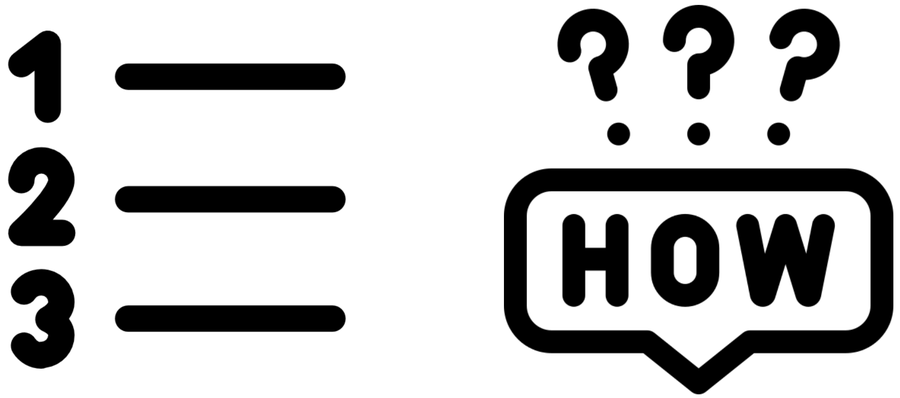
Declarative Static Analysis for Multilingual Programs using CodeQL

Dongjun Youn — KAIST
Sungho Lee — CNU
Sukyoung Ryu — KAIST

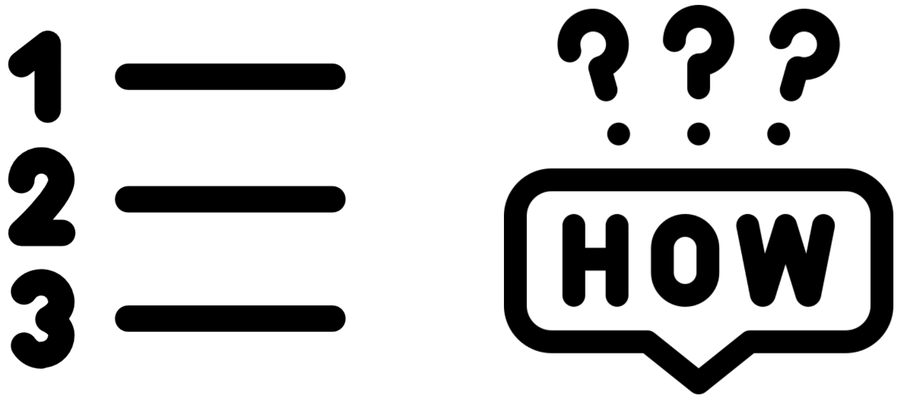
2023.07.06

PLRG Programming Language Research Group

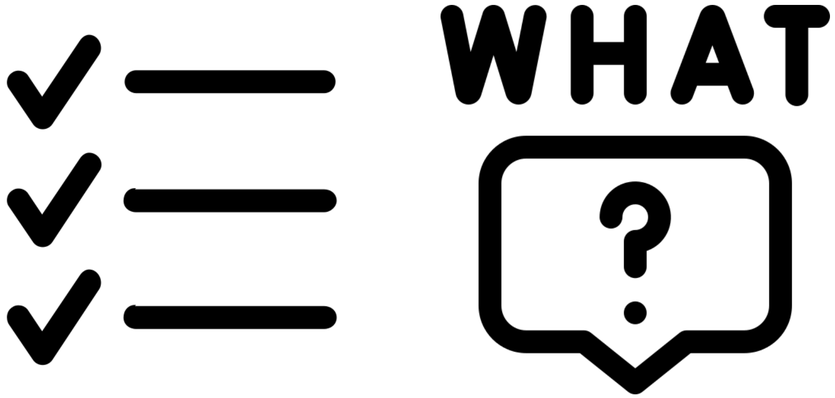
KAIST



Imperative languages



Imperative languages



Declarative languages

Q. Check if `p` is a prefix of `s`

Q. Check if `p` is a prefix of `s`

```
def is_prefix(p, s):
    l = len(p)
    for i in range(l):
        if p[i] != s[i]:
            return False
    return True
```

Imperative languages

Q. Check if p is a prefix of s

```
def is_prefix(p, s):
    l = len(p)
    for i in range(l):
        if p[i] != s[i]:
            return False
    return True
```

Imperative languages

```
predicate is_prefix(p, s){
  exists(x | p + x = s)
}
```

Declarative languages

Q. Check if `p` is a prefix of `s`

```
def is_prefix(p, s):
    l = len(p)
    for i in range(l):
        if p[i] != s[i]:
            return False
    return True
```

Imperative languages

```
predicate is_prefix(p, s){
  exists(x | p + x = s)
}
```

Declarative languages

Declarative static analyzers

Doop - Framework for Java Pointer and Taint Analysis (using P/Taint)

This document contains instructions for invoking the main driver of Doop.

- For an introduction to Datalog, please consult Datalog-101.
- For a more detailed tutorial on using the results of Doop analyses, please consult Doop-101.
- For an introduction to pointer analysis using Datalog, you can read a research-level tutorial.
- For the architecture of Doop, see docs/documentation.md.

CodeQL

Discover vulnerabilities across a codebase with CodeQL, our industry-leading semantic code analysis engine. CodeQL lets you query code as though it were data. Write a query to find all variants of a vulnerability, eradicating it forever. Then share your query to help others do the same.

CodeQL is free for research and open source.

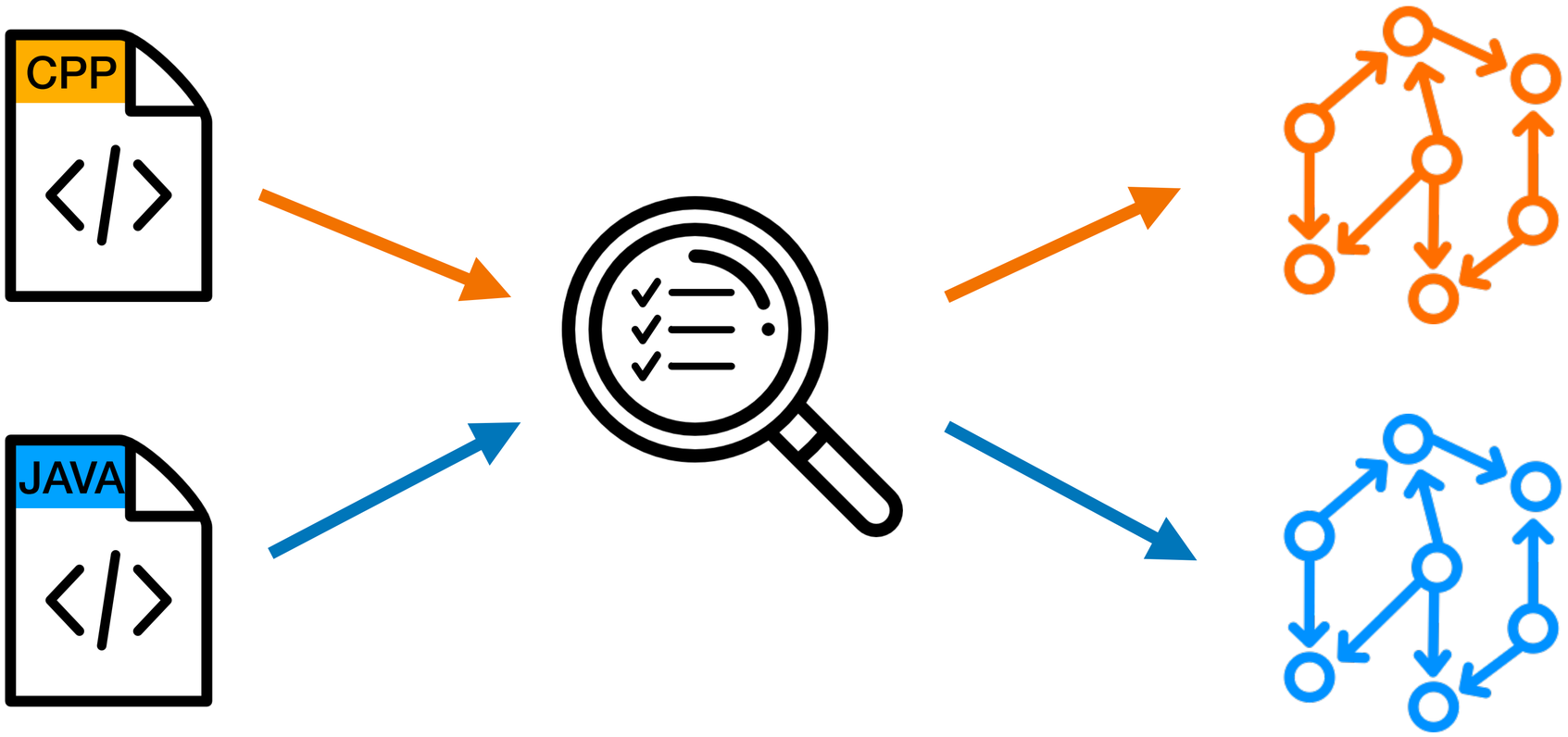
Glean

System for collecting, deriving and querying facts about source code

Declarative static analyzers, supporting multiple languages

Declarative static analyzers, supporting multiple languages

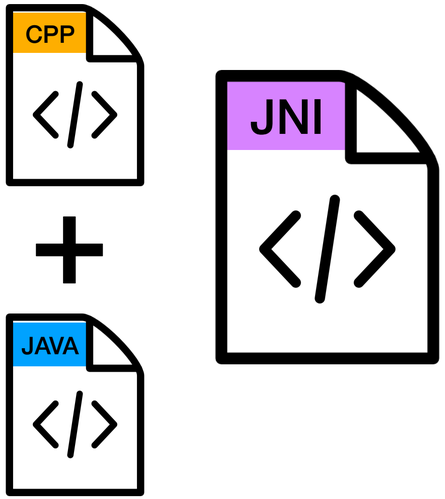
Declarative static analyzers, supporting multiple languages



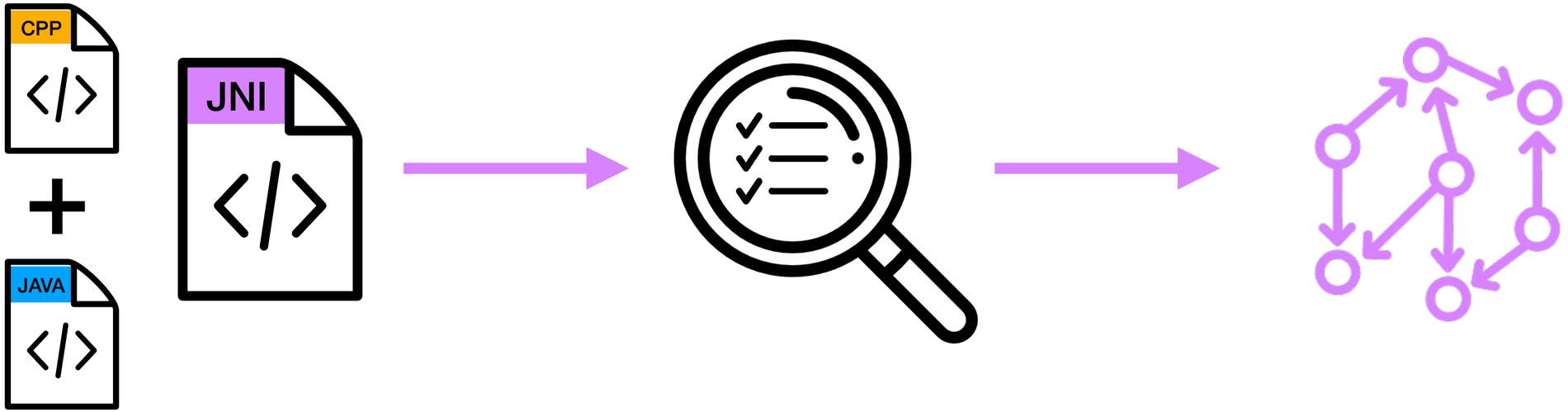
PLRG
Programming Language Research Group

KAIST

Declarative static analyzers, supporting multilingual program?



Declarative static analyzers, supporting multilingual program?



Contributions

Contributions

- Suggest how to easily merge two analyzers
  - Reuse components!

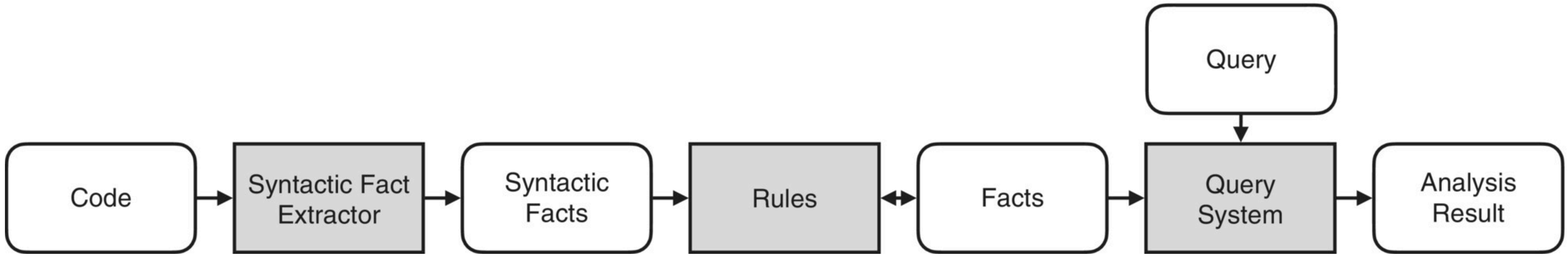
Contributions

- Suggest how to easily merge two analyzers
  - Reuse components!
- Implement MultiQL
  - Analyzer for JNI / C-Python programs

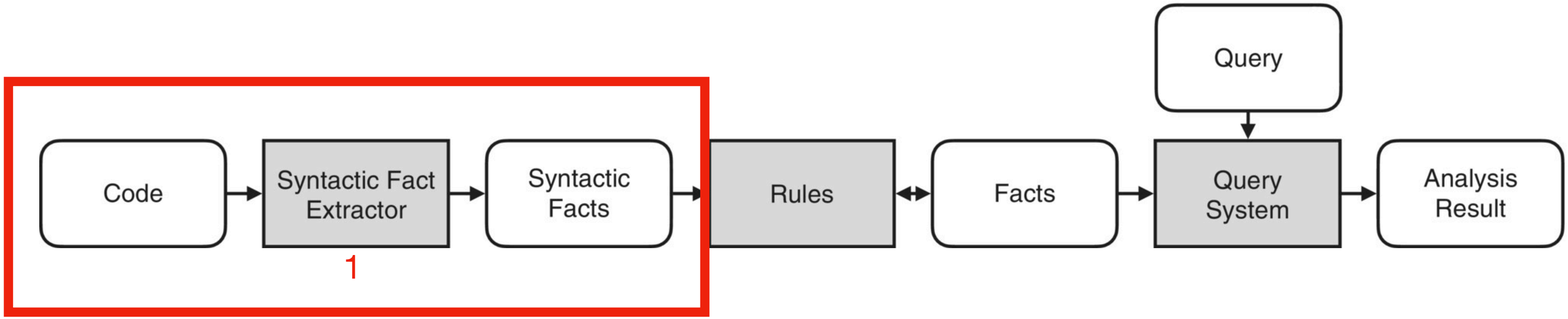
Contributions

- Suggest how to easily merge two analyzers
  - Reuse components!
- Implement MultiQL
  - Analyzer for JNI / C-Python programs
- Find bugs in real programs
  - Including new bugs

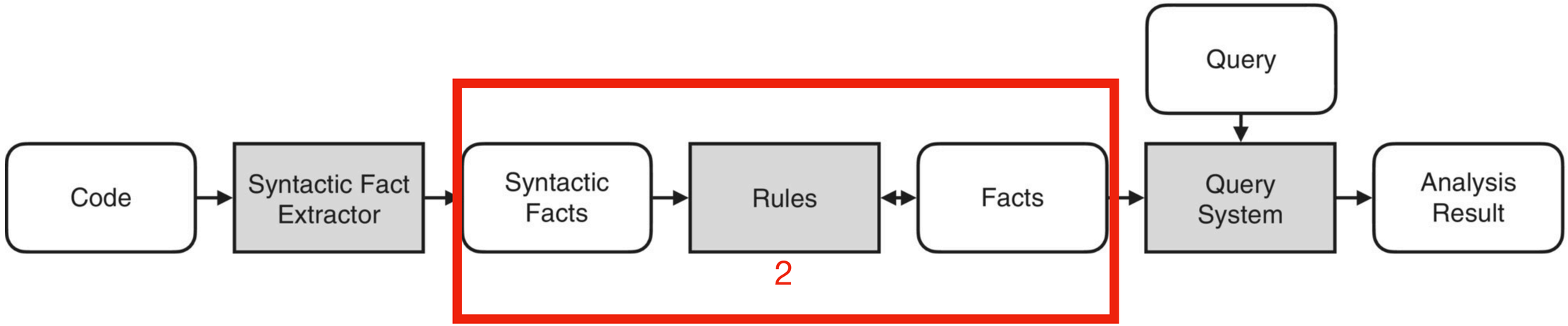
Declarative static analyzer for one language



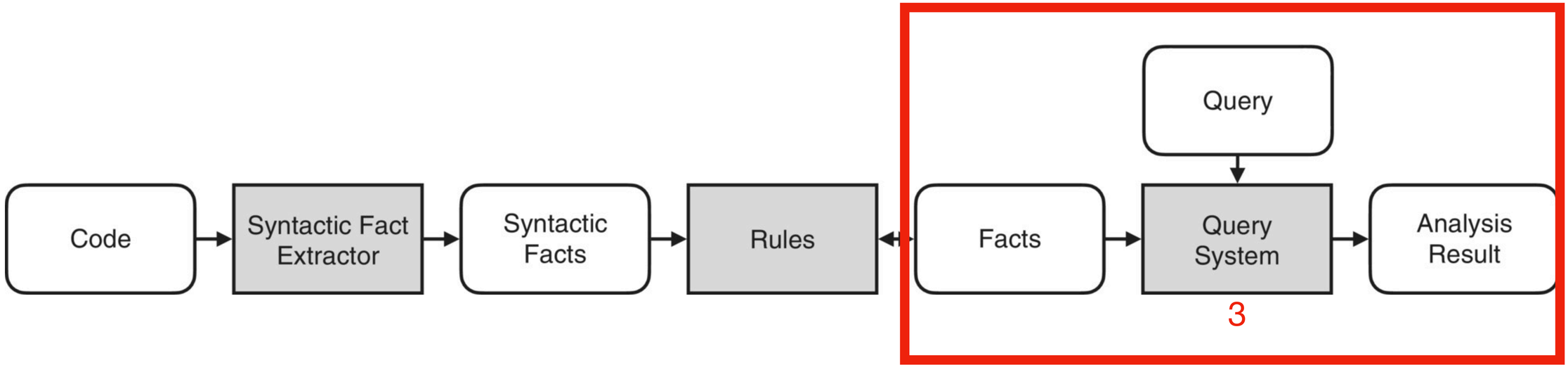
Declarative static analyzer for one language



Declarative static analyzer for one language



Declarative static analyzer for one language



1. Extracting syntactic facts

Syntactic Fact Extractor

1. Extracting syntactic facts

Named tuples

Syntactic Fact Extractor

1. Extracting syntactic facts

Named tuples

Code:

```
int f() {
  return 42;
}

int val = f();
```

→ Syntactic Fact Extractor

1. Extracting syntactic facts

Named tuples

Code:

```
int f() {
  return 42;
}

int val = f();
```

→

Syntactic Fact Extractor

→

Syntactic facts:

```
Return("f", 42)
Call(f(), "f")
Assign(val, f())
```

2. Derive new facts using rules

Rules

2. Derive new facts using rules

Rules

```
Step(from, to) :- Return(name, from), Call(to, name)
Step(from, to) :- Assign(to, from)

Flow(from, to) :- Step(from, to)
Flow(from, to) :- Step(from, mid), Flow(mid, to)
```

2. Derive new facts using rules

Syntactic facts:

```
Return("f", 42)
Call(f(), "f")
Assign(val, f())
```

Rules

```
Step(from, to) :- Return(name, from), Call(to, name)
Step(from, to) :- Assign(to, from)

Flow(from, to) :- Step(from, to)
Flow(from, to) :- Step(from, mid), Flow(mid, to)
```

PLRG Programming Language Research Group

KAIST

2. Derive new facts using rules

Syntactic facts:

```
Return("f", 42)
Call(f(), "f")
Assign(val, f())
```

Rules

Facts:

```
Step(42, f())
Step(f(), val)
Flow(42, f())
Flow(42, val)
```

```
Step(from, to) :- Return(name, from), Call(to, name)
Step(from, to) :- Assign(to, from)

Flow(from, to) :- Step(from, to)
Flow(from, to) :- Step(from, mid), Flow(mid, to)
```

PLRG Programming Language Research Group

KAIST

2. Derive new facts using rules

Syntactic facts:

```
Return("f", 42)
Call(f(), "f")
Assign(val, f())
```

Rules

Facts:

```
Step(42, f())
Step(f(), val)
Flow(42, f())
Flow(42, val)
```

```
Step(from, to) :- Return(name, from), Call(to, name)
Step(from, to) :- Assign(to, from)

Flow(from, to) :- Step(from, to)
Flow(from, to) :- Step(from, mid), Flow(mid, to)
```

PLRG Programming Language Research Group

KAIST

3. Perform query

Query system

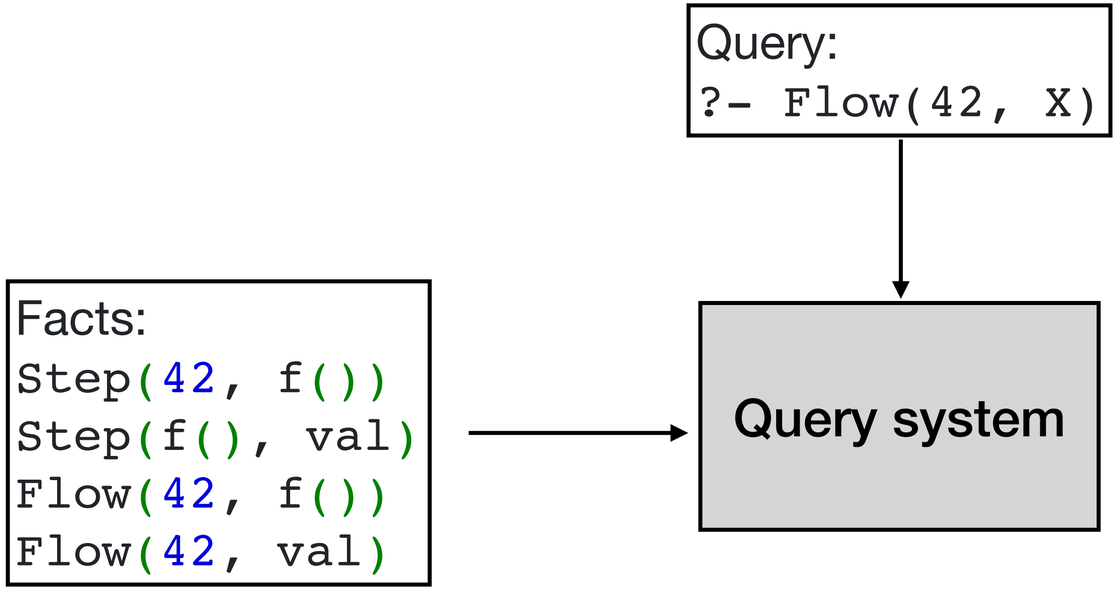
3. Perform query

Facts:

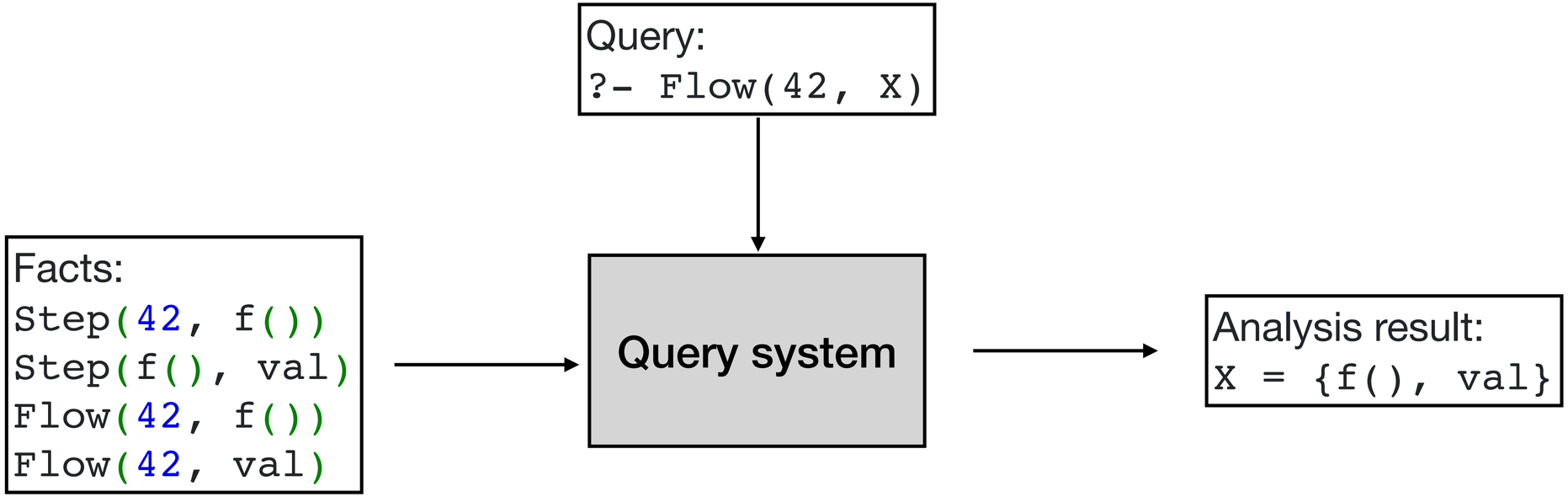
```
Step(42, f())
Step(f(), val)
Flow(42, f())
Flow(42, val)
```

→ Query system

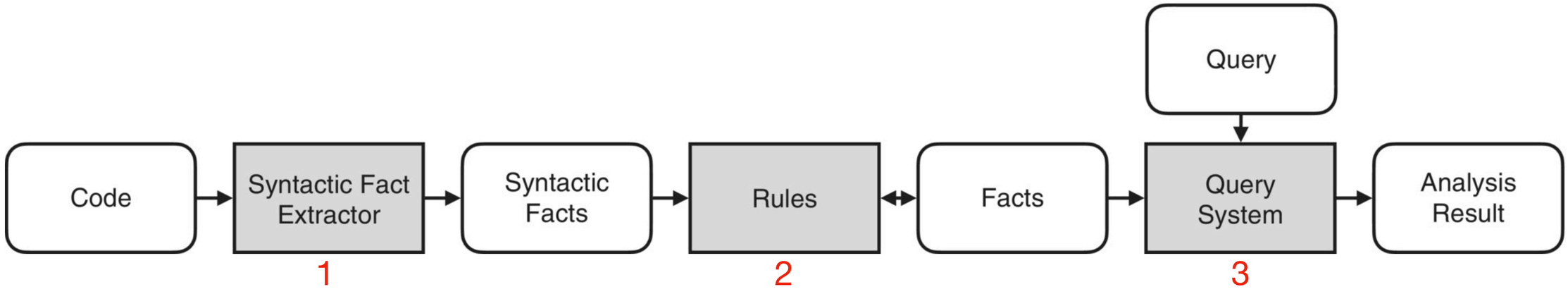
3. Perform query



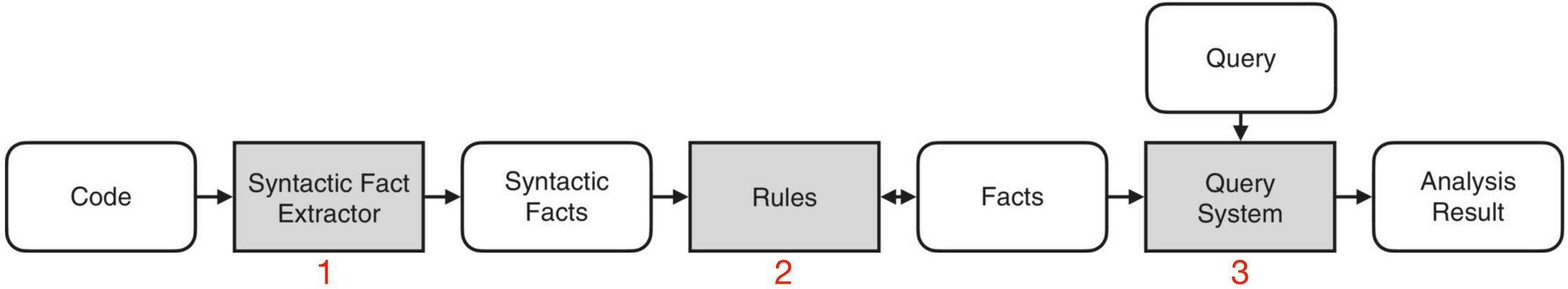
3. Perform query

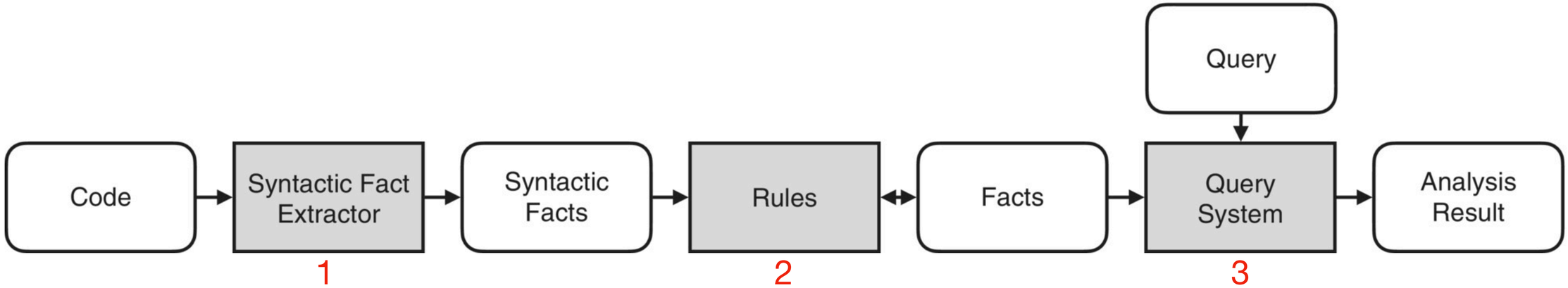


Declarative static analyzer for one language

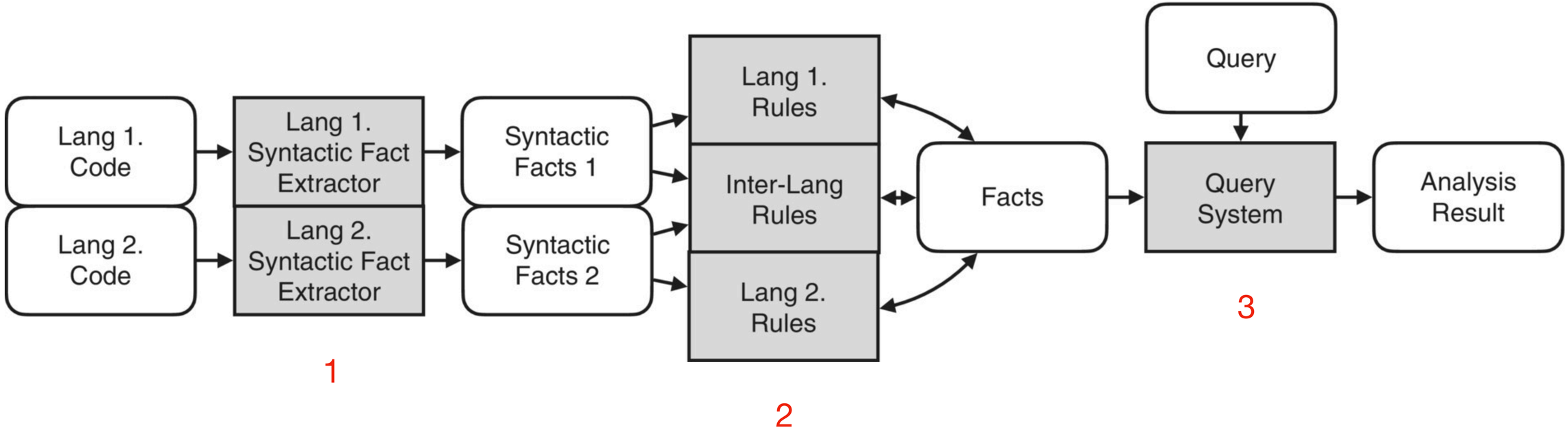


Declarative static analyzer for two language

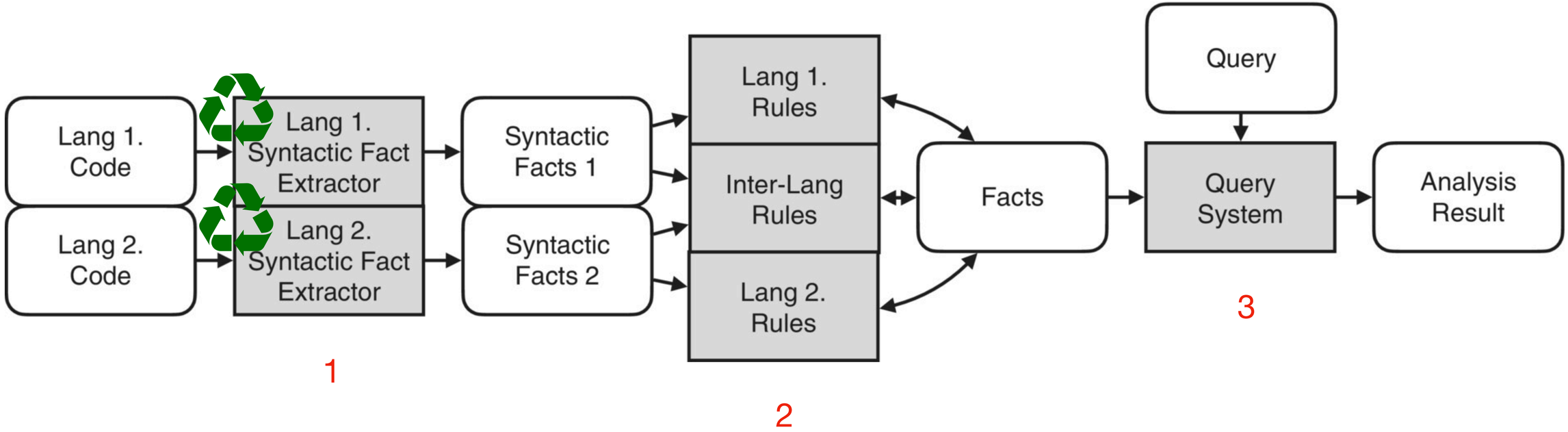




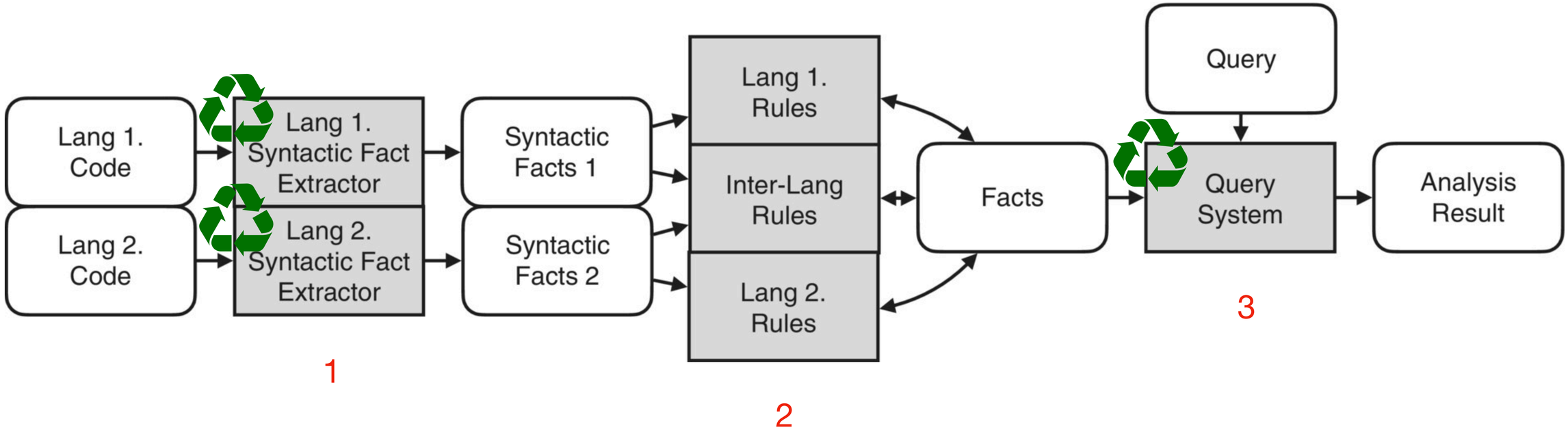
Declarative static analyzer for multilingual program



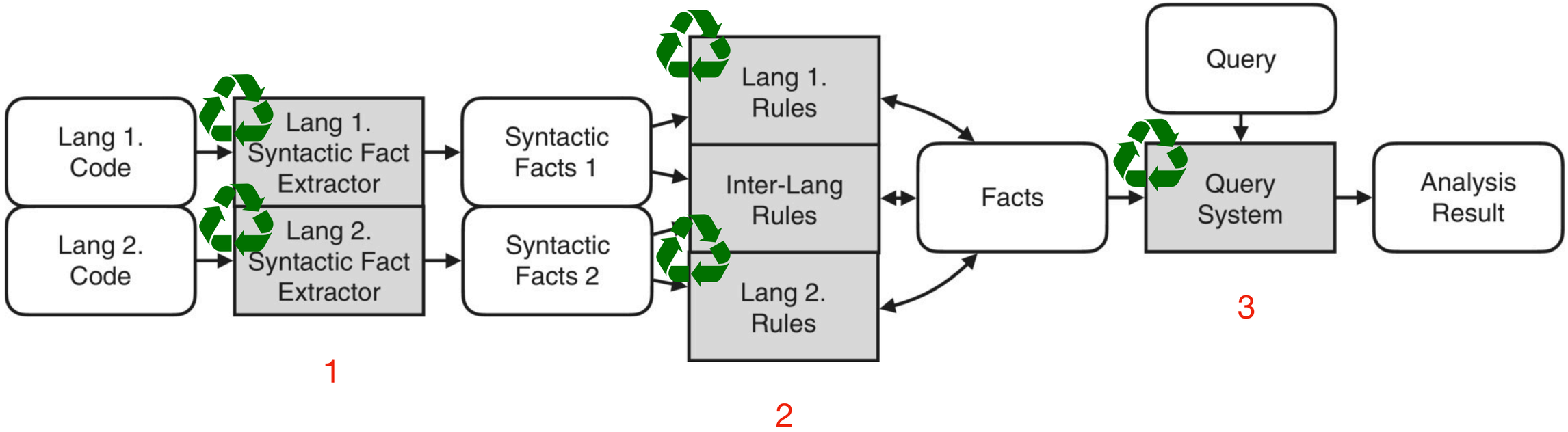
Declarative static analyzer for multilingual program



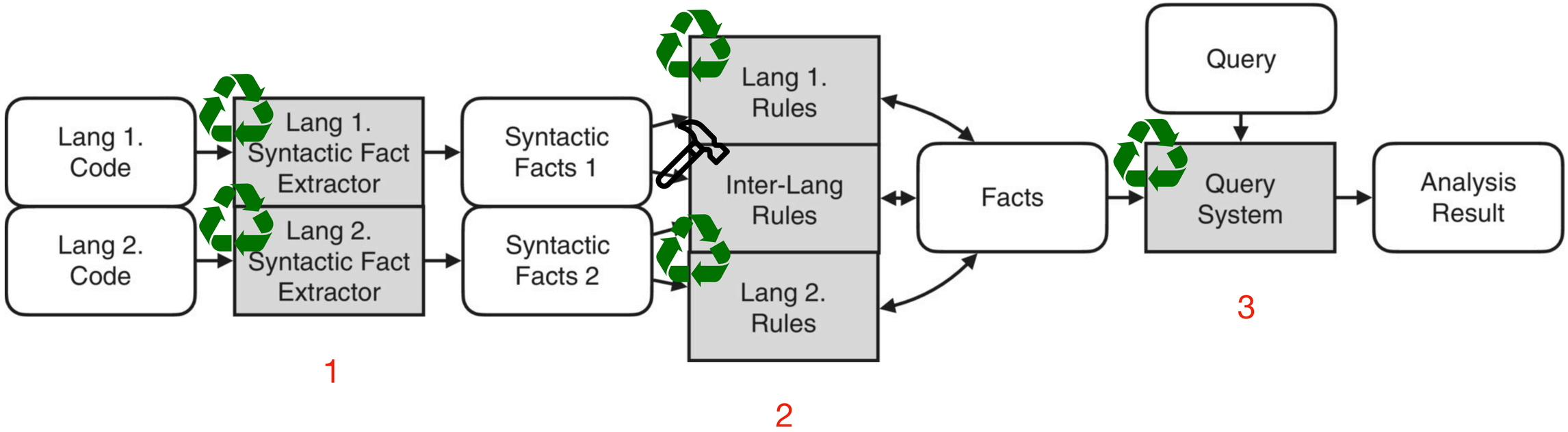
Declarative static analyzer for multilingual program



Declarative static analyzer for multilingual program



Declarative static analyzer for multilingual program



1. Extracting syntactic facts

Code 1:

```
def m1():
  return f()
def m2():
  return 42

int val = m1();
```

→ **Syntactic Fact Extractor** →

Syntactic facts 1:

```
Return_A(“m1", f())
Return_A(“m2", 42)
Call_A(f(), “f")
Call_A(m1(), “m1")
Assign_A(val, m1())
```

Code 2:

```
int f() {
  return m2();
}
```

→ **Syntactic Fact Extractor** →

Syntactic facts 2:

```
Return_B(“f", m2())
```

2. Derive new facts using rules

```
Step_1(from, to) :- ...
Step_2(from, to) :- ...
Step_12(from, to) :- Return_1(name, from), Call_2(to, name)
Step_21(from, to) :- Return_2(name, from), Call_1(to, name)

Step(from, to) :- Step_1(from, to)
Step(from, to) :- Step_2(from, to)
Step(from, to) :- Step_12(from, to)
Step(from, to) :- Step_21(from, to)

Flow(from, to) :- ...
```

2. Derive new facts using rules

Intra-lang rule

```
Step_1(from, to) :- ...
Step_2(from, to) :- ...
Step_12(from, to) :- Return_1(name, from), Call_2(to, name)
Step_21(from, to) :- Return_2(name, from), Call_1(to, name)

Step(from, to) :- Step_1(from, to)
Step(from, to) :- Step_2(from, to)
Step(from, to) :- Step_12(from, to)
Step(from, to) :- Step_21(from, to)

Flow(from, to) :- ...
```

PLRG Programming Language Research Group

KAIST

2. Derive new facts using rules

Intra-lang rule

Inter-lang rule

```
Step_1(from, to) :- ...
Step_2(from, to) :- ...
Step_12(from, to) :- Return_1(name, from), Call_2(to, name)
Step_21(from, to) :- Return_2(name, from), Call_1(to, name)

Step(from, to) :- Step_1(from, to)
Step(from, to) :- Step_2(from, to)
Step(from, to) :- Step_12(from, to)
Step(from, to) :- Step_21(from, to)

Flow(from, to) :- ...
```

2. Derive new facts using rules

Syntactic facts 1:

```
Return_1("m1", f())
Return_1("m2", 42)
Call_1(f(), "f")
Call_1(m1(), "m1")
Assign_1(val, m1())
```

Syntactic facts 2:

```
Call_2(m2(), "m2")
Return_2("f", m2())
```

Lang 1 Rules
+
Inter-lang Rules
+
Lang 2 Rules

```
Step(from, to) :- ...
Flow(from, to) :- ...
```

2. Derive new facts using rules

Syntactic facts 1:

```
Return_1("m1", f())
Return_1("m2", 42)
Call_1(f(), "f")
Call_1(m1(), "m1")
Assign_1(val, m1())
```

Syntactic facts 2:

```
Call_2(m2(), "m2")
Return_2("f", m2())
```

Lang 1 Rules
+
Inter-lang Rules
+
Lang 2 Rules

```
Step(from, to) :- ...
Flow(from, to) :- ...
```

2. Derive new facts using rules

Syntactic facts 1:

```
Return_1("m1", f())
Return_1("m2", 42)
Call_1(f(), "f")
Call_1(m1(), "m1")
Assign_1(val, m1())
```

Syntactic facts 2:

```
Call_2(m2(), "m2")
Return_2("f", m2())
```

Lang 1 Rules
+
Inter-lang Rules
+
Lang 2 Rules

```
Step(from, to) :- ...
Flow(from, to) :- ...
```

2. Derive new facts using rules

Syntactic facts 1:

```
Return_1("m1", f())
Return_1("m2", 42)
Call_1(f(), "f")
Call_1(m1(), "m1")
Assign_1(val, m1())
```

Syntactic facts 2:

```
Call_2(m2(), "m2")
Return_2("f", m2())
```

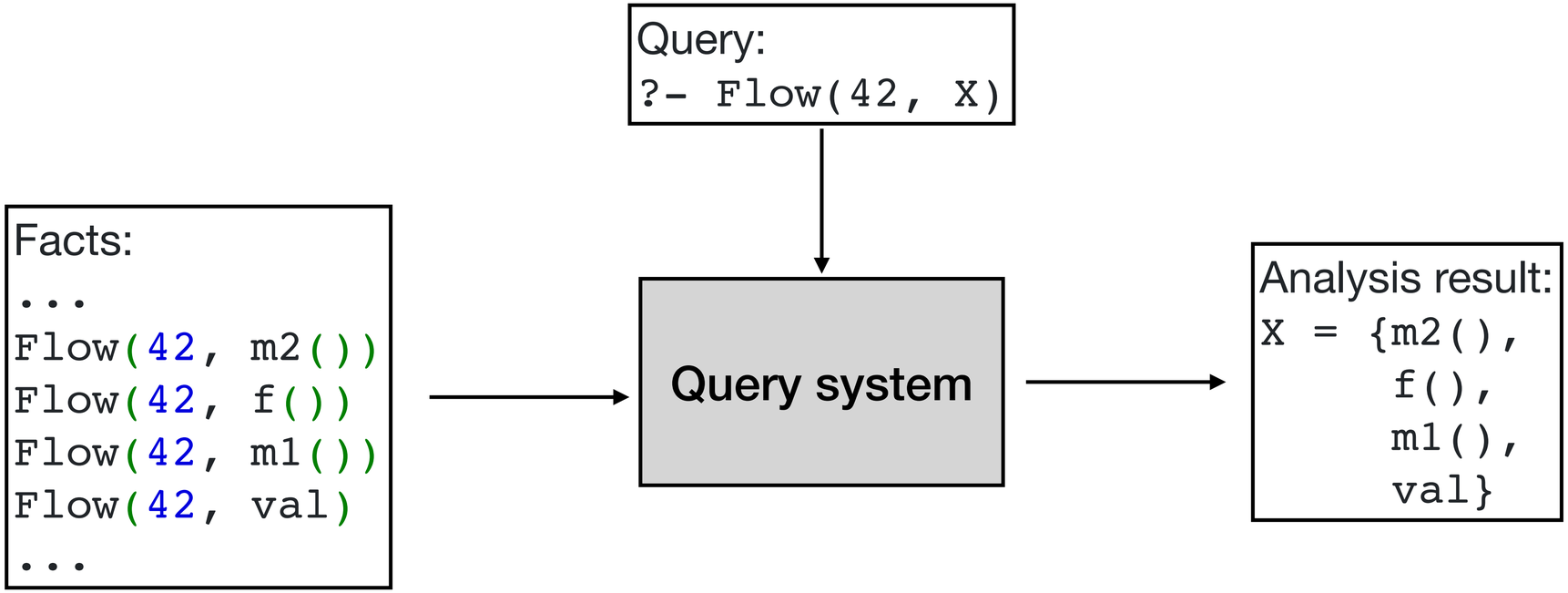
Lang 1 Rules
+
Inter-lang Rules
+
Lang 2 Rules

```
Step(from, to) :- ...
Flow(from, to) :- ...
```

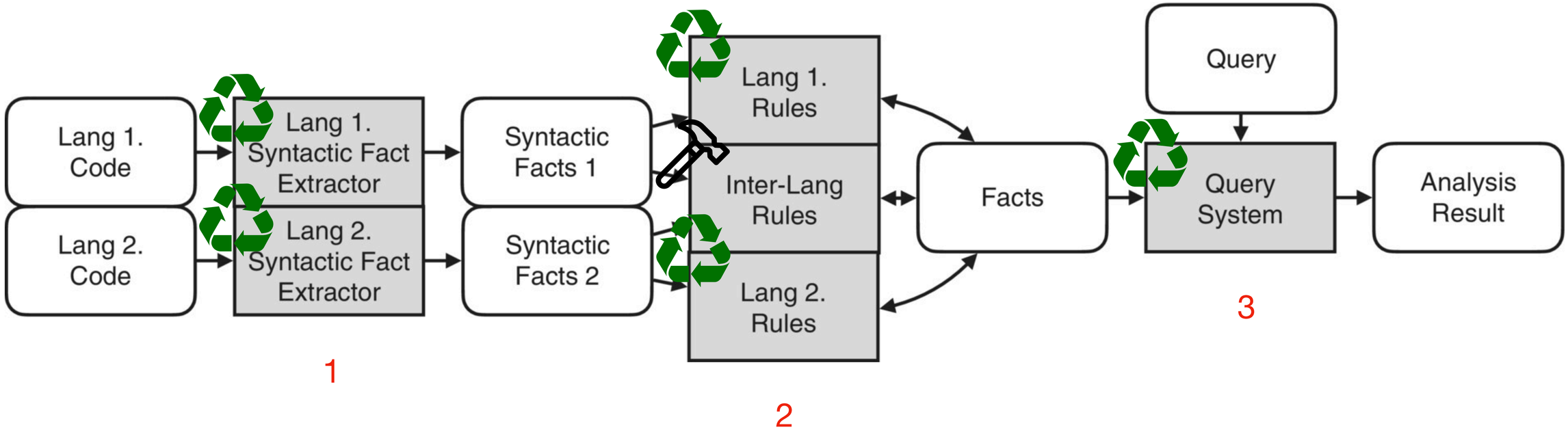
Facts:

```
...
Flow(42, m2())
Flow(42, f())
Flow(42, m1())
Flow(42, val)
...
```

3. Perform query

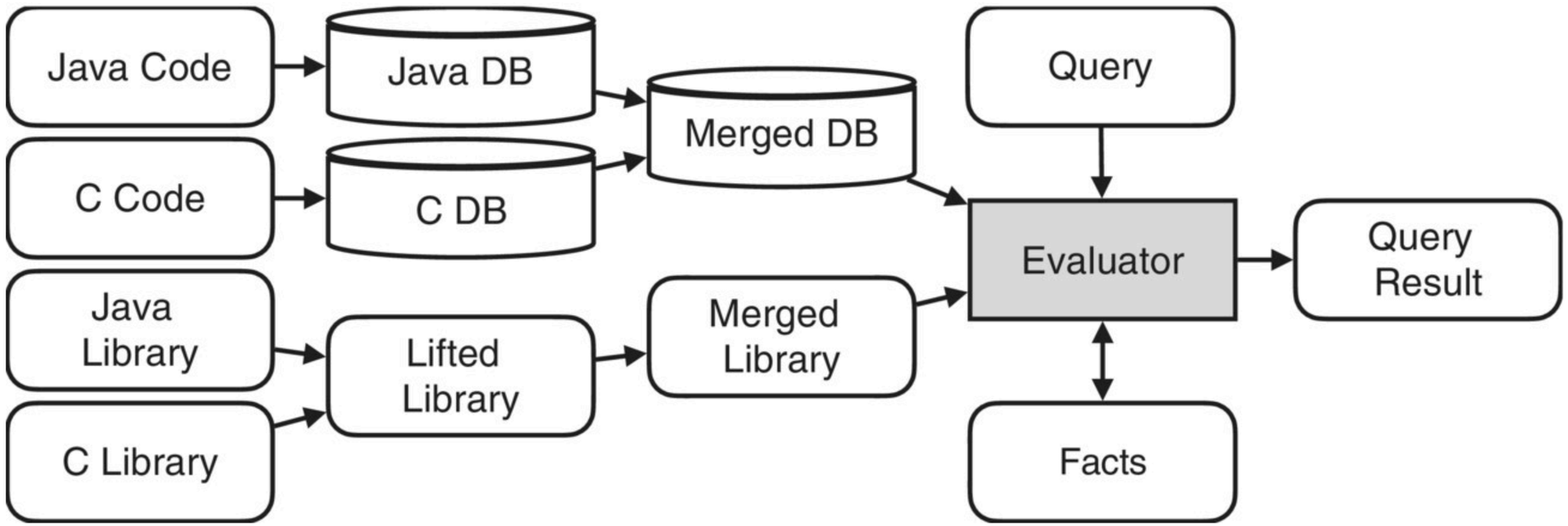


Declarative static analyzer for multilingual program



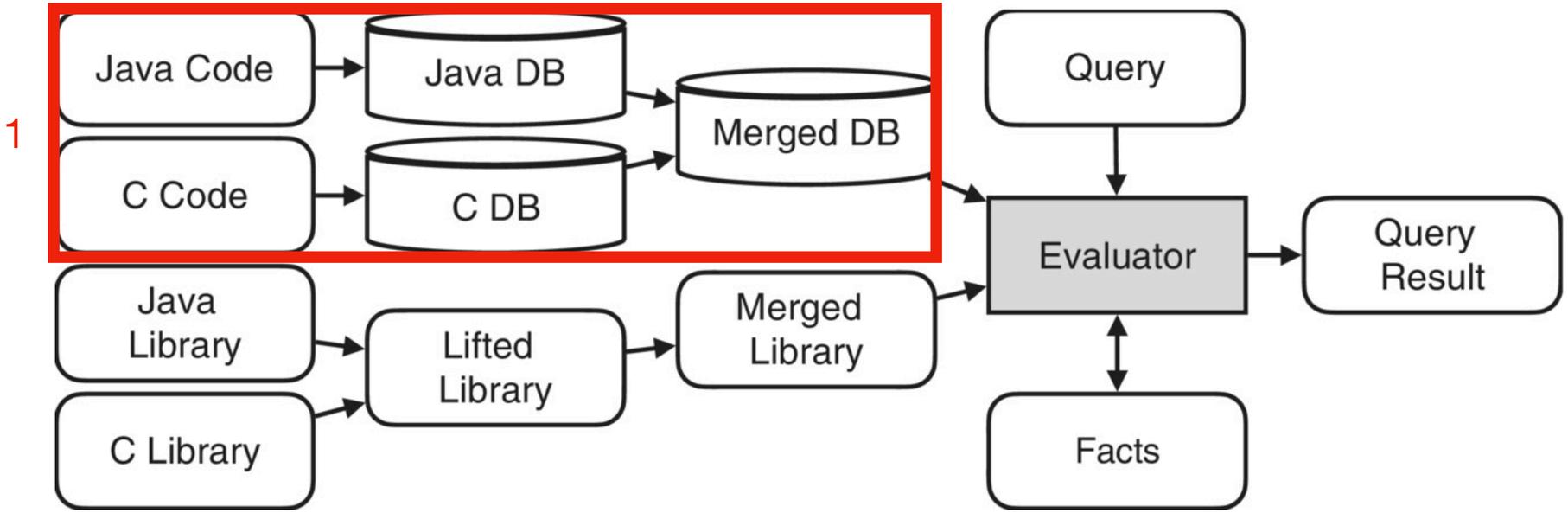
MultiQL

- Proof-of-concept dataflow analyzer for JNI programs / C-Python programs
- Based on CodeQL dataflow analyzer for C, Java, and Python



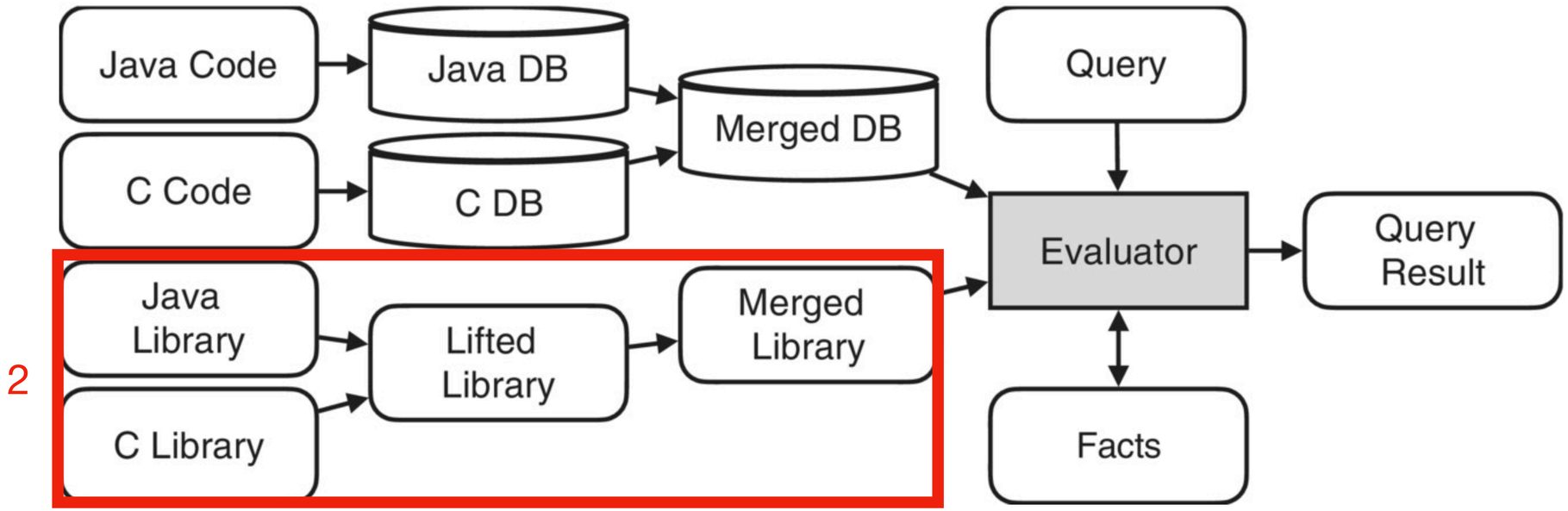
MultiQL

- Proof-of-concept dataflow analyzer for JNI programs / C-Python programs
- Based on CodeQL dataflow analyzer for C, Java, and Python



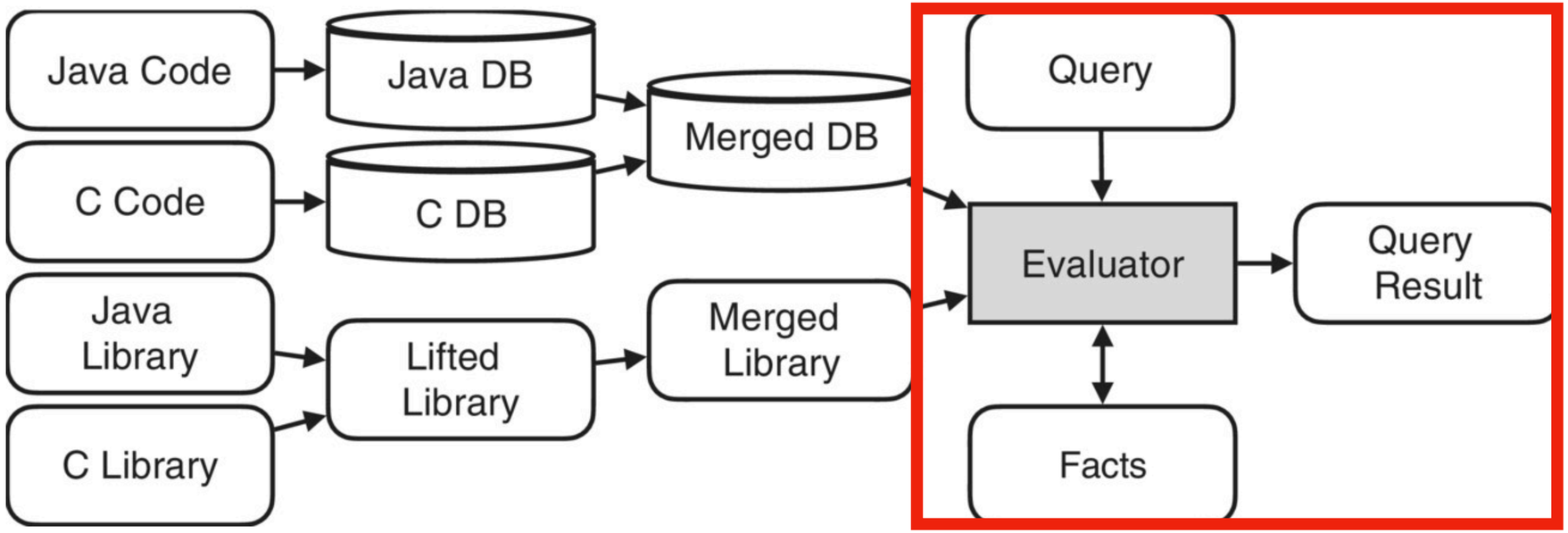
MultiQL

- Proof-of-concept dataflow analyzer for JNI programs / C-Python programs
- Based on CodeQL dataflow analyzer for C, Java, and Python



MultiQL

- Proof-of-concept dataflow analyzer for JNI programs / C-Python programs
- Based on CodeQL dataflow analyzer for C, Java, and Python



3

Test with small benchmarks

	JNI (NativeFlowBench)	C-Python (ExtModuleFlowBench)
MultiQL	19/23	14/20
JN-SAF	21/23	-

Ratio of passed tests

PLRG Programming Language Research Group

KAIST

Test with real-world benchmarks

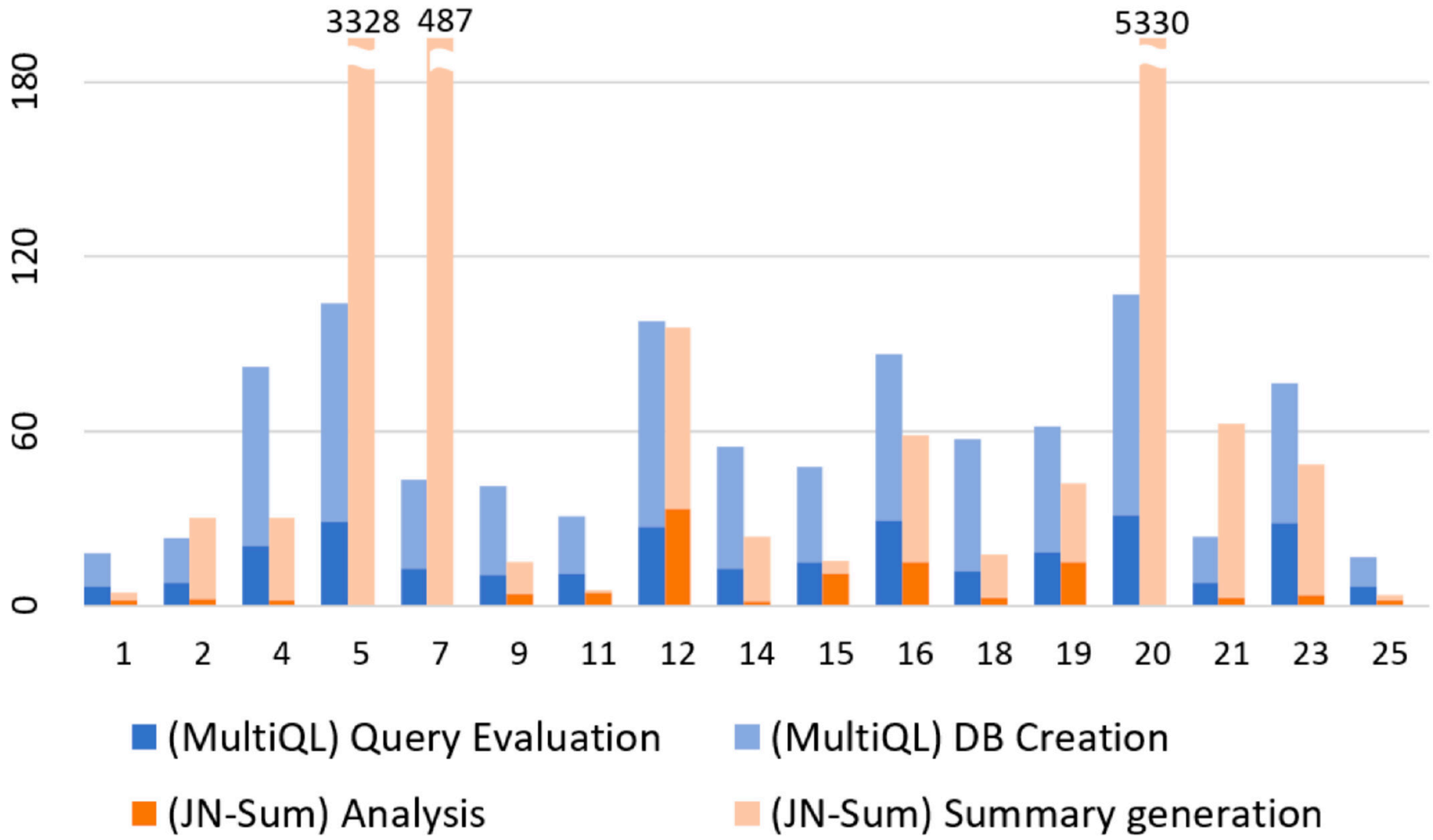
	C->Java API Call	C->Python API Call
MutiQL	92%	69%
JN-Sum	71%	-

Ratio of resolved API Calls

PLRG Programming Language Research Group

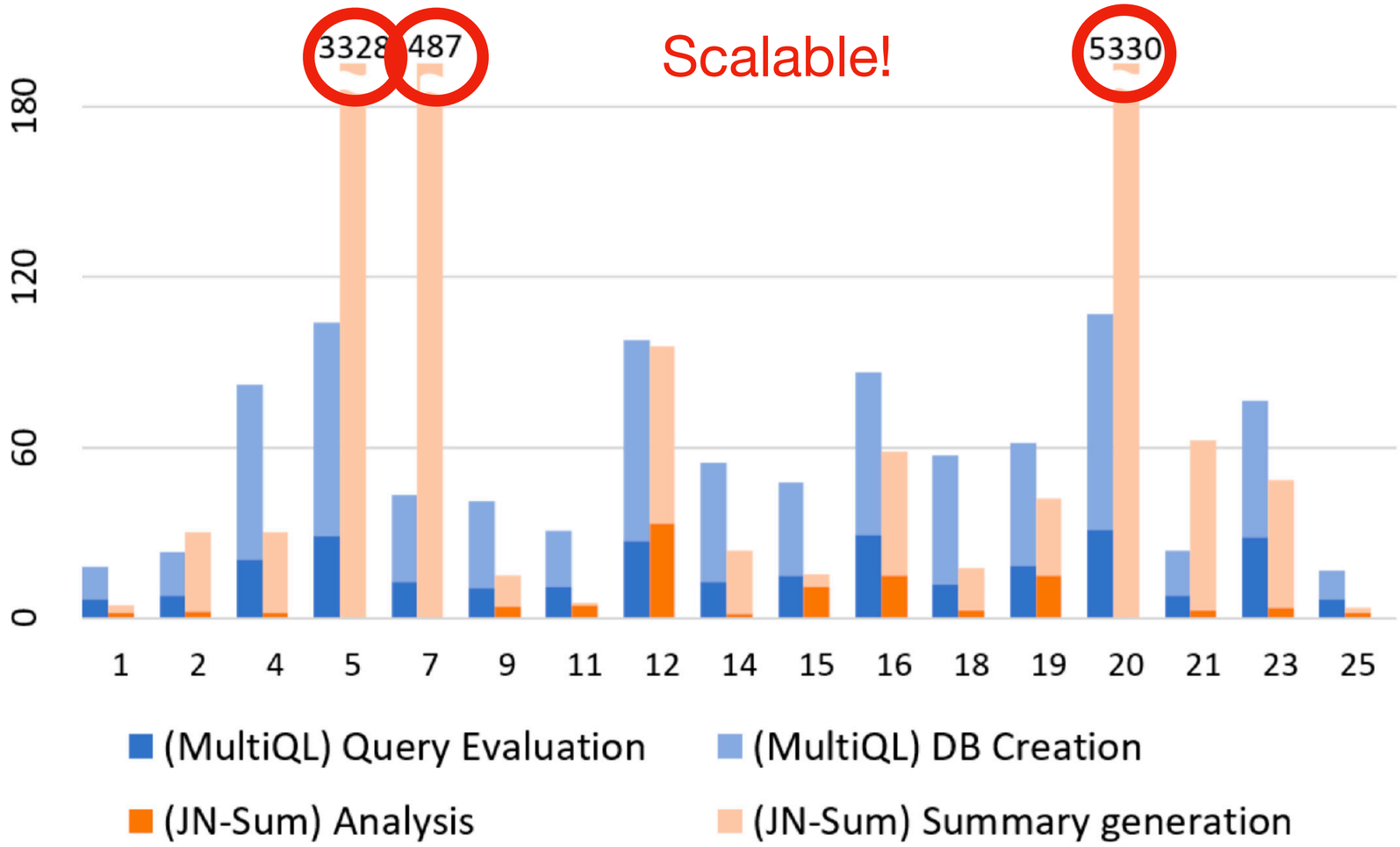
KAIST

Test with real-world benchmarks



Analysis time comparison

Test with real-world benchmarks



Analysis time comparison

Example of found bugs: NULL Dereference

```
1  //EmulatorActivity.java
2  String tmp = null;
3  String folder = Util.GetInternalAppStorage(activity);
4  if (folder != null) {
5    tmp = folder + "tmp";
6    Util.CreateDirectory(tmp);
7  }
8  EmulatorActivity.nativeInitGraph89(..., tmp);
```

```
1  //wrappercommonjni.c
2  void nativeInitGraph89(..., jstring tmp_dir) {
3     (*env)->GetStringUTFChars(env, tmp_dir, 0); ...
4  }
```

(A) NullDeref

Example of found bugs: NULL Dereference

```java
//EmulatorActivity.java
String tmp = null;
String folder = Util.GetInternalAppStorage(activity);
if (folder != null) {
  tmp = folder + "tmp";
  Util.CreateDirectory(tmp);
}
EmulatorActivity.nativeInitGraph89(..., tmp);
```

```c
//wrappercommonjni.c
void nativeInitGraph89(..., jstring tmp_dir) {
   (*env)->GetStringUTFChars(env, tmp_dir, 0); ...
}
```

(A) NullDeref

Example of found bugs: NULL Dereference

```
1  //EmulatorActivity.java
2  String tmp = null;
3  String folder = Util.GetInternalAppStorage(activity);
4  if (folder != null) {
5    tmp = folder + "tmp";
6    Util.CreateDirectory(tmp);
7  }
8  EmulatorActivity.nativeInitGraph89(..., tmp);
```

```
1  //wrappercommonjni.c
2  void nativeInitGraph89(..., jstring tmp_dir) {
3     (*env)->GetStringUTFChars(env, tmp_dir, 0); ...
4  }
```

(A) NullDeref

Example of found bugs: NULL Dereference

```
//EmulatorActivity.java
String tmp = null;
String folder = Util.GetInternalAppStorage(activity);
if (folder != null) {
  tmp = folder + "tmp";
  Util.CreateDirectory(tmp);
}
EmulatorActivity.nativeInitGraph89(..., tmp);
```

```
//wrappercommonjni.c
void nativeInitGraph89(..., jstring tmp_dir) {
   (*env)->GetStringUTFChars(env, tmp_dir, 0); ...
}
```

(A) NullDeref

Example of found bugs: NULL Dereference

```
//EmulatorActivity.java
String tmp = null;
String folder = Util.GetInternalAppStorage(activity);
if (folder != null) {
  tmp = folder + "tmp";
  Util.CreateDirectory(tmp);
}
EmulatorActivity.nativeInitGraph89(..., tmp);
```

```
//wrappercommonjni.c
void nativeInitGraph89(..., jstring tmp_dir) {
    (*env)->GetStringUTFChars(env, tmp_dir, 0); ...
}
```

(A) NullDeref

Example of found bugs: NULL Dereference

```
//EmulatorActivity.java
String tmp = null;
String folder = Util.GetInternalAppStorage(activity);
if (folder != null) {
  tmp = folder + "tmp";
  Util.CreateDirectory(tmp);
}
EmulatorActivity.nativeInitGraph89(..., tmp);
```

```
//wrappercommonjni.c
void nativeInitGraph89(..., jstring tmp_dir) {
    (*env)->GetStringUTFChars(env, tmp_dir, 0); ...
}
```

(A) NullDeref

Declarative Static Analysis for Multilingual Programs using CodeQL