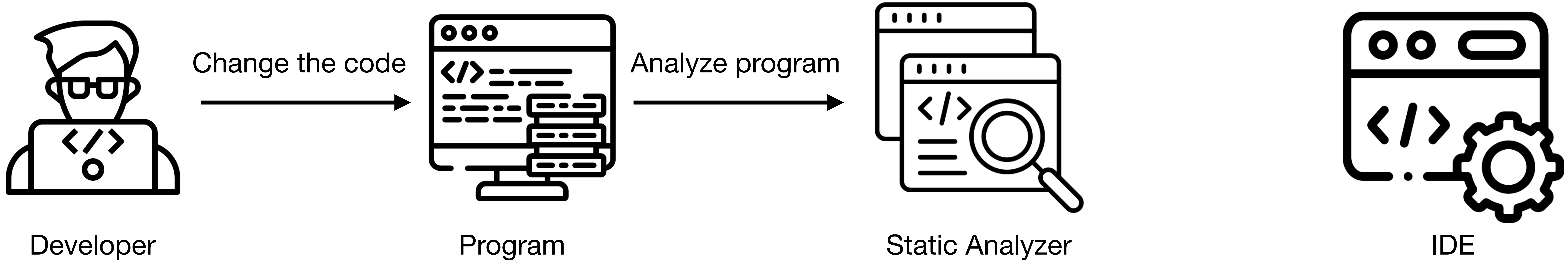

Static Analyzer


IDE

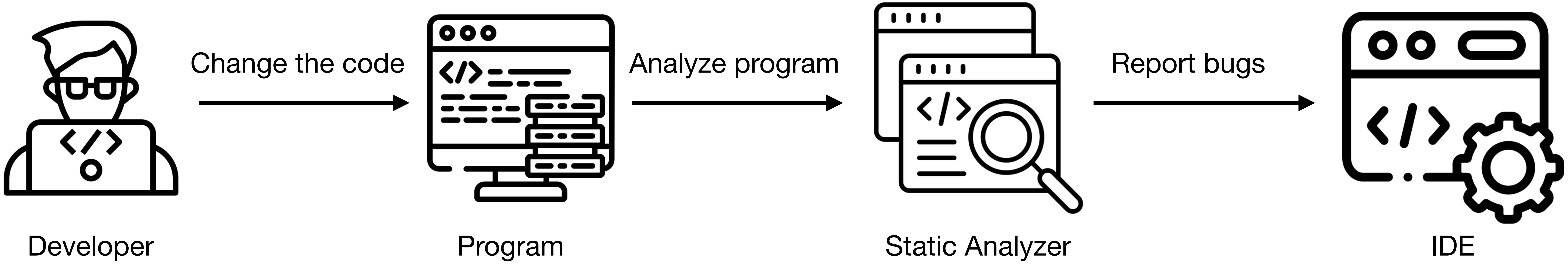
Motivation



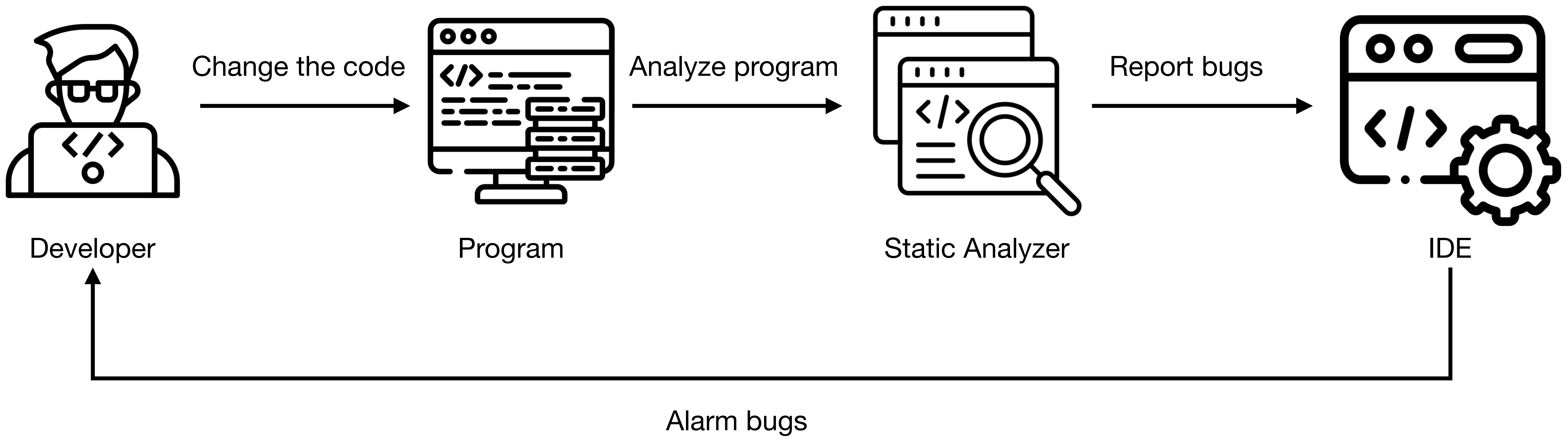
Motivation


Change the code
Analyze program
Developer
Program
Static Analyzer
IDE

Motivation

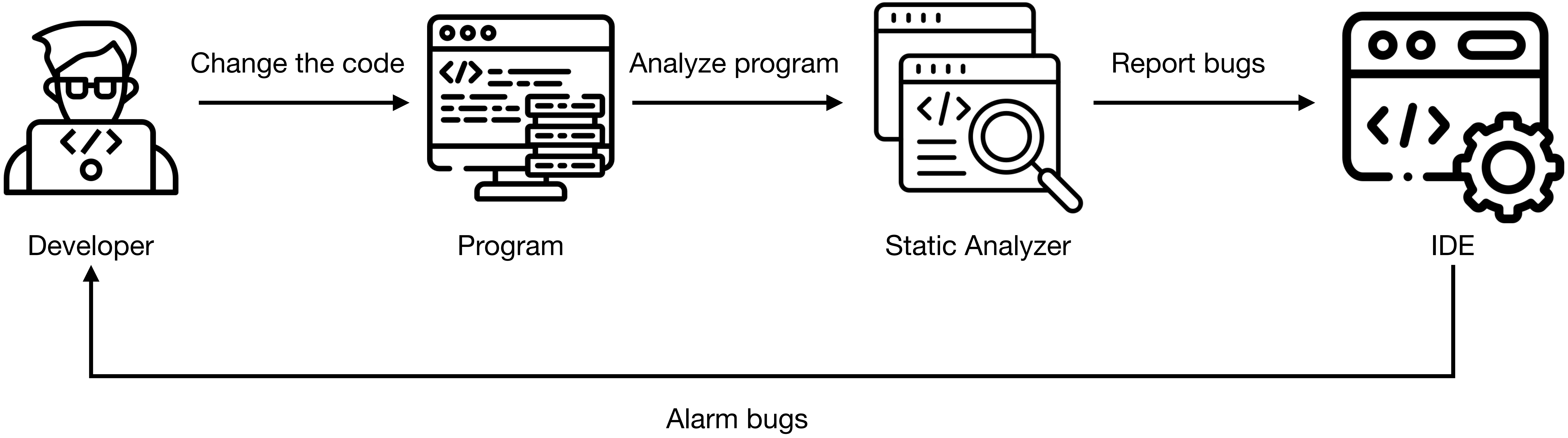

Change the code
Analyze program
Report bugs
Developer
Program
Static Analyzer
IDE

Motivation



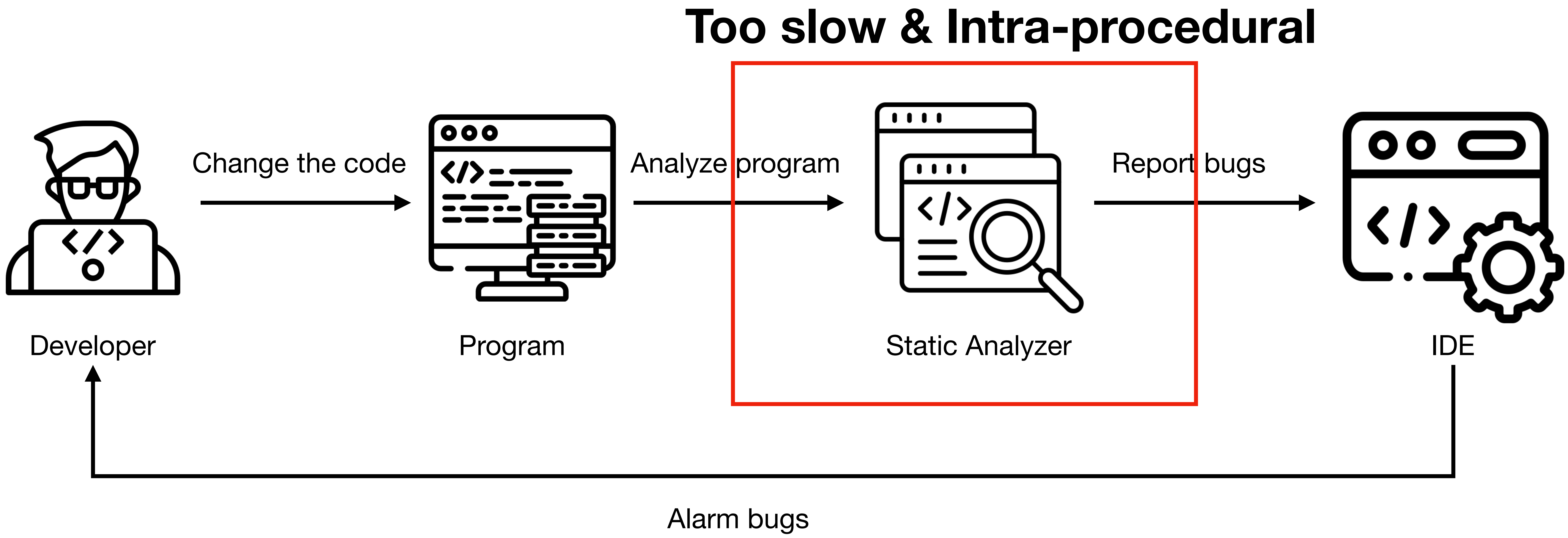
Change the code
Analyze program
Report bugs
Developer
Program
Static Analyzer
IDE
Alarm bugs

Motivation



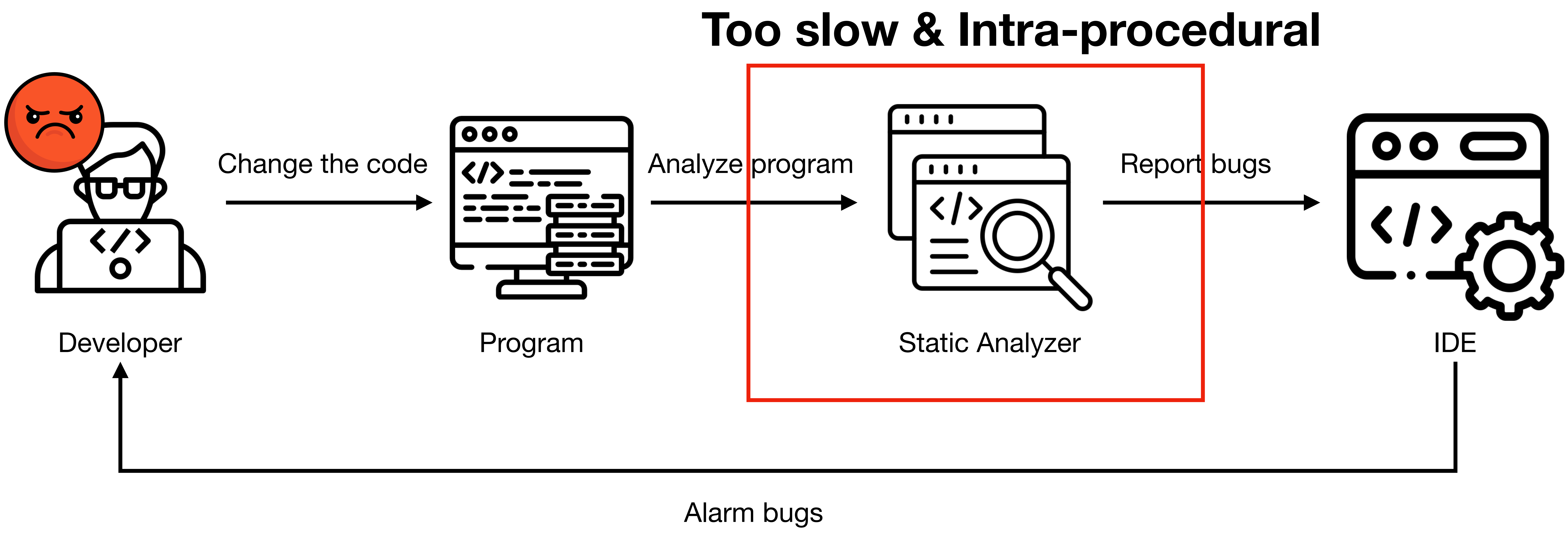
Should not break the development flow

Motivation



Should not break the development flow

Motivation



Should not break the development flow

Current Problem

- Previous works are only considered for **intra-procedural** analysis

Current Problem

- Previous works are only considered for **intra-procedural** analysis
- When the existing incremental analysis is performed for **inter-procedural,**
  - It showed bad performance

Current Problem

- Previous works are only considered for **intra-procedural** analysis
- When the existing incremental analysis is performed for **inter-procedural,**
  - It showed bad performance
- Incremental analysis results are
  - Up to **22 sec**, average **9 sec**
  - Too slow to analyze while developing code.

Current Problem

- Previous works are only considered for **intra-procedural** analysis
- When the existing incremental analysis performed for **inter-procedural**
  - It showed bad performance

They propose a new incremental Datalog solver called **Laddder**

- Incremental analysis results are
  - Up to **22 sec**, average **9 sec**
  - Too slow to analyze while developing code.

Laddder

- Ladder can perform **inter-procedural**
  - incremental analysis for the **whole program**.

Laddder

- Ladder can perform **inter-procedural**
  - incremental analysis for the **whole program**.
- Ladder solves existing problems with
  - **Differential Dataflow** and **Non-Standard Aggregation Semantics**.

Laddder

- Ladder can perform **inter-procedural**
  - incremental analysis for the **whole program**.
- Ladder solves existing problems with
  - **Differential Dataflow** and **Non-Standard Aggregation Semantics**.
- Ladder performs an incremental analysis
  - with less than **0.1 sec** in real-world Java code

Incrementalizability

- Our intuition is
  - **small changes** in the program have a **small impact** on the entire program
- If this intuition is incorrect, it is difficult to perform incremental analysis
  - for the whole program

Impact Measurement

- To show the intuition is true, they measure the impact
- Points-to analysis for Java source code (with Doop[1])
    - MiniJavac, antlr, etc
- "The **impact** of an input change is the number of output tuples that are deleted or inserted in the observable relations because of the change"
- "A computation can only be **incrementalizable** if the vast majority of small input changes have low impact."

[1] Doop - Framework for Java Pointer and Taint Analysis, https://github.com/KnowSciEng/doop

Impact Measurement

PT(a1, Alpha)
PT(b1, Beta)

```
Alpha a1 = new Alpha();
Beta b1 = new Beta();
```

Impact Measurement

PT(a1, Alpha)
PT(b1, Beta)

```
Alpha a1 = new Alpha();
Beta b1 = new Beta();
```

The number of output tuple is 2

Impact Measurement

PT(a1, Alpha)
PT(b1, Beta)

```
Alpha a1 = new Alpha();
Beta b1;
```

Impact Measurement

PT(a1, Alpha)
~~PT(b1, Beta)~~

```
Alpha a1 = new Alpha();
Beta b1;
```

The number of output tuple is 1

Impact Measurement

PT(a1, Alpha)
~~PT(b1, Beta)~~

```
Alpha a1 = new Alpha();
Beta b1;
```

The number of output tuple is 1

↓

The impacted tuple is **1**

Impact Measurement

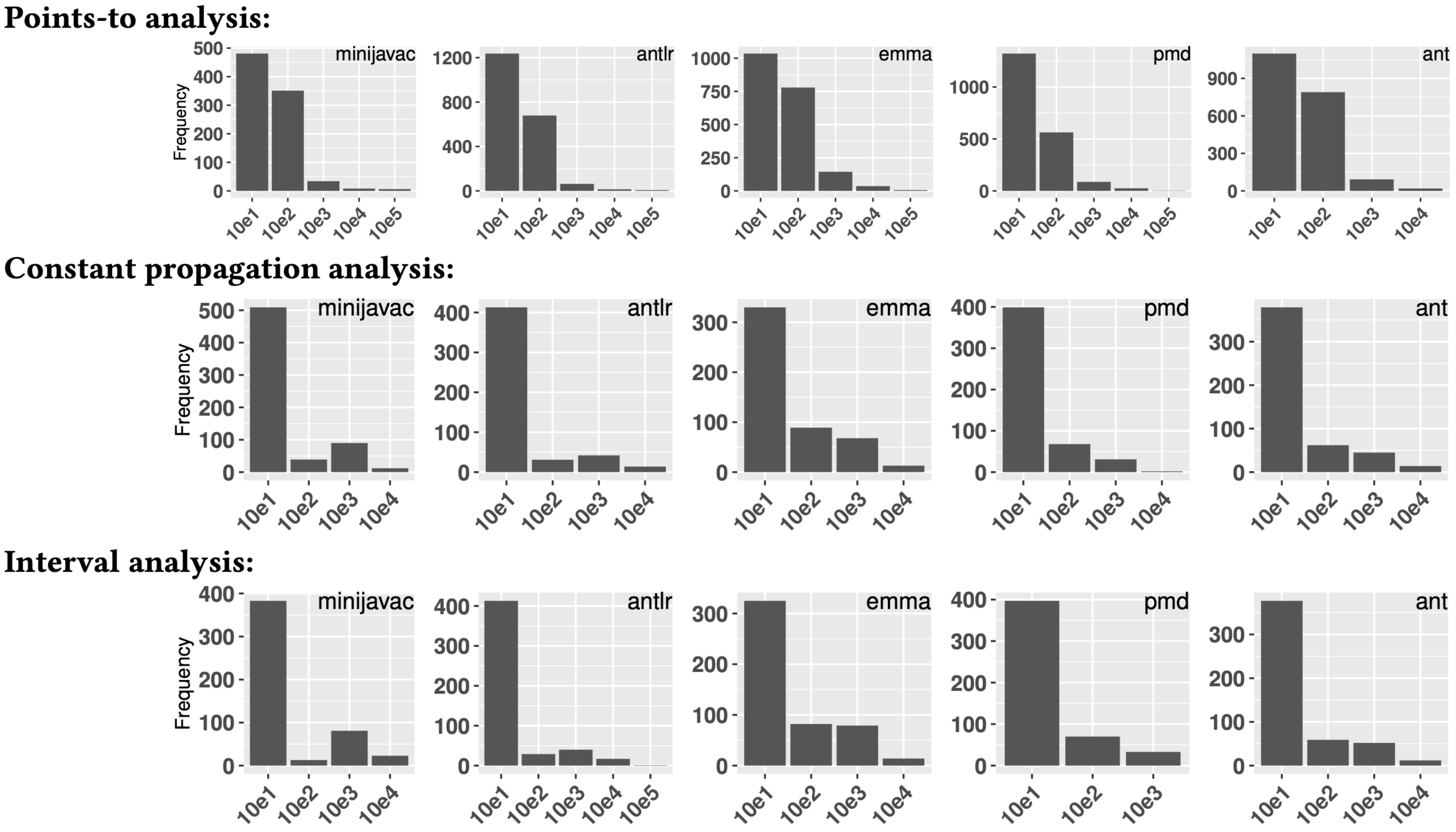
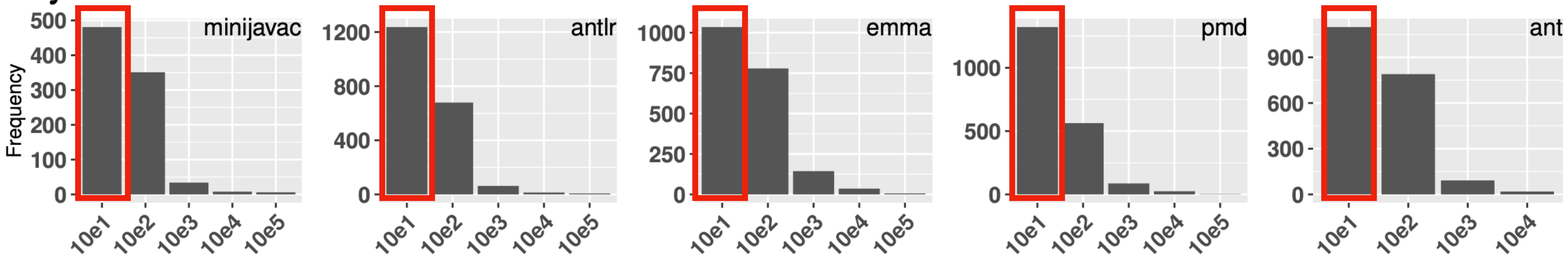


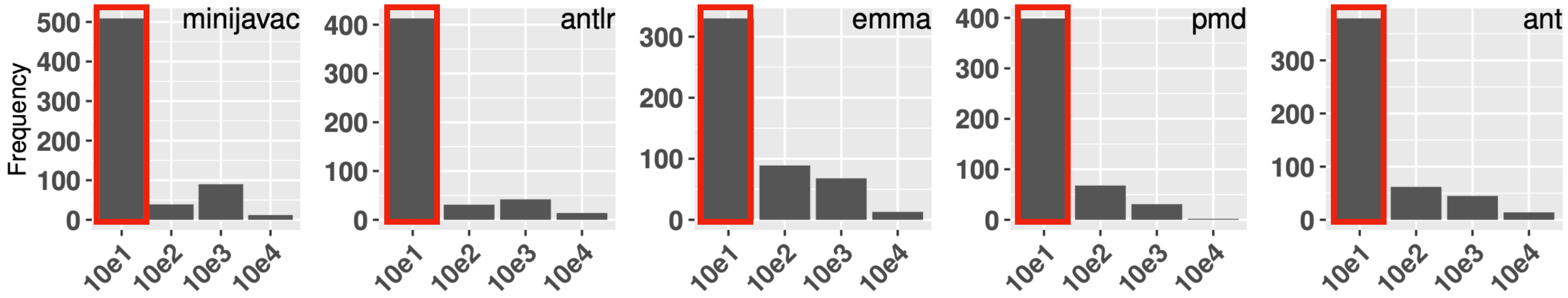
Figure 2. Whole-program analyses are incrementalizable because small input changes have low impact.

Impact Measurement

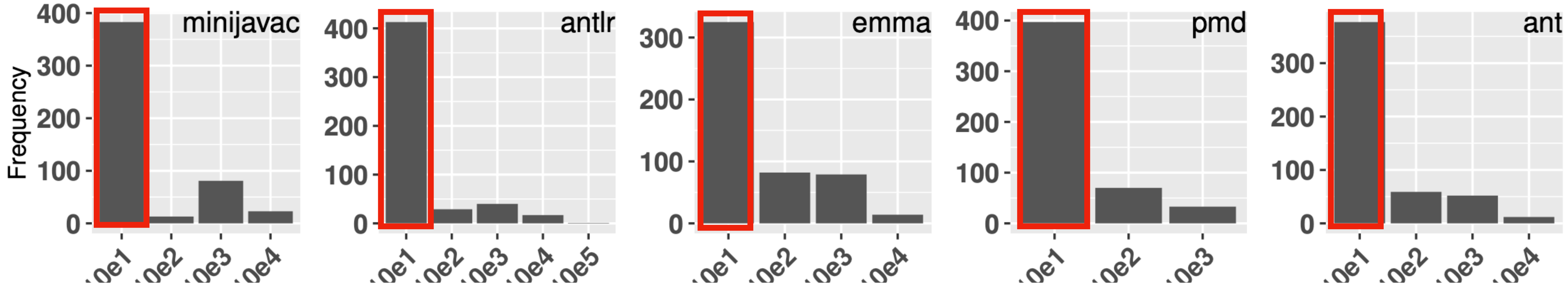
Points-to analysis:



Constant propagation analysis:

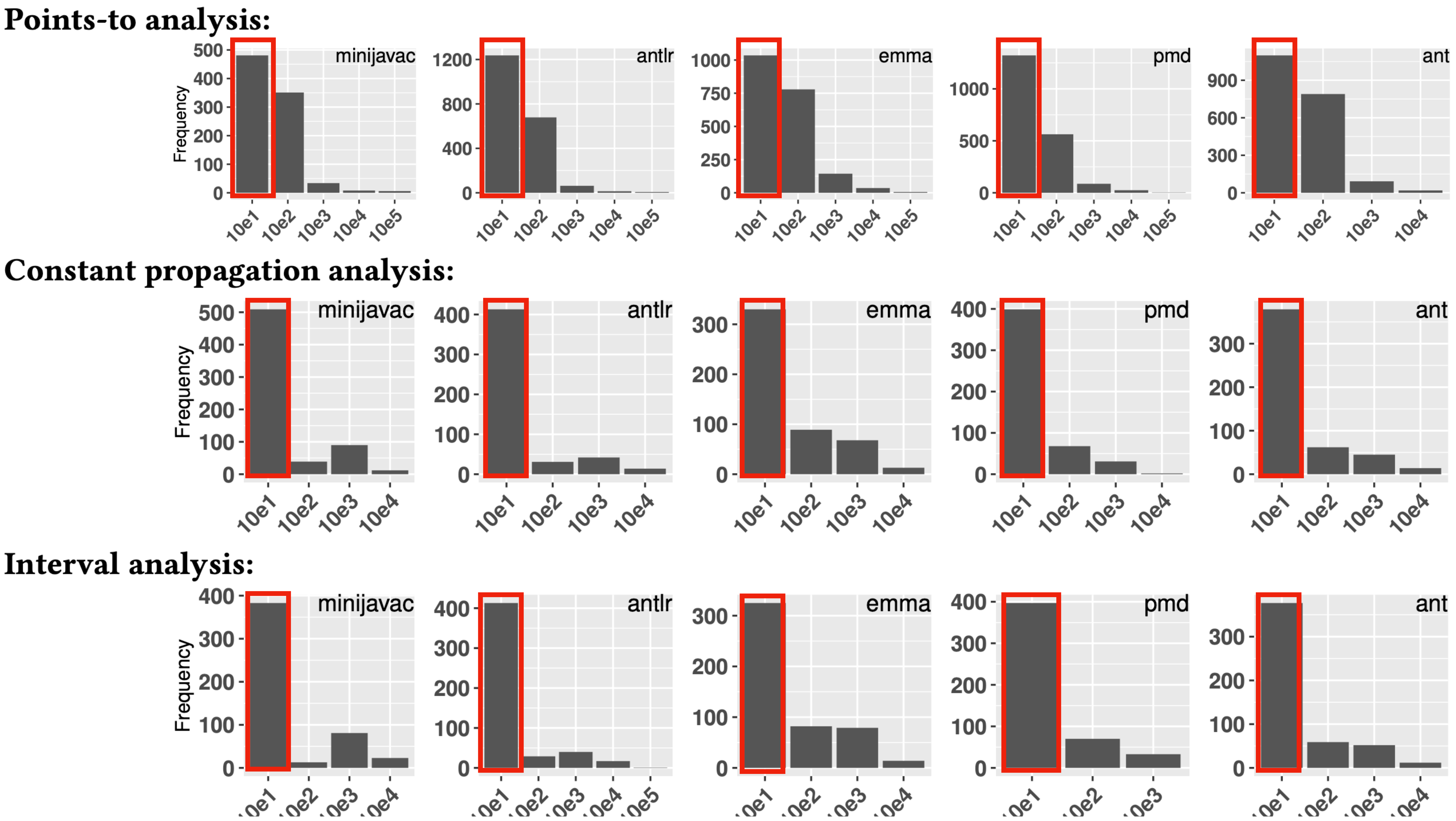


Interval analysis:



When the code is changed, the number of tuples affected is mostly between 1 and 10

Impact Measurement



When the code is changed, the number of tuples affected is mostly between 1 and 10

Thus, we can assume the analysis for whole program is **incrementalizable**

Problem #1: Scalability

- It's too slow
- This is a datalog solver problem called DRed[2] that was used before
- Previously, all related dependent tuples were deleted

[2] Ashish Gupta et al. 1993. Maintaining Views Incrementally.
In Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (Washington, D.C., USA) (SIGMOD ’93)

Problem #1: Scalability

Analysis results

```
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1 = s;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Problem #1: Scalability

Analysis results

PT(s, S)

```
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1 = s;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Problem #1: Scalability

Analysis results

PT(s, S)

PT(s1, S)

PT(s2, S)

```
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1 = s;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Problem #1: Scalability

Analysis results

PT(s, S)

PT(s1, S)

PT(s2, S)

Reach(proc)

```
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1 = s;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Problem #1: Scalability

Analysis results

PT(s, S)

PT(s1, S)

PT(s2, S)

Reach(proc)

PT(this, S)

```java
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1 = s;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```java
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Problem #1: Scalability

Analysis results

PT(s, S)

PT(s1, S)

PT(s2, S)

Reach(proc)

PT(this, S)

```
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Problem #1: Scalability

Analysis results

PT(s, S)	PT(s1, S)
PT(s2, S)	Reach(proc)
PT(this, S)	

```
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Previously, the current results were integrated into one, so to delete PT(s1, S)
After deleting all the relationships that depend on it, we derive the tuple again.

Problem #1: Scalability

Analysis results

re-computation & re-derive

```
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Previously, the current results were integrated into one, so to delete PT(s1, S)
After deleting all the relationships that depend on it, we derive the tuple again.

Solution #1: DDF

- DDF: Differential Data Flow
    - Maintains <u>timelines</u> and <u>support counts</u>
- Maintain accurate order of updates between dependent data
- Even if one path is invalidated,
    - if there is another path, the result of the analysis remains

Solution #1: DDF

Analysis results

Timestamp	Partial result
1	PT(s, S)

```
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1 = s;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Solution #1: DDF

Analysis results

Timestamp	Partial result
1	PT(s, S)
2	PT(s1, S), PT(s2, S)

```
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1 = s;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Solution #1: DDF

Analysis results

Timestamp	Partial result
1	PT(s, S)
2	PT(s1, S), PT(s2, S)
3	Reach(proc) @2

```
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1 = s;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Solution #1: DDF

Analysis results

Timestamp	Partial result
1	PT(s, S)
2	PT(s1, S), PT(s2, S)
3	Reach(proc) @2
4	PT(this, S)

```
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1 = s;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Solution #1: DDF

Analysis results

Timestamp	Partial result
1	PT(s, S)
2	PT(s1, S), PT(s2, S)
3	Reach(proc) @2
4	PT(this, S)
5	Reach(proc)

```
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1 = s;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Solution #1: DDF

Analysis results

Timestamp	Partial result
1	PT(s, S)
2	PT(s1, S), PT(s2, S)
3	Reach(proc) @2
4	PT(this, S)
5	Reach(proc)

```
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Solution #1: DDF

Analysis results

Timestamp	Partial result
1	PT(s, S)
2	PT(s2, S)
3	Reach(proc) @2
4	PT(this, S)
5	Reach(proc)

```
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Solution #1: DDF

Analysis results

Timestamp	Partial result
1	PT(s, S)
2	PT(s2, S)
3	Reach(proc) @1
4	PT(this, S)
5	Reach(proc)

```
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Solution #1: DDF

Analysis results

Timestamp	Partial result
1	PT(s, S)
2	PT(s2, S)
3	Reach(proc) @1
4	PT(this, S)
5	Reach(proc)

```
class Executor {
    public static void run(Env env){
        Session s = new Session();
        ...
        if (...) {
            Session s1;
            ...
            s1.proc();
        } else {
            Session s2 = s;
            ...
            s2.proc();
}}}
```

```
class Session {
    public void proc(){
        ...
        if (...) {
            this.proc();
        }
    ...
}}
```

Instead of ignoring previous results and inducing new tuples,

this method efficiently utilizes existing analysis results to speed up

Problem #2: Lattice-based aggregation

- Bot ⊑ O(obj) ⊑ C(cls)
- If it has to be changed to C(cls) instead of O(obj),
  - 1) **Delete** the previous O(obj),
  - 2) C(cls) should be **inserted**
  - In the **same stage** of the fixpoint computation
- However, previous datalog solver **does not guarantee**
  - appear at the same stage

Problem #2: Lattice-based aggregation

Analysis results

PT(f, O(F1))

PT(c, O(F2))

```java
class Session {
    public void proc() {
    Factory f;
    if (...) {
        f = new DefaultFactory(); F1
    } else {
        Factory c = new CustomFactory();
        f = c;                           F2
    }
    f.init();
    if (...) {
    this.proc();
}}}
```

```java
abstract class Factory {
    abstract void init();
}
class DefaultFactory extends Factory {
    @Override void init() { ... }
}
class CustomFactory extends Factory {
    @Override void init() { ... }
}
class DelegatingFactory extends Factory {
    @Override void init() { ... }
}
```

Problem #2: Lattice-based aggregation

Analysis results

PT(f, O(F1))

PT(c, O(F2))

PT(f, O(F2))

```
class Session {
    public void proc() {
    Factory f;
    if (...) {
        f = new DefaultFactory();  F1
    } else {
        Factory c = new CustomFactory();
        f = c;                              F2
    }
    f.init();
    if (...) {
    this.proc();
}}}
```

```
abstract class Factory {
    abstract void init();
}
class DefaultFactory extends Factory {
    @Override void init() { ... }
}
class CustomFactory extends Factory {
    @Override void init() { ... }
}
class DelegatingFactory extends Factory {
    @Override void init() { ... }
}
```

Problem #2: Lattice-based aggregation

Analysis results

PT(f, O(F1))

PT(f, O(F2))

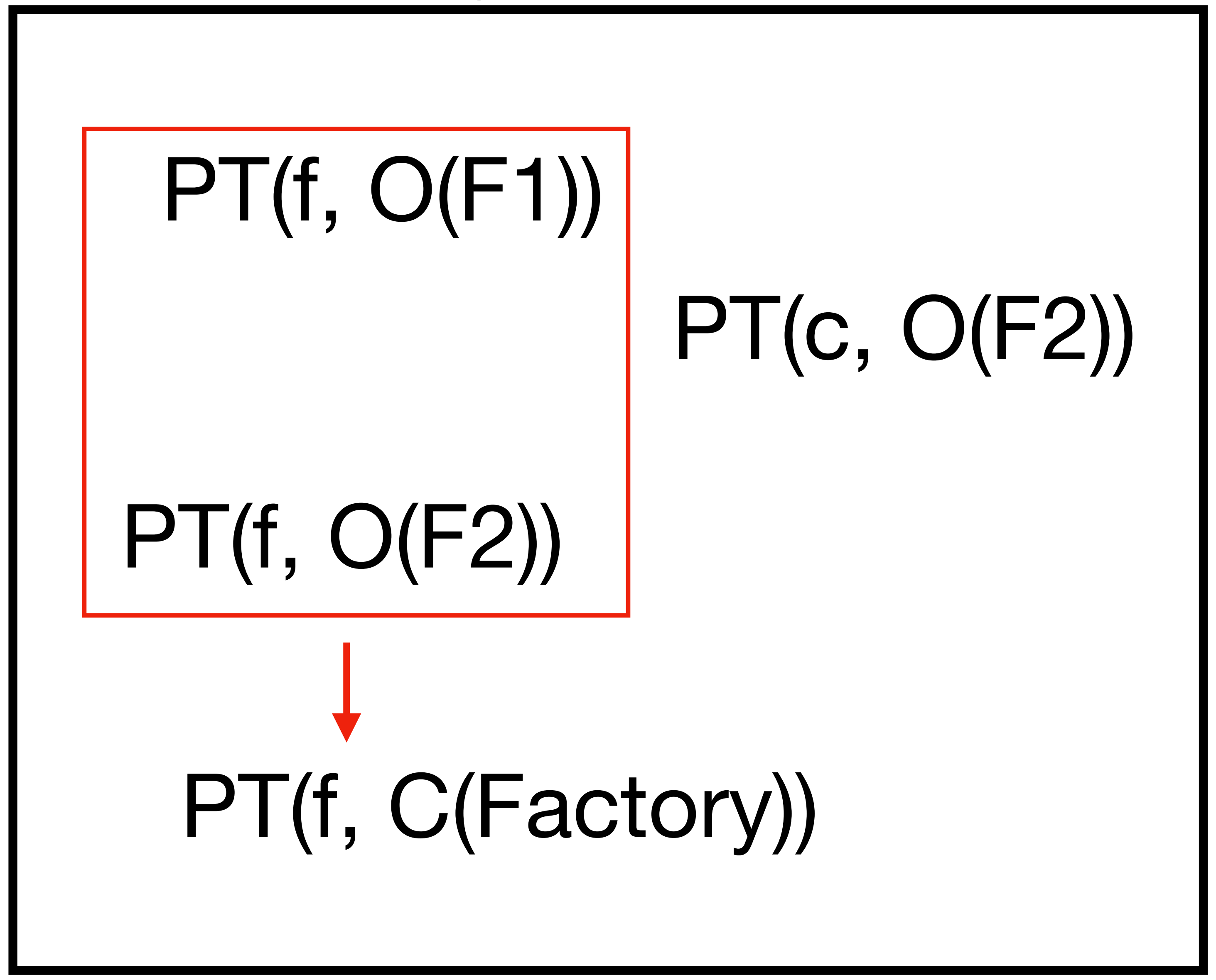
PT(c, O(F2))

```
class Session {
    public void proc() {
    Factory f;
    if (...) {
        f = new DefaultFactory();  F1
    } else {
        Factory c = new CustomFactory();
        f = c;                            F2
    }
    f.init();
    if (...) {
    this.proc();
}}}
```

```
abstract class Factory {
    abstract void init();
}
class DefaultFactory extends Factory {
    @Override void init() { ... }
}
class CustomFactory extends Factory {
    @Override void init() { ... }
}
class DelegatingFactory extends Factory {
    @Override void init() { ... }
}
```

Problem #2: Lattice-based aggregation

Analysis results




```
class Session {
    public void proc() {
    Factory f;
    if (...) {
        f = new DefaultFactory();  F1
    } else {
        Factory c = new CustomFactory();
        f = c;                          F2
    }
    f.init();
    if (...) {
    this.proc();
}}}
```

```
abstract class Factory {
    abstract void init();
}
class DefaultFactory extends Factory {
    @Override void init() { ... }
}
class CustomFactory extends Factory {
    @Override void init() { ... }
}
class DelegatingFactory extends Factory {
    @Override void init() { ... }
}
```

Problem #2: Lattice-based aggregation

Analysis results

① Delete previous result

$\boxed{\text{PT(f, O(F1))} \quad \text{PT(f, O(F2))}}$

PT(c, O(F2))

↓

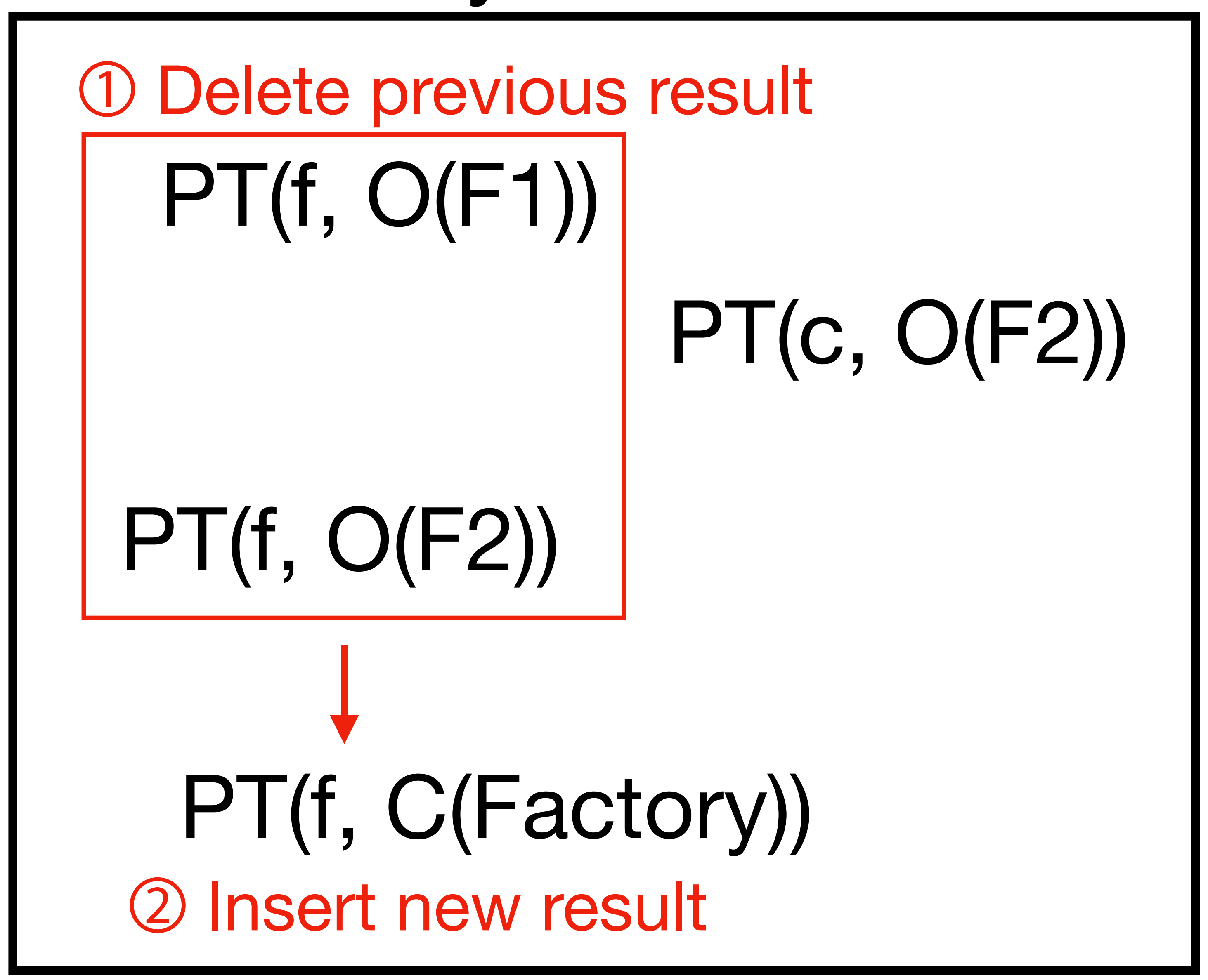
PT(f, C(Factory))

```
class Session {
    public void proc() {
    Factory f;
    if (...) {
        f = new DefaultFactory();  F1
    } else {
        Factory c = new CustomFactory();
        f = c;                             F2
    }
    f.init();
    if (...) {
    this.proc();
}}}
```

```
abstract class Factory {
    abstract void init();
}
class DefaultFactory extends Factory {
    @Override void init() { ... }
}
class CustomFactory extends Factory {
    @Override void init() { ... }
}
class DelegatingFactory extends Factory {
    @Override void init() { ... }
}
```

Problem #2: Lattice-based aggregation

Analysis results




```
class Session {
    public void proc() {
    Factory f;
    if (...) {
        f = new DefaultFactory();  F1
    } else {
        Factory c = new CustomFactory();
        f = c;                            F2
    }
    f.init();
    if (...) {
    this.proc();
}}}
```

```
abstract class Factory {
    abstract void init();
}
class DefaultFactory extends Factory {
    @Override void init() { ... }
}
class CustomFactory extends Factory {
    @Override void init() { ... }
}
class DelegatingFactory extends Factory {
    @Override void init() { ... }
}
```

Problem #2: Lattice-based aggregation

- O(F1) → C(Factory) is monotone so no problem
  - O(obj) ⊑ C(cls)
- However, -O(F1) and +C(Factory) are not guaranteed
  - to compute on the same stage
- Fixed point may not converge to O(F1) or C(Factory)
  - continue to fluctuate and may become non-terminated.

Solution #2: Eventual-monotonicity with inflationary semantics

- Inflation: don't delete old aggregate results
- eventual-monotonicity: only propagete "largest" value

Analysis results

Timestamp	Partial result
1	PT(f, O(F1)), PT(c, O(F2)
2	PT(f, O(F2))
3	PT(f, C(Factory))
4	...
5	...

Solution #2: Eventual-monotonicity with inflationary semantics

- Inflation: don't delete old aggregate results
- eventual-monotonicity: only propagete "largest" value

Analysis results

Timestamp	Partial result
1	PT(f, O(F1)), PT(c, O(F2)
2	PT(f, O(F2))
3	PT(f, C(Factory))
4	...
5	...

aggregands

Solution #2: Eventual-monotonicity with inflationary semantics

- Inflation: don't delete old aggregate results
- eventual-monotonicity: only propagete "largest" value

Analysis results

Timestamp	Partial result	
1	PT(f, O(F1)), PT(c, O(F2))	aggregands
2	PT(f, O(F2))	
3	PT(f, C(Factory))	Final result
4	...	
5	...	

Evaluation

Research Question

- RQ1: Can LADDDER provide quick feedback for lattice-based inter-procedural analyses?
- RQ2: Is the extra memory consumption of LADDDER acceptable for IDE usage?

Benchmark

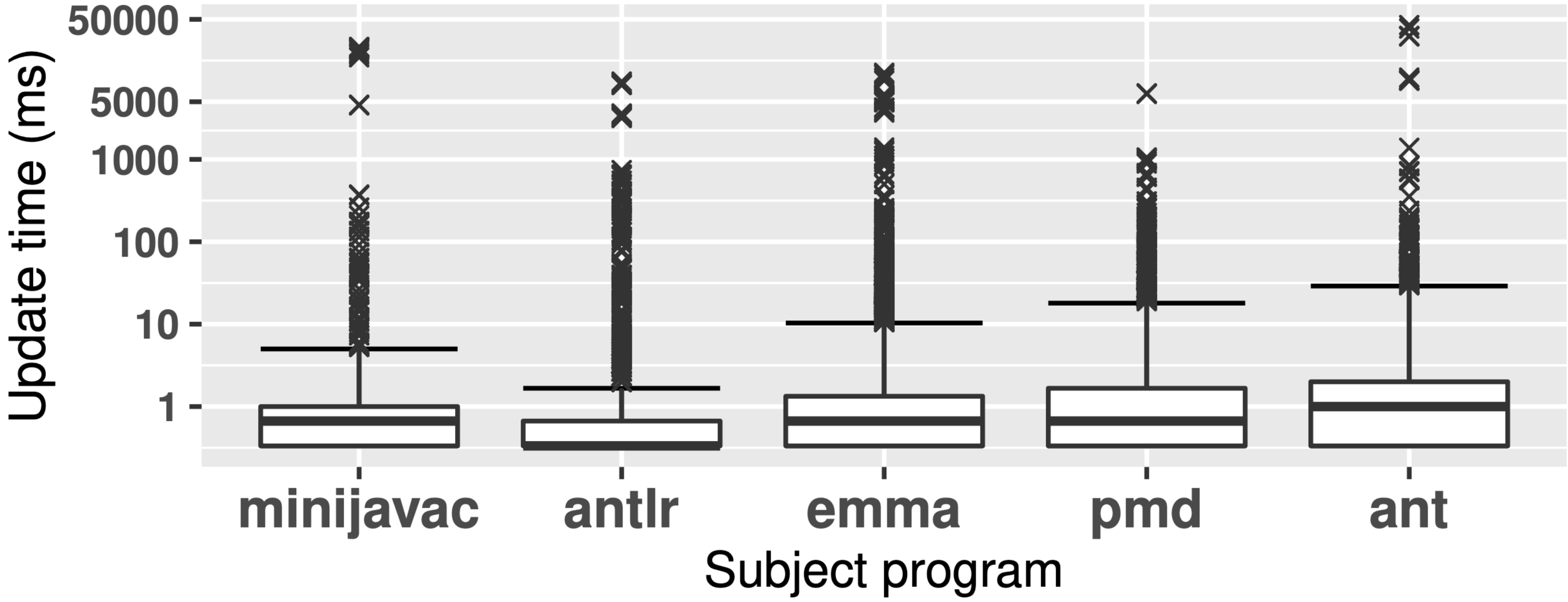
- Real-world java program from Qualitas Corpus
  - antlr (22k LOC), emma (26k LOC), pmd (61k LOC), ant (105k LOC).
  - minijavac (6.5k LOC)

Evaluation

RQ1: provide quick feedback

- Randomly delete and re-insert 1000 object allocation sites

k-update points-to analysis:

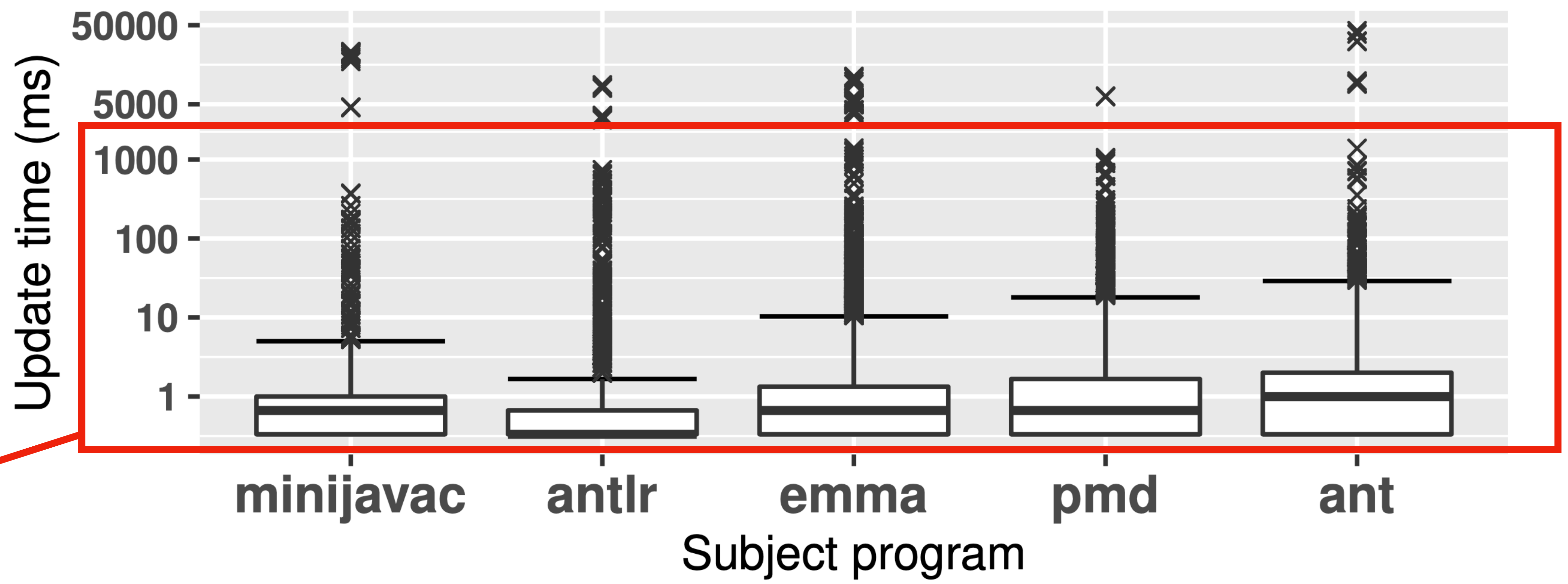


Evaluation

RQ1: provide quick feedback

- Randomly delete and re-insert 1000 object allocation sites

k-update points-to analysis:



99% of the changes are handled in less than 1000 ms

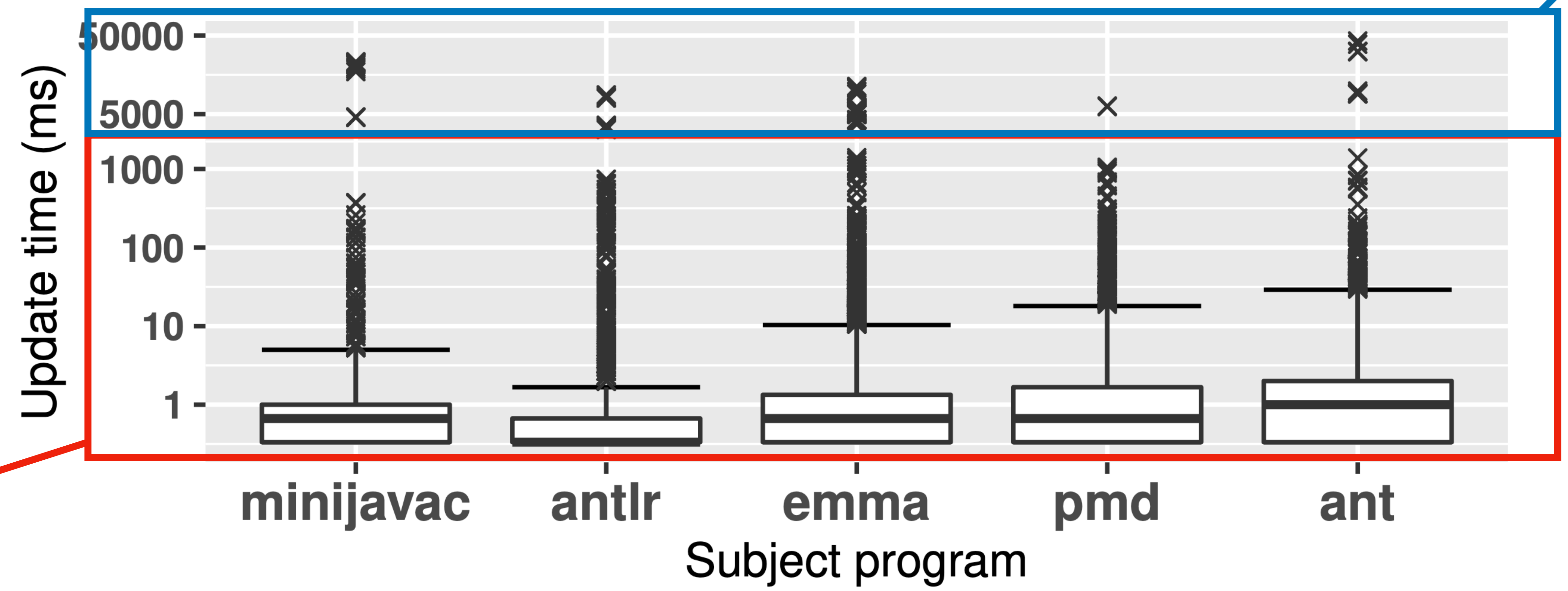
Evaluation

RQ1: provide quick feedback

- Randomly delete and re-insert 1000 object allocation sites

Some outliers reach 50 sec

k-update points-to analysis:



99% of the changes are handled in less than 1000 ms

Evaluation

RQ2: memory usage

- Memory use of LADDDER on the 5 code bases:
  - k-update points-to: 3.7 - 8.7 GB
  - constant propagation 0.6 - 2.3 GB
  - interval 0.8 - 2.9 GB
- "These values may seem high, but we emphasize that the analyses are inter-procedural, also analyzing parts of the JRE."

Summary

- **Incremental analysis** provides analysis results for the modified code part
  - not the entire code.
- Previous work has **scalability** and **lattice-based aggregation** problems.
- LADDDER solves these problems with
  - **Differential Dataflow**
  - **Eventual-monotonicity** with inflationary semantics
- LADDDER performs incremental analysis in **0.1s** and
  - performs with a **convincing amount of memory**