Pluggable type systems reconsidered

ISSTA 2018 Impact Paper Award for “Practical Pluggable Types for Java”

Michael D. Ernst
University of Washington

https://checkerframework.org/
Type Checking

```java
int x = "hello world";  
```

Compiler error
Impacts

160+ citations
40+ type systems built using the Checker Framework
Used at Amazon, Google, Uber, startups, Wall Street, ...
Java has syntax to support the Checker Framework

For practitioners: more robust and correct code

For researchers: easier experimentation ⇒ better theory
and more impact

For both: appreciation of type systems
Outline

Credits
Motivation
Contributions
Predicting impact
Ideas and results
Research approach
Outline

Credits

Motivation

Contributions

Predicting impact

Ideas and results

Research approach
Who deserves credit for this paper?

Practical Pluggable Types for Java

Matthew M. Papi  Mahmood Ali  Telmo Luis Correa Jr.  Jeff H. Perkins  Michael D. Ernst
MIT Computer Science and Artificial Intelligence Lab, Cambridge, MA, USA
{mpapi,mali,telmo,jhp,mernst}@csail.mit.edu

Abstract

This paper introduces the Checker Framework, which supports adding pluggable type systems to the Java language in a backward-compatible way. A type system designer defines type qualifiers and their semantics, and a compiler plug-in enforces the semantics. Programmers can write the type qualifiers in their programs and use the plug-in to detect or prevent errors. The Checker Framework is useful both to programmers who wish to write error-free code, and to type system designers who wish to evaluate and deploy their type systems. The plug-in permits more expressive compile-time checking and guarantees the absence of additional errors.

A pluggable type framework serves two key constituencies. It enables a programmer to write type qualifiers in a program and to run a type checker that verifies that the program respects the type system. It enables a type system designer to define type qualifiers, to specify their semantics, and to create the checker used by the programmer.

Programmers wish to improve the quality of their code without disrupting their workflow. To be useful to programmers, a plug-
Who deserves credit for this work?

Who deserves credit for this work?

Werner Dietl

Suzanne Millstein
Outline

Credits

Motivation

Contributions

Predicting impact

Ideas and results

Research approach
Context for the paper

2001-2008 was Static Typing Winter
Dynamic types: flexible, fast development
Type systems:
● Hard to understand
● Many false positives
● Inapplicable to the most important problems

Today:
Type systems:
● Rich, expressive, precise type systems
● Simple, usable
● Address real-world problems
● Errors and security vulnerabilities matter
Why did I work on pluggable type-checking?

**Project:** automated SAT translation (idea: Kautz & Selman)

Implementation optimizations: sharing rather than purely functional

**Problem:** undesired side effects

**Solution:** integrate functional & imperative
Controlling side effects

**Project:** programmer-controlled, statically-enforced immutability

A type system must be:
- Sound: proofs
- Useful: experiments, integration with a language

2001: compiler implementation (Java extension)
2002-2009: 7 more compilers, for 5 languages

**Problem:** huge implementation effort

**Solution:** framework for defining a type system
Pluggable type-checking

**Project:** Type system implementation framework

**Goals:** expressiveness for rich type systems  
conciseness when possible

Express the four parts of a type system:

- Type hierarchy
- Type rules
- Type introduction
- Type refinement

**Problem:** No syntax for pluggable types

**Solution:** Change the language (Java)
Type qualifiers in Java

**Project:** Java language extension
2004: Proposal
2006: Proposal accepted
2005-2014: Implement in javac, draft Java specification
2014: Approved
Today: Consulting on spec and implementation

```java
@Present Optional<String> maidenName;

@Untainted String query;
List<String> strings;
myGraph = (@Immutable Graph) tmp;
class UnmodifiableList<T> implements @Readonly List<T> {}
```
Motivation #2: Formal verification $\supseteq$ type systems

Beautiful, compelling idea
Provides guarantees that testing cannot
Many research papers show successes

Not used in practice
I tried to use it and couldn’t

Exception: Type systems
- Lightweight, practical, familiar
- Partial verification

Can this bring verification to all programmers?
Type-systems Specification and verification

Specifications can be complicated
Verification is hard; complex reasoning

Programmers are reluctant
Not appropriate for every project

Benefits:
● Reliability
● Documentation
● Efficiency
● Reasoning
● IDE tooling
● Leads to simpler designs
Motivation #2a: Teaching

Don’t teach methodology without practical application

Don’t teach methodology without tool support

Experiment: Students using the Checker Framework had more correct programs and higher grades
- No difference in time spent
Motivation #3: Research methodology

A type system must have two properties:

- **Sound**
  - Provides a guarantee
  - No loopholes (except explicit ones)
  - Precision is essential
  - Formal proofs can be useful
Formalizations

\[ h \in \text{Heap} \quad \Rightarrow \quad \text{Addr} \rightarrow \text{Obj} \]
\[ \tau \in \text{Expr} \quad \Rightarrow \quad \text{Set of Addresses} \cup \{\text{null}_a\} \]
\[ \text{Obj} = \text{rType, Fields} \]
\[ \text{OwnerAddr ClassId}_\text{rType} \]
\[ \text{FieldId} \rightarrow \text{Addr} \]
\[ \text{Addr} \cup \{\text{any}_a\} \]
\[ \text{TVarId} \in \text{rType}; \text{ParId} \in \text{Addr} \]

\[ h, \tau, e_0 \leadsto h', \nu_0 \]
\[ \nu_0 \neq \text{null}_a \]
\[ \nu_0 \neq \text{null}_a \]
\[ h_0, \tau', e_2 \leadsto h_2, \nu \]
\[ h' = h_2[\nu_0.f := \nu] \]

\[ \text{OS-Read} : h, \tau, e_0.f \leadsto h', \nu \]

\[ \Gamma \vdash e_0 : N_0 \quad N_0 = u_0.C_0<-> \]
\[ T_1 = f\text{Type}(C_0, \nu) \]
\[ \Gamma \vdash e_2 : N_0 \triangleright T_1 \]
\[ \Gamma \vdash e_0.f = e_2 : N_0 \triangleright T_1 \]

\[ \Gamma \vdash e_0 : N_0 \quad N_0 = - \]

\[ \text{GT-Read} : \Gamma \vdash e_0 : N_0 \quad N_0 = - \]

\[ \Gamma \vdash e_0.f = e_2 : N_0 \triangleright f\text{Type}(C_0, \nu) \]

\[ \Gamma \vdash \text{GT-Update} : \Gamma \vdash e_0.f = e_2 : N_0 \triangleright T_1 \]

\[ \nu_0 \neq \text{any} \]

\[ \text{rp}(u_0, T_1) \]

\[ \Gamma \vdash e_0 : N_0 \quad N_0 = - \]

\[ \Gamma \vdash e_0.f = e_2 : N_0 \triangleright T_1 \]

\[ \Gamma \vdash \text{GT-Update} : \Gamma \vdash e_0.f = e_2 : N_0 \triangleright T_1 \]

\[ \nu_0 \neq \text{any} \]

\[ \text{rp}(u_0, T_1) \]
Motivation #3: Research methodology

A type system must have two properties:

- **Sound**
  - Provides a guarantee
  - No loopholes (except explicit ones)
  - Precision is essential
  - Formal proofs can be useful

- **Useful**
  - Solves a real problem
  - Simple to explain
  - Low usage burden
  - Applicable to real languages, programs, development model
  - Evaluated experimentally

Goal: Make implementation easy
Better experimentation $\Rightarrow$ better theory

Too much research omits one!
Outline

Credits
Motivation

Contributions
Predicting impact
Ideas and results
Research approach
Contributions

- **Syntax** for type qualifiers in Java
- **Checker Framework** for writing type checkers
- **5 checkers** written using the framework
- **Case studies** enabled by the infrastructure
- **Insights** about the type systems
Implementing a type system

Example: Ensure encrypted communication

```java
void send(@Encrypted String msg) {...}
@Encrypted String msg1 = ...;
send(msg1); // OK
String msg2 = ....;
send(msg2); // Warning!
```

The complete checker:

```java
@Target(ElementType.TYPE_USE)
@SubtypeOf(Unqualified.class)
public @interface Encrypted {}
Sample type checkers

- Basic checker (subtyping)
- Null dereferences (@NonNull)
- Errors in equality testing (@Interned)
- Reference immutability (Javari)
- Reference & object immutability (IGJ)

< 500 LOC per checker
Case studies

• Annotated existing Java programs
• Found bugs in every codebase
  – Verified by a human and fixed
• As of summer 2007: 360 KLOC
  – Now: > 1 MLOC
  – Scales to > 200 KLOC

Easy to use
Not too verbose
Not too many false positives
Better than competing tools
Lessons learned

• Type systems
  – Interning
  – Nullness
  – Javari
  – IGJ
• Polymorphism
• Framework design
• Others: supertype qualifiers, simple type systems, inference, syntax, language integration, toolchain, ...

Our mistakes

Analysis on AST
- Common approach
- Solution: build a CFG
- Our dataflow analysis was adopted by Google, Uber, etc.

Poor performance
- Compilation time doubles or worse
- We lost users

Limitation: Generics
- Complex specification
- Difficult to implement correctly
Example type systems

Null dereferences (@NonNull)
>200 errors in javac, Google Collections, ...

Equality tests (@Interned)
>200 problems in Lucene, Xerces, ...

Concurrency / locking (@GuardedBy)
>500 errors in Guava, Tomcat, BitcoinJ, Derby, ...

Fake enumerations / typedefs (@Fenum)
problems in Swing, JabRef

Array indexing (@IndexFor)
89 bugs in Guava, JFreeChart, plume-lib
String contents

Regular expression syntax (@Regex)
56 errors in Apache, etc.; 200 annotations required

Printf format strings (@Format)
104 errors, only 107 annotations in 2.8 MLOC

Method signature format (@FullyQualified)
28 errors in OpenJDK, ASM, AFU

Compiler messages (@CompilerMessageKey)
8 wrong keys in Checker Framework
Security type systems

Command injection vulnerabilities (@OsTrusted)
  5 missing validations in Hadoop

Information flow privacy (@Source)
  SPARTA detected malware in Android apps

Industrial use

You can write your own type system!

Many problems can be expressed as a type system

No run-time overhead; standard VM and tools
Previous work

CQual: Jeff Foster, Alex Aiken, others (1999-2006)
Type qualifiers for the C programming language


ESC/Java: Cormac Flanagan, Rustan Leino, others (2002)
Lightweight verification, programmer-written partial specs

Chose the problem and approach on my own

Closely read to learn lessons, avoid repeating mistakes, give credit, make comparisons
Practical Pluggable Types for Java

Create, evaluate, and use custom type systems
An effective verification methodology

For practitioners: more robust and correct code

For researchers: easier experimentation
⇒ better theory and more impact
Outline

Credits
Motivation
Contributions
Predicting impact
Ideas and results
Research approach
Finding Bugs in Dynamic Web Applications

Shay Artzi, Adam Kiežun, Julian Dobry, Frank Tip, Danny Dig, Amit Paradkar, Michael D. Ernst

MIT CSAIL, {artzi,akiezun,dannydig,ernst}@csail.mit.edu
IBM T.J. Watson Research Center, {dobry,ftip,paradkar}@us.ibm.com

Abstract
Web script crashes and malformed dynamically-generated Web pages are common errors, and they seriously impact usability of Web applications. Current tools for Web-page validation cannot handle the dynamically-generated pages that are ubiquitous on today's Internet. In this work, we apply a dynamic test generation technique, based on combined concrete and symbolic execution, to the domain of dynamic Web applications. The technique generates tests automatically, uses the tests to detect failures, and minimizes the conditions on the inputs exposing each failure, so that the resulting bug reports are small and useful in finding and fixing the underlying faults. Our tool Apollo implements the technique for PHP. Apollo generates test inputs for the Web application, monitors the application for crashes, and validates that the output conforms to the HTML specification. This paper presents Apollo’s algorithms and implementation, and an experimental evaluation that revealed 214 faults in 4 PHP Web applications.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification; D.2.5 [Software Engineering]: Testing and Debugging;
General Terms Reliability, Verification
Keywords Software Testing, Web Applications, Dynamic Analysis, PHP

1. Introduction

manifested as Web application crashes or as malformed HTML. Some faults may terminate the application, such as when a Web application calls an undefined function or reads a nonexistent file. In such cases, the HTML output presents an error message and the application execution is halted.

More commonly in deployed applications, a Web application creates output that is not syntactically well-formed HTML, for example by generating an opening tag without a matching closing tag. Web browsers are designed to tolerate some degree of malformedness in HTML, but this merely masks underlying failures. Malformed HTML is less portable across browsers and is vulnerable to breaking on new browser releases. An application that creates invalid (but displayable) HTML during testing may create undisplayable HTML on different executions. More seriously, browsers’ attempts to compensate for malformed Web pages may lead to crashes and security vulnerabilities. A browser might also succeed in displaying only part of a malformed webpage, silently discarding important information. Search engines may have trouble indexing incorrect pages. Standard HTML renders on more browsers, and valid pages are more likely to look as expected, including on future versions of Web-browsers. Standard HTML renders faster. For example, in Mozilla, “improper tag nesting […] triggers residual style handling to try to produce the expected visual result, which can be very expensive” [25].

Web developers widely recognize the importance of creating legal HTML. Many websites are validated using HTML validators (from the ISSTA’08 website displays the W3C HTML compliance checker).
Reaction at ISSTA 2008

Reviews:

“There is nothing particularly novel”

“The general idea of pluggable types is not new”

“JQual is superior ... [very incomplete list of JQual limitations] ... JQual could be easily enhanced to handle them”

Useful reviews!
Reviewer concerns

Declarative syntax
Inadequate for rich type systems

Results claimed by previous work
The reviewers believed them

Simple, clear explanation ⇒ trivial

Too much engineering
Only 2 new type systems

Programmer writes specs
The reviewers preferred inference
It’s hard to predict impact

160 papers

3 impact awards

10 best-paper awards
Outline

Credits
Motivation
Contributions
Predicting impact

Ideas and results
Research approach
Ideas and results

Idea: a thought or suggestion

Result: evidence that answers a question or provides information

Results lead to the best research

Most ideas are terrible*

Most ideas are not novel*

* including my ideas!
Crisis of reproducibility

Most published findings are false

- bias
- testing by independent teams
- size of studies
- effect sizes
- number of tested relationships
- vagueness/flexibility of definitions
- financial interests
Multiple discovery

Calculus
Oxygen
Evolution

Undefinability theorem, universal computing machine, integrated circuit, Kolmogorov complexity, packet switching, CFG parsing, KMP string searching, Cook-Levin theorem, RSA algorithm, elliptic curve cryptography, distributed hash tables, ...

I have been scooped, and I have scooped others
Results vs. ideas: not a new debate

"Should computer scientists experiment more?"

Critique: Requiring experiments will
● Slow dissemination of ideas
● Slow scientific progress
● Waste resources

Translation:
● My ideas are uniquely valuable
● My ideas are obviously good
● I don’t like the work of evaluation
We should publish some ideas

Some are good!
Some are novel!

Explore the design space and justify your choice
Draw connections
Recognize the limitations of an idea

Different kinds of results are valuable
- Including evaluation, generalization, filtering
Danger in publishing too many ideas

Proposing lots of ideas indiscriminately
- Good for tenure
- Bad for science

A trivial or contrived evaluation is worse than no evaluation

If you believe in your idea, evaluate and implement it
Don’t expect credit for science fiction

The filtering of conferences is valuable
Don’t fetishize algorithmic novelty
Outline

Credits

Motivation

Contributions

Predicting impact

Ideas and results

Research approach
My approach to research

- Work with undergraduates
- Scratch your own itch
- Don’t give up
- Publish your implementations
- Never say “easy” or “obvious”
- Focus on results (pursue ideas to results)
- Community actions

Disclaimer: Your mileage may vary
Work with undergraduates

I got my start because faculty took a gamble on me

CRA-E Undergraduate Research Faculty Mentoring Award, 2018

Be a programmer

Generates lots of good ideas

Ensures your work is relevant

Leads to impact

Fun! (cf. writing grant proposals and grading exams)
Scratch your own itch

Choose problems you care about
Use your tool
- If not, do you believe in it?
- If not, do you understand the domain?

Don’t pursue fads

Don’t use the literature to suggest a project
- If you do, you may solve the wrong problem
- Good for understanding existing techniques
- May inspire new ideas

Choose problems that are grounded in programming
Ask, “Why do I care?” and “How is this actionable?”
Don’t give up

It takes time for your great work to be recognized
It takes time to turn ideas into results
Impact comes from follow-through
Work on ideas you believe in
Believe in yourself, too
Also know when to declare victory and move on
Checker Framework maintenance

As of the ISSTA talk:
● 22 public releases
● >1MLOC of code type-checked

Today:
● Release every month
● Closed 264 issues in the past year
● 30 talks in the last 3.5 years
Publish your implementations

Science is built on reproducibility

Lets others work faster

Increases confidence in your work, citations, impact

Community should prioritize this
Reusable frameworks

Don't claim reusability until you have multiple real instantiations

Checker Framework, as of ISSTA 2008 talk:
- 5 full type-checkers that had found bugs
- 9 universities building type systems
Never say “easy” or “obvious”

“It would be easy to …”
If it's so easy, why don't you do it?

“Conceptually easy, but uninteresting engineering.”
Why don’t you automate or abstract it?
Why don’t you just do it?

“It’s obvious that …”
It’s obvious that the earth is flat
and the sun revolves around it.

Your intuition is wrong beyond human scale
Avoid Aristotelian science
Checker Framework was not easy or obvious

Several previous attempts had failed

Differences in design

Value in case studies to assess pluggable type-checking

Reusable engineering
Focus on results (pursue ideas to results)

There is no substitute for experiments

Rationally assess the value of your ideas
Get external feedback

Publish fewer papers rather than more
  ● Make every paper outstanding

Assess impact rather than counting papers
Learn to read and think, not just to count
  ● Your dean needs to learn this too
Community actions

Give credit where it is due (not necessarily first publication)
Publish some idea papers, but recognize their value
Don't denigrate results (anti-intellectual)

Encourage experimentation, replication, and extensions
Accept papers whose results are convincing but not perfect

Don't blindly believe the claims of related work
Reviewers, hold papers to their claims
Outline

Credits
Motivation
Contributions
Predicting impact
Ideas and results
Research approach
Try the Checker Framework today

You have nothing to lose but your bugs

https://checkerframework.org/