Lightweight Verification via Specialized Typecheckers

Martin Kellogg
University of Washington
Bugs in software

<table>
<thead>
<tr>
<th>Hours</th>
<th>Seconds</th>
<th>Calculation Time [seconds]</th>
<th>Inaccuracy [seconds]</th>
<th>Approximate Shift in range gate [meters]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3600</td>
<td>3600.000000000000000000000</td>
<td>0.0034</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>28800</td>
<td>28800.000000000000000000</td>
<td>0.0275</td>
<td>55</td>
</tr>
<tr>
<td>24</td>
<td>72000</td>
<td>72000.0000000000000000000</td>
<td>0.0667</td>
<td>197</td>
</tr>
<tr>
<td>48</td>
<td>172800</td>
<td>172800.000000000000000000</td>
<td>0.1668</td>
<td>330</td>
</tr>
<tr>
<td>72</td>
<td>259200</td>
<td>259200.000000000000000000</td>
<td>0.2472</td>
<td>494</td>
</tr>
<tr>
<td>100²</td>
<td>360000</td>
<td>360000.0000000000000000000000000</td>
<td>0.3433</td>
<td>687</td>
</tr>
</tbody>
</table>

etc.
Bugs in software
Goal: every developer uses verification
Preventing bugs: a gross oversimplification

Can find all the bugs?

Usability
Preventing bugs: a gross oversimplification

Usability

Can find all the bugs?

Goal
Preventing bugs: a gross oversimplification

Can find all the bugs?

Testing

Usability

Goal
Preventing bugs: a gross oversimplification

“Testing can only show the presence of bugs, not their absence”
Preventing bugs: a gross oversimplification

Can find all the bugs?

Verification

Testing

Usability

Goal
Preventing bugs: a gross oversimplification

Testing

Usability

Verification

Can find all the bugs?

Pluggable types

Goal
Preventing bugs: a gross oversimplification

Goal #1: make pluggable types easier to adopt
Preventing bugs: a gross oversimplification

Goal #1: make pluggable types easier to adopt

Goal #2: make pluggable types more expressive
Preventing bugs: a gross oversimplification

Goal #1: make pluggable types easier to adopt

Goal #2: make pluggable types more expressive
A new domain: compliance

- Certificates that a company follows a ruleset
  - PCI DSS for credit card transactions
  - HIPAA for healthcare information
  - FedRAMP for US government cloud vendors
  - SOC for information security vendors
  - etc.
A new domain: compliance

- Certificates that a company follows a ruleset
  - PCI DSS for credit card transactions
  - HIPAA for healthcare information
  - FedRAMP for US government cloud vendors
  - SOC for information security vendors
  - etc.

- State-of-the-practice is manual audits of source code
A new domain: compliance

● Certificates that a company follows a ruleset
  ○ PCI DSS for credit card transactions
  ○ HIPAA for healthcare information
  ○ FedRAMP for US government cloud vendors
  ○ SOC for information security vendors
  ○ etc.

● State-of-the-practice is **manual audits** of source code

Developers hate doing this work
A new domain: compliance

- Certificates that a company follows a ruleset
  - PCI DSS for credit card transactions
  - HIPAA for healthcare information
  - FedRAMP for US government cloud vendors
  - SOC for information security vendors
  - etc.

- State-of-the-practice is **manual audits** of source code

- **Insight:** specialized checkers can replace manual audits

A new domain: compliance

- Certificates that a company follows a ruleset
  - PCI DSS for credit card transactions
  - HIPAA for healthcare information
  - FedRAMP for US government cloud vendors
  - SOC for information security vendors
  - etc.

- State-of-the-practice is **manual audits** of source code

- **Insight:** specialized checkers can replace manual audits

> Developers love this, because it saves work

Specialized compliance checkers, industry

Run on 76M NCNB LoC

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Verified</td>
<td>37,315 pkgs</td>
</tr>
<tr>
<td>True pos.</td>
<td>173 pkgs</td>
</tr>
<tr>
<td>False pos.</td>
<td>1 pkg</td>
</tr>
</tbody>
</table>

Specialized compliance checkers, industry

![Continuous Compliance](image)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Verified</td>
<td>37,315 pkgs</td>
</tr>
<tr>
<td>True pos.</td>
<td>173 pkgs</td>
</tr>
<tr>
<td>False pos.</td>
<td>1 pkg</td>
</tr>
</tbody>
</table>

Run on 76M NCNB LoC

Only 23 annotations

Specialized compliance checkers, industry

Run on 76M NCNB LoC

<table>
<thead>
<tr>
<th>Verified</th>
<th>37,315 pkgs</th>
</tr>
</thead>
<tbody>
<tr>
<td>True pos.</td>
<td>173 pkgs</td>
</tr>
<tr>
<td>False pos.</td>
<td>1 pkg</td>
</tr>
</tbody>
</table>

- Auditors accepted output of typecheckers as evidence during a **real audit**
- Checkers **integrated** into build process

---

*Kellogg, Schaef, Tasiran, Ernst. Continuous Compliance. ASE 2020.*
Types vs. other approaches

Recall

- Ours: 100%
- SpotBugs
- Coverity
- CrySL
- CryptoGuard

Precision

- Ours: 100%
- SpotBugs
- Coverity
- CrySL
- CryptoGuard
Preventing bugs: a gross oversimplification

Goal #1: make pluggable types easier to adopt

Goal #2: make pluggable types more expressive
Harder problem: array indexing

```c
T[] a = ...;

int i = ...;

... a[i] ...
```

We need to show that:
● $i$ is an index for $a$
Harder problem: array indexing

```java
T[] a = ...;
int i = ...;
...
a[i] ...
```

We need to show that:
- $i$ is an index for $a$
- $i \geq 0$
- $i < a.length$
Harder problem: array indexing

```java
T[] a = ...;
int i = ...;
... a[i] ...
```

**Insight:** treat array indexing as a collection of problems

We need to show that:
- `i` is an index for `a`
- `i ≥ 0`
- `i < a.length`
Harder problem: array indexing

T[] a = ...;

int i = ...;

... a[i] ...

We need to show that:
- i is an index for a
- i ≥ 0
- i < a.length

Insight: treat array indexing as a collection of problems

build many analyses instead of just one

Kellogg, Dort, Millstein, Ernst. Lightweight Verification of Array Indexing. ISSTA 2018
Cooperating specialized checkers: array indexing

- Linear inequalities: $i < j$
- Negative indices: $|i| < a.length$
- Equal lengths: $a.length = b.length$
- Minimum lengths: $a.length > 10$
- Lower bounds: $i \geq 0$
- Upper bounds: $i < a.length$

*Kellogg, Dort, Millstein, Ernst. Lightweight Verification of Array Indexing. ISSTA 2018*
Summary of results

- Found bugs in industrial codebases (Google Guava)
Summary of results

- Found bugs in industrial codebases (Google Guava)
- vs prior verification approaches (KeY, Clousot):
  - more sound in microbenchmarks
  - equally precise on large codebases
  - more scalable - 10 min vs 3 hrs to check 100k LoC
Preventing bugs: a gross oversimplification

Usability

Testing

Goal #1: make pluggable types easier to adopt

Pluggable types

Goal #2: make pluggable types more expressive

Verification

Can find all the bugs?
Preventing bugs: a gross oversimplification

Goal #1: make pluggable types easier to adopt

Goal #2: make pluggable types more expressive

Usability

Testing

Pluggable types

Verification

Can find all the bugs?
Typestate analysis

```java
File f = ...;
f.open();
File f2 = f;
f.close();
f2.read();
```
File f = ...;
f.open();
File f2 = f;
f.close();
f2.read();

Typestate analysis
Typestate analysis

File f = ...;
f.open();
File f2 = f;
f.close();
f2.read();

Aliasing!
The builder pattern

```java
UserIdentity identity = 
    UserIdentity.builder()
    .name(username)
    .id(generateRandom(32))
    .build();
```

The builder pattern

```java
UserIdentity identity = 
    UserIdentity.builder()
    .name(username)
    .id(generateRandom(32))
    .build();
```

The builder pattern

```java
UserIdentity identity = 
    UserIdentity.builder()
    .name(username)
    .id(generateRandom(32))
    .build();
```

```java
UserIdentity identity = 
    UserIdentity.builder()
    .name(username)
    .build();
```

---

The builder pattern

```java
UserIdentity identity = UserIdentity.builder()
    .name(username)
    .id(generateRandom(32))
    .build();
```

```java
UserIdentity identity = UserIdentity.builder()
    .name(username)
    .build();
```

The builder pattern

**Key insight:**
No loops in this FSM!
(except self loops)

*Kellogg, Ran, Sridharan, Schaef, Ernst. Verifying Object Construction. ICSE 2020.*
The builder pattern

“accumulation analysis”

Key insight:
No loops in this FSM!
(except self loops)
Accumulation analysis

A typestate analysis whose state representation is a *monotonically-increasing set*.
Accumulation analysis

A typestate analysis whose state representation is a _monotonically-increasing set_.

Advantages:
- Does **not** require alias analysis for soundness
- Modular

_Kellogg, Ran, Sridharan, Schaef, Ernst. Verifying Object Construction. ICSE 2020._
User study

Task: add a new required field to a builder

Results:
- +50% success rate
- ~50% faster

Accumulation for resource leaks

```java
try {
    Socket s = new Socket(address, port);
    ...
    s.close();
} catch (IOException e) {
}
```

Accumulation for resource leaks

```java
try {
    Socket s = new Socket(address, port);
    ...
} catch (IOException e) {
    // Missing call to close()
}
```
Accumulation for resource leaks

3 stage checker:

1. taint-tracker over-approximates methods that need to be called
2. accumulation under-approximates methods that have been called
3. dataflow analysis compares the two at “going out-of-scope” points
Accumulation for resource leaks: results

For full results, come to our talk on 26 August, 4pm Athens time ;)

Accumulation for resource leaks: results

Recall

- RLC
- Eclipse: 100%
- Grapple

Precision

- RLC
- Eclipse: 100%
- Grapple

Time

- RLC
- Eclipse
- Grapple: ~37 hrs

Accumulation: future plans
Accumulation: future plans

**Big question:** How much of typestate is accumulation?
How much of typestate is accumulation?

What we know for sure is accumulation:

- **builders**
- **resource leaks**
How much of typestate is accumulation?

Plan #1: survey the literature
How much of typestate is accumulation?

Plan #1: survey the literature

Example: Dwyer, Avrunin, and Corbett (ICSE 1999) split finite-state properties into 8 patterns
How much of typestate is accumulation?

Plan #1: survey the literature

Example: Dwyer, Avrunin, and Corbitt (ICSE 1999) split finite-state properties into 8 patterns

5/8 can be expressed as accumulation
How much of typestate is accumulation?

Plan #1: survey the literature

Example: Dwyer, Avrunin, and Corbitt (ICSE 1999) split finite-state properties into 8 patterns. 5/8 can be expressed as accumulation. 60% of specifications they found in the wild!
How much of typestate is accumulation?

**Plan #1**: survey the literature

**Plan #2**: look for real problems solved with typestate
How much of typestate is accumulation?

Plan #1: survey the literature

Plan #2: look for real problems solved with typestate

Example: Qi & Myers, POPL 2009 introduced “masked types” for safe object initialization
How much of typestate is accumulation?

Plan #1: survey the literature

Plan #2: look for real problems solved with typestate

Example: Qi & Myers, POPL 2009 introduced “masked types” for safe object initialization

Masked types are an accumulation analysis that accumulates **fields** rather than method calls
Related work

- Testing
- Verification

Usability

Pluggable types

Can find all the bugs?

Goal
Related work

Usability

Testing

Pluggable types

Can find all the bugs?

Verification

Goal
Related work

- continuous integration (Fowler & Foemmel 2006)
- fuzzing (e.g. Zawelski 2014, Padyhe et al. 2019)
- oracle generation (e.g. Blasi et al. 2018)

Testing

Usability

Verification

Can find all the bugs?

Pluggable types

Goal
Related work

Testing

Unsound static analysis
- heuristic bug-finding (e.g. Ayewah et al. 2008)
- symbolic execution (e.g. Bessey et al. 2010)
- intentionally-unsound variants of sound analyses (e.g. Bannerjee et al. 2019, Rahaman et al. 2019, Emmi et al. 2021)

Pluggable types

Verification

Usability

Can find all the bugs?
Related work

- Testing
- Usability

Goal

Can find all the bugs?

Pluggable types
Related work

- Dataflow analysis:
  - via graph-reachability (Reps et al. 1995, Sagiv et al. 1996; Soot/Heros)
  - via abstract interpretation (Cousot & Cousot 1977)
  - via types (i.e. pluggable types, dependent types)
- Typestate (Strom & Yemeni 1986)
  - Heap-monotonic typestate (Fähndrich & Leino 2003)
- Language-based approaches (e.g. Plaid, Rust)

- Proof assistants (e.g. Coq, Isabelle/HOL, Lean)
- Automated theorem provers (e.g. Z3, KeY, ESC/Java, Dafny)

Testing

Usability

Pluggable types

Verification

Can find all the bugs?
Related work

Usability

Testing

Pluggable types

Can find all the bugs?

Verification

Goal
Related work

Usability

Testing
- Formalization (Foster et al. 1999)
- Checker Framework: (Papi et al. 2008)
- Checkers for:
  * Nullness (Dietl et al. 2011, Papi et al. 2008)
  * Immutability (Coblenz et al. 2017, Dietl et al. 2011, Papi et al. 2008)
  * Regular expressions (Spishak et al. 2012)
  * GUI effects (Gordon et al. 2013)
  * Locking discipline (Ernst et al. 2015)
  * Determinism (Mudduluru et al. 2021)
  * etc.

Verification

Pluggable types

Can find all the bugs?

Goal
Conclusion

- **Goal:** every developer uses verification
Conclusion

● **Goal:** every developer uses verification

Our contributions:

● Pluggable types are a **powerful and useable** kind of verification
● Using types in **new domains** makes devs want to do verification
● **Cooperating type systems** can solve hard problems
● **Accumulation** can often replace typestate