Finding Application Errors and Security Flaws Using PQL: A Program Query Language

Michael Martin, Benjamin Livshits,
Monica S. Lam
Stanford University
OOPSLA 2005

The Problem

- Lots of bug-finding research
  - Null dereferences
  - Buffer overruns
  - Data races
- Many bugs application-specific
  - Misuse of libraries
  - Violation of application logic

Solution: Division of Labor

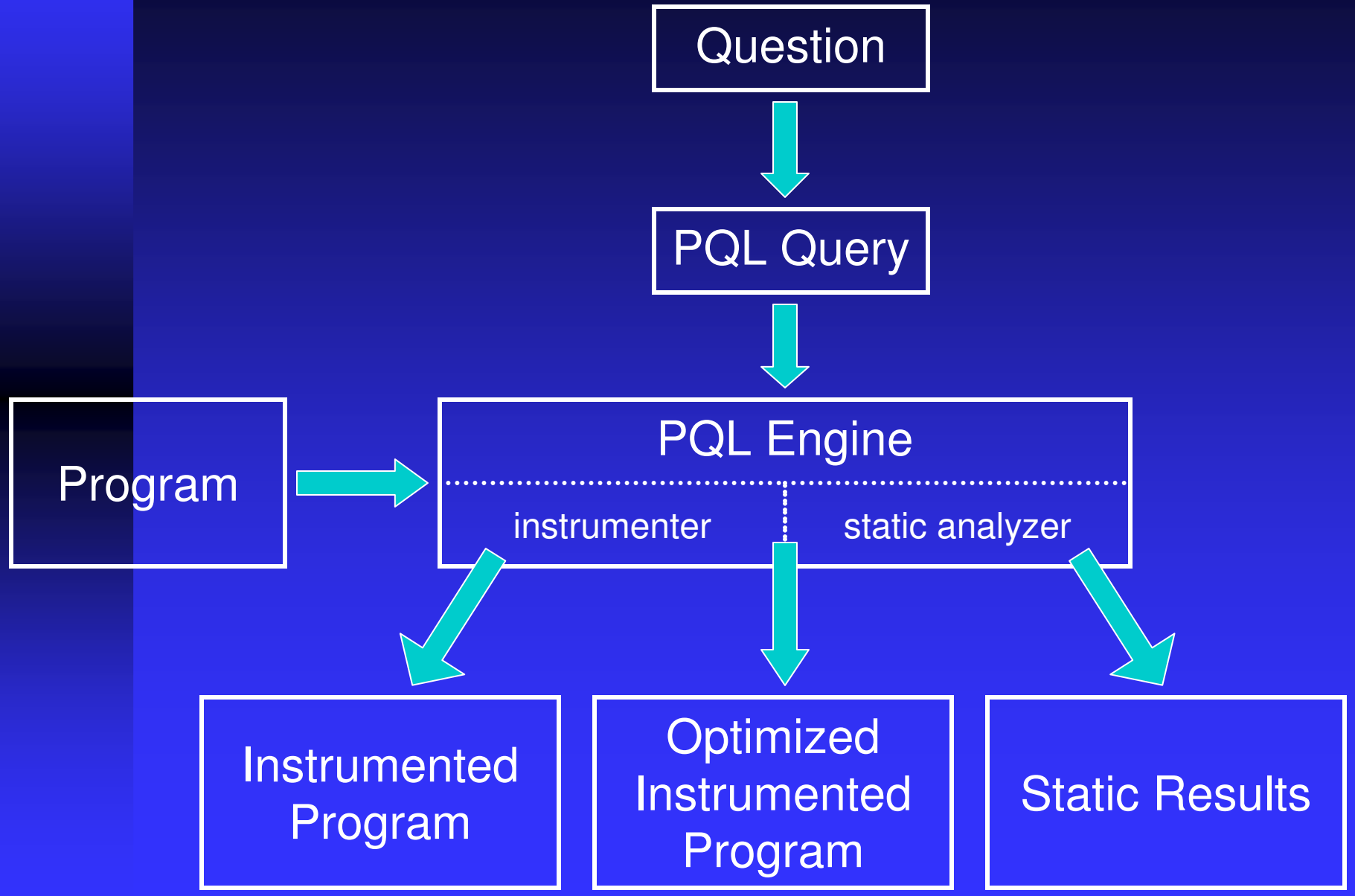
- Programmer
  - Knows target program
  - Doesn’t know analysis
- Program Analysis Specialists
  - Knows analysis
  - Doesn’t know specific bugs

Give the programmer a usable analysis

Program Query Language

- Queries operate on *program traces*
  - Sequence of events representing a run
  - Refers to object *instances*, not variables
  - Events matched may be widely spaced
- Patterns resemble Java code
  - Like a small matching code snippet
  - No references to compiler internals

System Architecture



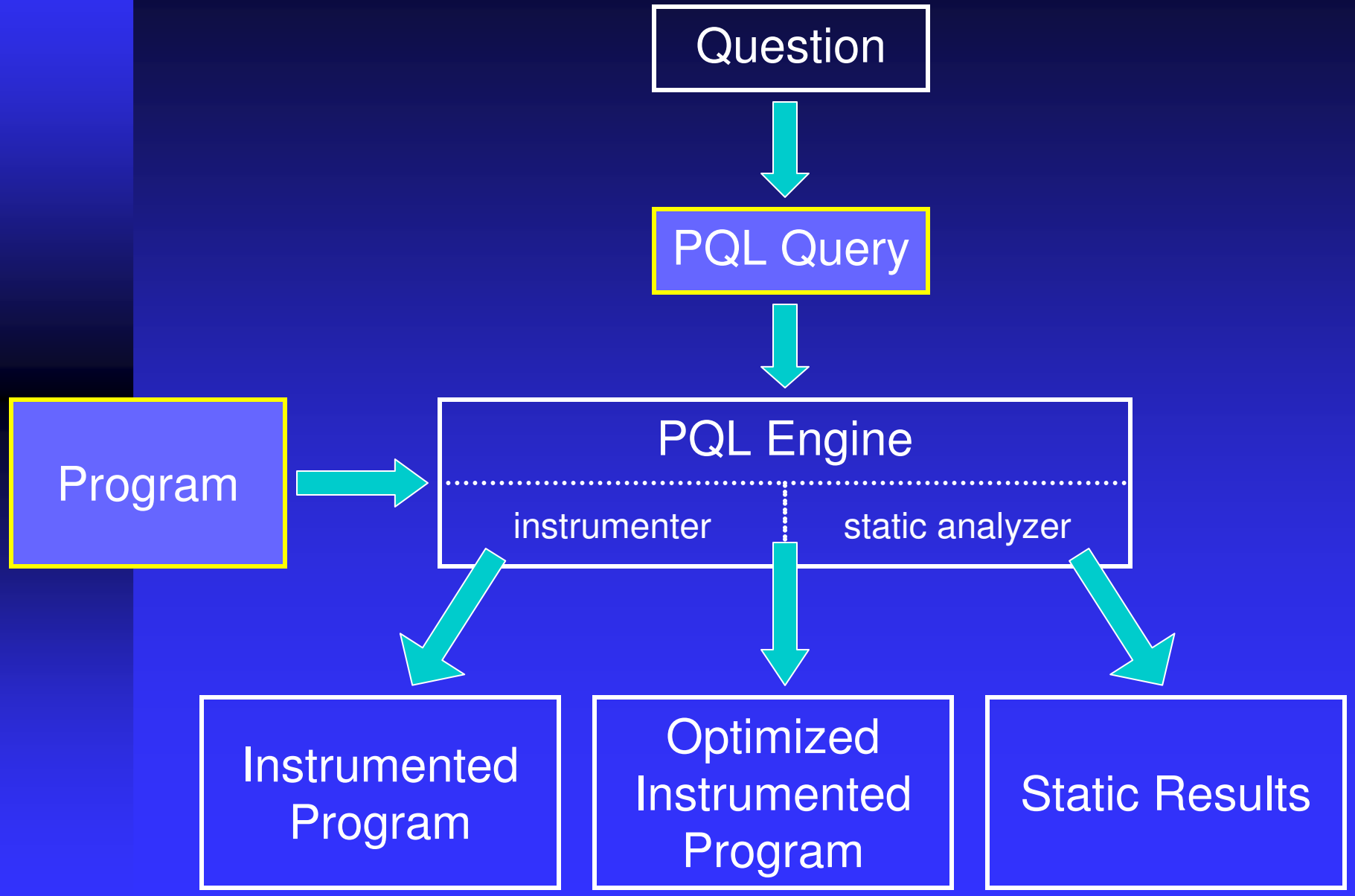
Complementary Approaches

- Dynamic analysis: finds matches at run time
  - After a match:
    - Can execute user code
    - Can fix code by replacing instructions
- Static analysis: finds all possible matches
  - Conservative: can prove lack of match
  - Results can optimize dynamic analysis

Experimental Results

- Explored a wide range of PQL queries
  - Bad session stores (API violations)
  - SQL injections (security flaws)
  - Mismatched calls (API violations)
  - Lapsed listeners (memory leaks)
- Automatically repaired and prevented many bugs at runtime
  - Fixed memory leaks, prevented security flaws
- Runtime overhead is reasonable
  - Overhead in the 9-125% range
  - Static optimization removed 82-99% of instr. points
- Found 206 bugs in 6 real-life Java applications
  - Eclipse, popular web applications
  - 60,000 classes combined

System Architecture



Running Example: SQL Injection

- Unvalidated user input passed to a database back-end for execution
- If SQL in embedded in the input, attacker can take over database
  - Unauthorized data reads
  - Unauthorized data modifications
- One of the top web security flaws

SQL Injection 1

```
HttpServletRequest req = /* ... */;
java.sql.Connection conn = /* ... */;
String q = req.getParameter("QUERY");
conn.execute(q);
```

```
1   CALL   o1.getParameter(o2)
2   RET    o2
3   CALL   o3.execute(o2)
4   RET    o4
```

SQL Injection 2

```
String read() {
  HttpServletRequest req = /* ... */;
    return req.getParameter("QUERY");
}
/* ... */
java.sql.Connection conn = /* ... */;
conn.execute(read());
```

1	CALL	read()
2	CALL	o_1.getParameter(o_2)
3	RET	o_3
4	RET	o_3
5	CALL	o_4.execute(o_3)
6	RET	o_5

Essence of Pattern the Same

```
1 CALL o1.getParameter(o2)
2 RET  o3

3 CALL o4.execute(o3)
4 RET  o5
```

```
1 CALL read()
2 CALL o1.getParameter(o2)
3 RET  o3
4 RET  o3
5 CALL o4.execute(o3)
6 RET  o5
```

The object *returned by* **`getParameter`** is then *argument 1 to* **`execute`**

Translates Directly to PQL

```
query main()
uses String x;
matches {
    x = HttpServletRequest.getParameter(_);

    Connection.execute(x);
}
```

- Query variables → heap objects
- Instructions need not be adjacent in trace

Alternation

```
query main()
uses String x;
matches {
    x = HttpServletRequest.getParameter(_)
  | x = HttpServletRequest.getHeader(_);
    Connection.execute(x);
}
```

SQL Injection 3

```
HttpServletRequest req = /* ... */;
n = getParameter("NAME");
p = getParameter("PASSWORD");
conn.execute(
        "SELECT * FROM logins WHERE name=" +
        n +
        " AND passwd=" +
        p
    );
```

- Compiler translates string concatenation into operations on `String` and `StringBuffer` objects

SQL Injection 3

```
1    CALL    o1.getParameter(o2)
2    RET     o3
3    CALL    o1.getParameter(o4)
4    RET     o5
5    CALL    StringBuffer.<init>(o6)
6    RET     o7
7    CALL    o7.append(o8)
8    RET     o7
9    CALL    o7.append(o3)
10   RET     o7
11   CALL    o7.append(o9)
12   RET     o7
13   CALL    o7.append(o5)
14   RET     o7
15   CALL    o7.toString()
16   RET     o10
17   CALL    o11.execute(o10)
18   RET     o12
```

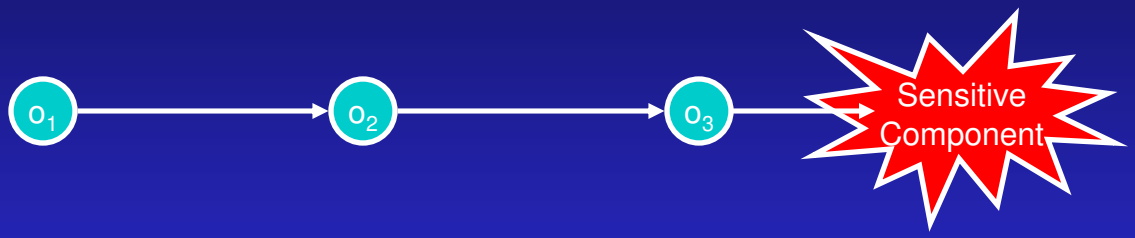
Old Pattern Doesn't Work

```
1    CALL    o1.getParameter(o2)
2    RET     o3
3    CALL    o1.getParameter(o4)
4    RET     o5

17   CALL    o11.execute(o10)
```

- o_{10} is neither o_3 nor o_5, so no match

Instance of Tainted Data Problem



- User-controlled input must be trapped and validated before being passed to database
- Objects derived from an initial input must also be considered user controlled
- Generalizes to many security problems: cross-site scripting, path traversal, response splitting, format string attacks...

Pattern Must Catch Derived `Strings`

```
1     CALL      o1.getParameter(o2)
2     RET       o3
3     CALL      o1.getParameter(o4)
4     RET       o5

9     CALL      o7.append(o3)
10    RET       o7
11    CALL      o7.append(o9)
12    RET       o7

15    CALL      o7.toString()
16    RET       o10
17    CALL      o11.execute(o10)
```

Pattern Must Catch Derived `Strings`

```
1    CALL    o1.getParameter(o2)
2    RET     o3
3    CALL    o1.getParameter(o4)
4    RET     o5

9    CALL    o7.append(o3)
10   RET     o7
11   CALL    o7.append(o9)
12   RET     o7

15   CALL    o7.toString()
16   RET     o10
17   CALL    o11.execute(o10)
```

Pattern Must Catch Derived `Strings`

```
1     CALL     o1.getParameter(o2)
2     RET      o3
3     CALL     o1.getParameter(o4)
4     RET      o5

9     CALL     o7.append(o3)
10    RET      o7
11    CALL     o7.append(o9)
12    RET      o7

15    CALL     o7.toString()
16    RET      o10
17    CALL     o11.execute(o10)
```

Derived String Query

```
query derived (Object x)
  uses Object temp;
  returns Object d;
matches {
  { temp.append(x); d := derived(temp); }
| { temp = x.toString(); d := derived(temp); }
| { d := x; }
}
```

New Main Query

```
query main()
uses String x, final;
matches {
    x = HttpServletRequest.getParameter(_)
  | x = HttpServletRequest.getHeader(_);
    final := derived(x);
    Connection.execute(final);
}
```

Defending Against Attacks

```
query main()
uses String x, final;
matches {
    x = HttpServletRequest.getParameter(_)
  | x = HttpServletRequest.getHeader(_);
    final := derived(x);
}
replaces Connection.execute(final) with
    SQLUtil.safeExecute(x, final);
```

- Sanitizes user-derived input
- Dangerous data cannot reach the database

Other PQL Constructs

- Partial order
  - `{ x.a(), x.b(), x.c(); }`
  - Match calls to `a`, `b`, and `c` on `x` in *any* order.
- Forbidden Events
  - Example: double-lock

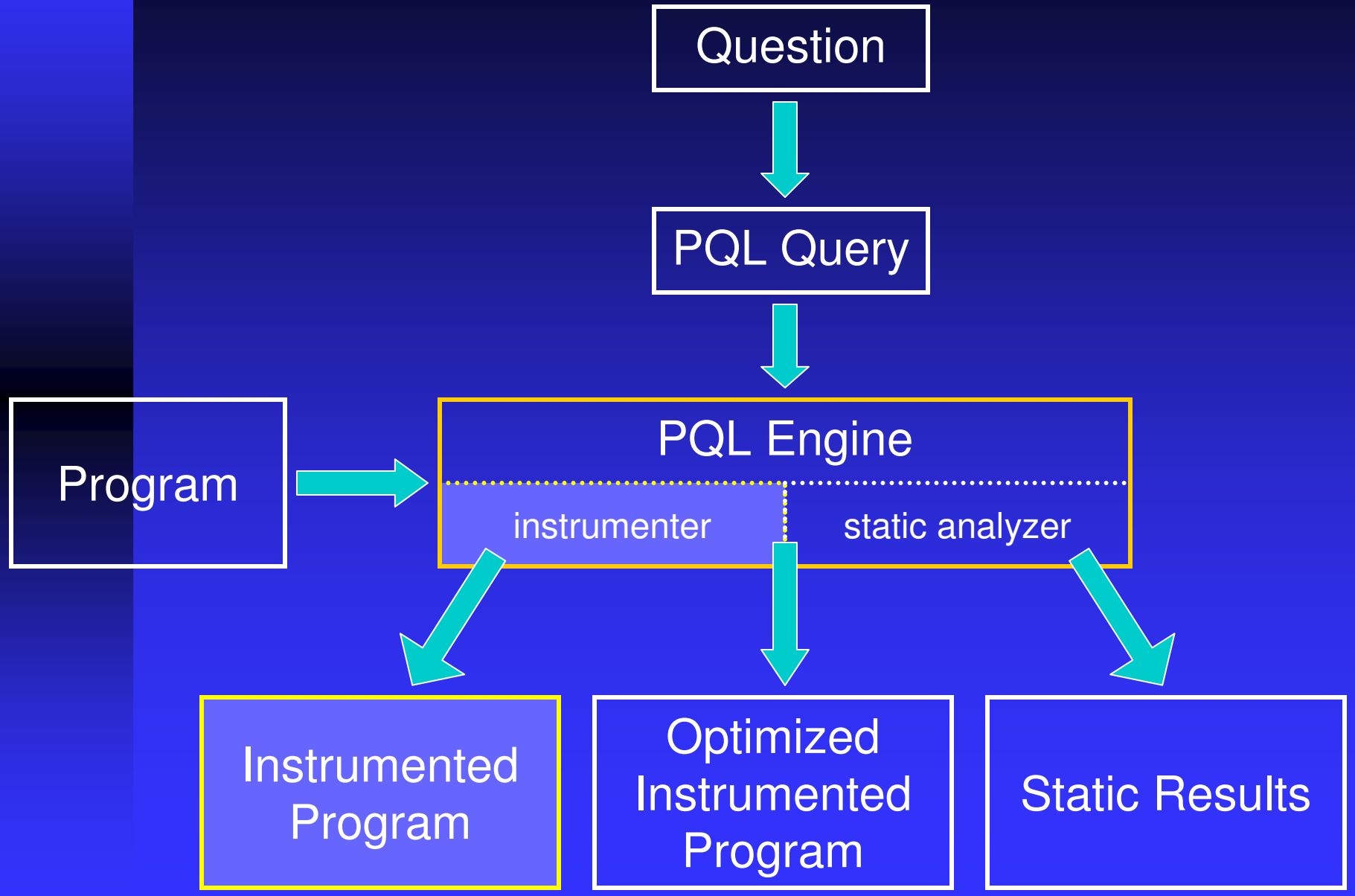
    ```
    x.lock(); ~x.unlock(); x.lock();
    ```

  - Single statements only

Expressiveness

- Concatenation + alternation = Loop-free regexp
- + Subqueries = CFG
- + Partial Order = CFG + Intersection
- Quantified over heap
  - Each subquery independent
  - Existentially quantified

System Architecture



Dynamic Matcher

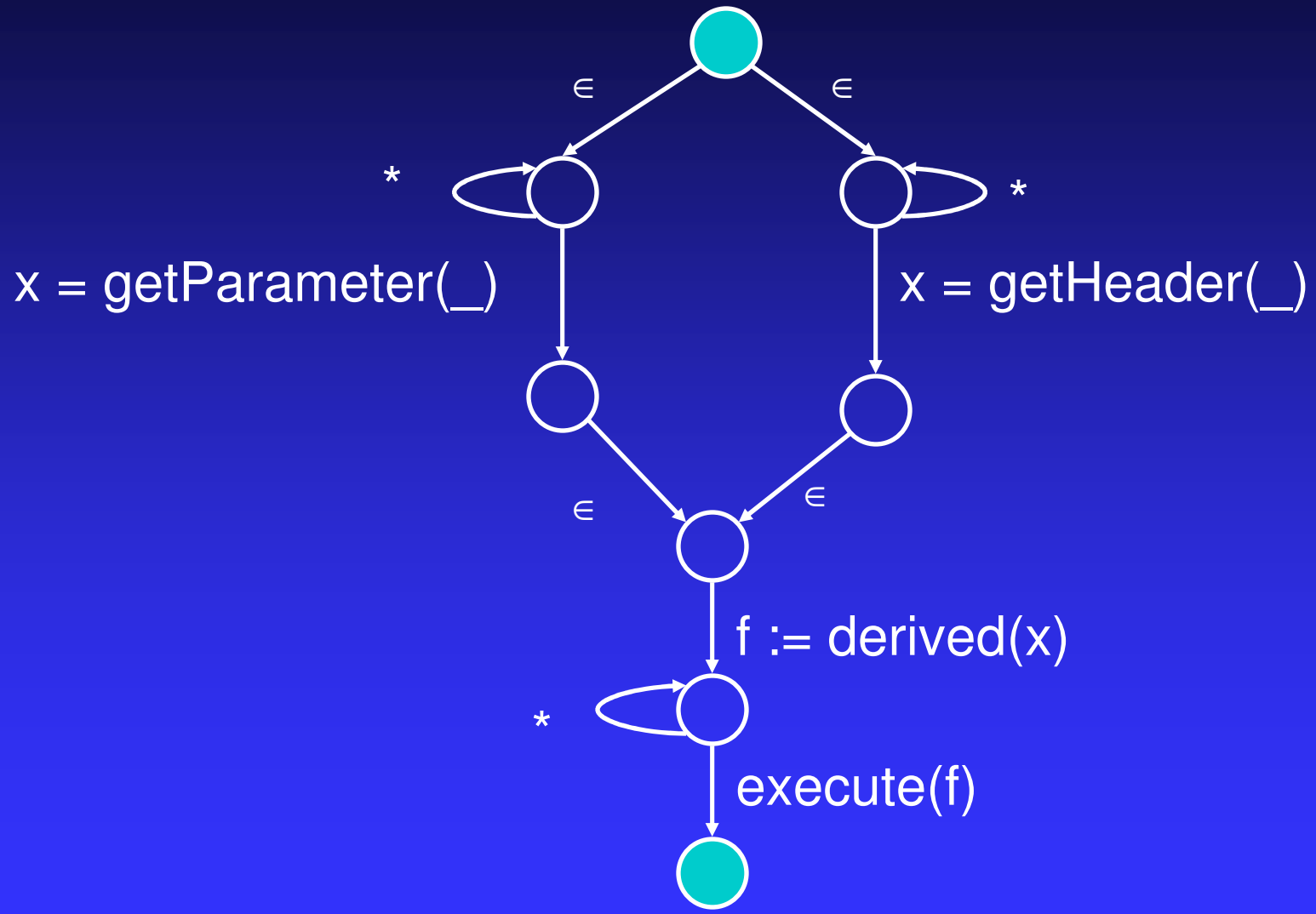
- Subquery → state machine
- Call to subquery → new instance of machine
- States carry “bindings” with them
  - Query variables → heap objects
  - Bindings are acquired as the variables are referenced for the first time in a match

Query to Translate

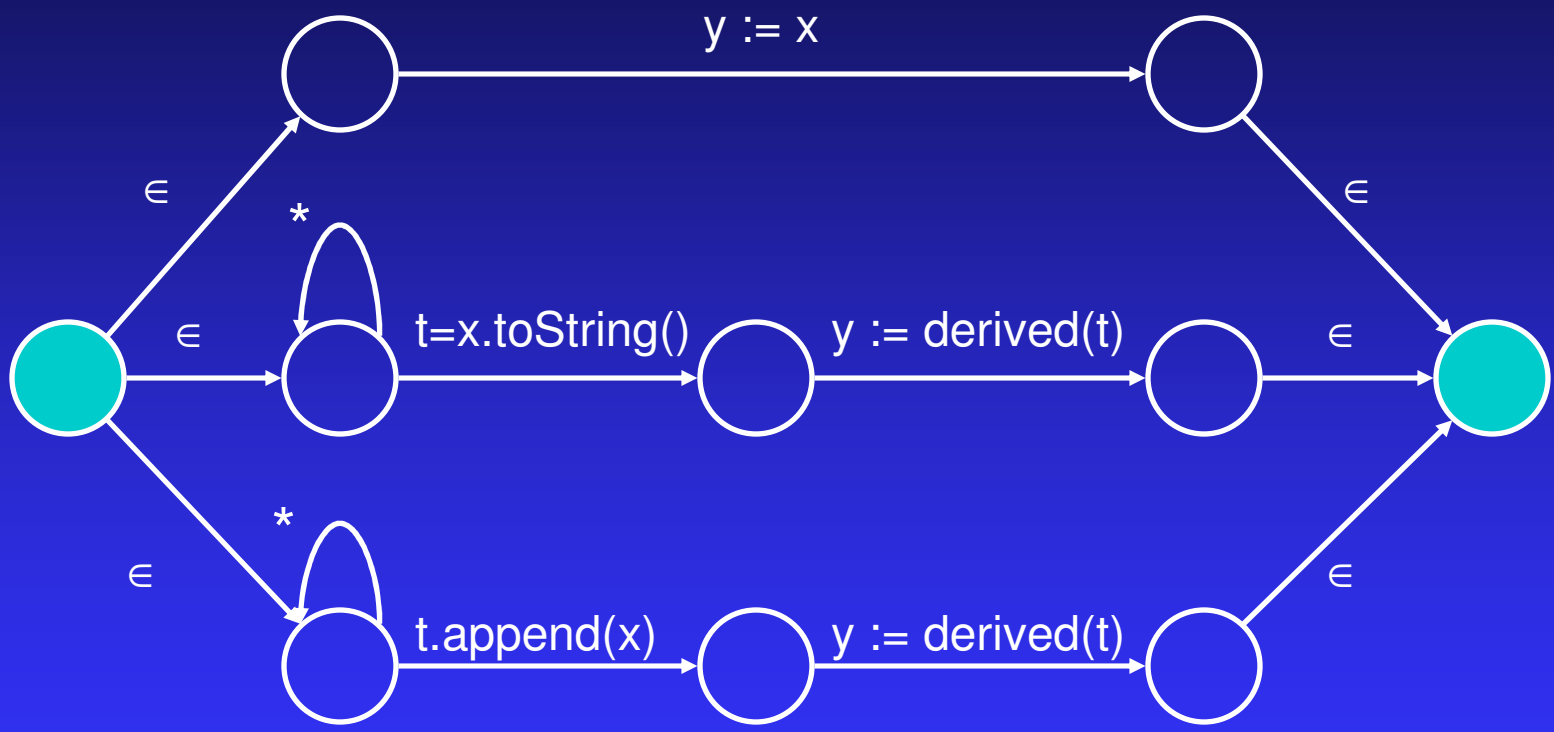
```
query main()
uses Object x, final;
matches {
    x = getParameter(_) | x = getHeader();
    f := derived (x); execute (f);
}

query derived(Object x)
uses Object t;
returns Object y;
matches {
    { y := x; }
  | { t = x.toString(); y := derived(t); }
  | { t.append(x); y := derived(t); }
}
```

`main()` Query Machine



`derived()` Query Machine



Example Program Trace

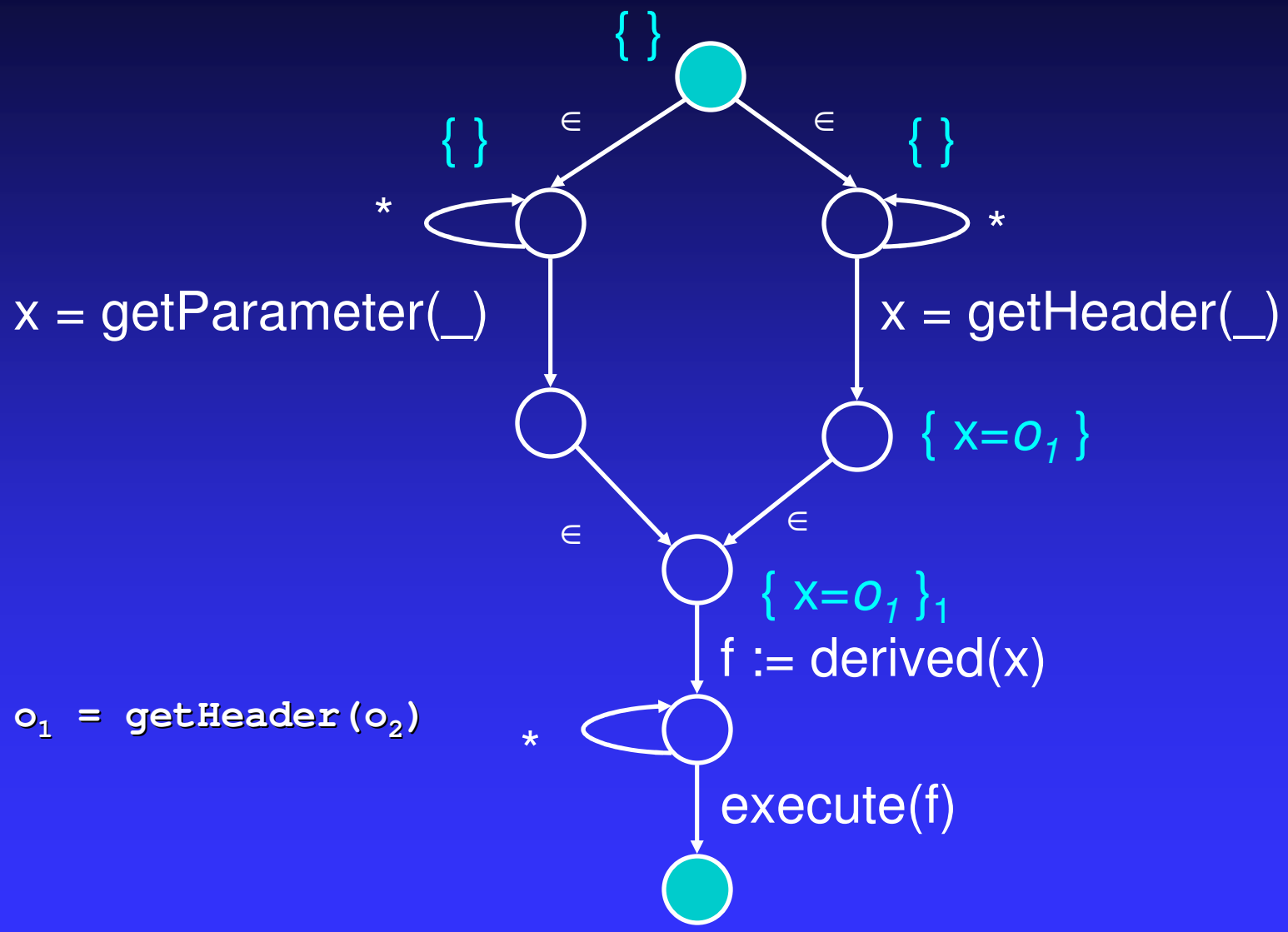
o_1 = getHeader(o_2)

o_3.append(o_1)

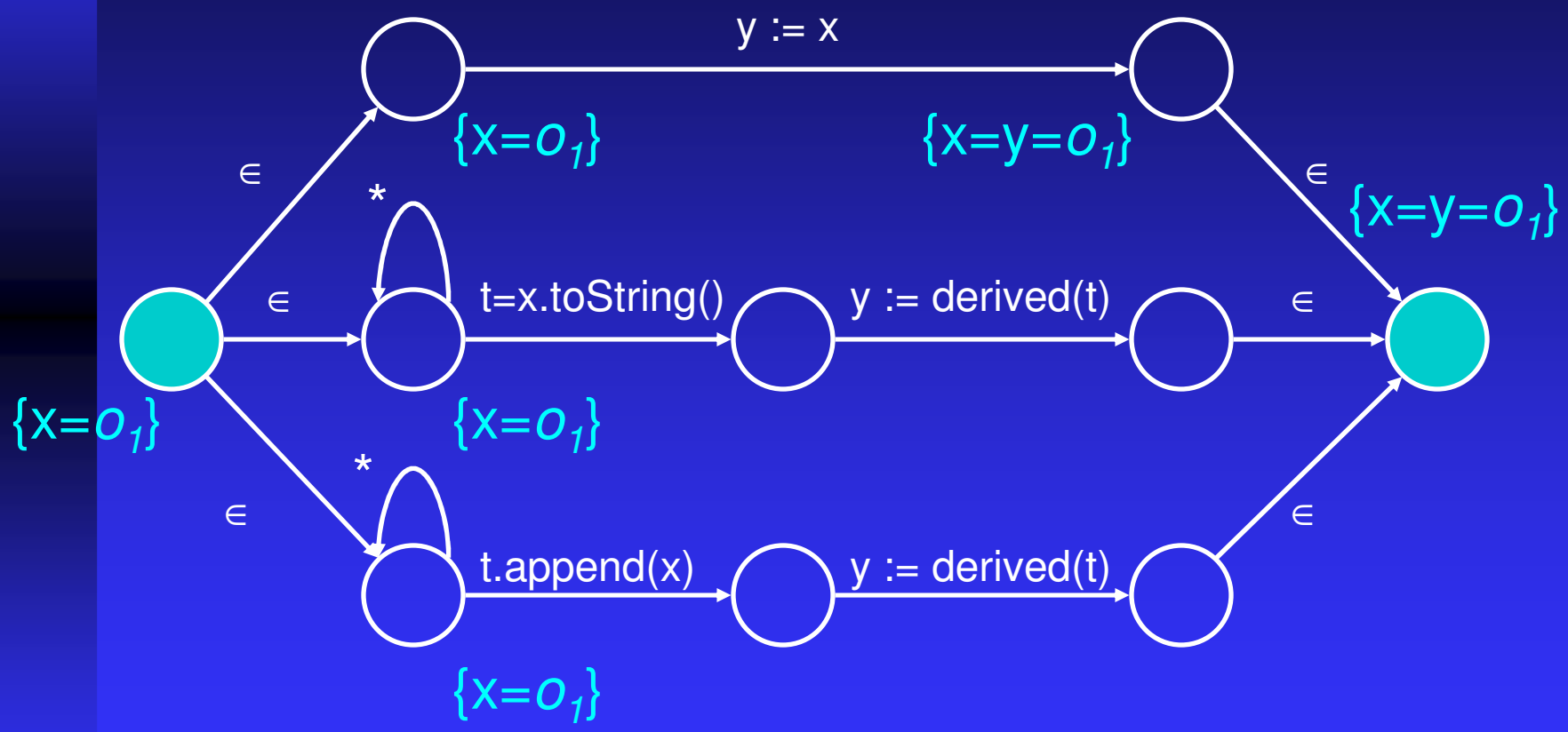
o_3.append(o_4)

o_5 = execute(o_3)

`main()`: Top Level Match

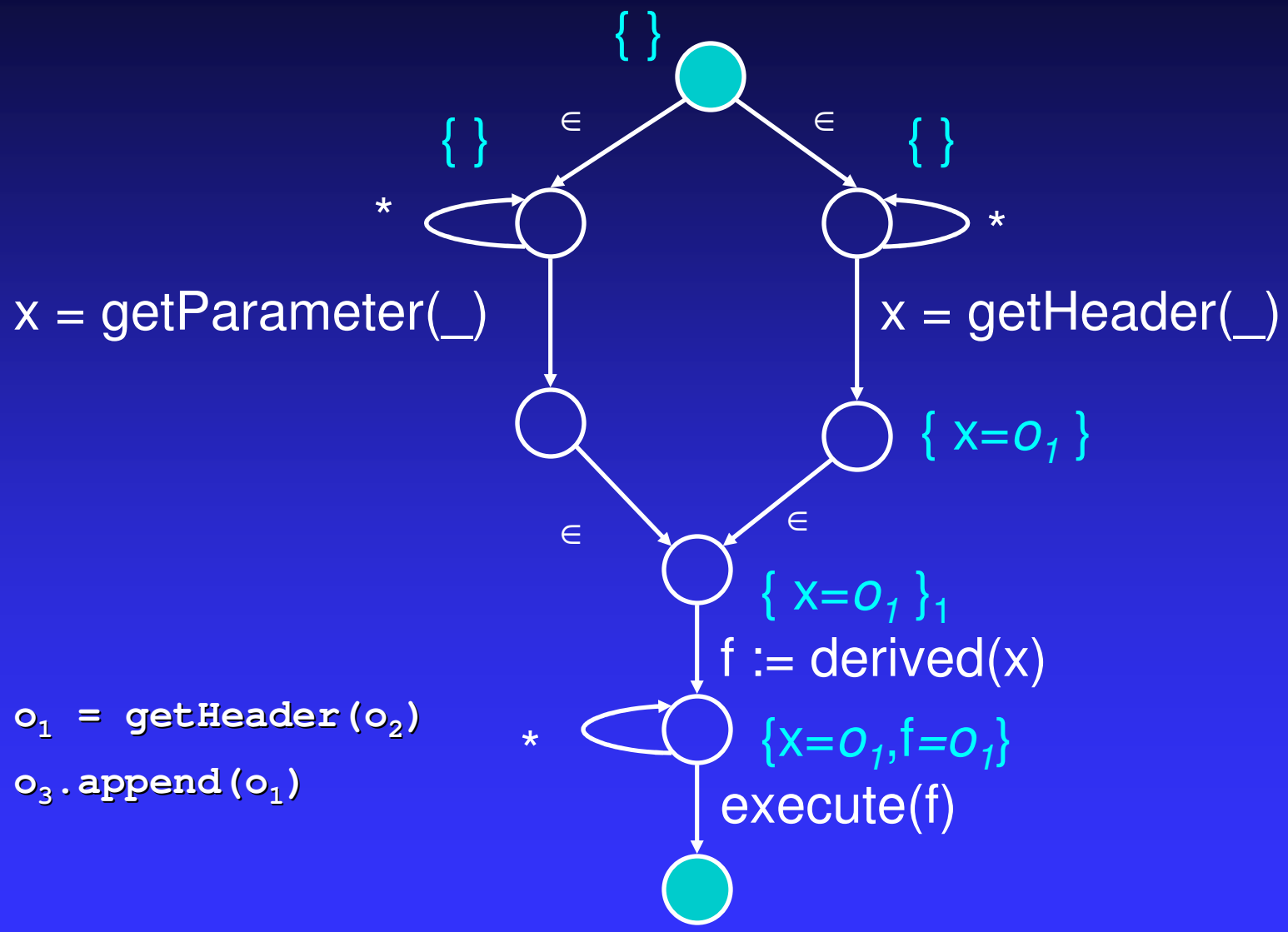


`derived()`: call 1



o_1 = `getHeader(`o_2`)`

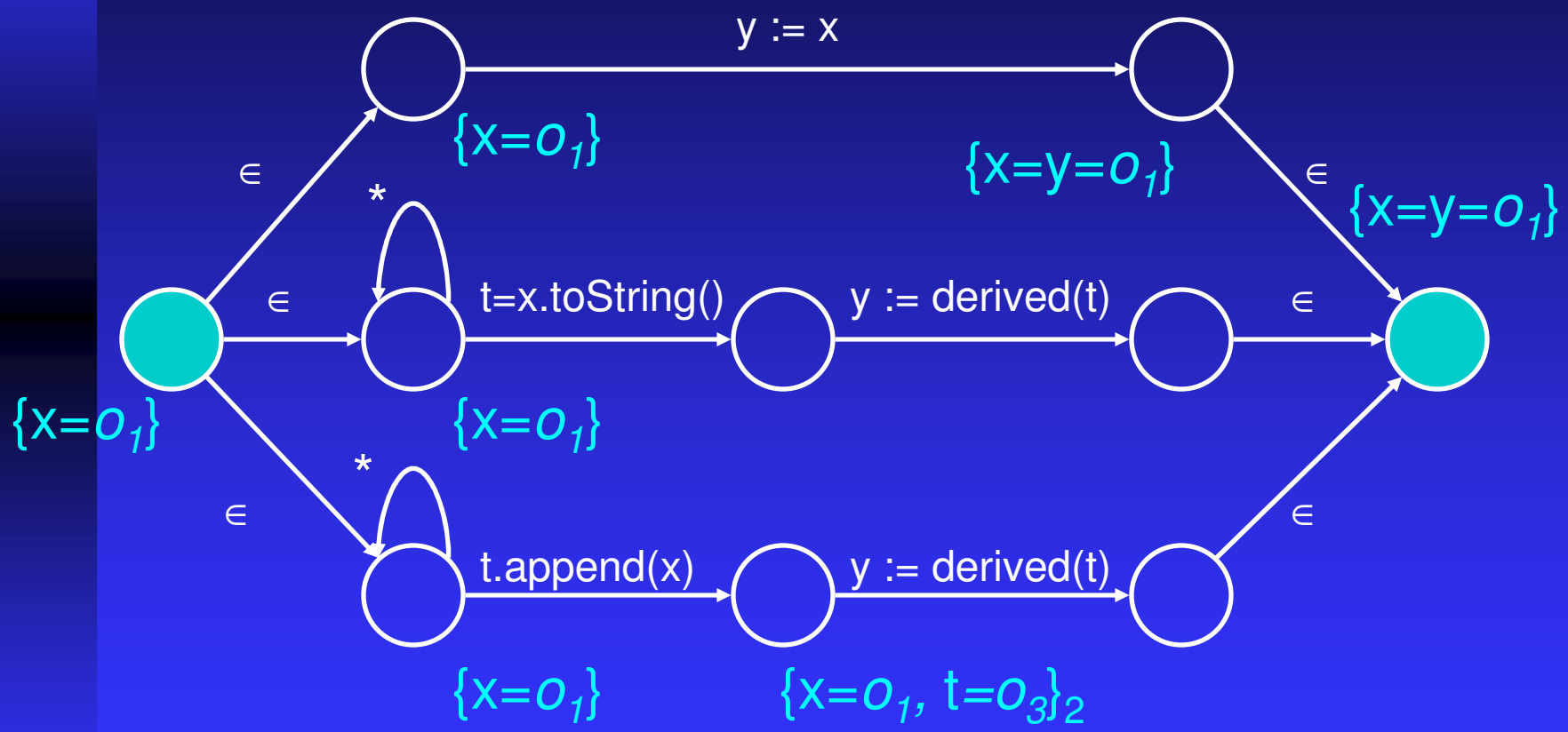
`main()`: Top Level Match



o_1 = **getHeader(o_2)**

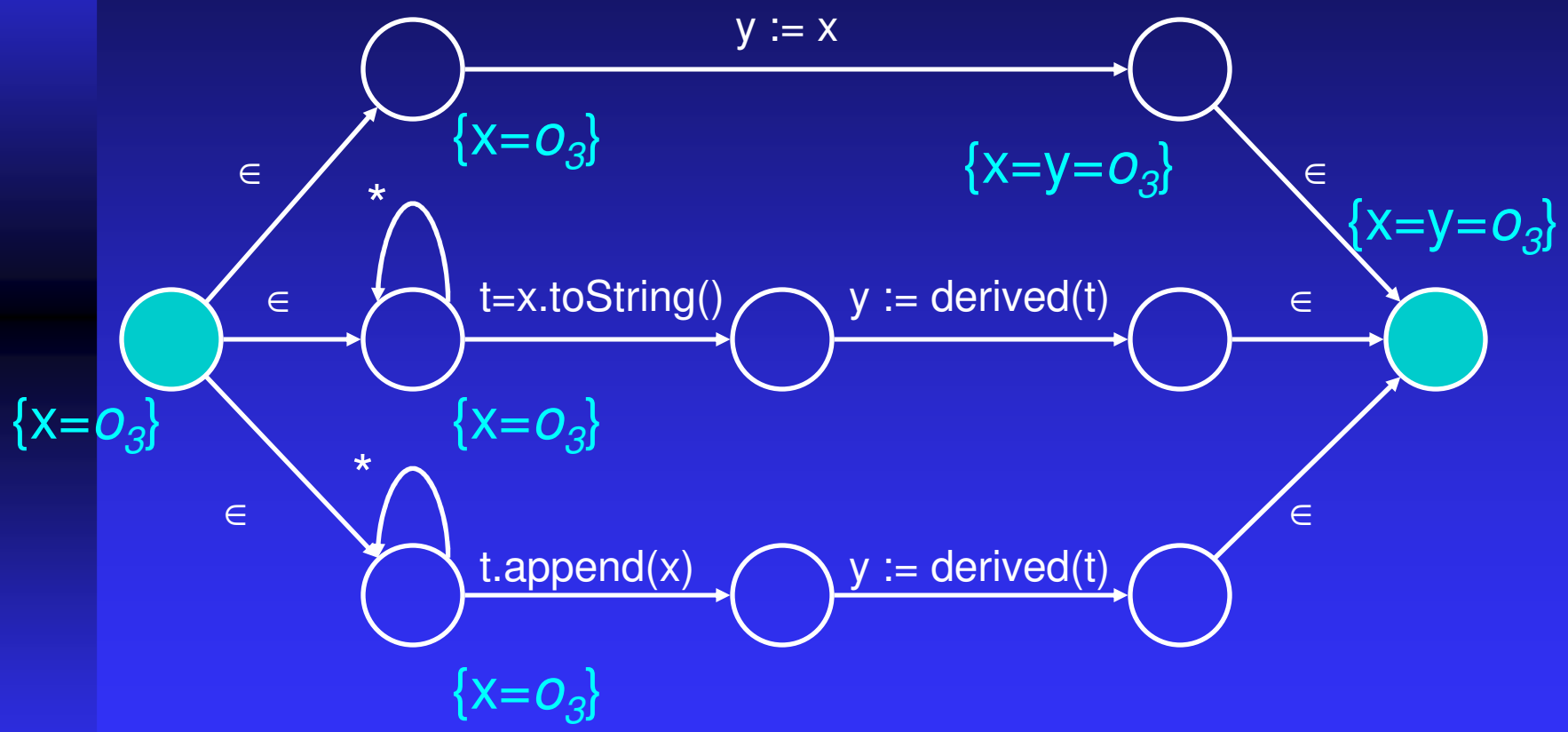
o_3**.append(o_1)**

`derived()`: call 1



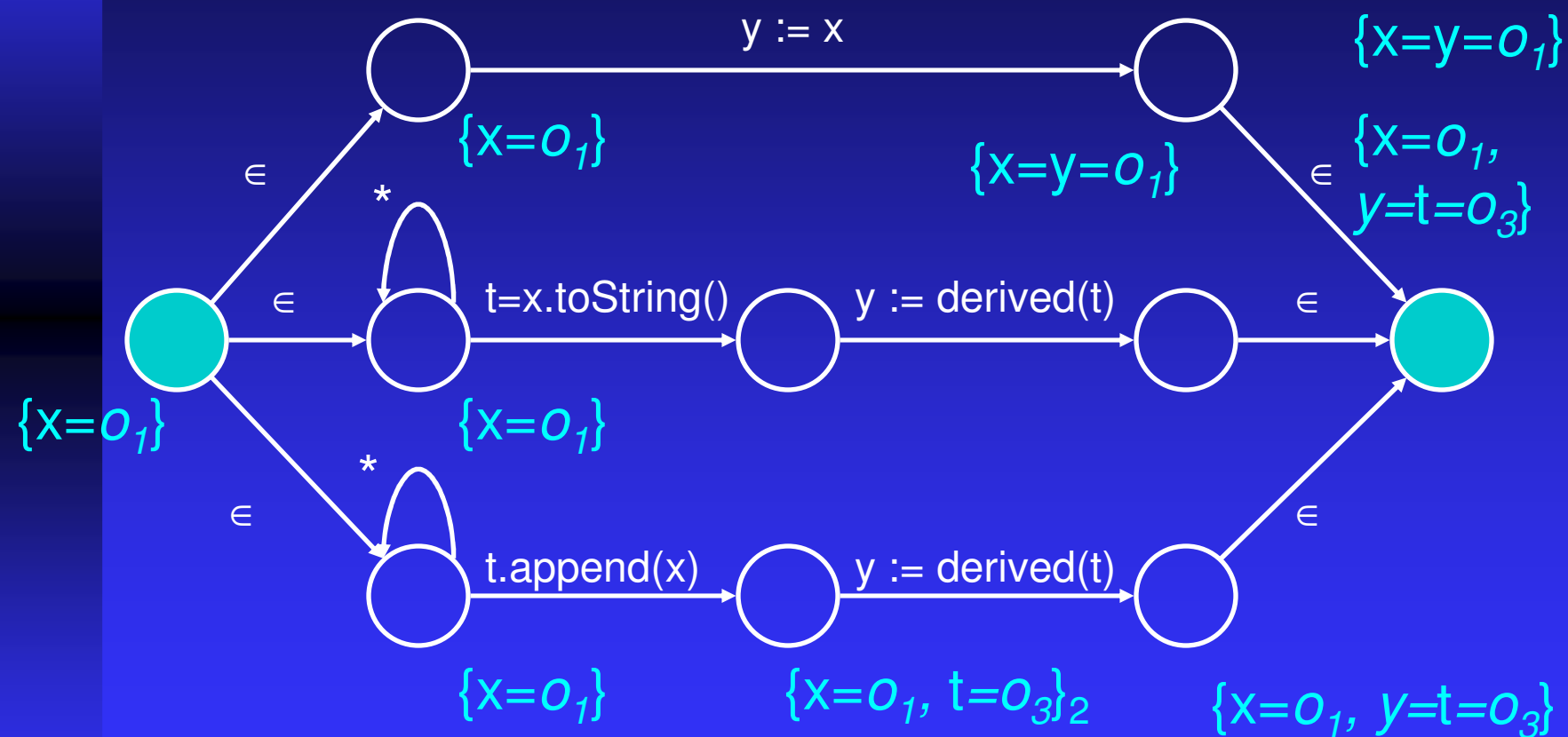

```
o1 = getHeader(o2)
o3.append(o1)
```

derived(): call 2



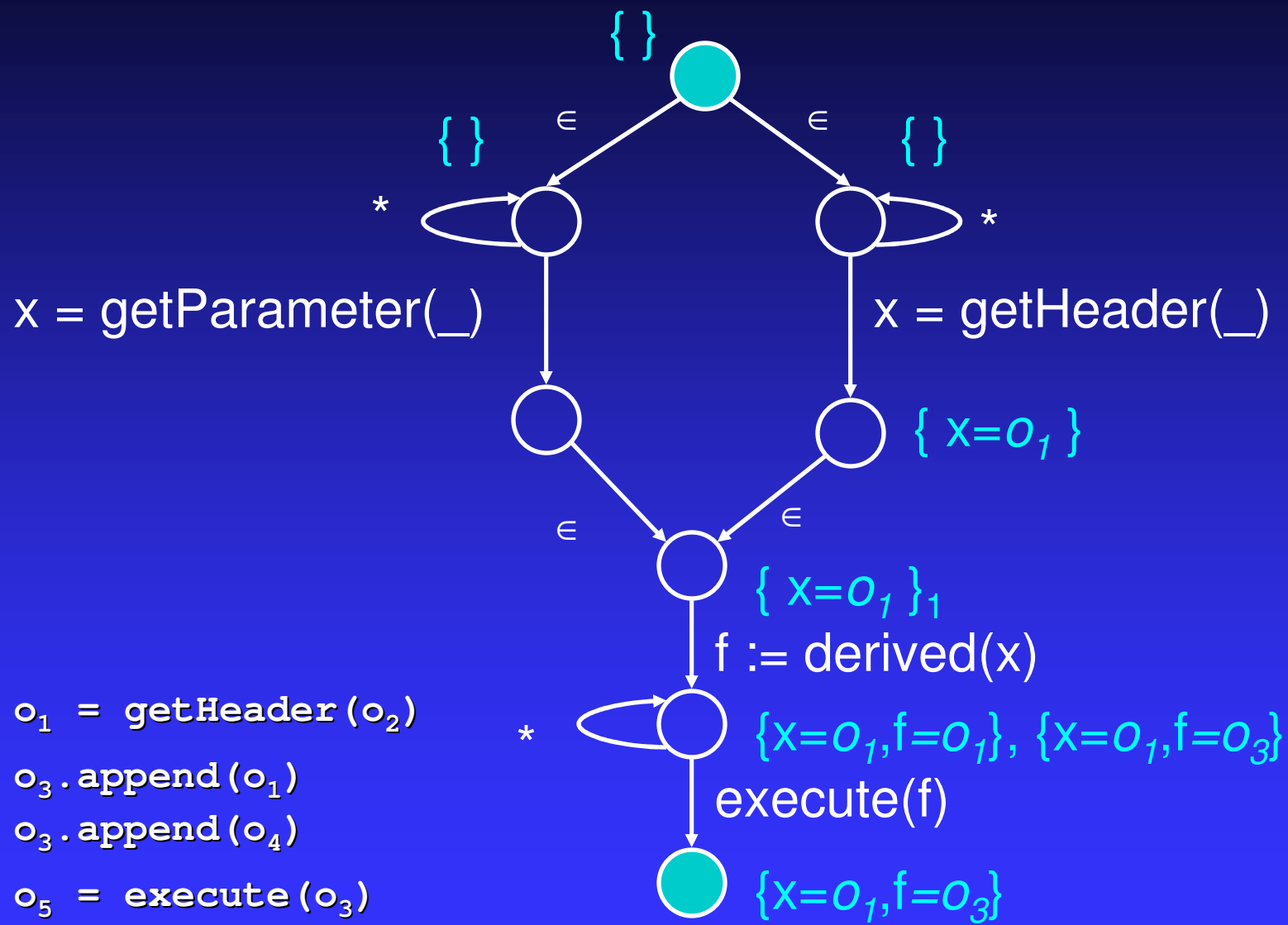

```
o1 = getHeader(o2)
o3.append(o1)
```

`derived()`: call 1




```
o1 = getHeader(o2)
o3.append(o1)
```

`main()`: Top Level Match

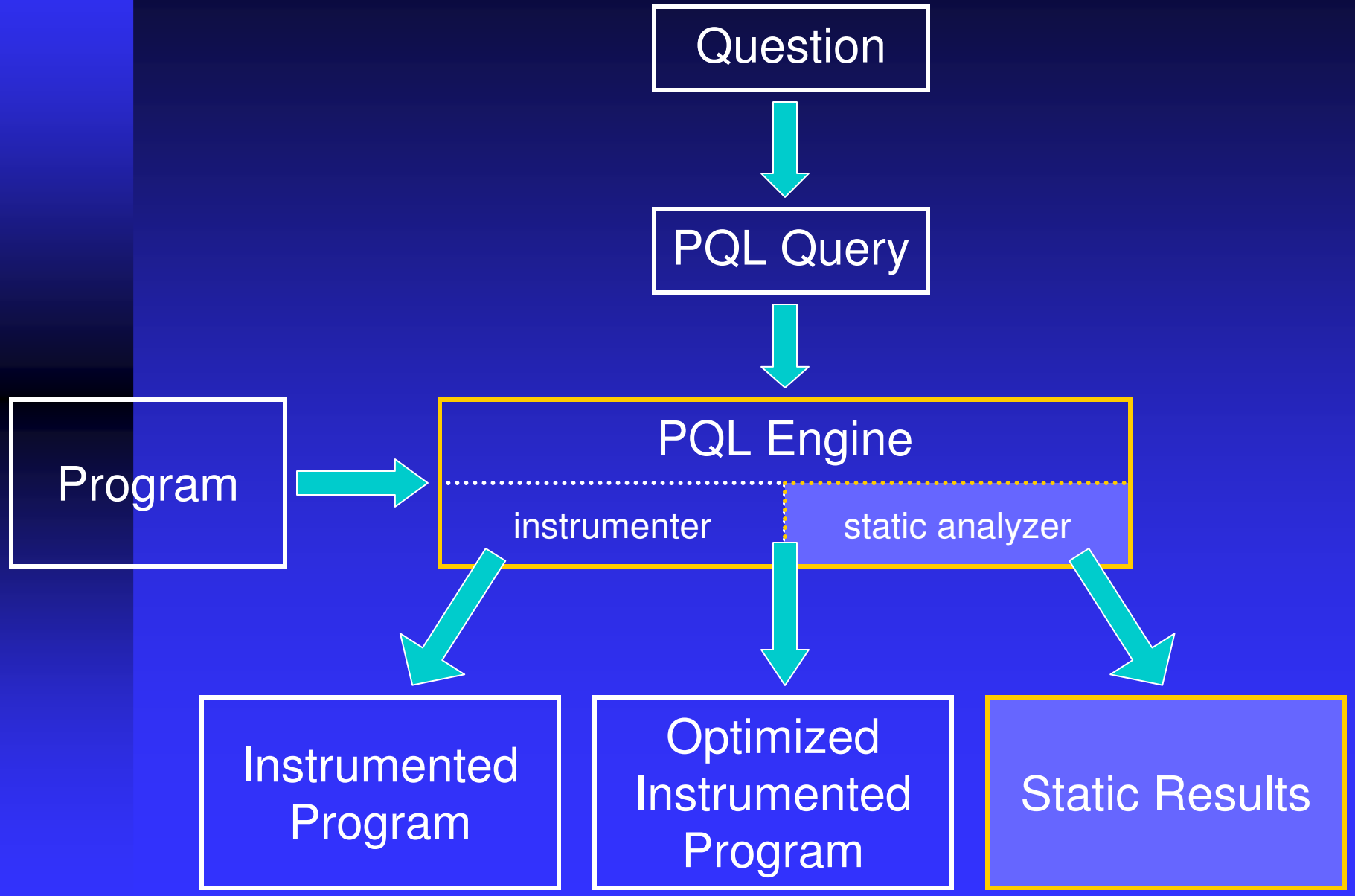



```
o1 = getHeader(o2)
o3.append(o1)
o3.append(o4)
o5 = execute(o3)
```

Find Relevance Fast

- Hash map for each transition
  - Every call-instance combined
- Key on known-used variables
- All used variables known-used → one lookup per transition

System Architecture



Static Analysis

- “Can this program match this query?”
- Undecidable in general
- We give a conservative approximation
- No matches found = None possible

Static Analysis

- PQL query automatically translated to query on pointer analysis results
- Pointer analysis is *sound* and *context-sensitive*
  - 10^{14} contexts in a good-sized application
  - Exponential space represented with BDDs
  - Analyses given in Datalog
- Whaley/Lam, PLDI 2004 (*bdddbddb*) for details

Static Results

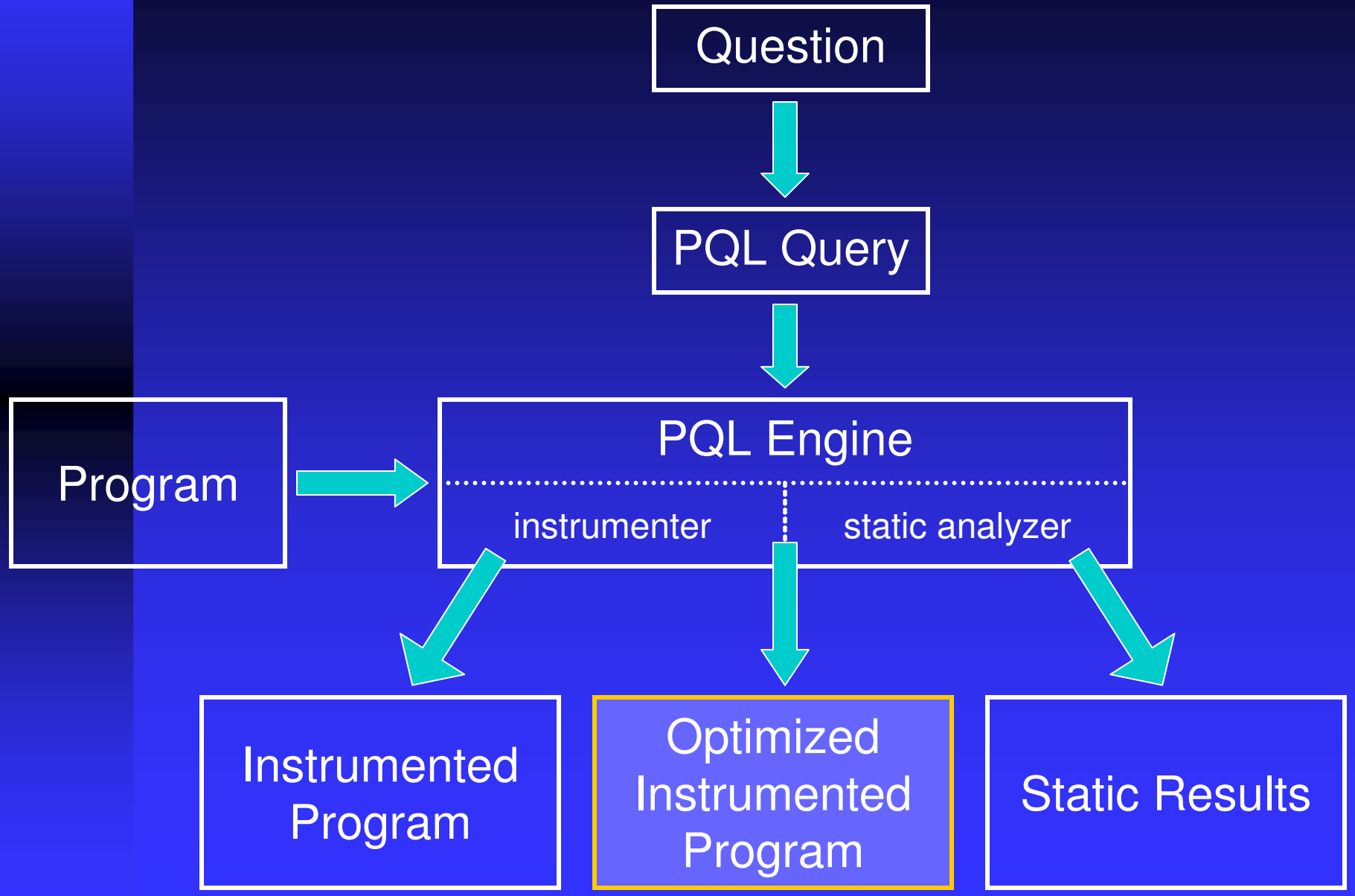
- Sets of objects and events that could represent a match

OR

- Program points that could participate in a match

- No results = no match possible!

System Architecture



Optimizing the Dynamic Analysis

- Static results conservative
  - So, point not in result → point never in any match
  - So, no need to instrument
- Usually more than 90% reduction

Experiment Topics

- Domain of Java Web applications
  - Serialization errors
  - SQL injection
- Domain of Eclipse IDE plugins
  - API violations
  - Memory leaks

Experiment Summary

Name	Classes	Inst Pts	Bugs
webgoat	1,021	69	2
personalblog	5,236	36	2
road2hibernate	7,062	779	1
snipsnap	10,851	543	8
roller	16,359	0	1
Eclipse	19,439	18,152	192
TOTAL	**59,968**	**19,579**	**206**

Session Serialization Errors

- Very common bug in Web applications
- Server tries to persist non-persistent objects
  - Only manifests under heavy load
  - Hard to find with testing
- One-line query in PQL
  - `HttpSession.setAttribute(_,!Serializable);`
- Solvable purely statically
  - Dynamic confirmation possible

SQL Injection

- Our running example
- Static optimizes greatly
  - 92%-99.8% reduction of points
  - 2-3x speedup
- 4 injections, 2 exploitable
  - Blocked both exploits
- Further applications and an improved static analysis in Usenix Security ’05

Full `derived()` Query

```
query derived (Object x)
returns Object y;
uses Object temp;
matches {
  y := x
| { temp = StringProp(x);
    y := derived(temp); }
}
```

```
query StringProp (Object * x)
returns Object y;
uses Object z;
matches {
  y.append(x, ...)
| x.getChars(_, _, y, _)
| y.insert(_, x)
| y.replace(_, _, x)
| y = x.substring(...)
| y = new java.lang.String(x)
| y = new java.lang.StringBuffer(x)
| y = x.toString()
| y = x.getBytes(...)
| y = _.copyValueOf(x)
| y = x.concat(_)
| y = _.concat(x)
| y = new java.util.StringTokenizer(x)
| y = x.nextToken()
| y = x.next()
| y = new java.lang.Number(x)
| y = x.trim()
| { z = x.split(...); y = z[]; }
| y = x.toLowerCase(...)
| y = x.toUpperCase(...)
| y = _.replaceAll(_, x)
| y = _.replaceFirst(_, x);
}
```

Eclipse

- IDE for Java
- Very large (tens of MB of bytecode)
  - Too large for our static analysis
- Purely interactive
  - Unoptimized dynamic overhead acceptable

Queries on Eclipse

- Paired Methods
  - register/deregister
  - createWidget/destroyWidget
  - install/uninstall
  - startup/shutdown
- Lapsed Listeners

Eclipse Results

- All paired methods queries were run simultaneously
  - 56 mismatches detected
- Lapsed listener query was run alone
  - 136 lapsed listeners
  - Can be automatically fixed

Current Status

- Open source and hosted on SourceForge
- http://pql.sf.net – standalone dynamic implementation

Related work

- PQL is a *query language*
  - JQuery
- on *program traces*
  - Partiqle, Dalek, ...
- Observing *behavior* and *finding bugs*
  - Metal, Daikon, PREfix, Clouseau, ...
- and *automatically add code to fix them*
  - AspectJ

Conclusions

- PQL – a Program Query Language
  - Match histories of sets of objects on a program trace
  - Targeting application developers
- Found many bugs
  - 206 application bugs and security flaws
  - 6 large real-life applications
- PQL provides a bridge to powerful analyses
  - Dynamic matcher
    - Point-and-shoot even for unknown applications
    - Automatically repairs program on the fly
  - Static matcher
    - Proves absence of bugs
    - Can reduce runtime overhead to production-acceptable