Principles of Software Construction: Objects, Design, and Concurrency

{Static & Dynamic} x {Typing & Analysis}

Jonathan Aldrich Bogdan Vasilescu

Carnegie Mellon University
School of Computer Science

S3D
Software and Societal
Systems Department

Quiz time!

- Go to Canvas and do the Lecture 22 Quiz
  - access code: patterns

Is this code buggy?

```
private static int getValue(Integer i) {
    return i.intValue();
}
```

How Do You Find Bugs?

- Run it?

```java
public class Fails {
    public static void main(String[] args) {
        getValue( i: null);
    }

    private static int getValue(Integer i) {
        return i.intValue();
    }
}
```

```
Exception in thread "main" java.lang.NullPointerException Create breakpoint : Cannot invoke "java.lang.Integer.intValue()" because "i" is null
    at misc.Fails.getValue(Fails.java:9)
    at misc.Fails.main(Fails.java:5)
```

How Else Can You Find Bugs?

```java
public class Fails {
    public static void main(String[] args) {
        getValue( i: null);
    }

    private static int getValue(Integer i) {
        return i.intValue();
    }
}
```

IntelliJ can look at this code and say:

```
public static void main(String[] args) {
    getValue( i: null);
}

private static int getValue(Integer i) {
    return i.intValue();
}
```

Passing 'null' argument to parameter annotated as @NotNull

(with annotations explicit)

Fails.java

```java
import org.jetbrains.annotations.NotNull;

public class Fails {
    private static int getValue(@NotNull Integer i) {
        return i.intValue();
    }

    public static void main(String [] args){
        getValue( null);
    }

}
```

Static Analysis!

How?

```java
public static void main(String[] args) {
    getValue( i: null);
}

private static int getValue(Integer i) {
    return i.intValue();
}
```

Passing 'null' argument to parameter annotated as @NotNull

Static Analysis!

How?

- We know at *compile time* where `getValue` gets routed to
- `getValue` calls a method on `i`
- `i` can be `null`

```
public static void main(String[] args) {
    getValue( i: null);
}

private static int getValue(Integer i) {
    return i.intValue();
}
```

Passing 'null' argument to parameter annotated as @NotNull

What about JS?

fails.js

```
function getValue(x) {
    return x.valueOf();
}
```

What about JS?

Run it: ✓

fails.js > ...

```javascript
function getValue(x) {
    return x.valueOf();
}

console.log(getValue("32"));
console.log(getValue(null));
```

PROBLEMS 3 OUTPUT TERMINAL DEBUG CONSOL

```
    return x.valueOf();
             ^

TypeError: Cannot read property 'valueOf' of null
```

Why no warning?

```
function getValue(x) {
    return x.valueOf();
}

console.log(getValue("32"));
console.log(getValue(null));
```

Another Java vs JS Example

```
class Foo {
    constructor(x) {
        this.x = x;
    }
}

function bar(foo) {
    return foo.x;
}

var foo = new Foo(3);
console.log(bar(foo));
console.log(bar(3));
```

```
class Foo {
    int x;
    Foo(int x) {
        this.x = x;
    }
}

public static void main(String[] args) {
    Foo foo = new Foo( x: 3);
    bar(foo);
    bar( foo: 3);
}

private static void bar(Foo foo) {
    System.out.println(foo.x);
}
```

Static vs. Dynamic Typing

- The more knowledge we inject in the code, the more bugs we can catch at compile time
  - Types, nullity annotations, invariants
- At compile-time:
  - Dynamically typed languages assume nothing
    - Types exist only for *values*
  - Static typing is not completely precise either
    - Objects have declared types and run-time types
    - Different “strength” type systems

Static vs. Dynamic Typing

- The more knowledge we inject in the code, the more bugs we can catch at compile time
  - Types, nullity annotations, invariants
- Is it worth it?
  - Dynamic typing can severely limit inference
  - But… static types are a lot of work

Static vs. Dynamic Typing

- The more knowledge we inject in the code, the more bugs we can catch at compile time
  - Types, nullity annotations, invariants
- Is it worth it?
  - Dynamic typing can severely limit inference
  - But… static types are a lot of work

Do Static Type Systems Improve the Maintainability of Software Systems? An Empirical Study

Sebastian Kleinschmager, Stefan Hanenberg
University of Duisburg-Essen
Essen, Germany
sebastian.kleinschmager@stud.uni-due.de
stefan.hanenberg@icb.uni-due.de

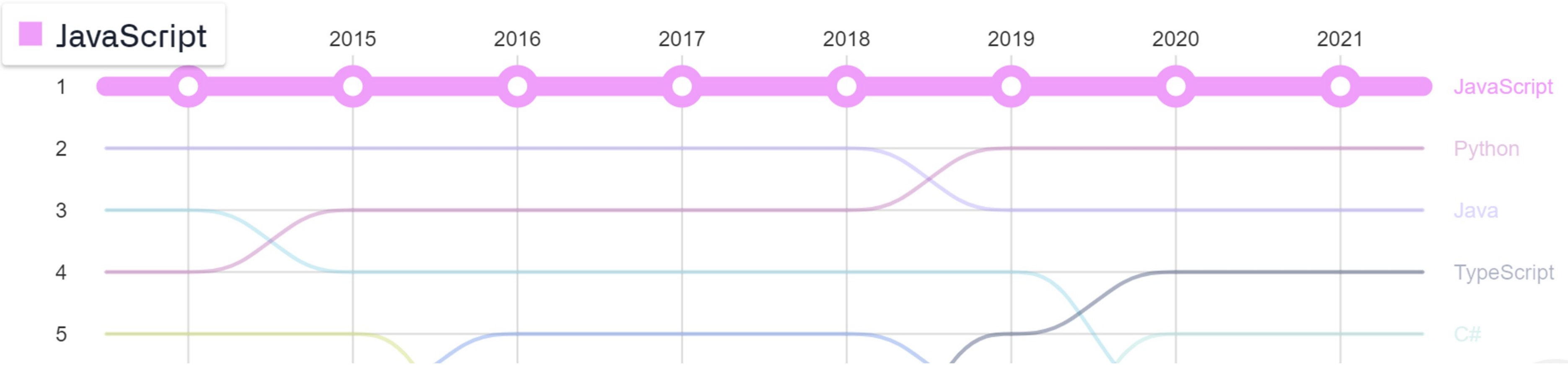
Romain Robbes, Éric Tanter
Computer Science Dept (DCC)
University of Chile, Chile
rrobbes@dcc.uchile.cl
etanter@dcc.uchile.cl

Andreas Stefik
Department of Computer Science
Southern Illinois University Edwardsville
Edwardsville, IL

Static vs. Dynamic Typing

Okay, but:

Top languages over the years



False Dichotomy?

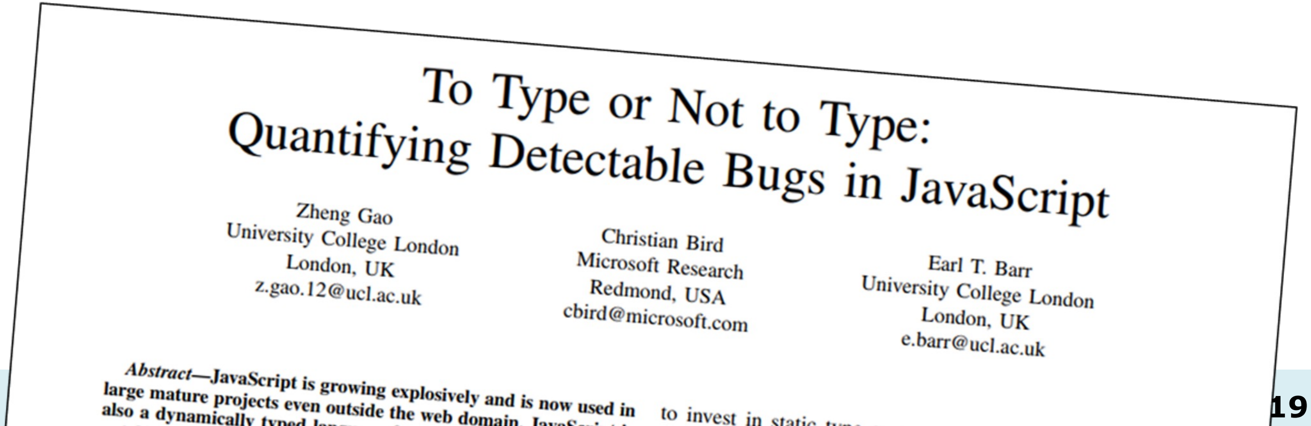
Yes, but:

Top languages over the years

https://octoverse.github.com/#geographical-distribution-of-active-users

Partial Types

- Low effort, some utility
  - Static types exist and are checked at compile-time
  - Dynamic types are used at run-time
    - So annotations get ignored!
  - Type checker can be shallow or deep; TS is shallow



To Type or Not to Type:
Quantifying Detectable Bugs in JavaScript

Zheng Gao
University College London
London, UK
z.gao.12@ucl.ac.uk

Christian Bird
Microsoft Research
Redmond, USA
cbird@microsoft.com

Earl T. Barr
University College London
London, UK
e.barr@ucl.ac.uk

Abstract—JavaScript is growing explosively and is now used in large mature projects even outside the web domain. also a dynamically typed

to invest in static

Types in TypeScript

```
function getValue(x: number) {
    return x.valueOf();
}

console.log(getValue(null));
```

```
Argument of type 'null' is not assignable to parameter of type
'number'. ts(2345)
```

View Problem   No quick fixes available

Types in TypeScript

```
function getValue(x: number | null) {
    return x.valueOf();
}

console.log(getValue(null));
```

```
Object is possibly 'null'. ts(2531)
(parameter) x: number | null
```

View Problem   No quick fixes available

Step Back

- Why do we care about types so much?

Step Back

- Why do we care about types so much?
  - #1 reason: automatically checked documentation
    - Probably dominates “bug finding” advantages in practice
  - But also: we care about *common mistakes*
  - Type errors happen to be very common
  - What else is common?

Step Back

- Why do we care about types so much?
  - #1 reason: automatically checked documentation
    - Probably dominates “bug finding” advantages in practice
  - But also: we care about *common mistakes*
  - Type errors happen to be very common
  - What else is common?
    - Nullity errors
    - Missing imports

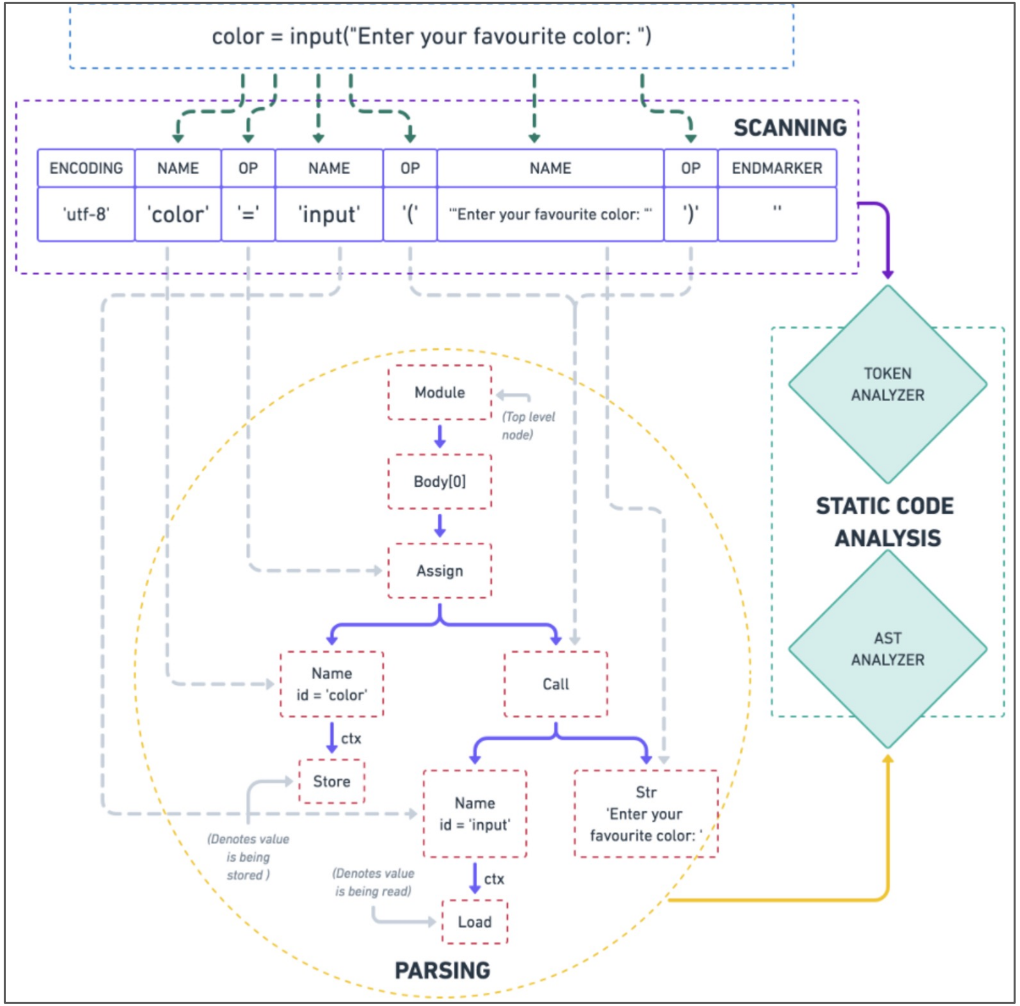
```java
public void forward(String sender) {
    if (sender == "me") {
        sendSelf();
    } else if (sender == "other") {
```

Static Analysis

- Detect real or plausible bugs based on code patterns
  - Plausible: look for risk-prone areas
    - Deeply nested loops
    - Overly general types (e.g,. ‘any’ in TS)
    - Dead code/unused variables
    - Any other places we often make mistakes?

Static Analysis

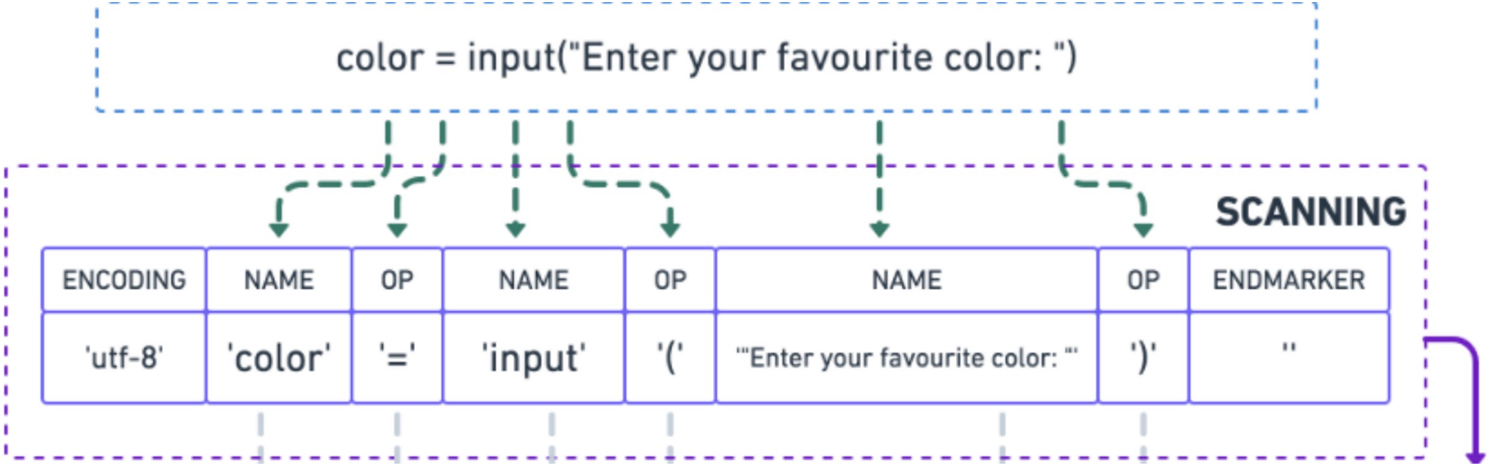
- How?
  - Program analysis + Vocabulary of patterns



https://deepsource.io/blog/introduction-static-code-analysis/

Static Analysis

- Step 1: Tokenization
  - Tokens are like the words of software
  - *Lexical* categories, incl. punctuation, identifiers, operators, strings

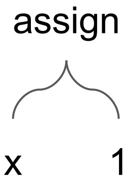


https://deepsource.io/blog/introduction-static-code-analysis/

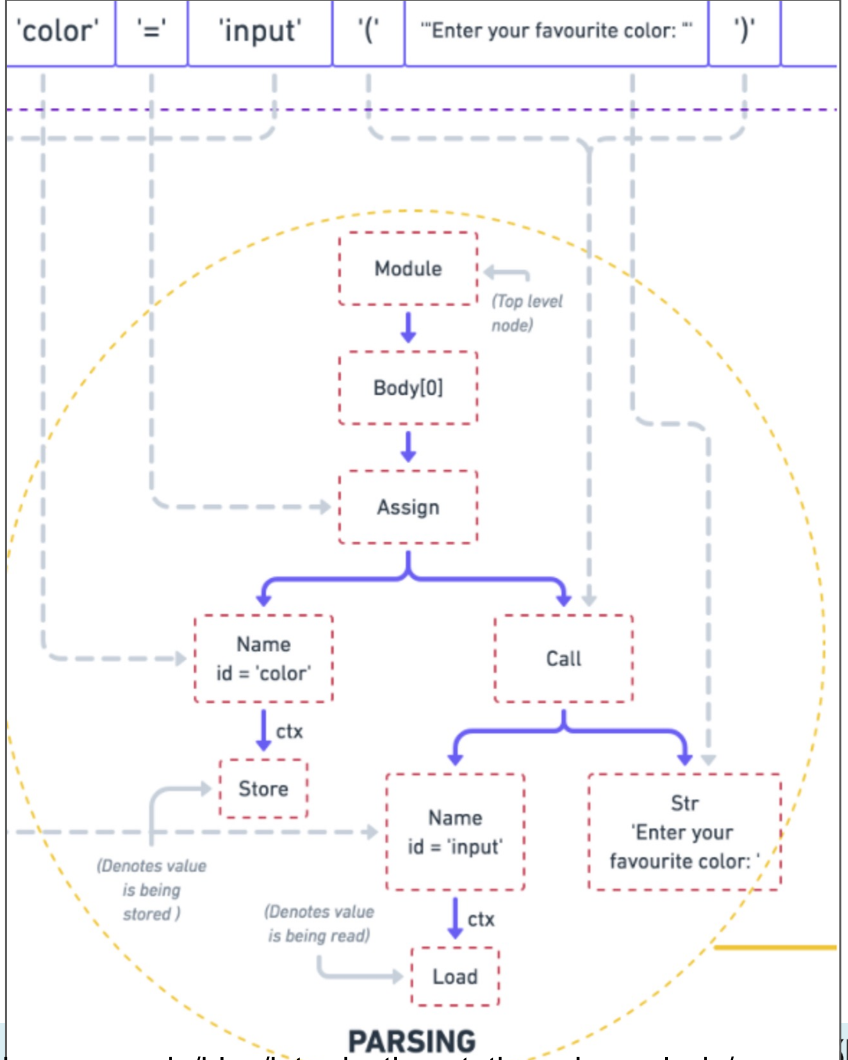
Static Analysis

- Step 2: Parsing
  - To the compiler/interpreter, software is a <u>tree</u>
  - Root node is file/module
  - Leaves mainly identifiers, literals
  - Internal nodes capture *structure*

x = 1 →



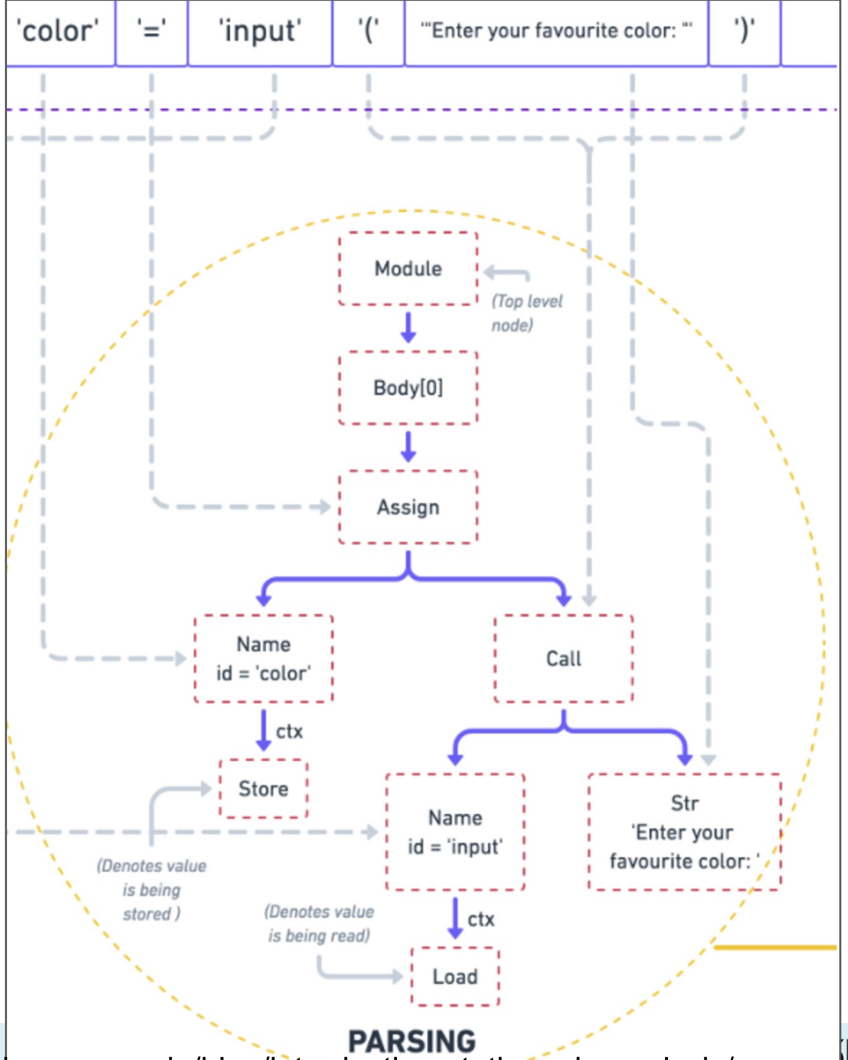
Consider checking out: https://ast.carlosroso.com/



https://deepsource.io/blog/introduction-static-code-analysis/

Static Analysis

- Step 2: Parsing
  - What does this get us?
  - Rich structure
    - Syntactic types (variables, method calls)
    - Dead code, deep nesting
  - A lot of type resolution
    - What vars are stored, loaded
    - Not complete!
    - Need to *build* to understand imports

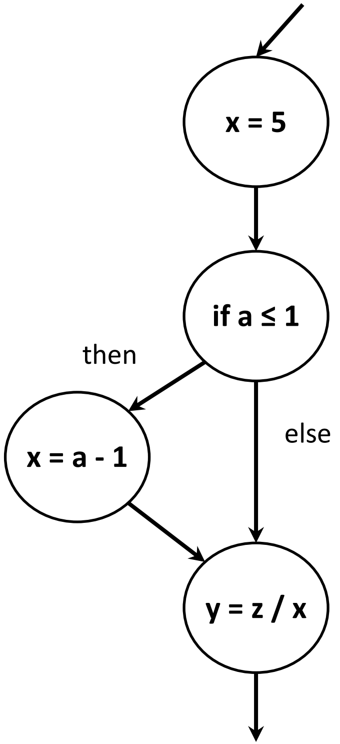


https://deepsource.io/blog/introduction-static-code-analysis/

Static Analysis

- Step 2b: Advanced Analysis
  - The compiler doesn't stop at parsing

```
public boolean div(int a, int z) {
   int x = 5;
   if (a <= 1) {
      x = a - 1;
   }
   return z / x;
}
```



Static Analysis

- Step 2b: Advanced Analysis
  - The compiler doesn't stop at parsing
  - There is *a lot* more down this rabbit hole
    - Control/data-flow, abstract interpretation, (dynamic) symbolic execution,
  - Consider a programming languages, compilers, or program analysis course

Static Analysis

- Step 3: register analyzers
  - At the core: walk the tree

```
class ListDefinitionChecker(BaseChecker):
    msg = "usage of 'list()' detected, use '[]' instead"

    def visit_Call(self, node):
        name = getattr(node.func, "id", None)
        if name and name == list.__name__ and not node.args:
            self.violations.append((self.filename, node.lineno, self.msg))
```

Static Analysis

- Step 3: register analyzers

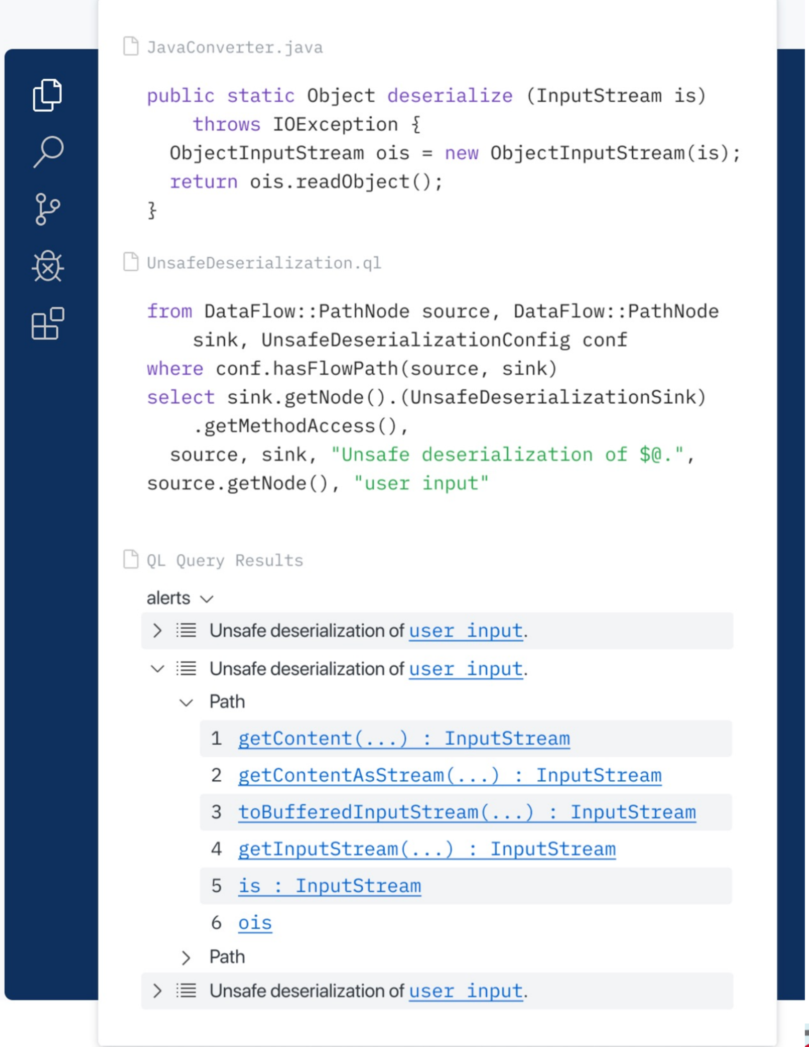
- Classic: walk a tree →
- Modern: build a database of code facts, express analysis as queries over that database.
    - This is how CodeQL works!

```python
class UnusedImportChecker(BaseChecker):
    def __init__(self):
        self.import_map = defaultdict(set)
        self.name_map = defaultdict(set)

    def _add_imports(self, node):
        for import_name in node.names:
            # Store only top-level module name ("os.path" -> "os").
            # We can't easily detect when "os.path" is used.
            name = import_name.name.partition(".")[0]
            self.import_map[self.filename].add((name, node.lineno))

    def visit_Import(self, node):
        self._add_imports(node)

    def visit_ImportFrom(self, node):
        self._add_imports(node)

    def visit_Name(self, node):
        # We only add those nodes for which a value is being read from.
        if isinstance(node.ctx, ast.Load):
            self.name_map[self.filename].add(node.id)
```

https://deepsource.io/blog/introduction-static-code-analysis/

Static Analysis

- Modern: build a database of code facts, express analysis as queries over that database.
- This is how CodeQL works!



Static Analysis

- Compared to Linters:
  - Linters mainly enforce style -- comments, quotes, idioms
    - This also requires static analysis! Just nothing particularly fancy
  - Some overlap; good conventions help avoid bugs

Static Analysis

- Compared to Parsers:
  - Parsers check for syntactic correctness
    - Can catch bugs as well, e.g. missing “;”
  - Parsing is often a key step in static analysis
    - Hard to do right with just text/regexes.
  - Parsing is a platform for further analyses
    - control-flow, data-flow

So… Static Analysis for Everything?

- Can we find every bug?
    - No! Rice's Theorem

      "Any nontrivial property about the language recognized by a Turing machine is undecidable.“ -- Henry Gordon Rice, 1953

    - Every static analysis is necessarily incomplete or unsound or undecidable (or multiple of these)

So… Static Analysis for Everything?

- Can we find every bug?
- Can we guarantee correctness?

So… Static Analysis for Everything?

- Can we find every bug?
- Can we guarantee correctness?
    - Yes (partial correctness anyway), but… much less useful

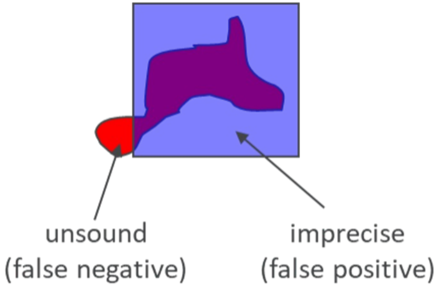
```
public class Fails {
    public static void main(String[] args) {
        getValue( i: null);
    }

    private static int getValue(Integer i) {
        return i.intValue();
    }
}
```

Soundness & Precision

- Since we can't perfectly analyze behavior statically
  - We may miss things by being cautious (unsound; false negative)
  - We might identify non-problems (imprecision, false positive)

Program state covered in actual execution

Program state covered by abstract execution with analysis



The Social Side

- How to deploy tools that are neither sound nor complete?

Static Analysis at Google

- Centered around FindBugs (succeeded by SpotBugs)
  - Essentially, a huge collection of risky patterns on Java bytecode
  - Annotated with five levels of concern


CONTRIBUTED ARTICLES

Lessons from Building Static Analysis Tools at Google

By Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, Ciera Jaspan
Communications of the ACM, April 2018, Vol. 61 No. 4, Pages 58-66
10.1145/3188720
Comments

VIEW AS:

SHARE:

SIGN IN for F

User Name

Password

» Forgot Passwo

Software bugs cost developers and software com

FindBugs™

Static Analysis at Google

- Three experiments in the early 2000s:
    1. **A dashboard:** run FindBugs overnight, report results in a centralized location
       Failed because: dashboard is outside the developer's workflow

Static Analysis at Google

- Three experiments in the early 2000s:
    1. **A dashboard:** run FindBugs overnight, report results in a centralized location
       Failed because: dashboard is outside the developer's workflow
    2. **Recurring FixIt events:** company-wide one-week effort to fix warnings
       Failed because: actually fixed some bugs, but FindBugs is too imprecise (44% of issues were "bugs", but only 16% mattered)

None of these worked!

- Three experiments in the early 2000s:
  1. **A dashboard:** run FindBugs overnight, report results in a centralized location

     Failed because: dashboard is outside the developer's workflow
  2. **Recurring FixIt events:** company-wide one-week effort to fix warnings

     Failed because: actually fixed some bugs, but FindBugs is too imprecise (44% of issues were "bugs", but only 16% mattered)
  3. **Add to Code Review:** run on every change, allow toggling warnings

     Failed because: too imprecise; suppressing FPs made it inconsistent

Static Analysis at Google

Okay so then what?

- What went wrong / what do we need?

Static Analysis at Google

Okay so then what?

- What went wrong / what do we need?
    1. Precision is key -- developers lose faith in inaccurate tools
    2. Provide timely warnings -- in-IDE or rapidly on builds
        a. Checkers are way more useful during coding
    3. Make a platform -- allow adding useful checks

Static Analysis at Google

Specifically:

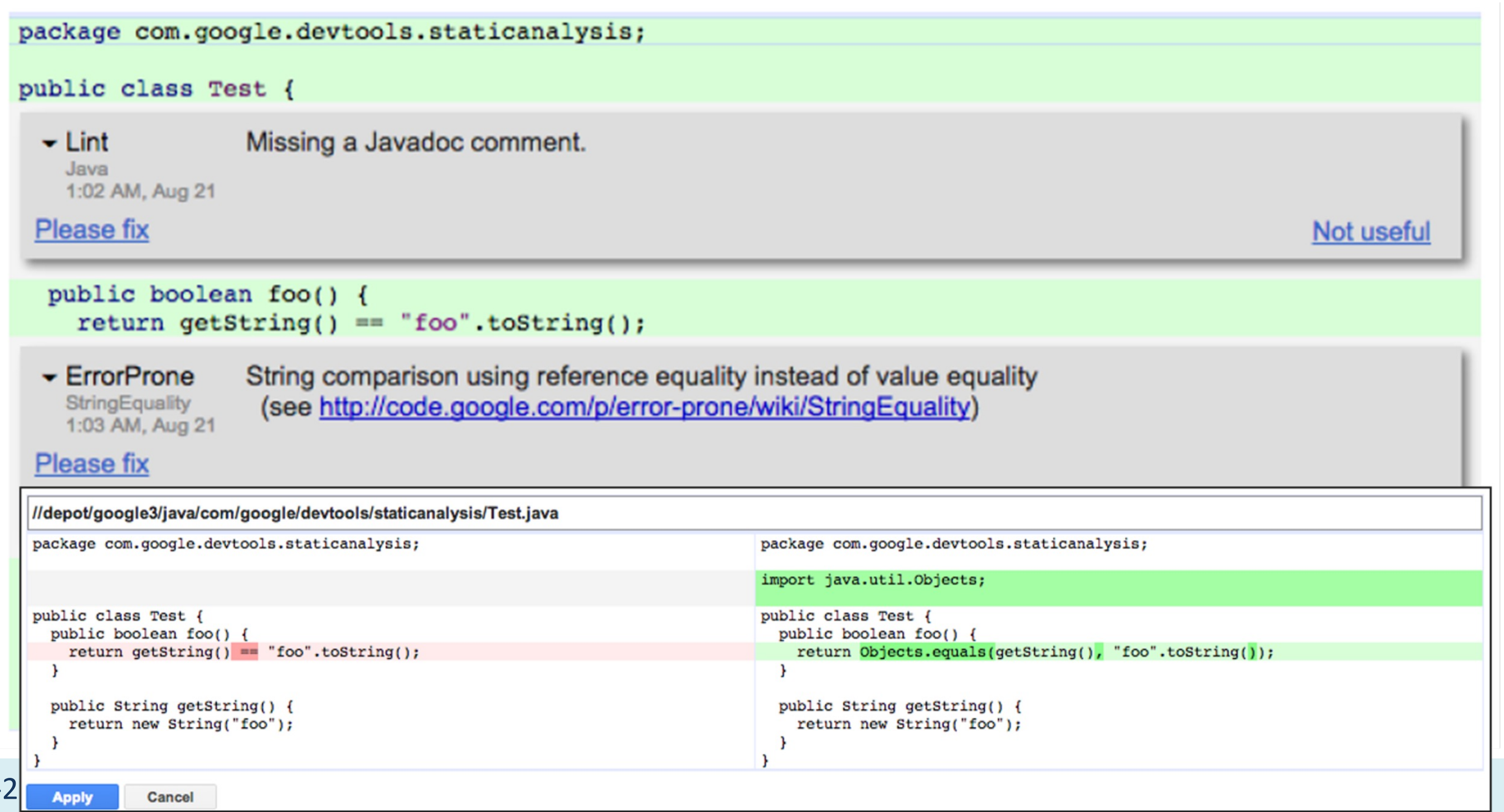
- At compile-time:
  - Perfectly Precise
    - **No** false-positives; never halt a build incorrectly
  - Simple
  - Actionable
    - Ideally to the point of auto-fix suggestions

Static Analysis at Google

Specifically:

- At review time: TriCoder
  - 90%+ precise
    - If it drops below, checker gets disabled! Onus on checker authors to fix
  - Actionable, but may require some work
  - Improve correctness or code quality
  - Some compile-time checks moved to review-time!
- Ran 50K times per day -- in 2018

TriCoder


package com.google.devtools.staticanalysis;
public class Test {
Lint
Java
1:02 AM, Aug 21
Missing a Javadoc comment.
Please fix
Not useful
public boolean foo() {
return getString() == "foo".toString();
ErrorProne
StringEquality
1:03 AM, Aug 21
String comparison using reference equality instead of value equality
(see http://code.google.com/p/error-prone/wiki/StringEquality)
Please fix
//depot/google3/java/com/google/devtools/staticanalysis/Test.java
package com.google.devtools.staticanalysis;
public class Test {
public boolean foo() {
return getString() == "foo".toString();
}
public String getString() {
return new String("foo");
}
}
package com.google.devtools.staticanalysis;
import java.util.Objects;
public class Test {
public boolean foo() {
return Objects.equals(getString(), "foo".toString());
}
public String getString() {
return new String("foo");
}
}
Apply
Cancel

Static Analysis at Google

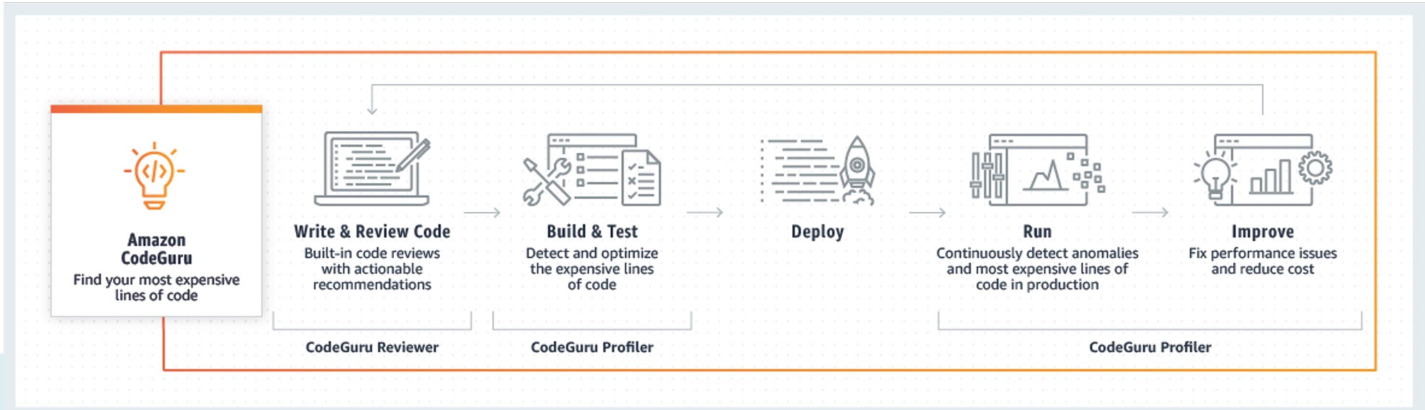
- The gist: Many simple precise checks
  - What else could one do?

Static Analysis at Google

- The gist: Many simple precise checks
  - What else could one do?
- Infer at Facebook
  - Built around separation logic; geared heavily towards tracking resources
    - Null-pointer dereferences, resource leaks, unintended data access
  - Google claims this won't (easily) scale to their multi-billion line mono-repo

Static Analysis at Google

- The gist: Many simple precise checks
  - What else could one do?
- Use AI?
  - Rule-mining from previous reviews
    - Detects typical vulnerabilities, bad patterns
  - Mostly fairly simple ML (details limited)

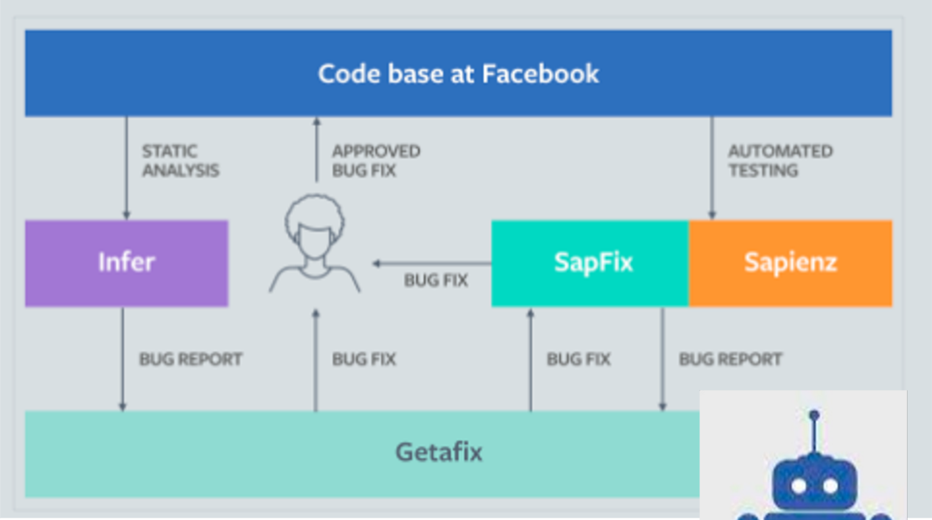


Static Analysis at Google

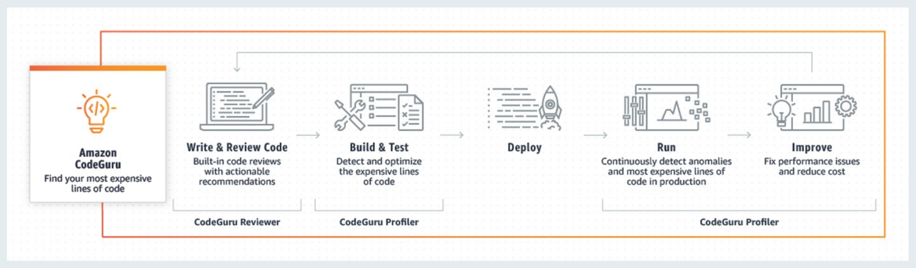
- The gist: Many simple precise checks
    - What else could one do?
- Use AI?
    - Microsoft's IntelliSense in VSCode
    - Mostly refactorings, code completions
    - Trained on large volumes of code

What else could we do?

- Integrate with bug fixing
  - Facebook: Getafix, also integrates with SapFix



- Fancy types / specifications
  - Simple example is Liquid Types – simple logical predicates on types
  - Extreme example: fully specified, statically verified code (cf. CompCert, seL4)
  - More work for developers, but also more payoff
  - Related research projects at CMU
    - Gradual Verification, Liquid Types for Java



Summary

- We all constantly make mistakes
  - Static analysis captures common issues
  - Choose suitable abstractions; consider trade-offs
    - E.g., dynamic vs. static typing; sound vs. precise
- At big-tech-scale, automated checks are key
  - Help normalize coding standards
  - Even rare bugs are common at scale
  - But: social factors are very important
- Active area of research, including here at CMU!
  - Take programming languages, program analysis courses to explore