IncA: A DSL for the Definition of Incremental Program Analyses

Tamás Szabó, Sebastian Erdweg, Markus Völter

itemis TU Delft mbeddr

Analyses are essential in IDEs

```
void measure(Environment env) {
  int32 temp;
  if (env.active) {
    readSensor(&temp);
  } else {
    error("Inactive system init!");
  } if
  calibrateEnv(env, temp);
} measure (function)
```

Error: Variable temp is not initialized!

Analyses are essential in IDEs

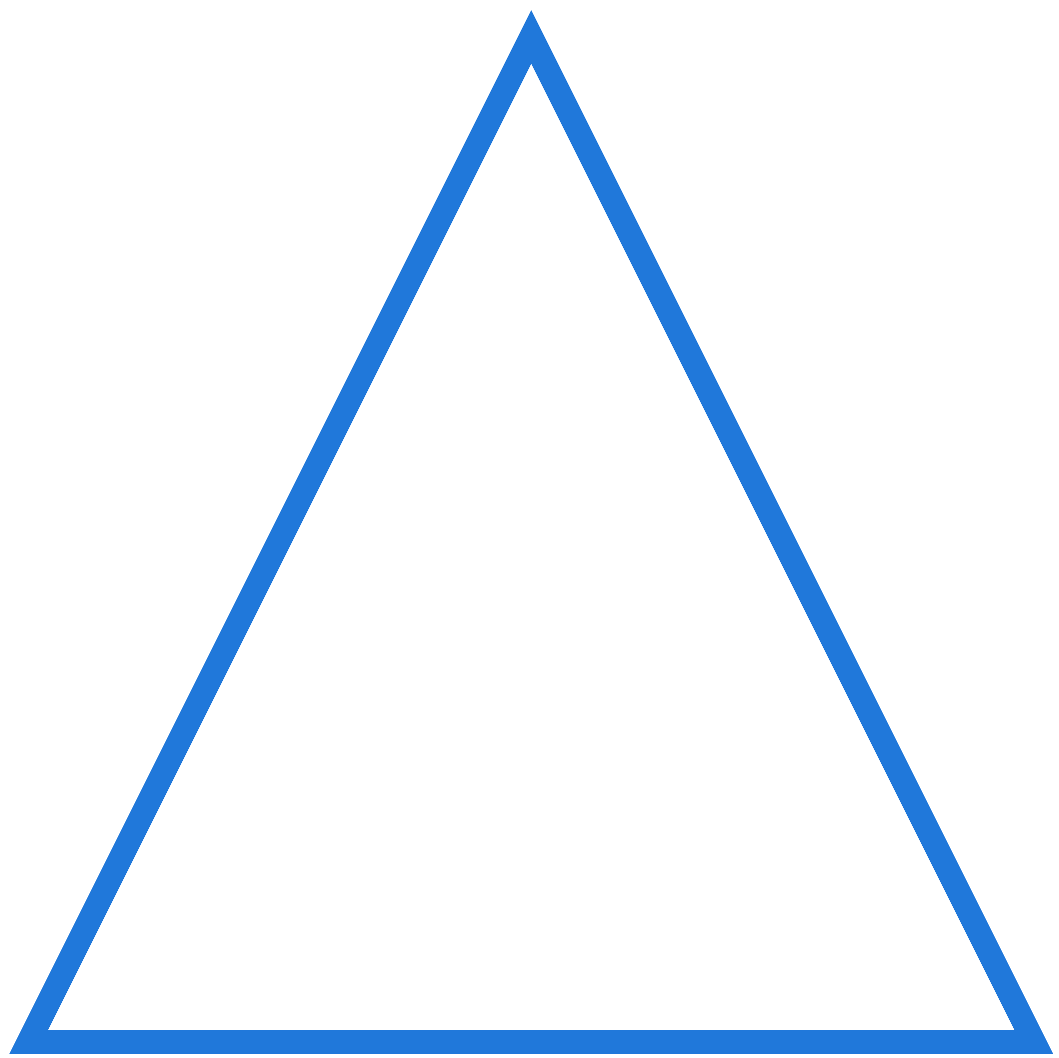
Fault Detection

Optimizations

Refactorings

We can't have it all!

Runtime Performance



*-sensitive
(Precision)

Memory Use

Problem Statement / Challenges

Support efficient program analyses for DSLs.

Problem Statement / Challenges

Support efficient program analyses for DSLs.

State-of-the-art DSLs also require sophisticated analyses just like GPLs.

Problem Statement / Challenges

Support efficient program analyses for DSLs.

State-of-the-art DSLs also require sophisticated analyses just like GPLs.

But they are developed with different economies compared to GPLs.

Problem Statement / Challenges

Real-time feedback in IDEs: an analysis should not interrupt the development flow.

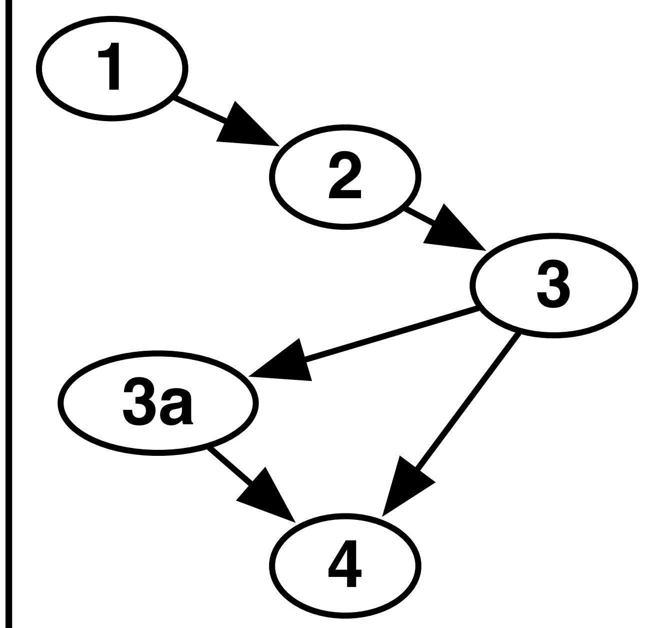
Solution Approach

IncA is a framework for the implementation and efficient evaluation of program analyses.

IncA - Example

Analyzed language: mbeddr C

```
pthread_mutex_lock(sensorLock);              (1)
int temp = readSensor(...);                  (2)
if (outOfRange(temp)) {                      (3)
  log("Beyond threshold %d!", temp);         (3a)
}
pthread_mutex_unlock(sensorLock);            (4)
```



Analyzed program

Analysis: control flow

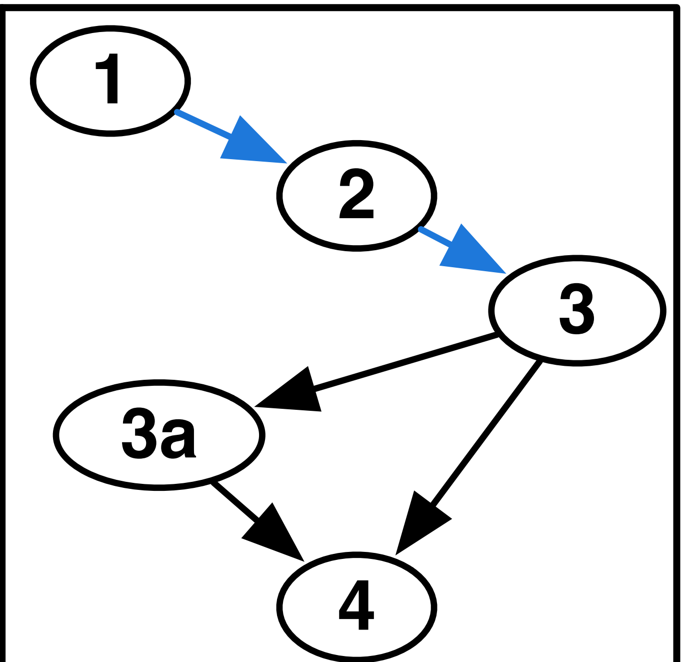
IncA - Example

Decompose the problem into two parts and define two relations.

IncA - Example

CSimple: CFG edges conforming to syntactic precedence.

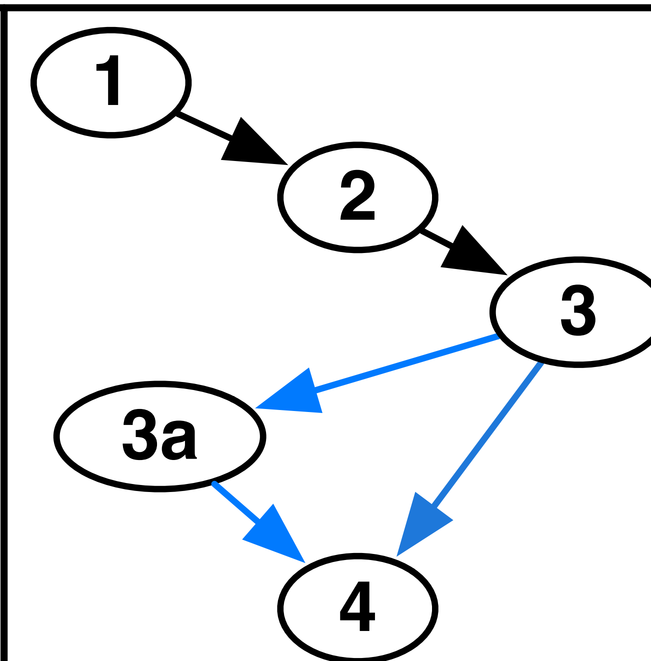
```
pthread_mutex_lock(sensorLock);          (1)
int temp = readSensor(...);              (2)
if (outOfRange(temp)) {                  (3)
  log("Beyond threshold %d!", temp);     (3a)
}
pthread_mutex_unlock(sensorLock);        (4)
```



IncA - Example

CIf: CFG edges leading into and out from if statements.

```
pthread_mutex_lock(sensorLock);                  (1)
int temp = readSensor(...);                      (2)
if (outOfRange(temp)) {                          (3)
  log("Beyond threshold %d!", temp);            (3a)
}
pthread_mutex_unlock(sensorLock);                (4)
```



IncA - DSL

```
def cSimple(trg : Statement): SimpleStatement={
  src := precedingStatement(trg)
  assert src instanceOf SimpleStatement
  return src
}
```

IncA - DSL

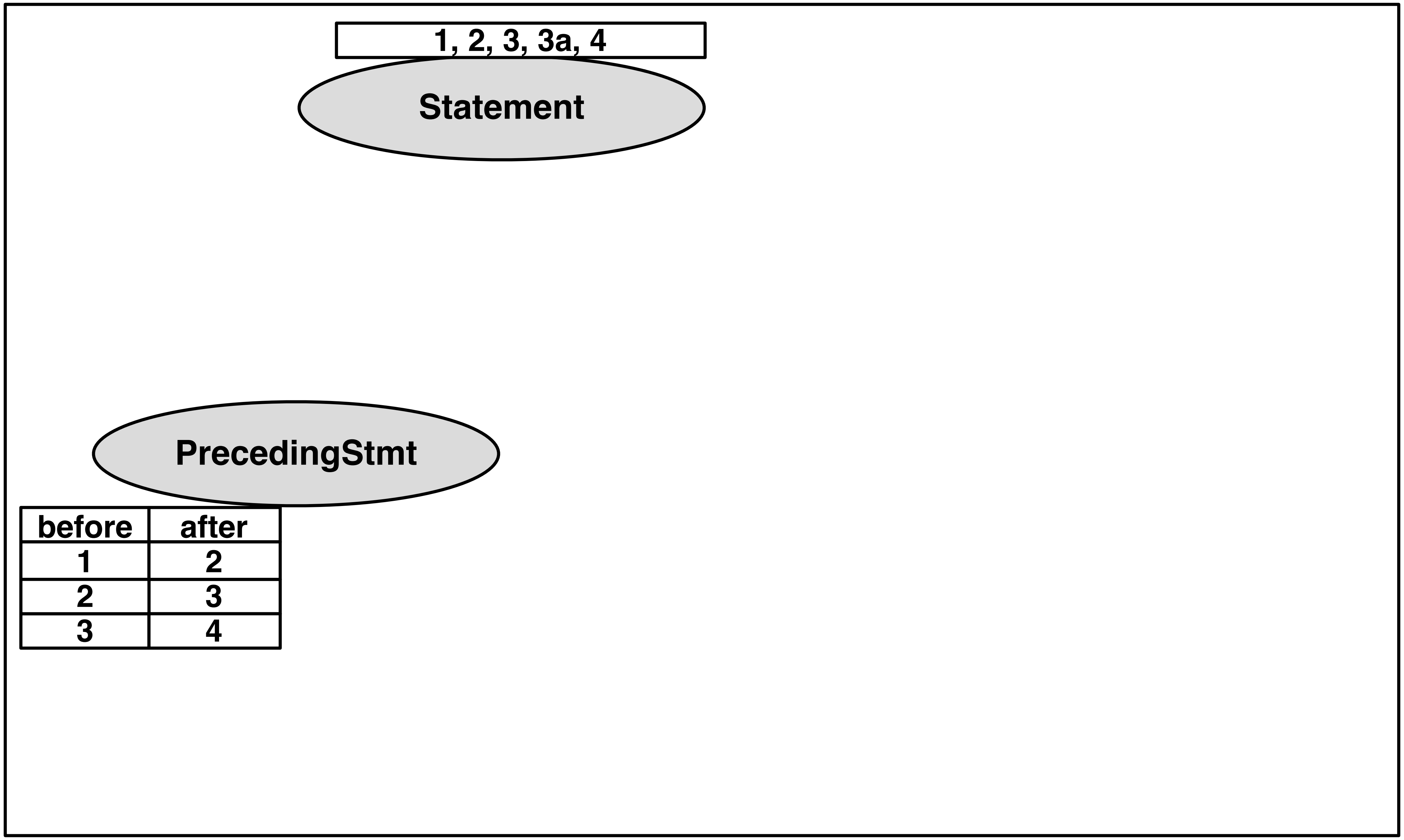
```
def cIf(trg : Statement): Statement = {
  src := precedingStatement(trg)
  assert src instanceOf IfStatement
  return lastStatement(src)
} alt {
  src := precedingStatement(trg)
  assert src instanceOf IfStatement
  assert undef src.else
  return src
} alt {
  assert undef precedingStatement(trg)
  parent := trg.parent
  assert parent instanceOf IfStatement
  return parent
}
```

IncA - DSL

```
def cFlow(trg : Statement) : Statement = {
  return cSimple(trg)
} alt {
  return cIf(trg)
}
```

IncA - Semantics

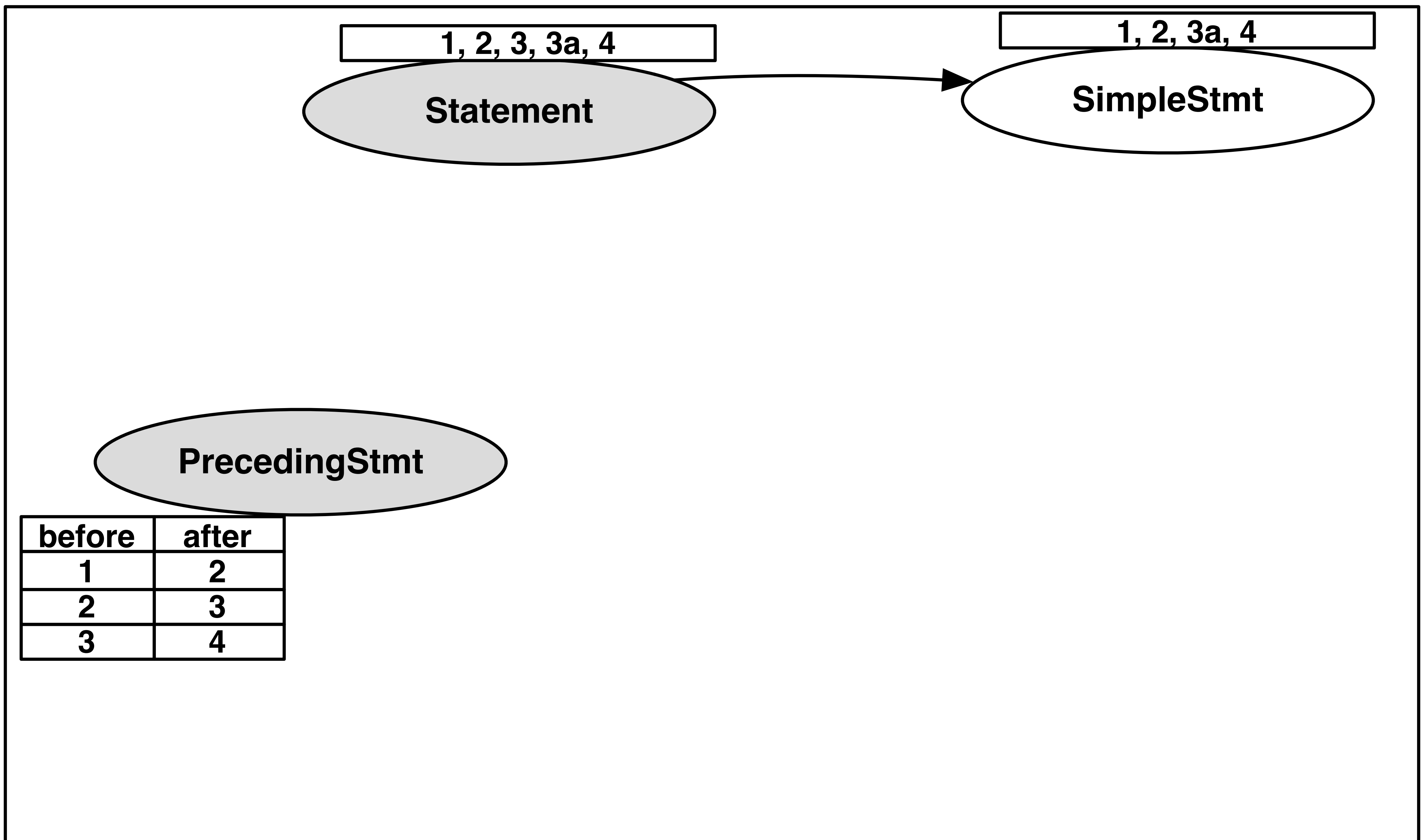
```
def cSimple(trg : Statement): SimpleStatement={
  src := precedingStatement(trg)
  assert src instanceOf SimpleStatement
  return src
}
```



before	after
1	2
2	3
3	4

IncA - Semantics

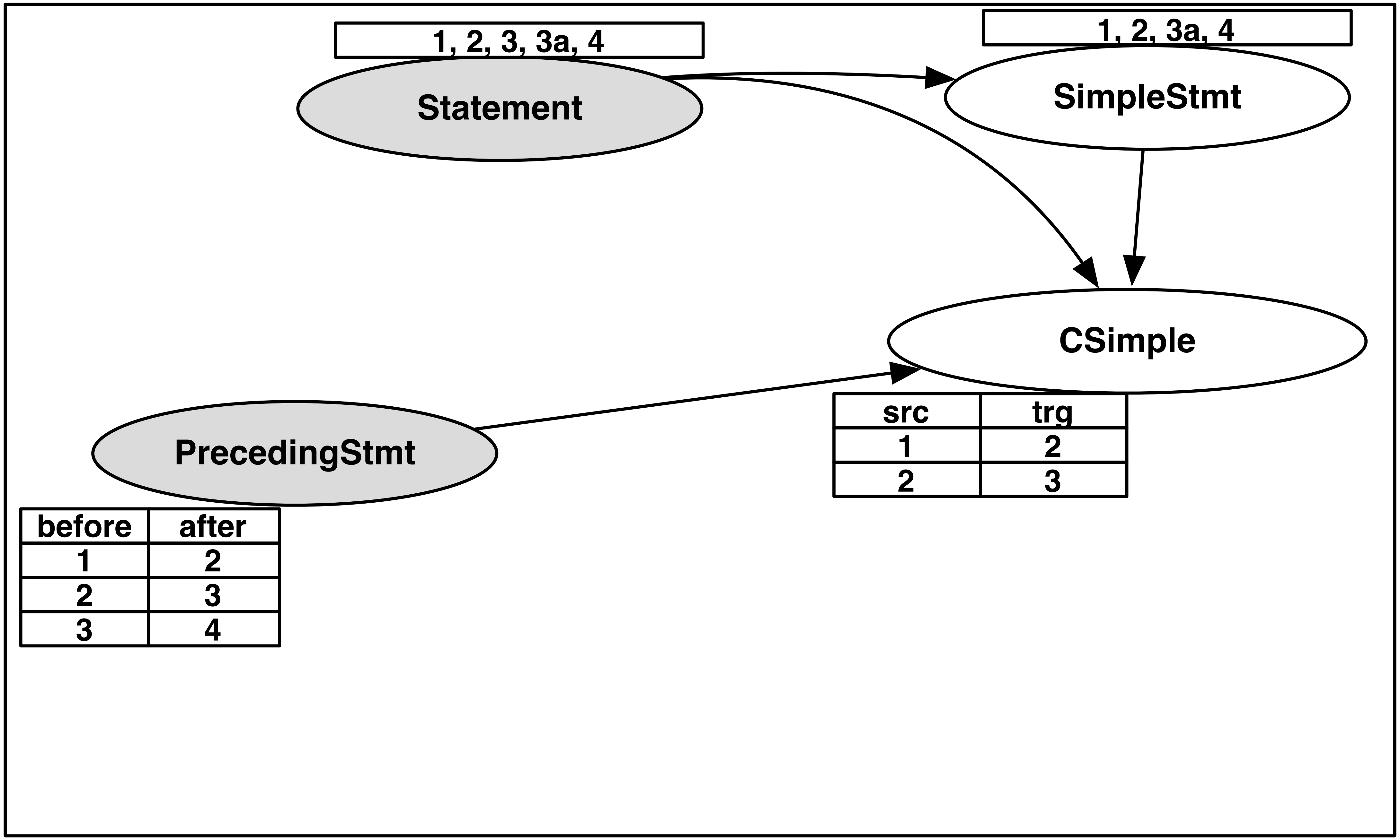
```
def cSimple(trg : Statement): SimpleStatement={
  src := precedingStatement(trg)
  assert src instanceOf SimpleStatement
  return src
}
```



before	after
1	2
2	3
3	4

IncA - Semantics

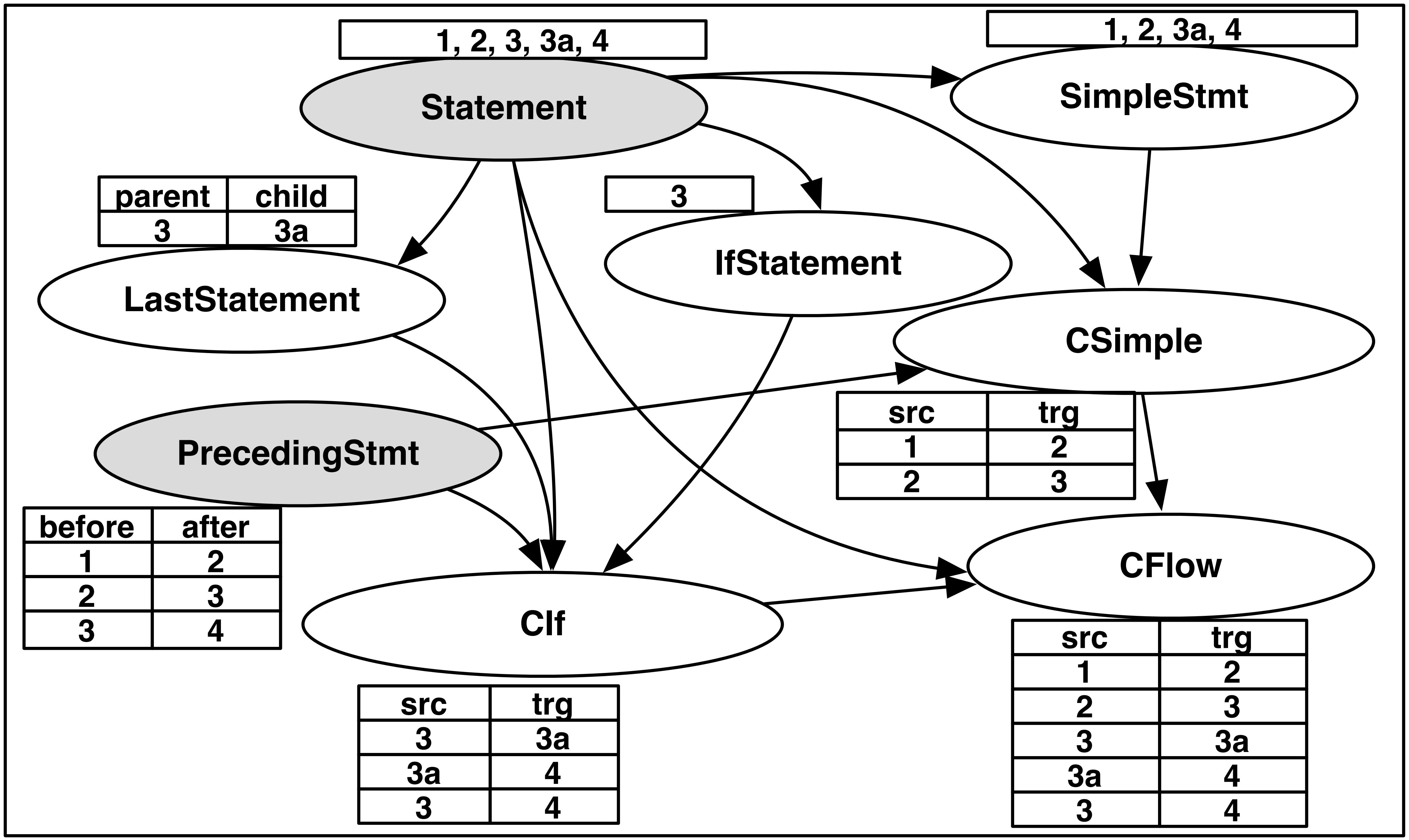
```
def cSimple(trg : Statement): SimpleStatement={
  src := precedingStatement(trg)
  assert src instanceOf SimpleStatement
  return src
}
```



src	trg
1	2
2	3

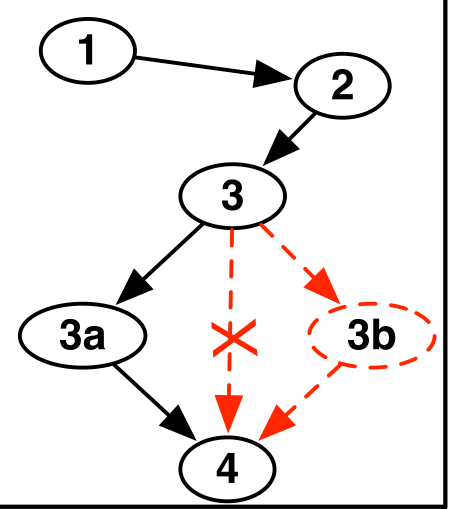
before	after
1	2
2	3
3	4

IncA - Semantics



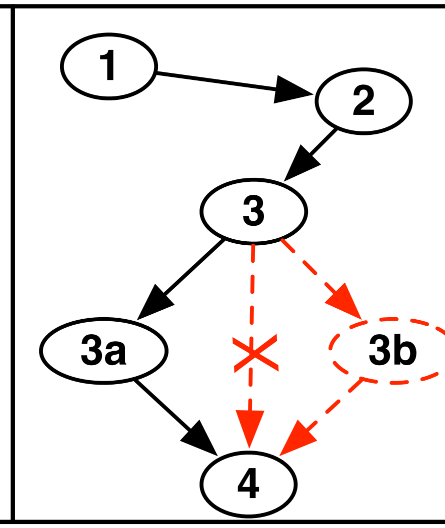
IncA - Incremental Semantics

```
pthread_mutex_lock(sensorLock);                    (1)
int temp = readSensor(...);                        (2)
if (outOfRange(temp)) {                            (3)
  log("Beyond threshold %d!", temp);               (3a)
} else {
  calibrate_env(temp, ...);                        (3b)
}
pthread_mutex_unlock(sensorLock);                  (4)
```



IncA - Incremental Semantics

```
pthread_mutex_lock(sensorLock);                 (1)
int temp = readSensor(...);                     (2)
if (outOfRange(temp)) {                         (3)
  log("Beyond threshold %d!", temp);            (3a)
} else {
  calibrate_env(temp, ...);                     (3b)
}
pthread_mutex_unlock(sensorLock);               (4)
```



+

1, 2, 3, 3a, 3b, 4

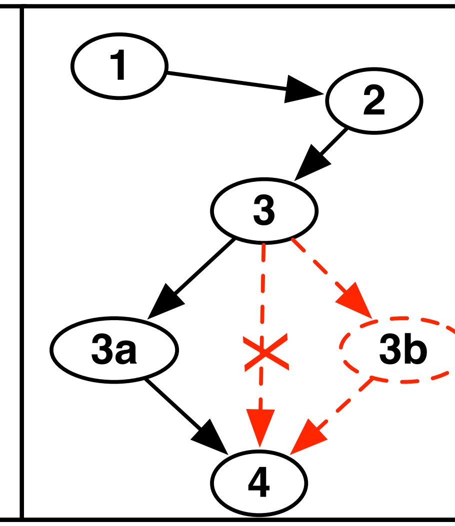
Statement

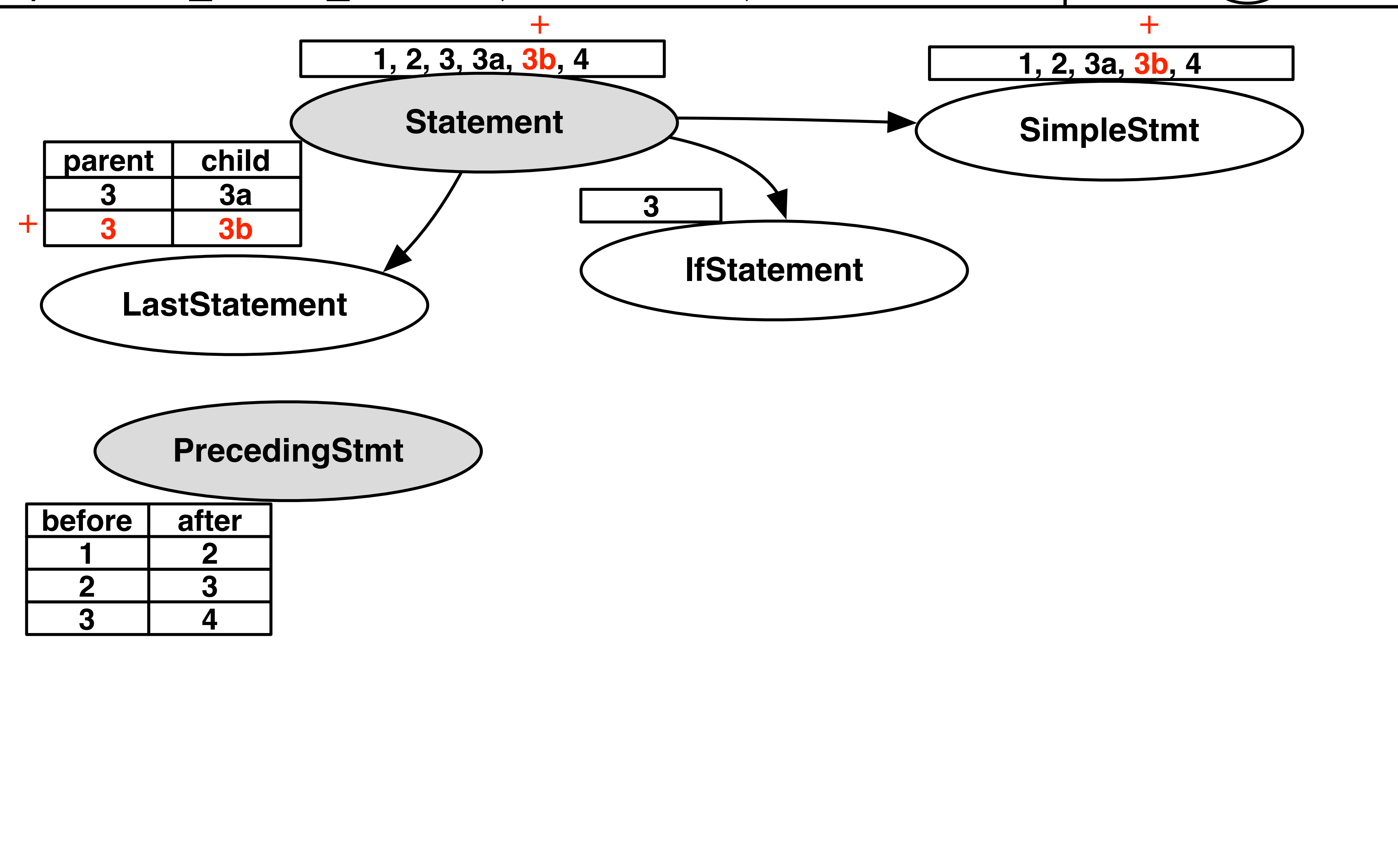
PrecedingStmt

before	after
1	2
2	3
3	4

IncA - Incremental Semantics

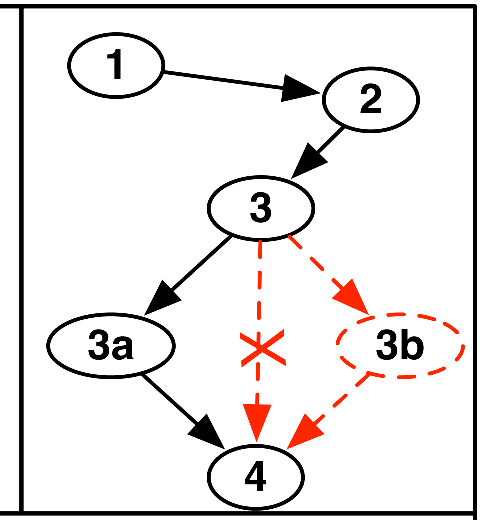
```
pthread_mutex_lock(sensorLock);                 (1)
int temp = readSensor(...);                     (2)
if (outOfRange(temp)) {                         (3)
  log("Beyond threshold %d!", temp);            (3a)
} else {
  calibrate_env(temp, ...);                     (3b)
}
pthread_mutex_unlock(sensorLock);               (4)
```

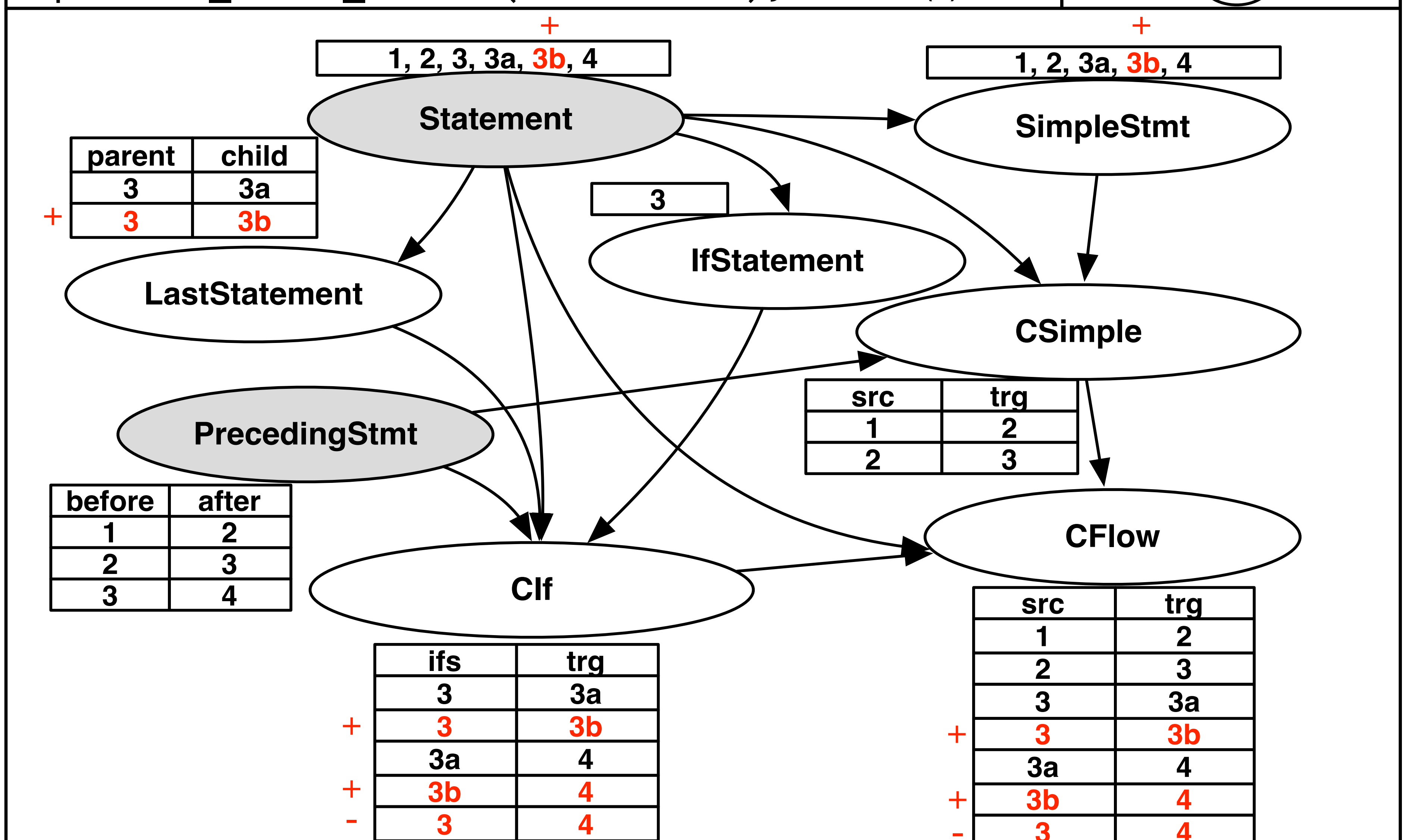




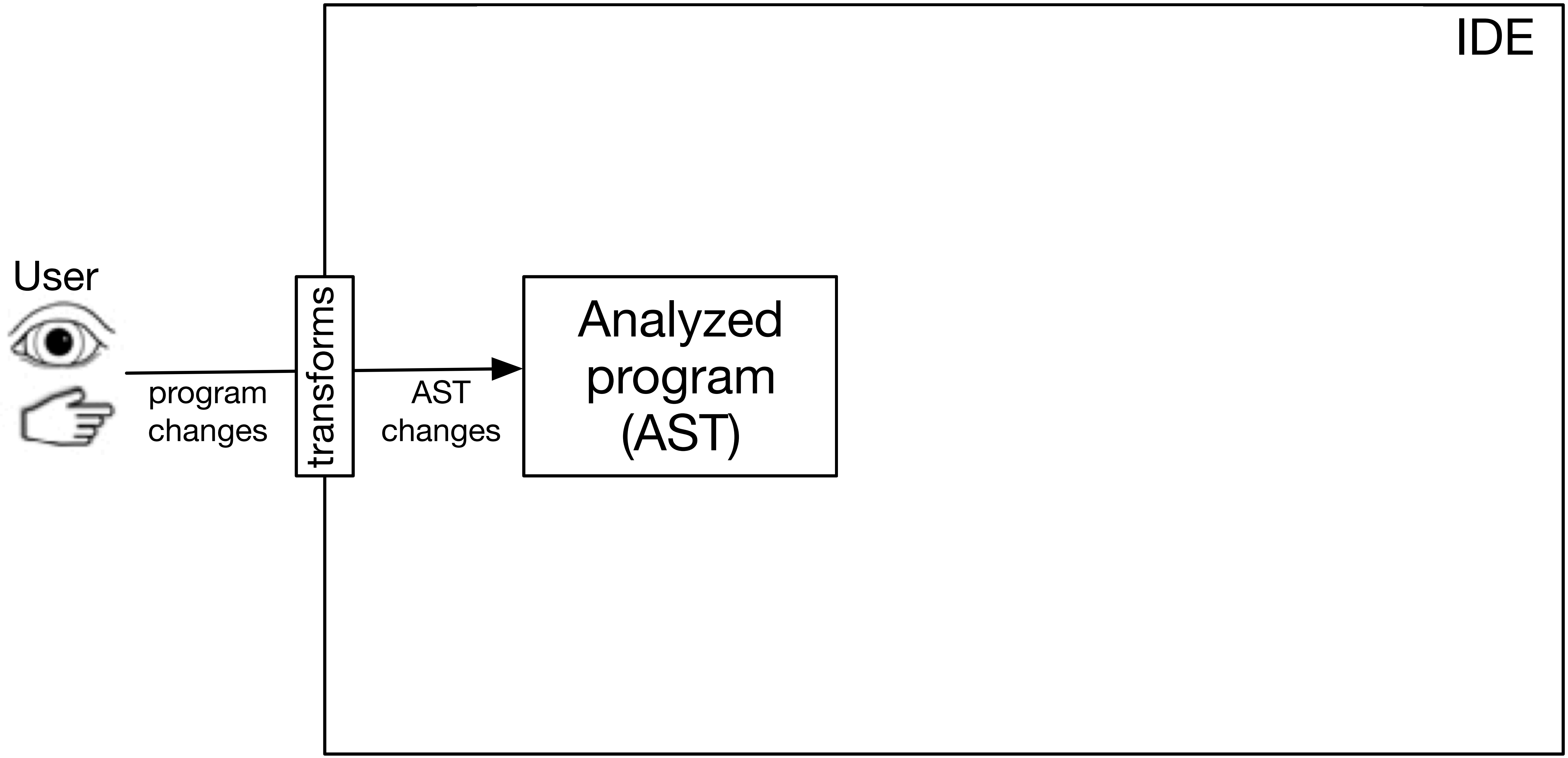
IncA - Incremental Semantics

```
pthread_mutex_lock(sensorLock);                 (1)
int temp = readSensor(...);                     (2)
if (outOfRange(temp)) {                         (3)
  log("Beyond threshold %d!", temp);            (3a)
} else {
  calibrate_env(temp, ...);                     (3b)
}
pthread_mutex_unlock(sensorLock);               (4)
```

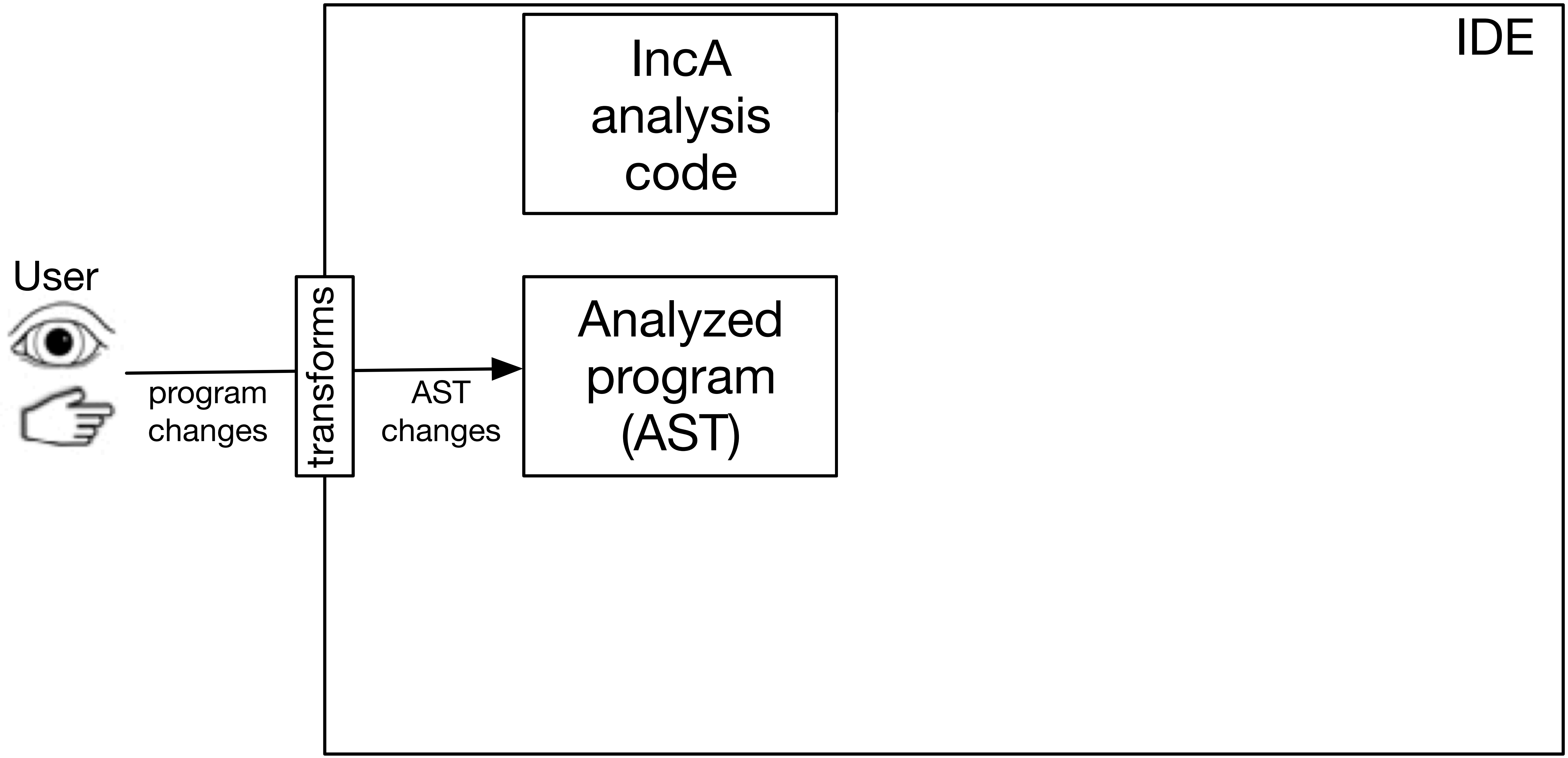




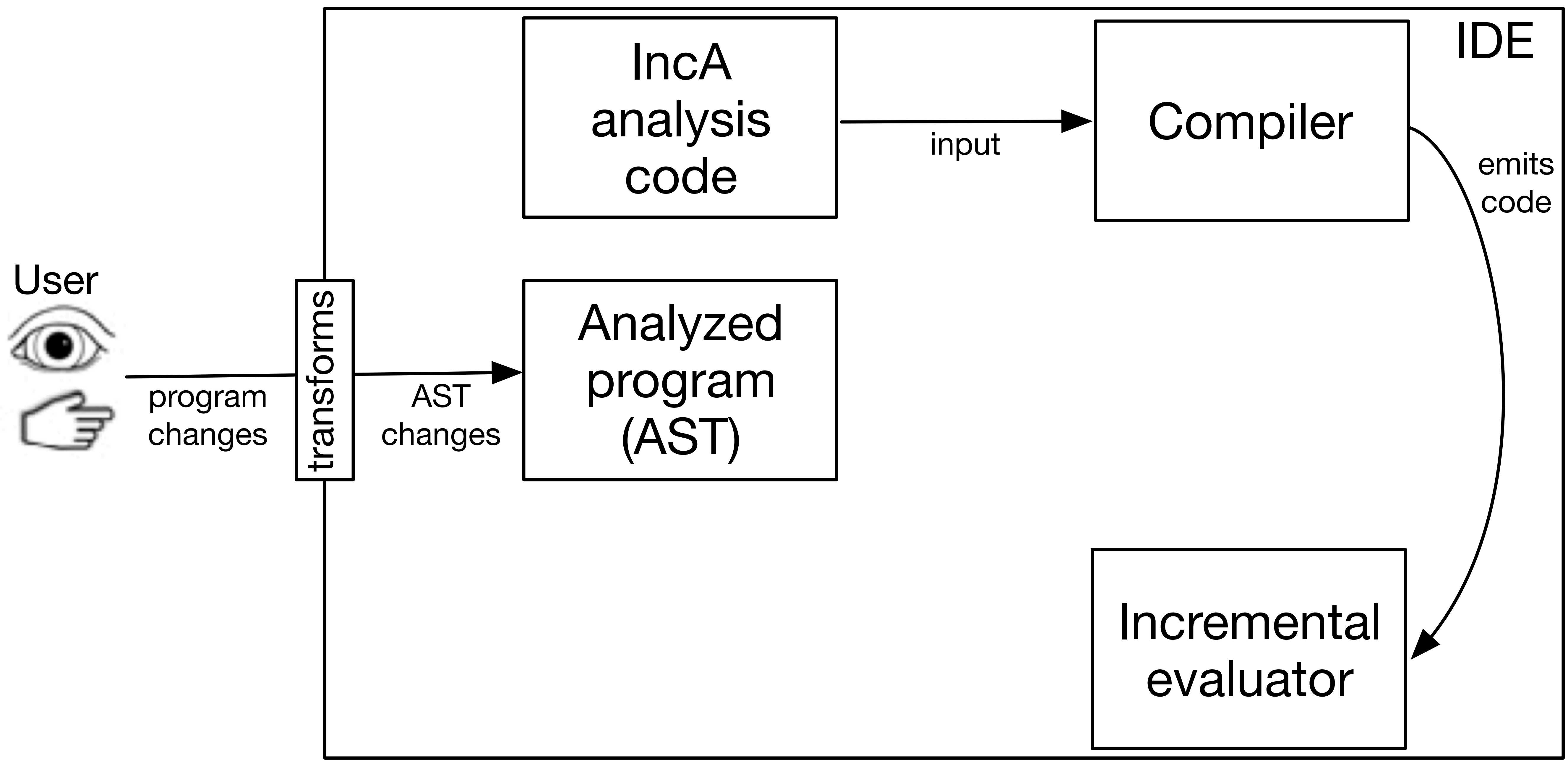
IncA - Architecture



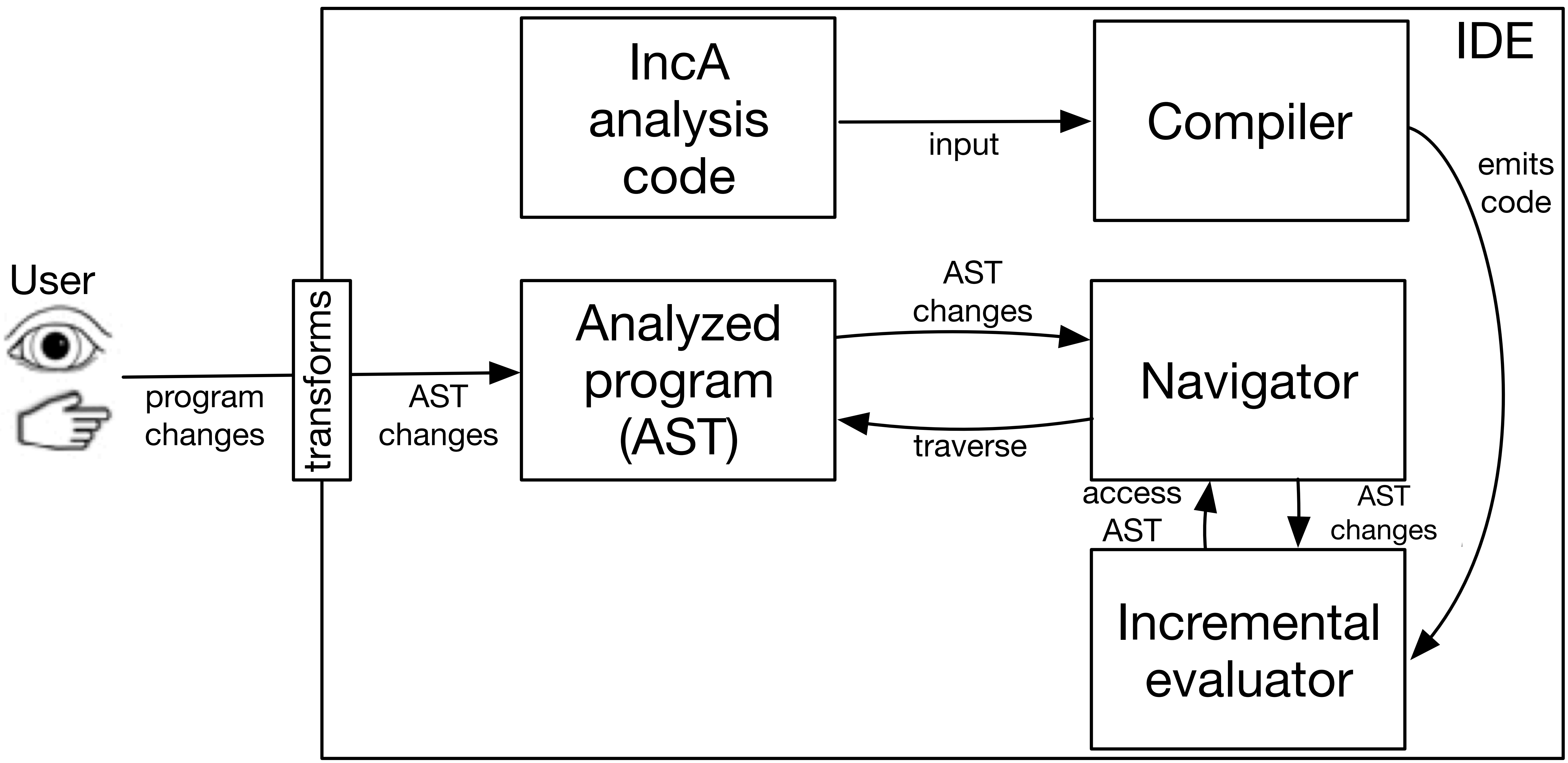
IncA - Architecture



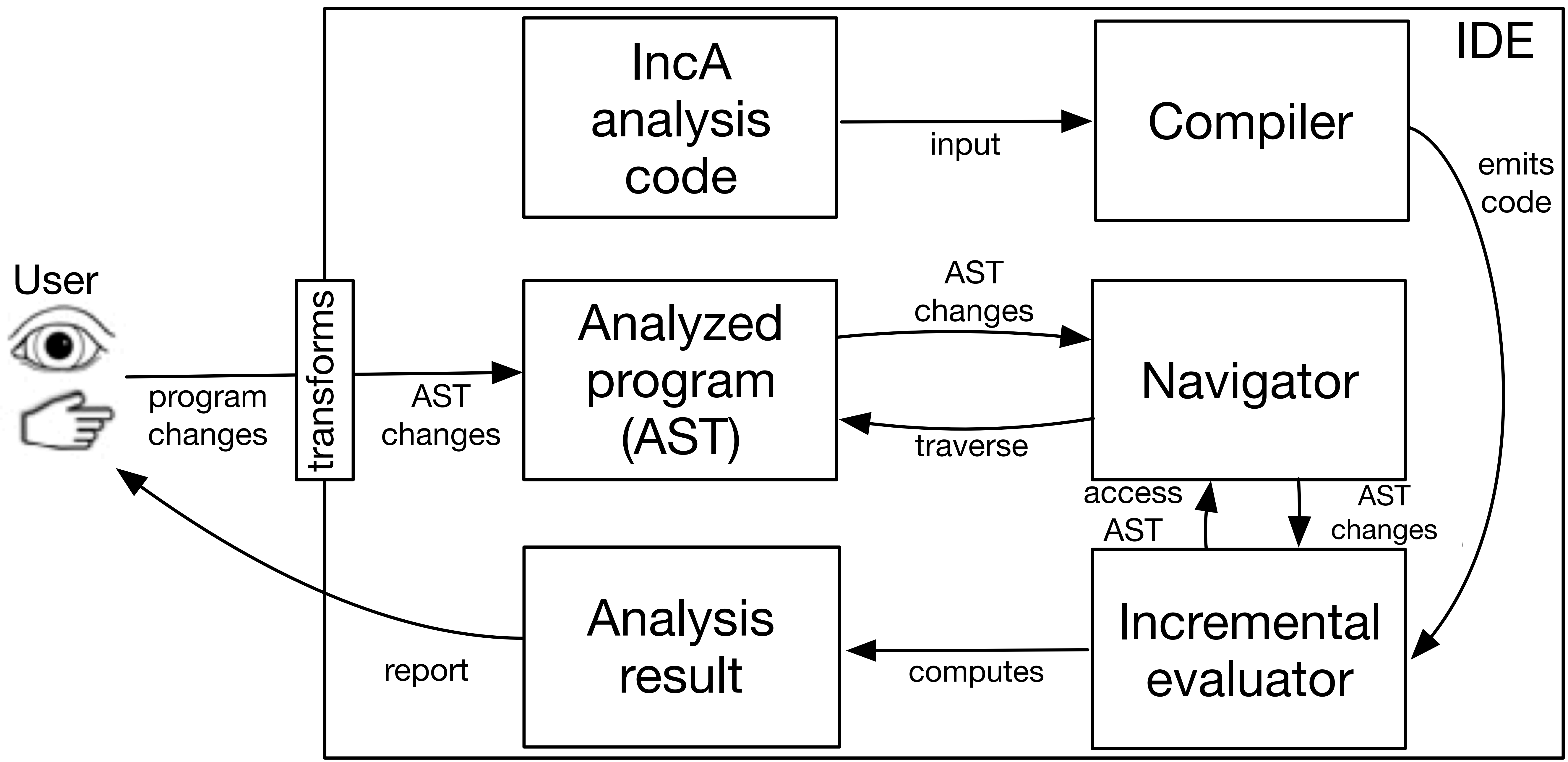
IncA - Architecture



IncA - Architecture



IncA - Architecture

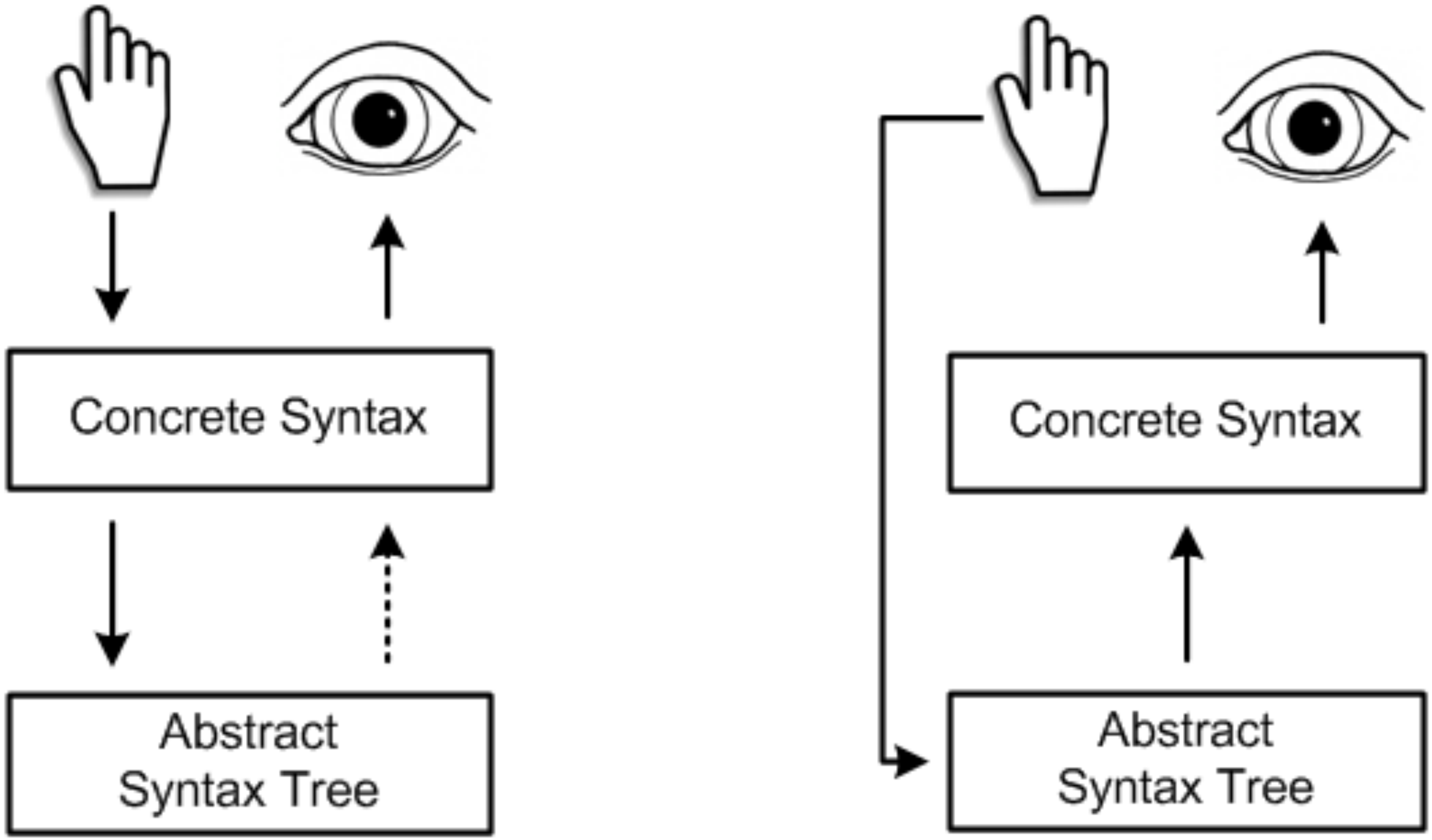


IncA - Implementation

Why MPS?

- Projectional editing is a perfect foundation for incrementalization!

IncA - Implementation



IncA - Implementation

Why MPS?

- Projectional editing is a perfect foundation for incrementalization!
- Use of DSLs for all language aspects is idiomatic

IncA - Implementation

Why MPS?

- Projectional editing is a perfect foundation for incrementalization!
- Use of DSLs for all language aspects is idiomatic

Other IDEs/LWBs?

- Efficiency depends on the granularity of incremental change notifications

IncA - Evaluation

Aspects:

- Run time performance
- Memory use

Method:

- Initialization
- Random changes

IncA - Evaluation

Aspects:

- Run time performance
- Memory use

Method:

- Initialization
- Random changes

Case studies and projects:

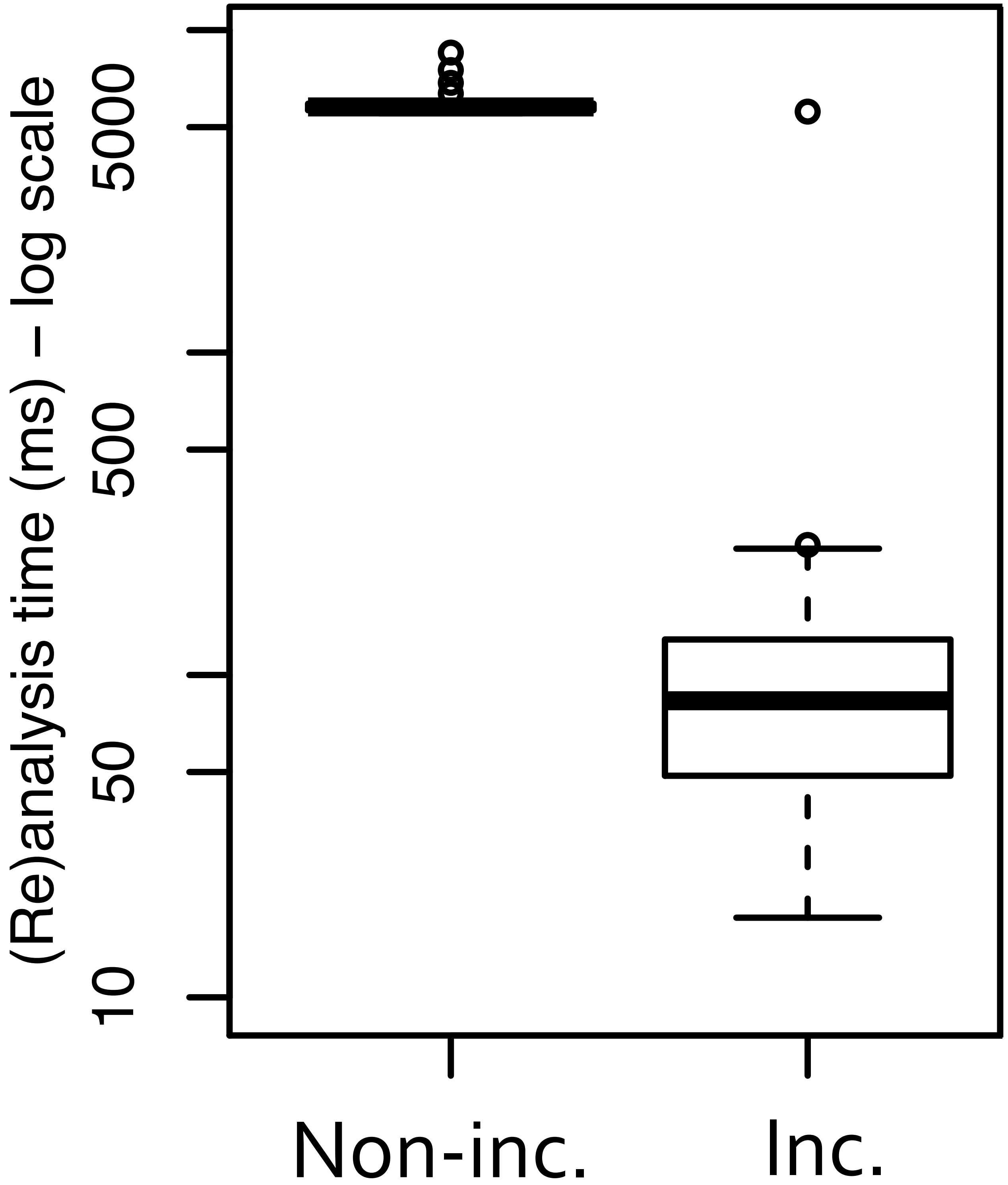
- CFG + points-to
- Well-formedness
- FindBugs

Toyota ITC (15 KLoC)

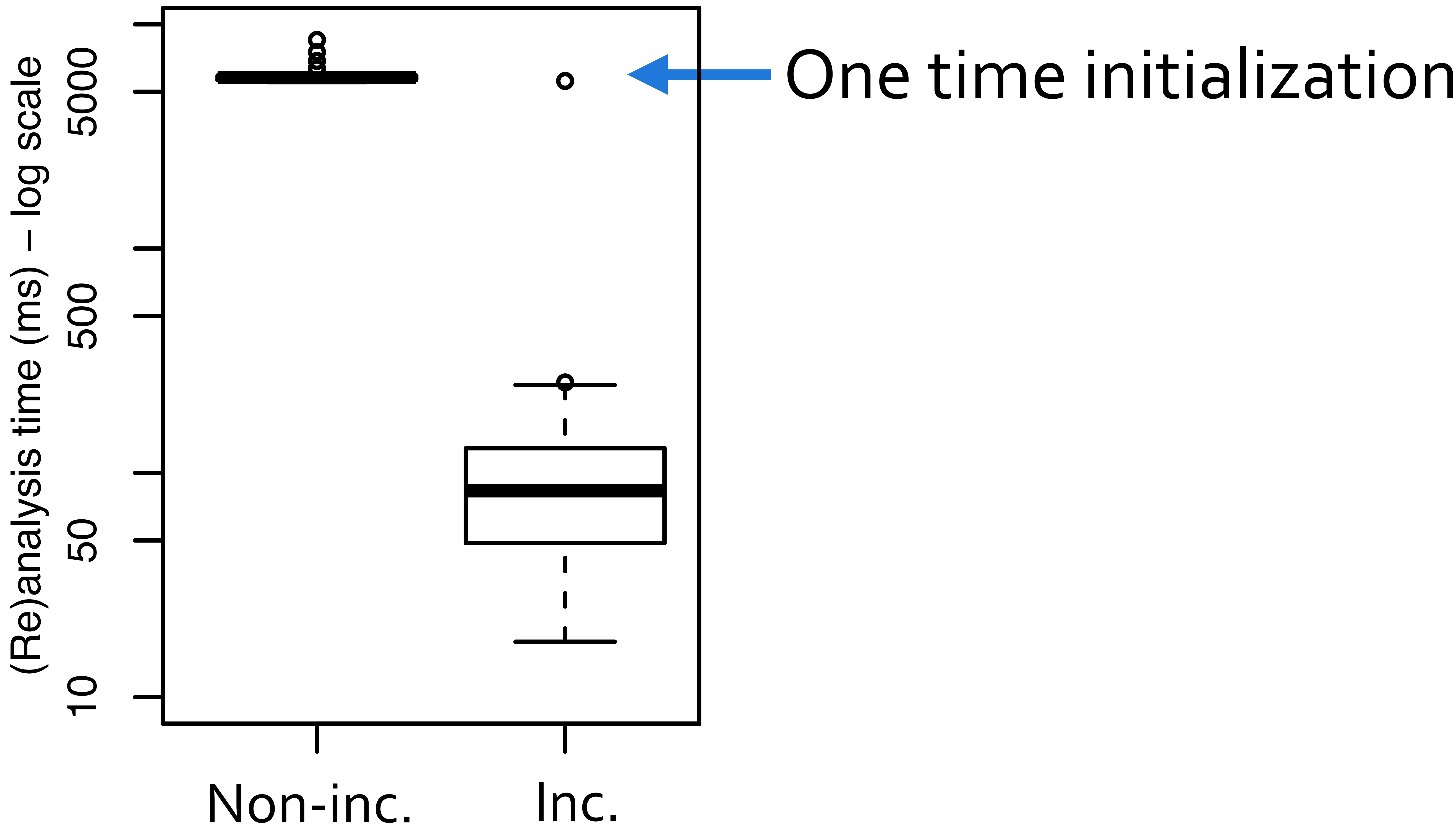
Smart Meter (44 KLoC)

mbeddr Importer (10 KLoC)

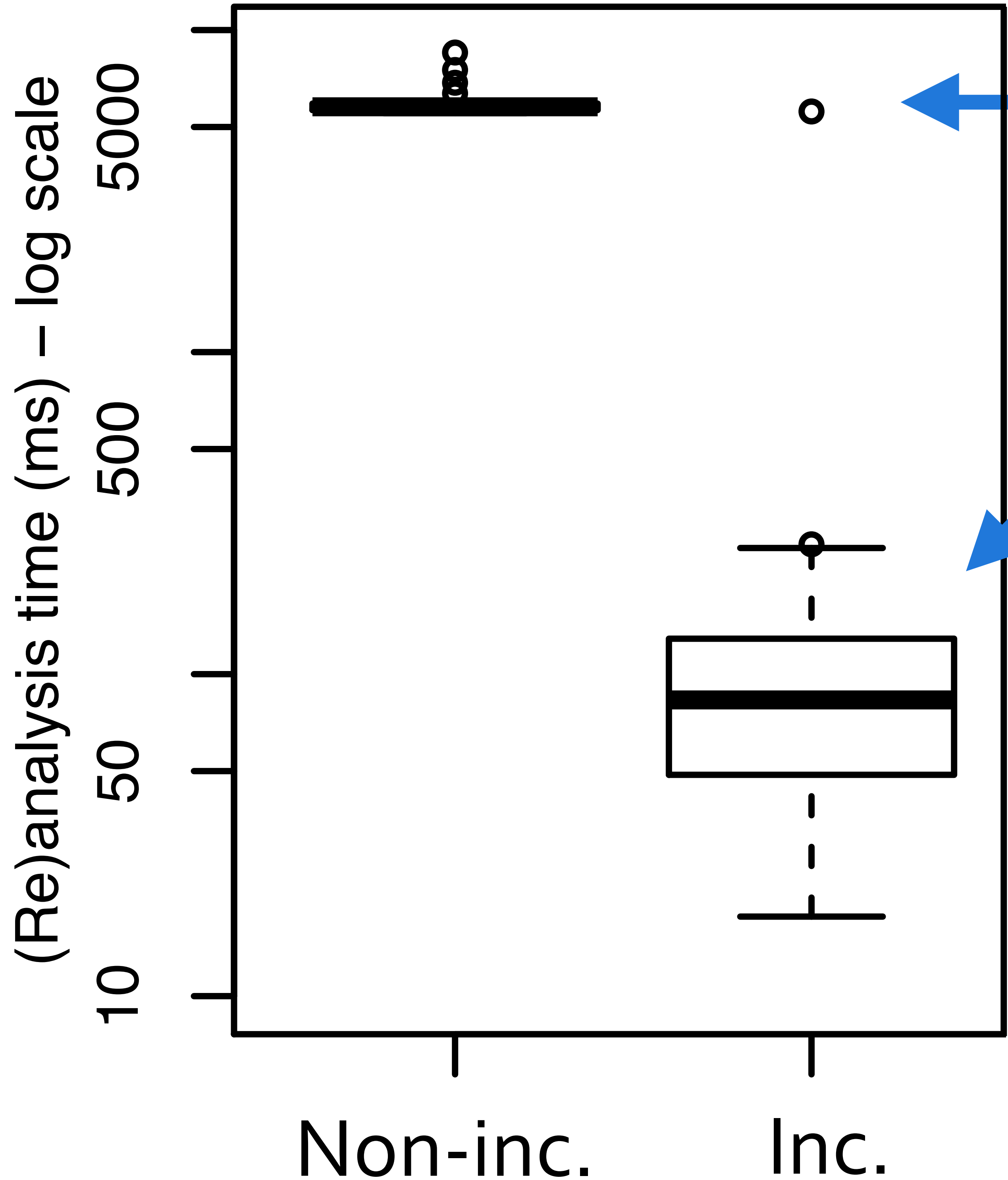
IncA - Points-to Run Times



IncA - Points-to Run Times



IncA - Points-to Run Times

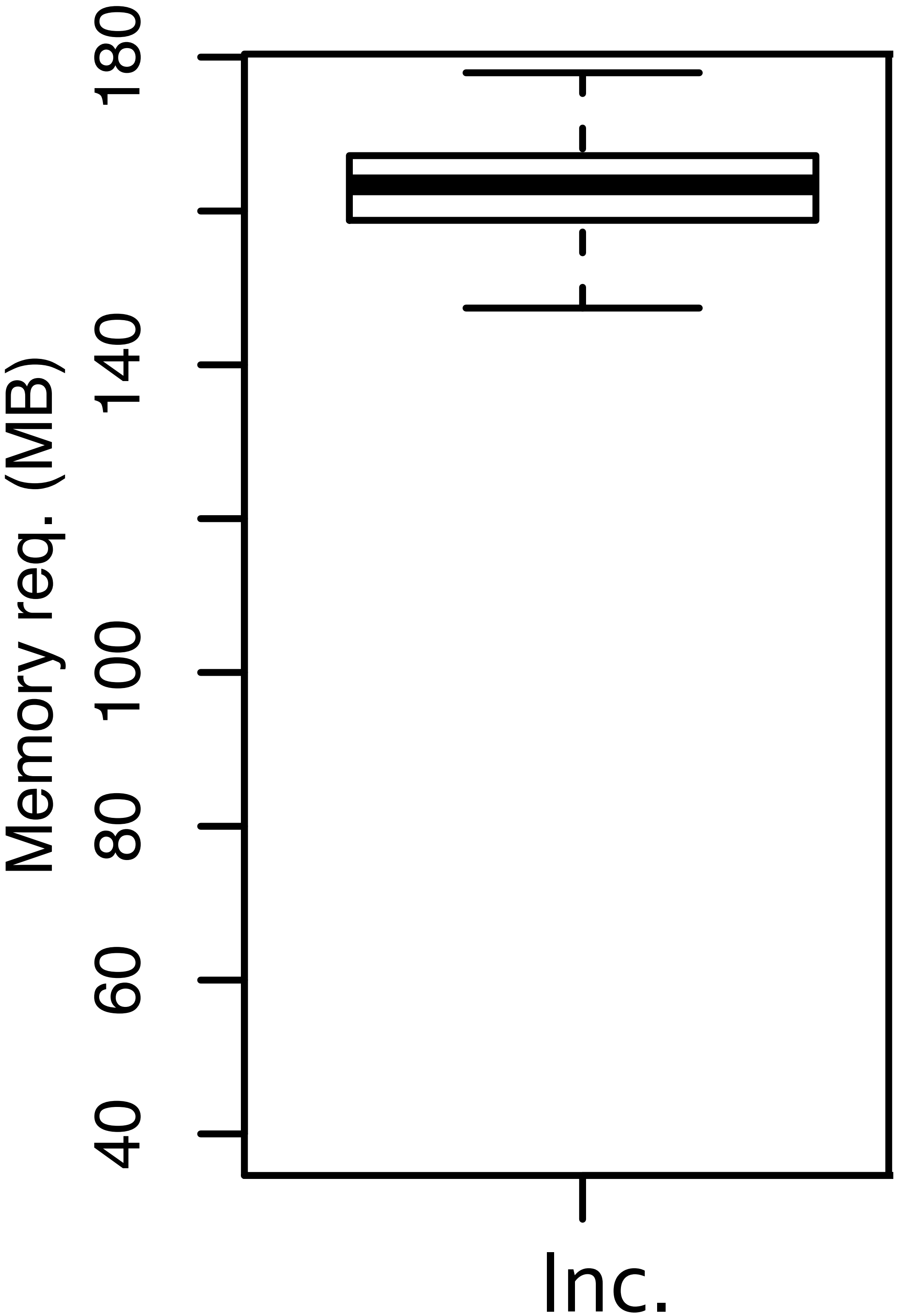


One time initialization

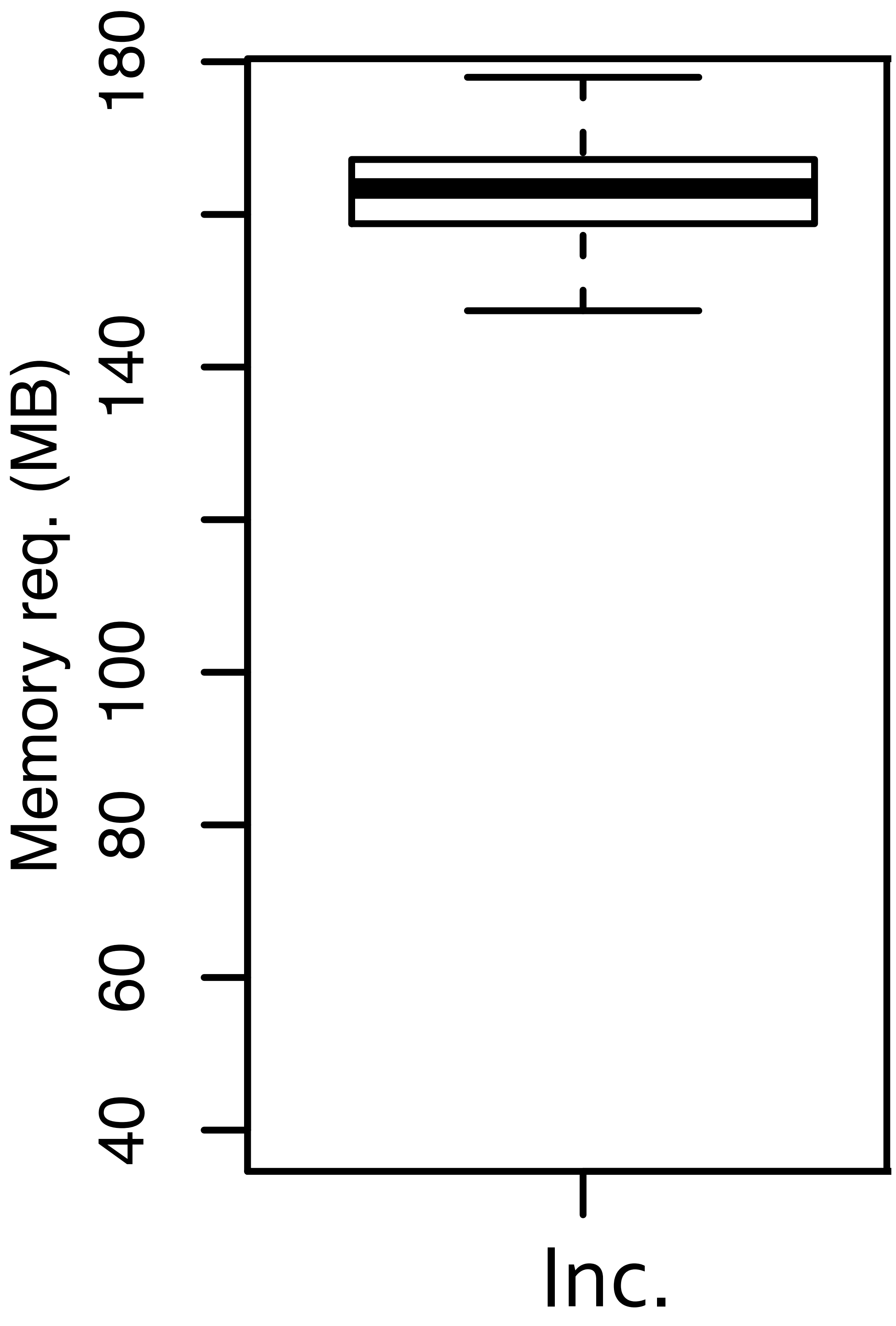
70x speedup after initialization

Considerable time savings compared to non-inc.

IncA - Well-form. Memory Use

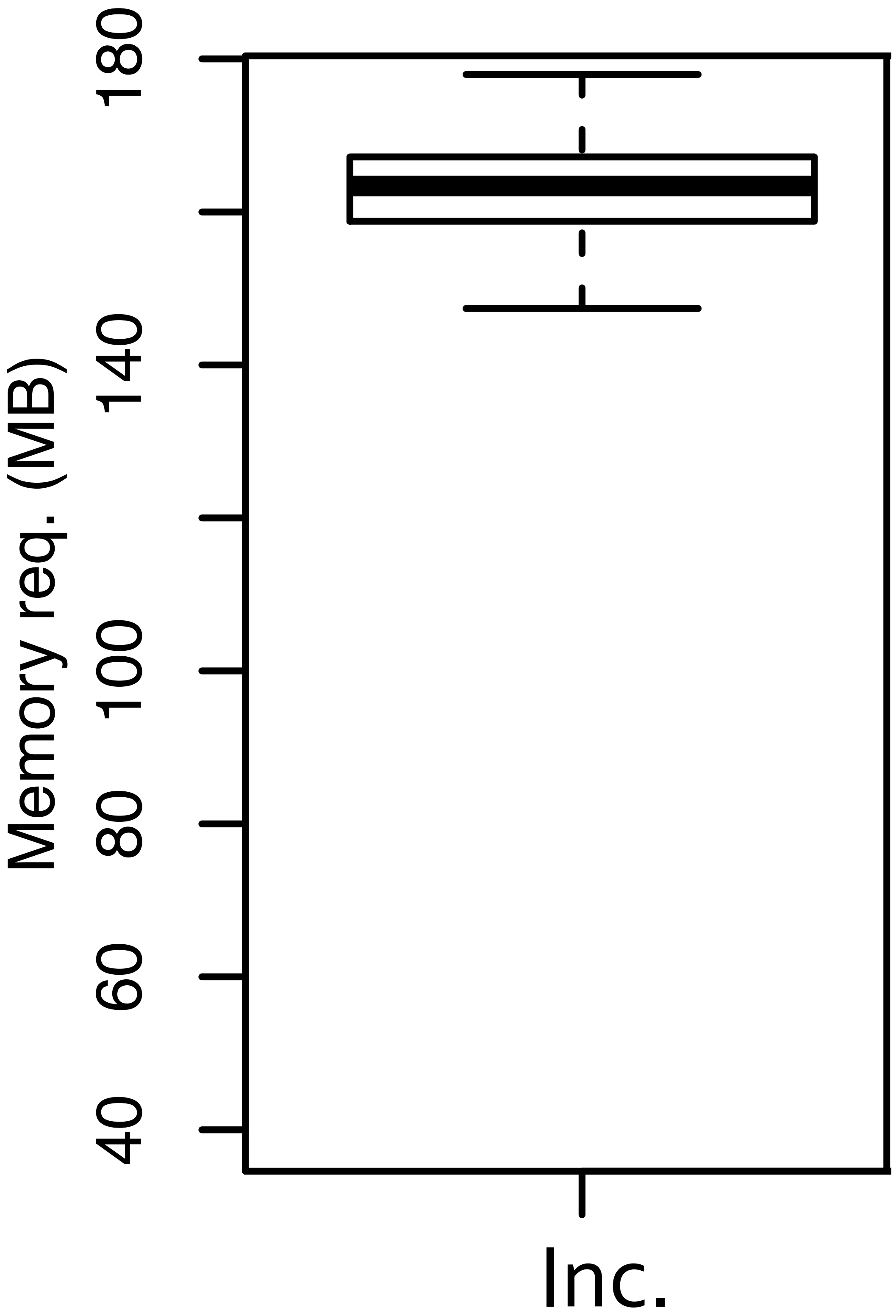


IncA - Well-form. Memory Use



Non-inc. does not perform caching.

IncA - Well-form. Memory Use



Non-inc. does not perform caching.

Typical workbench usage is around 1.2 GB.

Incrementalization induces an extra ~14 % of memory use.

IncA

These results already show the potential of incrementalization.

IncA

Can we do better?

IncA - Optimization

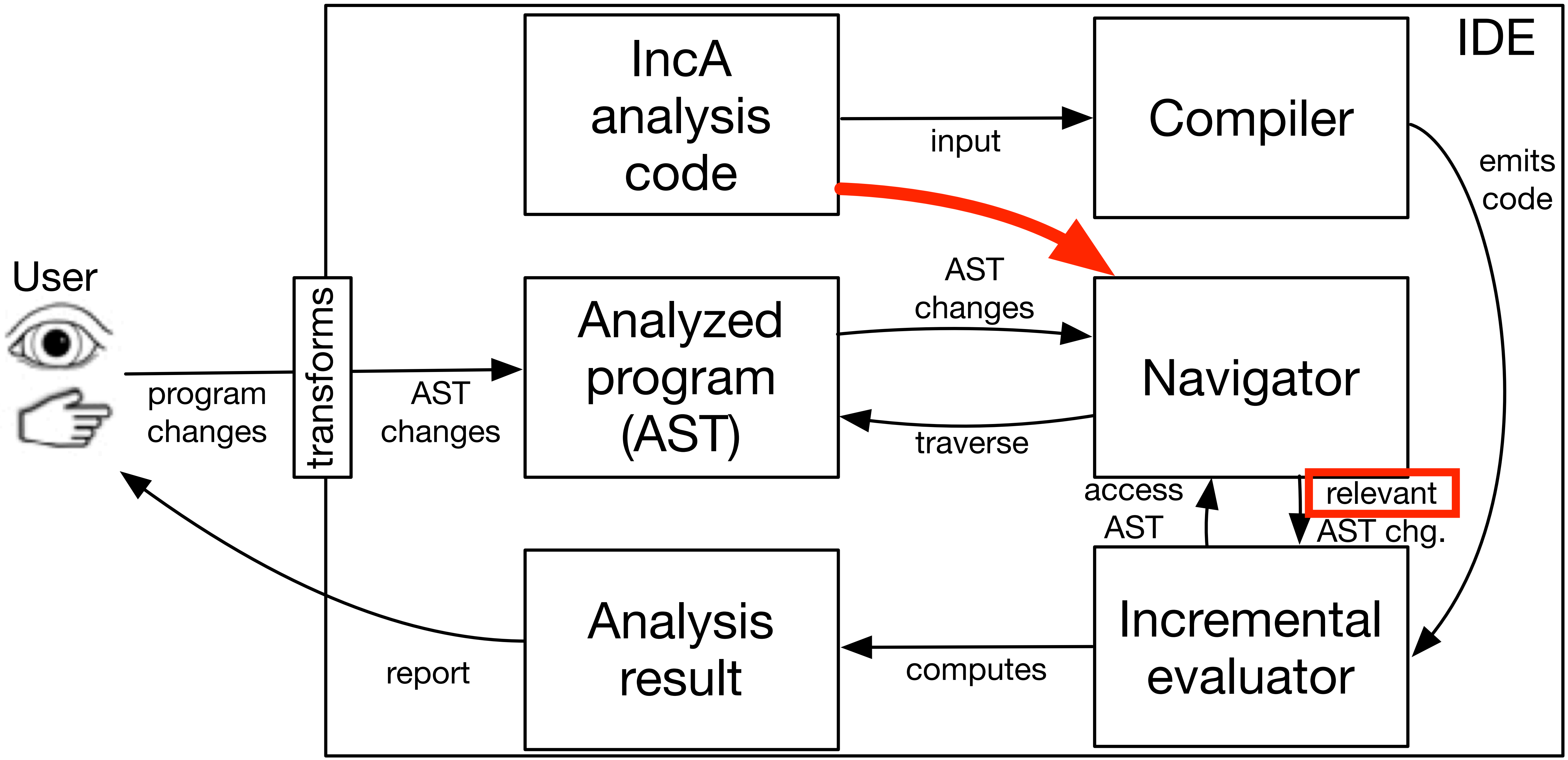
Differentiate relevant from irrelevant program changes.
An irrelevant program element does not need to be …

- processed -> save time!
- cached -> save memory!

The compiler performs an inter-procedural data-flow analysis on the analysis code.

Derive the most restrictive set of AST node types that can affect the analysis result.

IncA - Optimization



IncA - Optimization

Example: compute points-to targets (a = &b)

```
def pointsTo(s : Assignment) : (Var, Var) = {
  left := s.left
  right := s.right
  assert right instanceOf AddressOfExpr
  from := varInExpr(left)
  to := varInExpr(right.expr)
  return (from, to)
}
```

IncA - Optimization

```
private def varInExpr(e : Expression) : Var ={
  assert e instanceOf GlobalVarRef
  return e.var
} alt {
  assert e instanceOf LocalVarRef
  return e.var
}
```

IncA - Optimization

```
private def varInExpr(e : Expression) : Var ={
    assert e instanceOf GlobalVarRef
    return e.var
} alt {
    assert e instanceOf LocalVarRef
    return e.var
}
```

IncA - Optimization

Example: compute points-to targets (a = &b)

```
module pointsTo {
  def pointsTo(s : Assignment) : (Var, Var) = {
    left := s.left
    right := s.right
    assert right instanceOf AddressOfExpr
    from := varInExpr(left)
    to := varInExpr(right.expr)
    return (from, to)
  }

  private def varInExpr(e : Expression) : Var ={
    assert e instanceOf GlobalVarRef
    return e.var
  } alt {
    assert e instanceOf LocalVarRef
    return e.var
  }
}
```

right ∈ AddressOfExpr

left ∈ GlobalVarRef or LocalVarRef
from ∈ GlobalVar or LocalVar

right.expr ∈
GlobalVarRef or LocalVarRef
to ∈ GlobalVar or LocalVar

IncA - Optimization

Expression

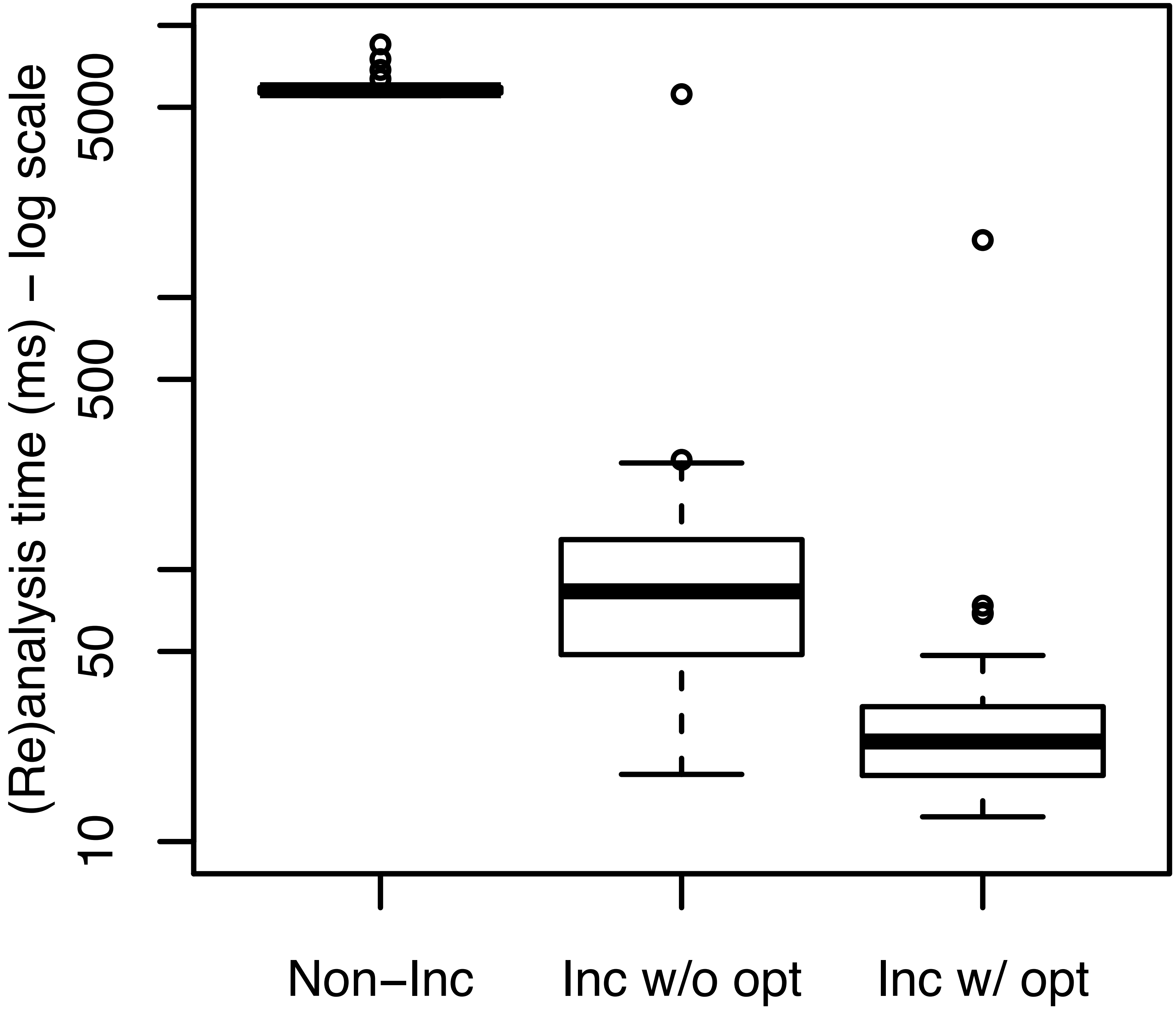
Assignment
GlobalVarRef
LocalVarRef
AddressOfExpr

Var

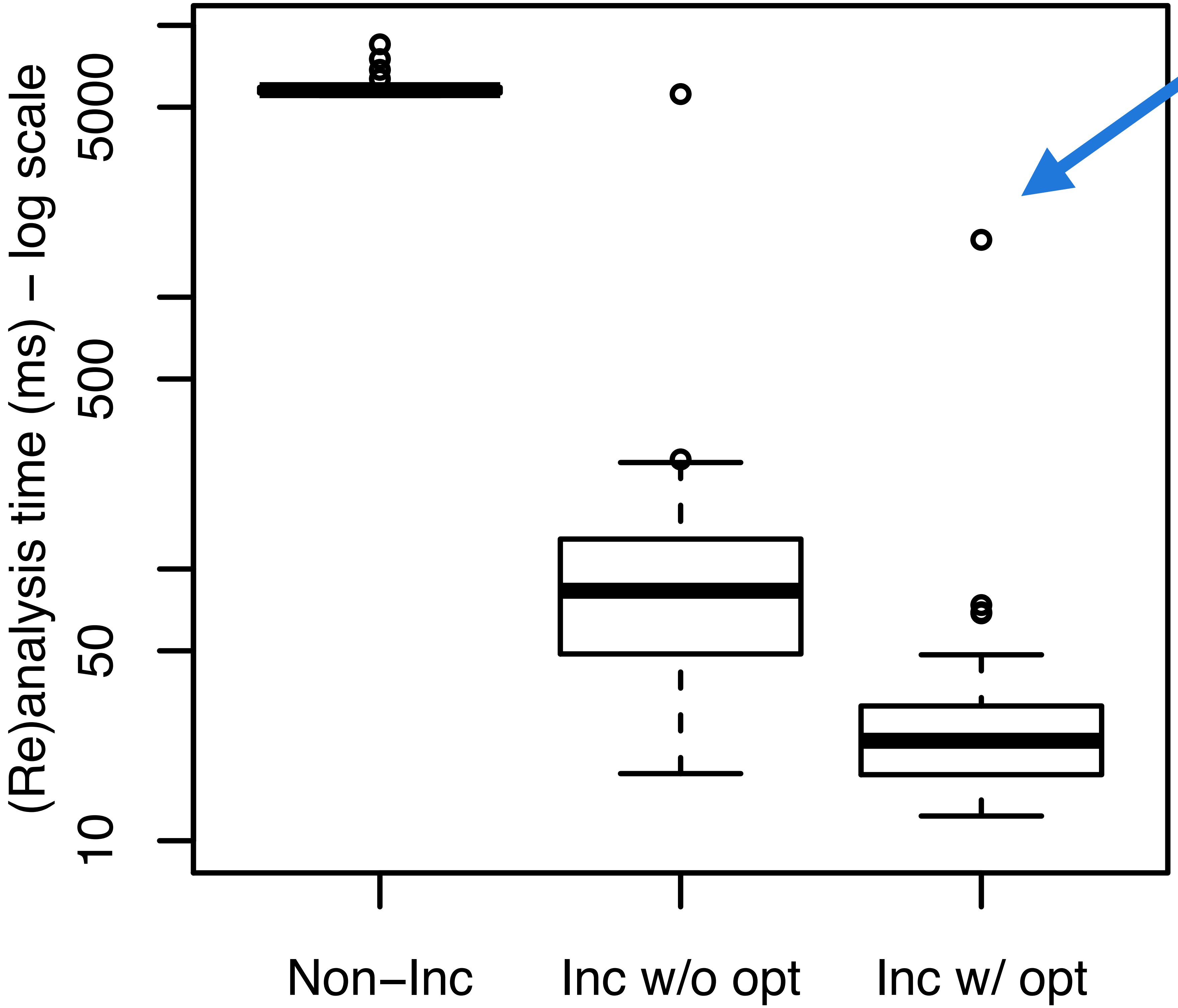
GlobalVar
LocalVar

Consider that mbeddr C & exts. come with few hundred different kinds of Expressions!

IncA - Points-to Run Times

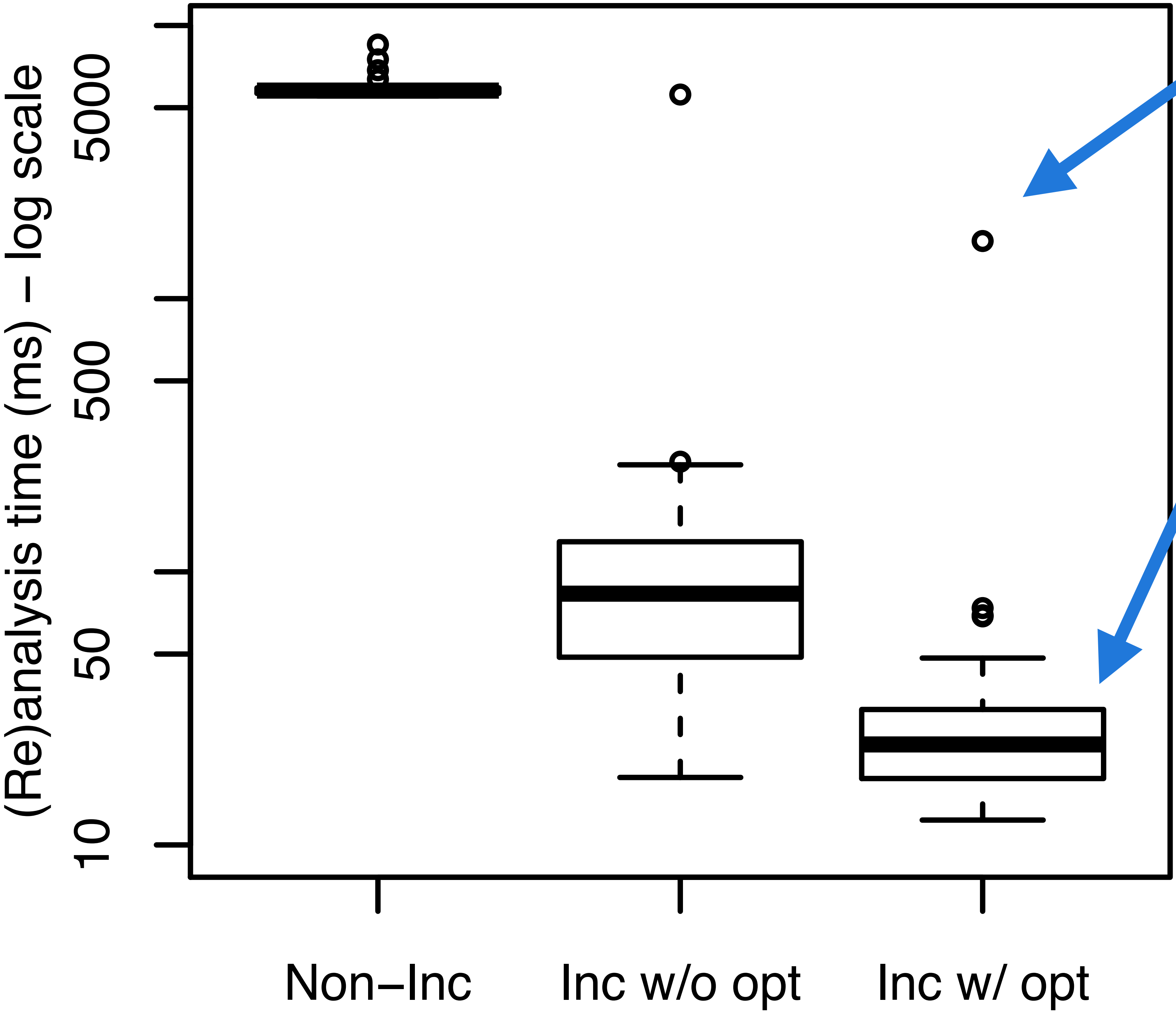


IncA - Points-to Run Times



Optimization effect on init.

IncA - Points-to Run Times

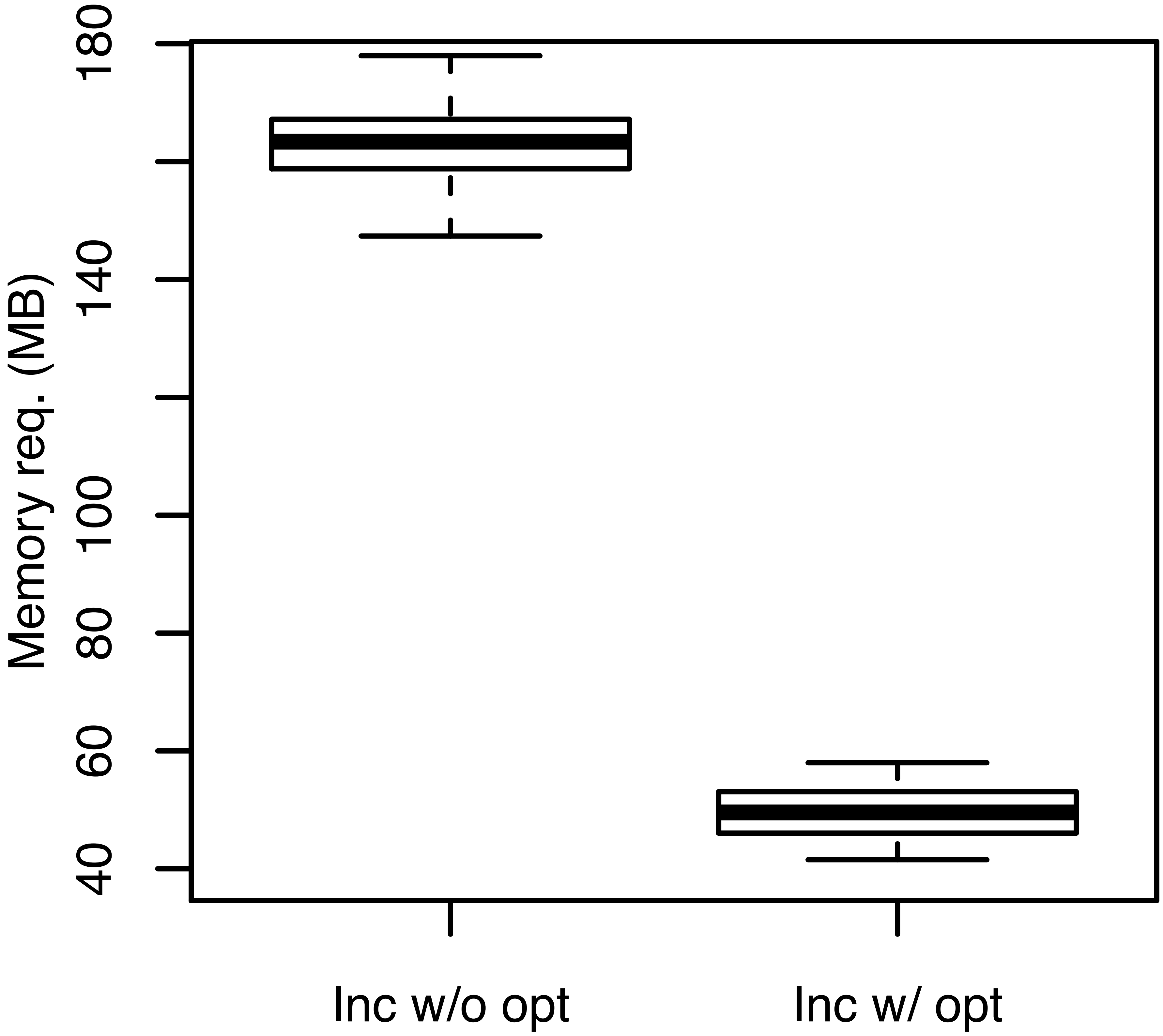


Optimization effect on init.

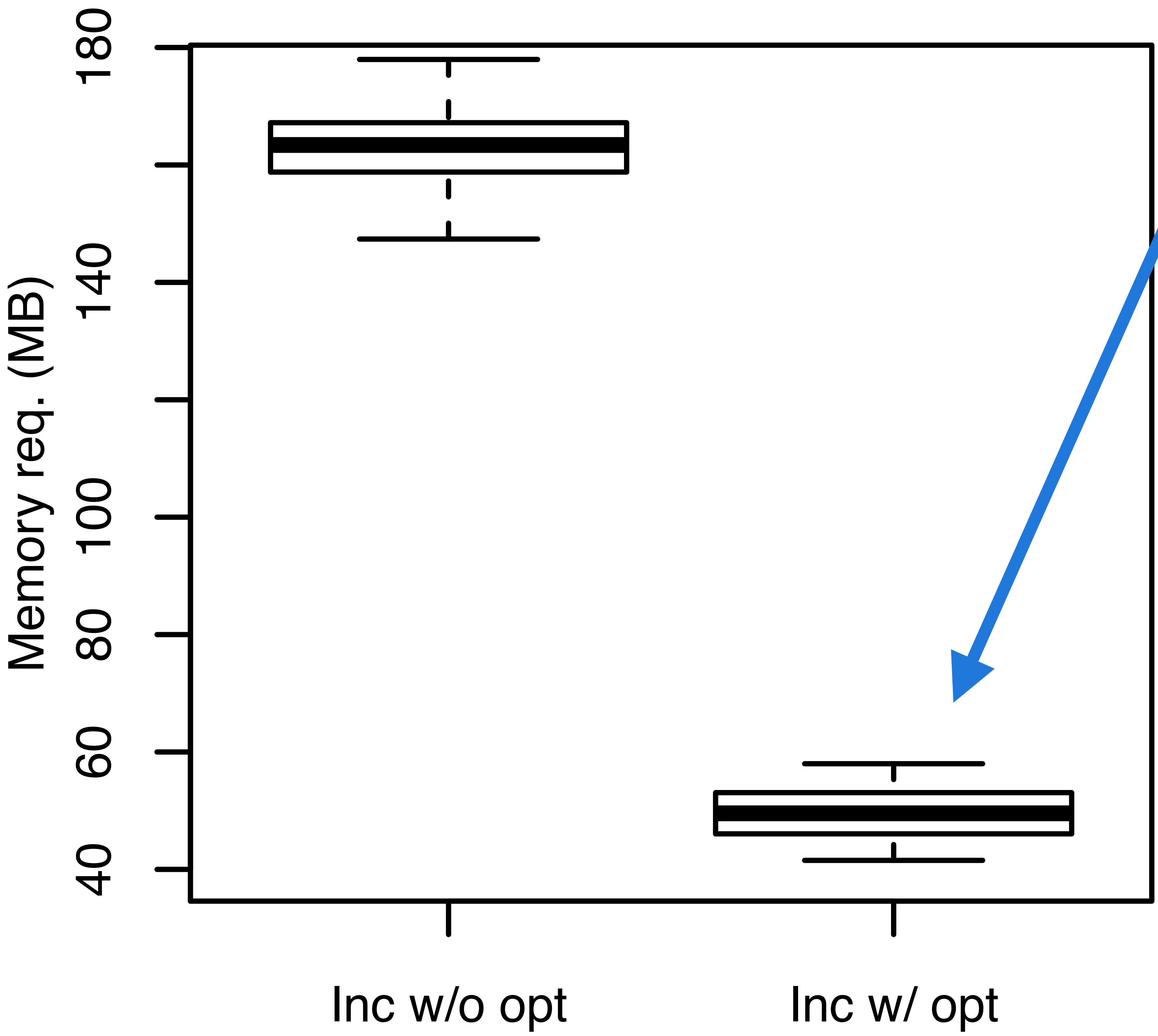
249x speedup after init.

Huge time savings compared to non-inc.

IncA - Well-form. Memory Use

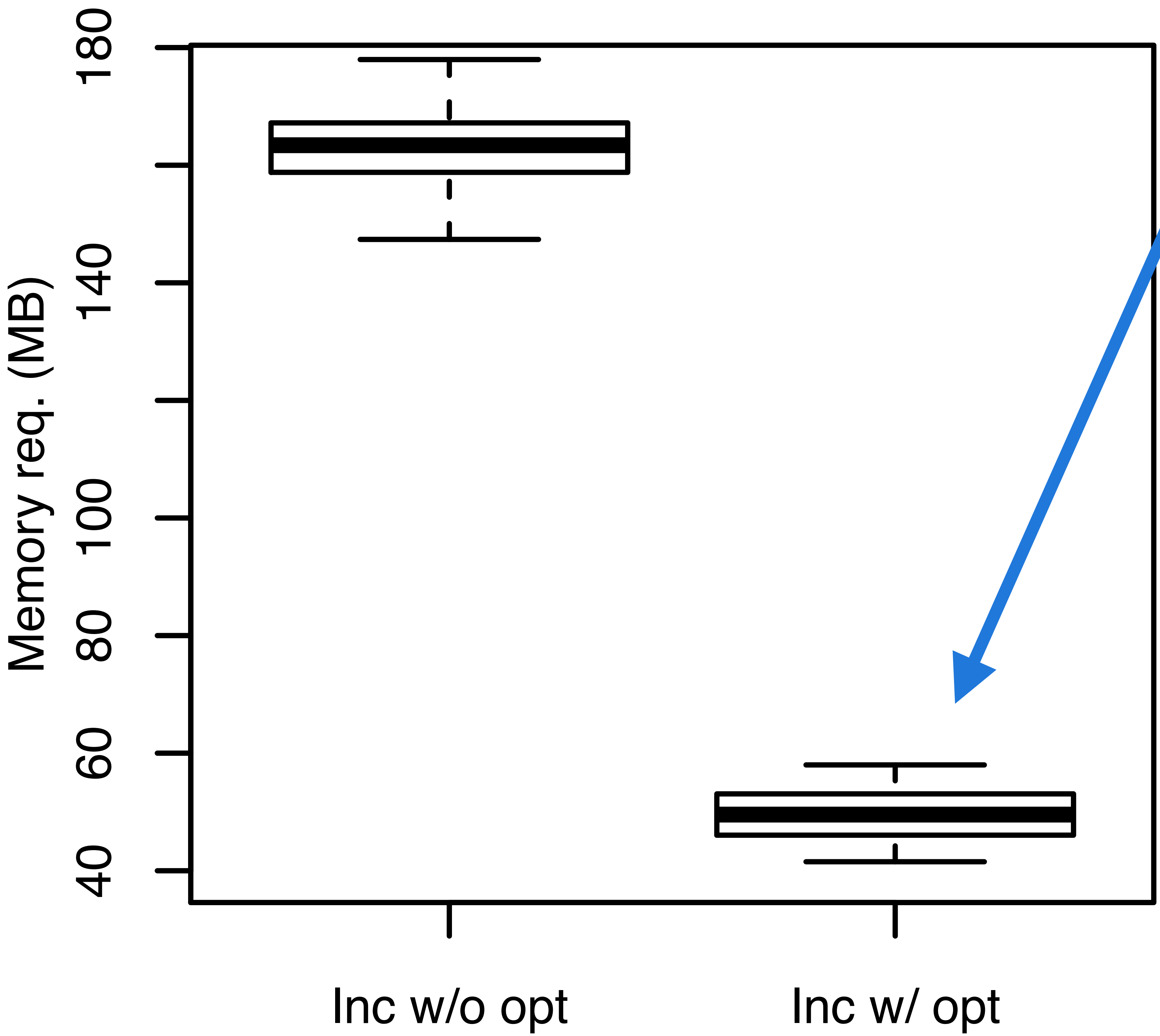


IncA - Well-form. Memory Use



69% memory reduction with optimization.

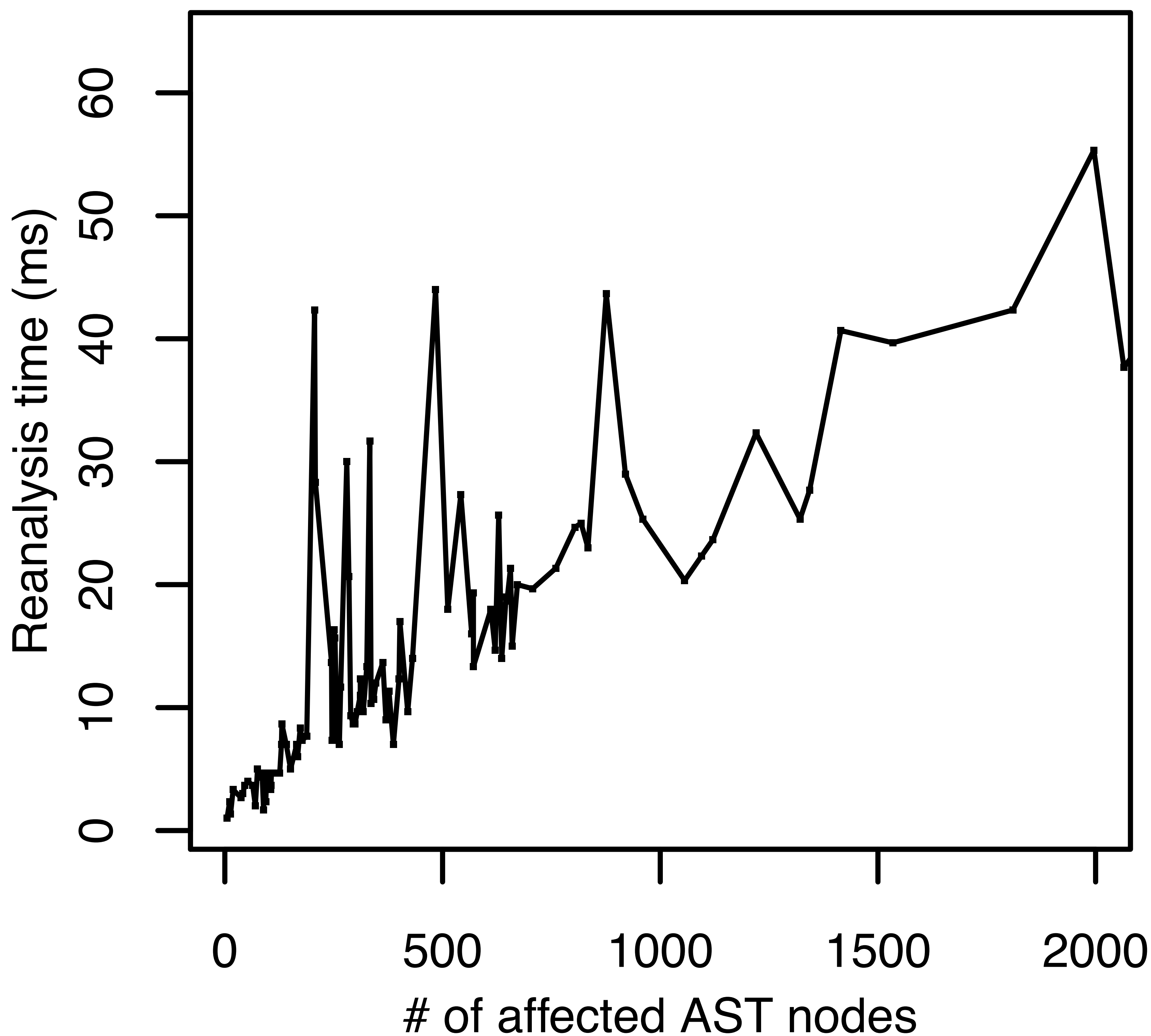
IncA - Well-form. Memory Use



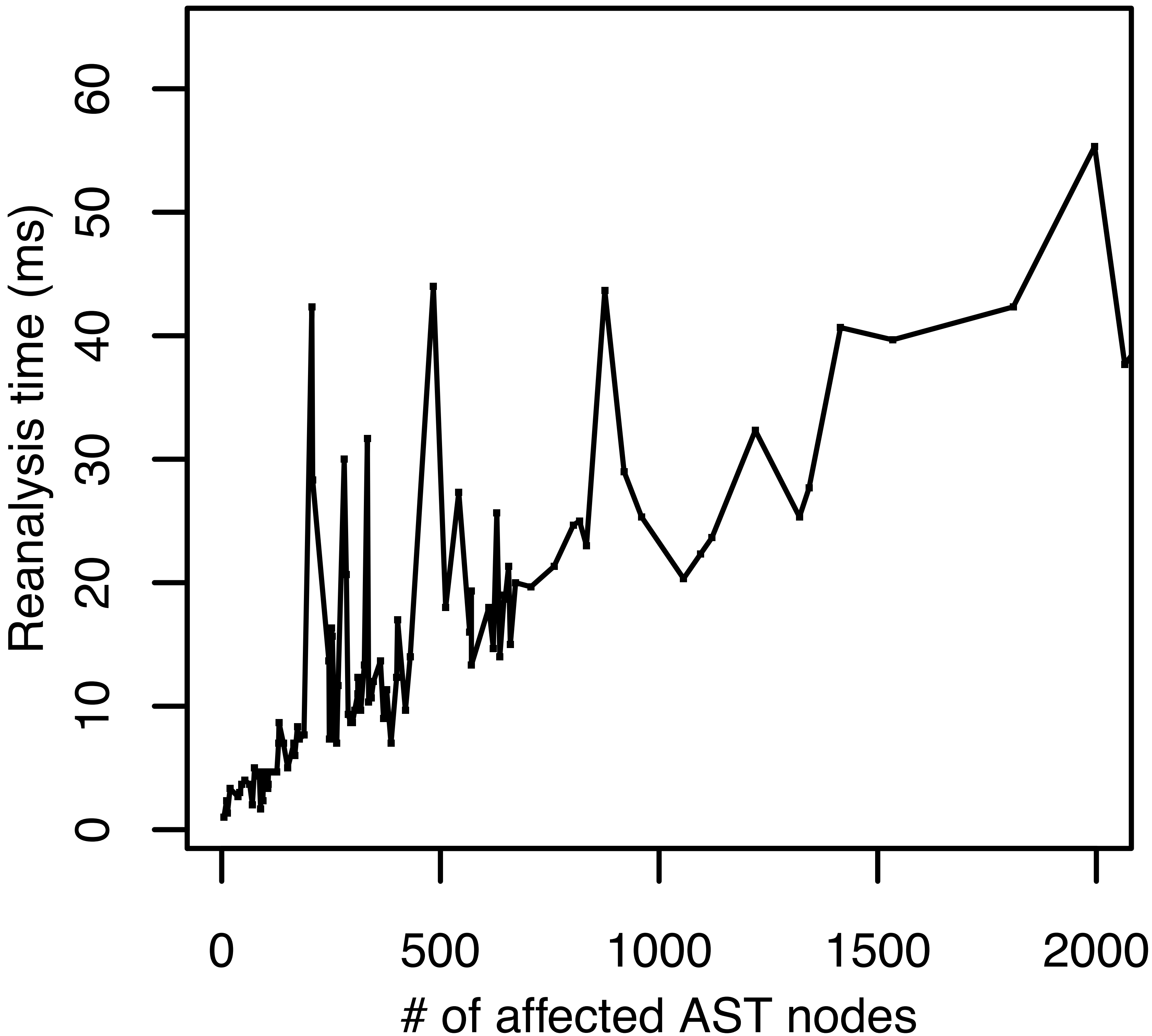
69% memory reduction with optimization.

Incrementalization overhead dropped down to ~4%.

IncA - Well-form. Scaling



IncA - Well-form. Scaling



Re-analysis times remain roughly linear with increasing change sizes.

Conclusions

- IncA DSL + runtime system for program analyses
- Generic optimization based on meta-analysis
- Several real-world case studies
- Significant speedups (up to 249X) for IncA analyses compared to non-inc.
- Considerable memory reduction (11-69%) due to optimization

Videos & docs: https://szabta89.github.io/projects/inca.html

Q & A

IncA - DSL Syntax

(module)	M	::=	**module** N **import** $\overline{m}$ { $\overline{F}$ }
(function)	F	::=	**def** N($\overline{Var : T_{in}}$) : T_{out} = $\overline{A}$
(alternative)	A	::=	$\overline{S}$
(statement)	S	::=	$\overline{Var}$:= E | **assert** C | **return** E
(condition)	C	::=	E == E | E != E | E **instanceOf** T | E **not instanceOf** T | **undef** E
(expression)	E	::=	Var | Val | E.N | f($\overline{E}$) | f^{+}(E)
(value)	Val	::=	number | string | enum | boolean
(variable)	Var	::=	N
(type)	T	::=	N
(name)	N	::=	<name>

Relation to Datalog

Datalog is widely used for program analyses

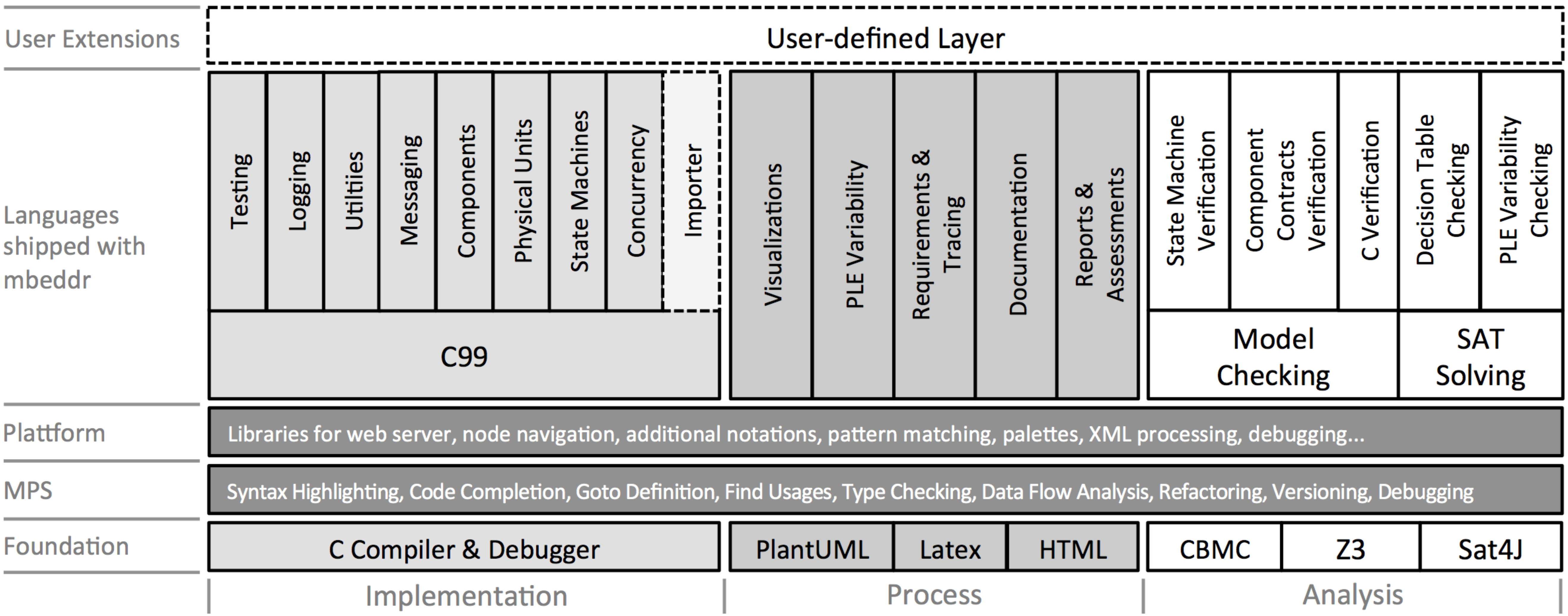
- There are specialized algorithms for incrementalizing Datalog
- But our system builds on existing incremental evaluators!

Expressive power? - FO(LFP) for IncA, what about Datalog?

Scalable Datalog backend? (LogicBlox, QL by Semmle)

Experience with customer projects shows that developers would rather rely on familiar abstractions (direction from input to output, assignments, functions).

The mbeddr stack



IncA - Case Studies

A wide variety of program analyses in the context of the mbeddr IDE

- Control flow and points-to analyses for C
- Well-formedness checks for DSLs
- FindBugs for Java
- Enforcement of secure coding standards (Misra, CERT)*