Hidden Truths in Dead Software Paths

Michael Eichberg, Ben Hermann, Mira Mezini, Leonid Glanz,

•••

Presented by: Weijie Huo

November 20th

Motivation

- Code smells make code hard to understand.
- It is not good for sustainable development of the program and it bring real bugs to the program.
- For large program, need to detect code smell automatically
- But the previous approaches used to detect code smell only targeted toward a very specific kind of issues.
- Thus, a generic approach that can detect code smells without targeting any specific kind of issues is necessary.

Terminology

- Unreachable code: part of the source code that can never be executed.
- Dead code: a section in the source code of a program which is executed but whose result is never used.
- Infeasible code: code that never occurs on a feasible execution within its containing procedure.
  Simplified: Combination of unreachable code and the code whose execution always cause the program terminates abnormally

The Approach

A new static analysis technique that exploits abstract interpretation to detect infeasible paths. The analysis is parametrized over abstract domains and the depth of call chains to follow inter-procedurally

Classification of Issues

Useless Code

Confused Conjunctions

Confused Language Semantics

Dead extensibility

Null Confusion

Forgotten Constant

Range double checks

Type Confusion

Unsupported Operation Usage

Unexpected Return Value

Classification of Issues

Obviously Useless Code

Confused Conjunctions

Forgotten Constant

Dead extensibility

Null Confusion

Confused Language Semantics

Type Confusion

Range double checks

Unsupported Operation Usage

Unexpected Return Value

Confused Conjunctions

```
if(maxBits > 4 || maxBits < 8) {
    maxBits = 8;
}
if(maxBits > 8) {
    maxBits = 16;
}
```

Confused Conjunctions

```
if(maxBits > 4 || maxBits < 8) {
    maxBits = 8;
}
if(maxBits > 8) {
    maxBits = 16;
}
```

Unreachable!!

Confused Language Semantics

```
doc = new CachedDocument(uri);

if(doc == null) {

    errorHandling();

}
```

Null Confusion

```
if(att != null) {
	types.add(att);
}
else {
	throw new IOException(paramOutsideWarning);
}
```

What if?

```
if(att != null) {
      types.add(att);
}
else {
      throw new IOException(paramOutsideWarning);
}
....(the value of att does not change in the middle)

if(att != null) {...}
```

What if?

```
if(att != null) {
      types.add(att);
}
else {
      throw new IOException(paramOutsideWarning);
}
```

....(the value of att does not change in the middle)

if(att != null) {...} Redundancy!!

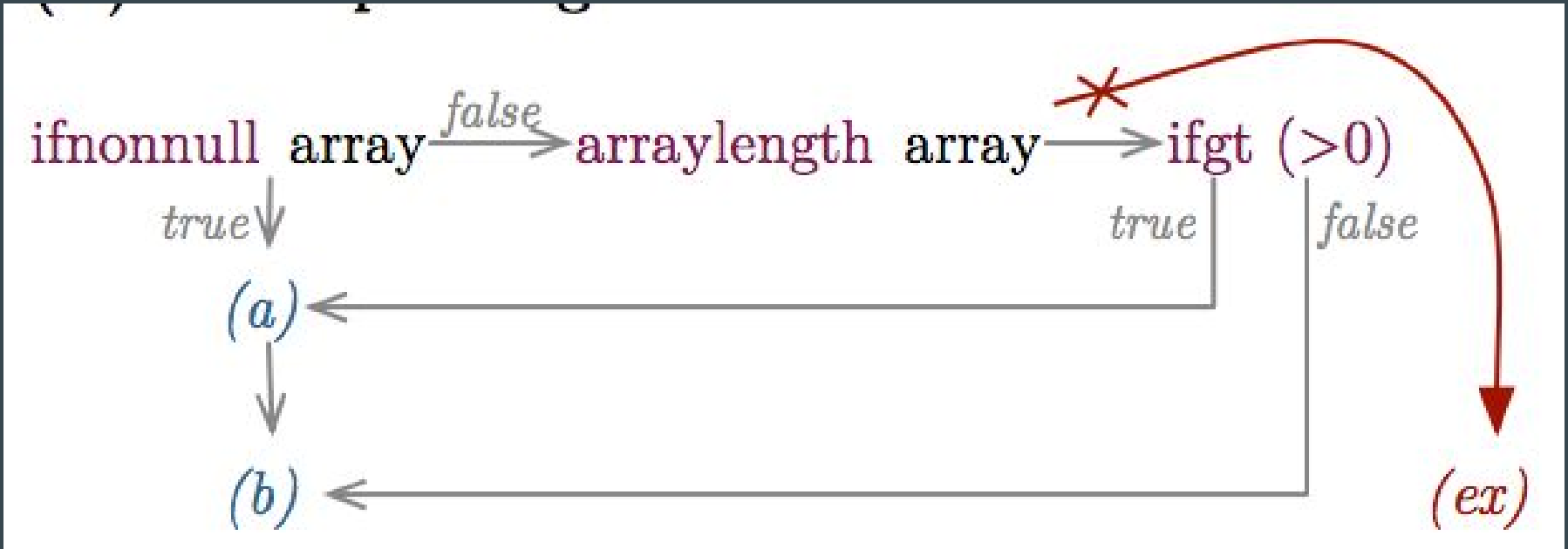
The Approach

- Construct a Control Flow Graph(CFG) for each method.
- Construct an Abstract Interpretation Flow Graph(AIFG).
- Remove all the edges of AIFG from CFG (The remaining edges in CFG are regarded as infeasible code candidates)
- Post-processing Analysis(avoid false positive)

Example

```
public static X doX(SomeType[] array) {
    if(array != null || array.length > 0) {
        ...(a)
    }
    ...(b)
}
```

Control Flow Graph



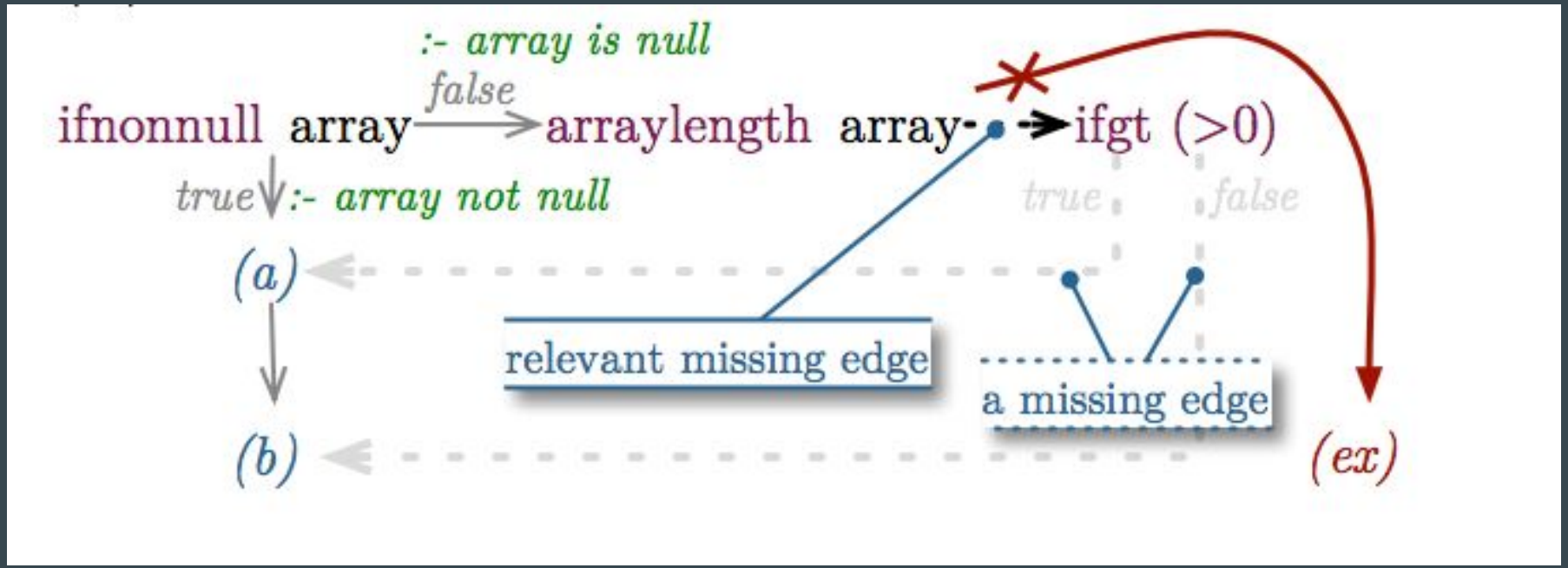
Abstract Interpretation Analysis

- Abstract domains used to represent and perform computations on value.

```
int i = 100;
if(...) i *= 10
else i *= 2        ===> i is in the range of [200, 1000]
```

- The maximum call chain length that the analysis follows.

Abstract Interpretation Flow Graph



Post-processing Analysis

Handle false positive cases such as:

```
Throwable doFail() {throw new Exception;}

Object compute(Object o) {
    if(o == null) {return doFail();}
    else return o;
}
```

Evaluation

Run 109 projects of Qualitas (Curated Collection of Java Code Empirical)

Increase the max call chain length ⇒ more identified issues, but takes more time.

Max Call Chain Length	Max Integer Ranges Cardinality	Issues Relevant	Issues Filtered	Issues Total	Time [s]	Aborted Methods
1	2	690	1157	1847	10	0
1	4	699	1172	1871	11	0
1	8	701	1180	1881	13	0
1	16	702	1182	1884	21	0
1	32	702	1182	1884	27	0
2	2	1078	1248	2326	25	0
2	4	1090	1269	2359	31	0
2	8	1093	1277	2370	43	0
2	16	1094	1279	2373	73	0
2	32	1094	1279	2373	149	1
3	2	1189	1252	2441	60	0
3	4	1201	1273	2474	78	0
3	8	1204	1281	2485	139	0
3	16	1205	1283	2488	289	1
3	32	1205	1283	2488	894	10
4	2	1224	1252	2476	156	0
4	4	1236	1273	2509	205	1
4	8	1237	1281	2518	438	7
4	16	1237	1283	2520	1259	27
4	32	1233	1283	2516	6117	63
5	2	1225	1252	2477	457	4
5	4	1235	1273	2508	990	7
5	8	1239	1281	2520	1566	20
5	16	1234	1283	2517	5482	80
5	32	1233	1283	2516	39274	143

Questions

What make their technique of finding unreachable code different from one performed by every good compiler like LLVM or gcc to find unreachable code and eliminated it from the code. They mentioned in “related work” section that compilers cannot do abstract interpretation but why they cannot do it

What additional categories of issues do you foresee the approach presented to be able to identify? Why do you think these categories are possible?