Pluggable type systems reconsidered

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

Michael D. Ernst
University of Washington

CHECKER framework

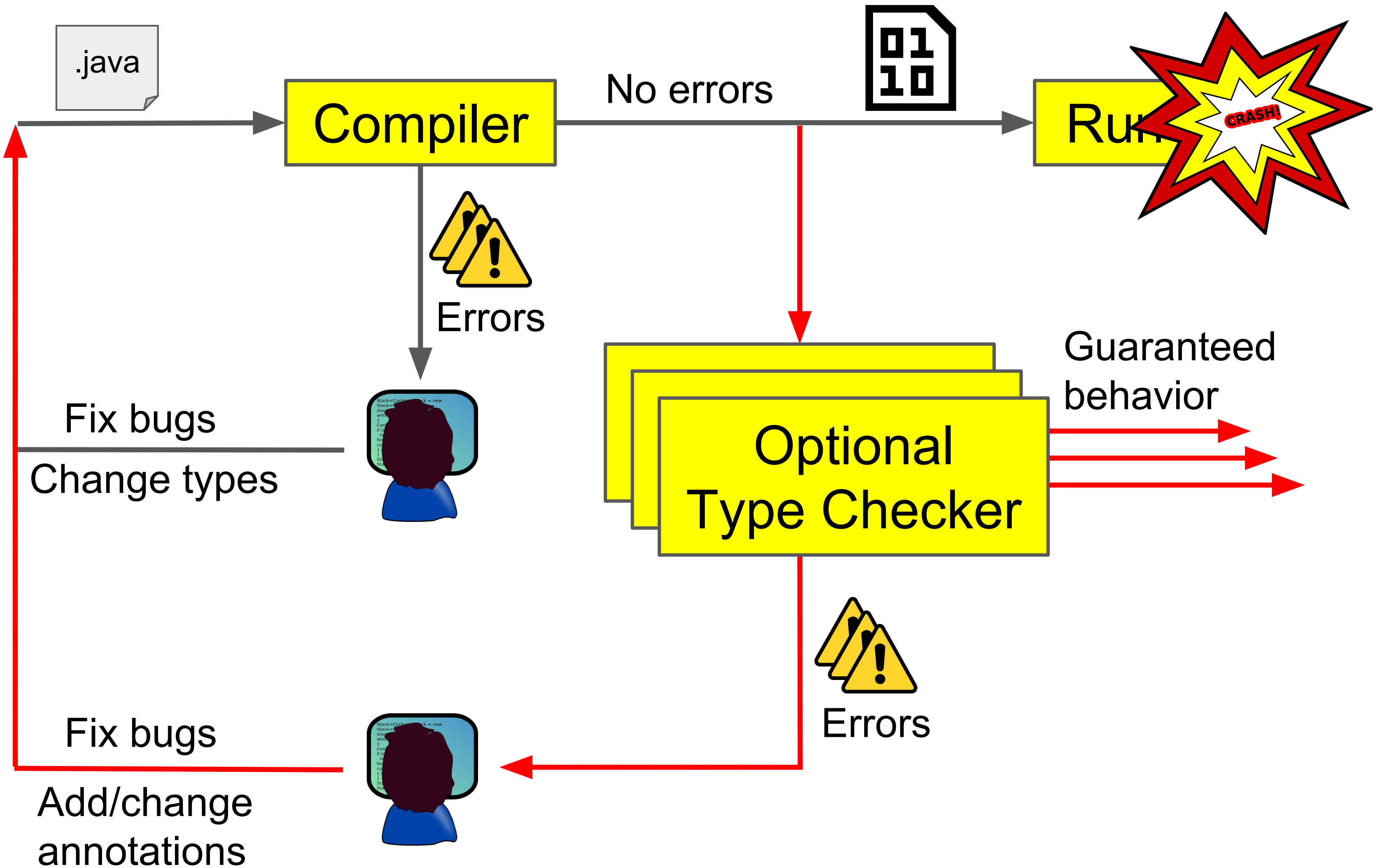
https://checkerframework.org/

Type Checking

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

Compiler error

Optional Type Checking



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?

Undergraduate

Undergraduate

Undergraduate

Programmer

Tenured

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 permit more expressive compile-time checking and guarantee 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

Who deserves credit for this work?

Abraham Lin, Alvin Abdagic, Anatoly Kupriyanov, Arie van Deursen, Arthur Baars, Ashish Rana, Asumu Takikawa, Atul Dada, Basil Peace, Bohdan Sharipov, Brian Corcoran, Calvin Loncaric, Charles Chen, Charlie Garrett, Christopher Mackie, Colin S. Gordon, Dan Brotherston, Dan Brown, David Lazar, David McArthur, Eric Spishak, Felipe R. Monteiro, Google Inc., Haaris Ahmed, Javier Thaine, Jeff Luo, Jianchu Li, Jiasen (Jason) Xu, Joe Schafer, John Vandenberg, Jonathan Burke, Jonathan Nieder, Kivanc Muslu, Konstantin Weitz, Lázaro Clapp, Liam Miller-Cushon, Luqman Aden, Mahmood Ali, Manu Sridharan, Mark Roberts, Martin Kellogg, Matt Mullen, Michael Bayne, Michael Coblenz, Michael Ernst, Michael Sloan, Mier Ta, Nhat Dinh, Nikhil Shinde, Pascal Wittmann, Patrick Meiring, Paulo Barros, Paul Vines, Philip Lai, Ravi Roshan, Renato Athaydes, René Just, Ruturaj Mohanty, Ryan Oblak, Sadaf Tajik, Shinya Yoshida, Stefan Heule, Steph Dietzel, Stuart Pernsteiner, Suzanne Millstein, Tony Wang, Trask Stalnaker, Jenny Xiang, Utsav Oza, Vatsal Sura, Vlastimil Dort, Werner Dietl.

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

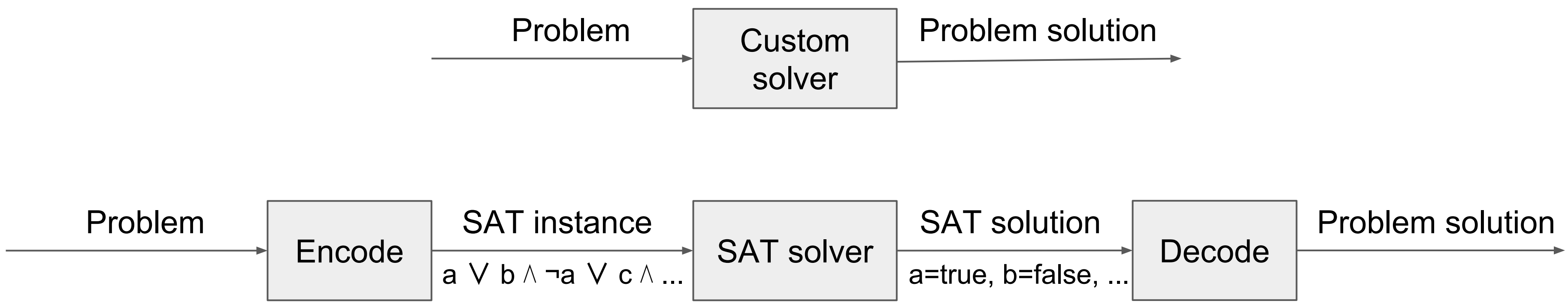
Today:

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?

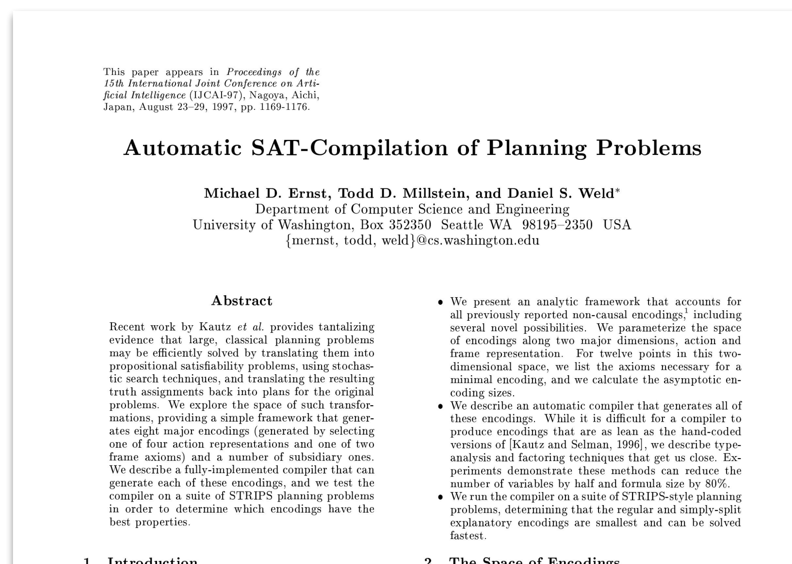
Project: automated SAT translation (idea: Kautz & Selman)



Implementation optimizations: sharing rather than purely functional

Problem: undesired side effects

Solution: integrate functional & imperative



This paper appears in *Proceedings of the 15th International Joint Conference on Artificial Intelligence* (IJCAI-97), Nagoya, Aichi, Japan, August 23–29, 1997, pp. 1169-1176.

Automatic SAT-Compilation of Planning Problems

Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld*
Department of Computer Science and Engineering
University of Washington, Box 352350 Seattle WA 98195–2350 USA
{mernst, todd, weld}@cs.washington.edu

Abstract

Recent work by Kautz *et al.* provides tantalizing evidence that large, classical planning problems may be efficiently solved by translating them into propositional satisfiability problems, using stochastic search techniques, and translating the resulting truth assignments back into plans for the original problems. We explore the space of such transformations, providing a simple framework that generates eight major encodings (generated by selecting one of four action representations and one of two frame axioms) and a number of subsidiary ones. We describe a fully-implemented compiler that can generate each of these encodings, and we test the compiler on a suite of STRIPS planning problems in order to determine which encodings have the best properties.

1 Introduction

- We present an analytic framework that accounts for all previously reported non-causal encodings,[1] including several novel possibilities. We parameterize the space of encodings along two major dimensions, action and frame representation. For twelve points in this two-dimensional space, we list the axioms necessary for a minimal encoding, and we calculate the asymptotic encoding sizes.
- We describe an automatic compiler that generates all of these encodings. While it is difficult for a compiler to produce encodings that are as lean as the hand-coded versions of [Kautz and Selman, 1996], we describe type-analysis and factoring techniques that get us close. Experiments demonstrate these methods can reduce the number of variables by half and formula size by 80%.
- We run the compiler on a suite of STRIPS-style planning problems, determining that the regular and simply-split explanatory encodings are smallest and can be solved fastest.

2 The Space of Encodings

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

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

Type qualifier

Java basetype

Type

```
@Untainted String query;
List<@NonNull String> strings;
myGraph = (@Immutable Graph) tmp;
class UnmodifiableList<T> implements @Readonly List<T> {}
```

Motivation #2: Formal verification ⊇ 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

$$\begin{array}{llll}
\texttt{h} & \in & \texttt{Heap} & = \texttt{Addr} \to \texttt{Obj} \\
\iota & \in & \texttt{Addr} & = \text{Set of Addresses} \cup \{\texttt{null}_a\} \\
\texttt{o} & \in & \texttt{Obj} & = {}^{\mathtt{r}}\texttt{Type},\ \texttt{Fields} \\
{}^{\mathtt{r}}\texttt{T} & \in & {}^{\mathtt{r}}\texttt{Type} & = \texttt{OwnerAddr ClassId<}\overline{{}^{\mathtt{r}}\texttt{Type}}\texttt{>} \\
\texttt{Fs} & \in & \texttt{Fields} & = \texttt{FieldId} \to \texttt{Addr} \\
\iota & \in & \texttt{OwnerAddr} & = \texttt{Addr} \cup \{\texttt{any}_a\} \\
{}^{\mathtt{r}}\Gamma & \in & {}^{\mathtt{r}}\texttt{Env} & = \overline{\texttt{TVarId}\ {}^{\mathtt{r}}\texttt{Type}};\ \overline{\texttt{ParId Addr}}
\end{array}$$

$$\begin{array}{llll}
\texttt{P} & \in & \texttt{Program} & ::= \overline{\texttt{Class}},\ \texttt{ClassId},\ \texttt{Expr} \\
\texttt{Cls} & \in & \texttt{Class} & ::= \texttt{class ClassId<}\overline{\texttt{TVarI}} \\
 & & & \quad \texttt{extends ClassId<}\overline{{}^{\mathtt{s}}\texttt{Typ}} \\
 & & & \quad \texttt{\{ }\overline{\texttt{FieldId}\ {}^{\mathtt{s}}\texttt{Type};}\ \texttt{Met} \\
{}^{\mathtt{s}}\texttt{T} & \in & {}^{\mathtt{s}}\texttt{Type} & ::= {}^{\mathtt{s}}\texttt{NType} \mid \texttt{TVarId} \\
{}^{\mathtt{s}}\texttt{N} & \in & {}^{\mathtt{s}}\texttt{NType} & ::= \texttt{OM ClassId<}\overline{{}^{\mathtt{s}}\texttt{Type}}\texttt{>} \\
\texttt{u} & \in & \texttt{OM} & ::= \\
\texttt{mt} & \in & \texttt{Meth} & ::= \\
 & & \texttt{MethSig} & ::= \\
\texttt{w} & \in & \texttt{Purity} & ::= \\
\texttt{e} & \in & \texttt{Expr} & ::= \\
 & & & \quad \texttt{Expr.MethId<}\overline{{}^{\mathtt{s}}\texttt{Type}}\texttt{>(}\overline{\texttt{Expr}}\texttt{)} \mid \\
 & & & \quad \texttt{new}\ {}^{\mathtt{s}}\texttt{Type} \mid (\,{}^{\mathtt{s}}\texttt{Type})\ \texttt{Expr} \\
{}^{\mathtt{s}}\Gamma & \in & {}^{\mathtt{s}}\texttt{Env} & ::= \overline{\texttt{TVarId}\ {}^{\mathtt{s}}\texttt{NType}};\ \overline{\texttt{ParId}\ {}^{\mathtt{s}}\texttt{Type}}
\end{array}$$

$$\text{OS-Read}\ \frac{\begin{array}{c}\texttt{h}, {}^{\mathtt{r}}\Gamma, \texttt{e}_0 \rightsquigarrow \texttt{h}', \iota_0 \\ \iota_0 \neq \texttt{null}_a \\ \iota = \texttt{h}'(\iota_0)\downarrow_2(\texttt{f})\end{array}}{\texttt{h}, {}^{\mathtt{r}}\Gamma, \texttt{e}_0.\texttt{f} \rightsquigarrow \texttt{h}', \iota}$$

$$\text{OS-Upd}\ \frac{\begin{array}{c}\texttt{h}, {}^{\mathtt{r}}\Gamma, \texttt{e}_0 \rightsquigarrow \texttt{h}_0, \iota_0 \\ \iota_0 \neq \texttt{null}_a \\ \texttt{h}_0, {}^{\mathtt{r}}\Gamma, \texttt{e}_2 \rightsquigarrow \texttt{h}_2, \iota \\ \texttt{h}' = \texttt{h}_2[\iota_0.\texttt{f} := \iota]\end{array}}{\texttt{h}, {}^{\mathtt{r}}\Gamma, \texttt{e}_0.\texttt{f}{=}\texttt{e}_2 \rightsquigarrow \texttt{h}',}$$

$$\text{GT-Upd}\ \frac{\begin{array}{c}\Gamma \vdash \texttt{e}_0 : \texttt{N}_0 \qquad \texttt{N}_0 = \texttt{u}_0\ \texttt{C}_0\texttt{<\_>} \\ \texttt{T}_1 = fType(\texttt{C}_0, \texttt{f}) \\ \Gamma \vdash \texttt{e}_2 : \texttt{N}_0 \triangleright \texttt{T}_1 \\ \texttt{u}_0 \neq \texttt{any} \qquad rp(\texttt{u}_0, \texttt{T}_1)\end{array}}{\Gamma \vdash \texttt{e}_0.\texttt{f}{=}\texttt{e}_2 : \texttt{N}_0 \triangleright \texttt{T}_1}$$

$$\text{GT-Read}\ \frac{\Gamma \vdash \texttt{e}_0 : \texttt{N}_0 \qquad \texttt{N}_0 = \_\ (}{\Gamma \vdash \texttt{e}_0.\texttt{f} : \texttt{N}_0 \triangleright fType(\texttt{C}_0, \texttt{f})}$$

$$\left.\begin{array}{l}\texttt{h} \vdash {}^{\mathtt{r}}\Gamma : {}^{\mathtt{s}}\Gamma \\ \texttt{h} \vdash \iota_1 : dyn({}^{\mathtt{s}}\texttt{N}, \texttt{h}, {}^{\mathtt{r}}\iota\,) \\ \texttt{h} \vdash \iota_2 : dyn({}^{\mathtt{s}}\texttt{T}, \iota_1, \texttt{h}(\iota_1)\downarrow_1)\end{array}\right\} \Longrightarrow \texttt{h} \vdash \iota_2 : dyn({}^{\mathtt{s}}\texttt{N} \triangleright {}^{\mathtt{s}}\texttt{T}, \texttt{h}, {}^{\mathtt{r}}\Gamma)$$

$$\begin{array}{l}{}^{\mathtt{s}}\texttt{N} = \texttt{u}_N\ \texttt{C}_N\texttt{<>} \\ \texttt{u}_N = \texttt{this}_u \Rightarrow {}^{\mathtt{r}}\Gamma(\texttt{this}) \\ free({}^{\mathtt{s}}\texttt{T}) \subseteq dom(\texttt{C}_N)\end{array}$$

$$\text{DYN}\ \frac{\begin{array}{ccc}{}^{\mathtt{r}}\texttt{T} = \iota'\ \_\texttt{<>} & \iota \vdash {}^{\mathtt{r}}\texttt{T}\ {}^{\mathtt{r}}\!\!<:\ \iota'\ \texttt{C<}\overline{{}^{\mathtt{r}}\texttt{T}}\texttt{>} & \iota \vdash {}^{\mathtt{r}}\texttt{T}\ {}^{\mathtt{r}}\!\!<:\ \iota'\ \texttt{C<}\overline{{}^{\mathtt{r}}\texttt{T}_a}\texttt{>} \Rightarrow \iota \vdash \overline{{}^{\mathtt{r}}\texttt{T}}\ {}^{\mathtt{r}}\!\!<:\ \overline{{}^{\mathtt{r}}\texttt{T}_a} \\ \multicolumn{2}{c}{dom(\texttt{C}) = \overline{\texttt{X}}} & free({}^{\mathtt{s}}\texttt{T}) \subseteq \overline{\texttt{X}} \circ \overline{\texttt{X}'}\end{array}}{dyn({}^{\mathtt{s}}\texttt{T}, \iota, {}^{\mathtt{r}}\texttt{T}, (\overline{\texttt{X}'\ {}^{\mathtt{r}}\texttt{T}'}; \_)) = {}^{\mathtt{s}}\texttt{T}[\iota'/\texttt{this}, \iota'/\texttt{peer}, \iota/\texttt{rep}, \texttt{any}_a/\texttt{any}_u, \overline{{}^{\mathtt{r}}\texttt{T}/\texttt{X}}, \overline{{}^{\mathtt{r}}\texttt{T}'/\texttt{X}'}]}$$

Too much research omits one!

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 ⇒ better theory

Helmuth von Moltke the Elder

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

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

The complete checker:

```
@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)

Today, 4 KLOC

< 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

“Pluggable type systems” term: Gilad Bracha (2004)

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

CHECKER framework

https://checkerframework.org/





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

Reaction at ISSTA 2008

My ISSTA 2008 paper received expedited journal publication ("best paper honorable mention")

Finding Bugs in Dynamic Web Applications

Shay Artzi[†] Adam Kieżun[†] Julian Dolby[‡]
Frank Tip[‡] Danny Dig[†] Amit Paradkar[‡] Michael D. Ernst[†]

[†] MIT CSAIL, {artzi,akiezun,dannydig,mernst}@csail.mit.edu
[‡] IBM T.J. Watson Research Center, {dolby,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[1]. 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[2]. 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[3] (even the ISSTA'08 website displays the W3C HTML compliance

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

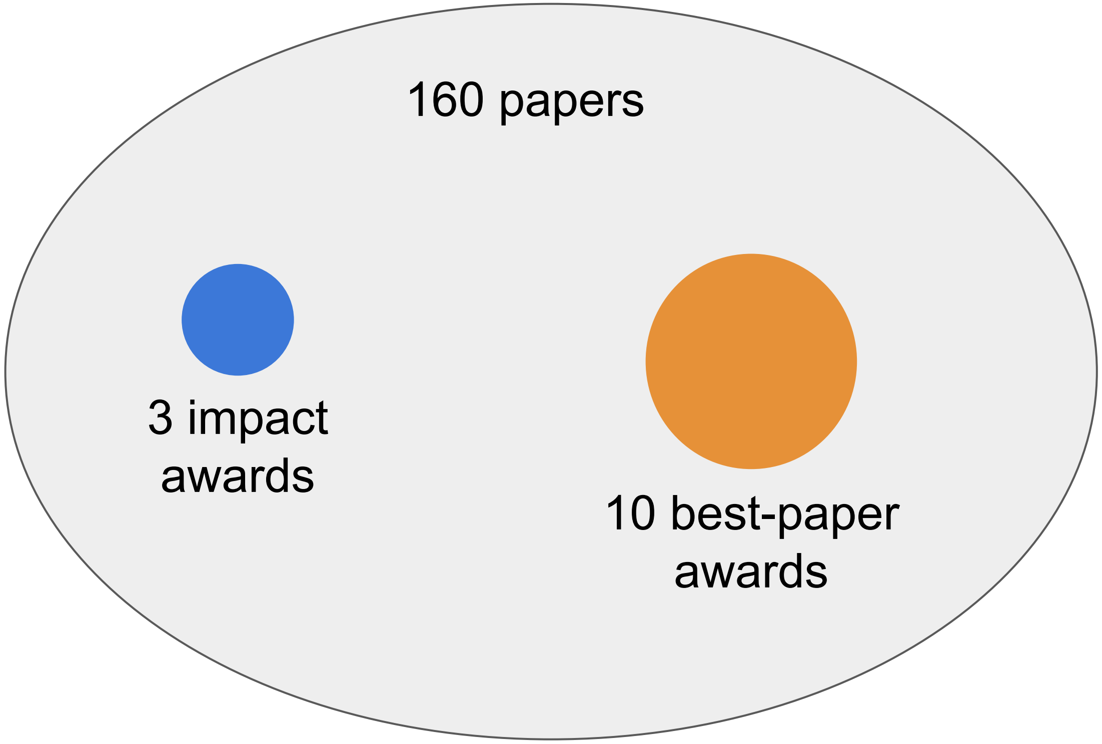
Results claimed by **previous work**
The reviewers believed them

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



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

The New York Times

MIND

Psychology Itself Is Under Scrutiny

Many famous studies of human behavior cannot be reproduced. Even so, they revealed aspects of our inner lives that feel true.

By Benedict Carey

July 16, 2018

102

Open access, freely available online

Essay

Why Most Published Research Findings Are False

John P. A. Ioannidis

Summary

There is increasing concern that most current published research findings are false. The probability that a research claim is true may depend on study power and bias, the number of other studies on the same question, and, importantly, the ratio of true to no relationships among the relationships probed in each scientific field. In this framework, a research finding is less likely to be true when the studies conducted in a field are smaller; when effect sizes are smaller; when there is a greater number and lesser preselection of tested relationships; where there is greater flexibility in designs, definitions, outcomes, and analytical modes; when there is greater financial and other

factors that influence this problem and some corollaries thereof.

Modeling the Framework for False Positive Findings

Several methodologists have pointed out [9–11] that the high rate of nonreplication (lack of confirmation) of research discoveries is a consequence of the convenient, yet ill-founded strategy of claiming conclusive research findings solely on the basis of a single study assessed by formal statistical significance, typically for a *p*-value less than 0.05. Research is not most appropriately represented and summarized by *p*-values, but, unfortunately, there is a widespread notion that medical research articles

is characteristic of the field and can vary a lot depending on whether the field targets highly likely relationships or searches for only one or a few true relationships among thousands and millions of hypotheses that may be postulated. Let us also consider, for computational simplicity, circumscribed fields where either there is only one true relationship (among many that can be hypothesized) or the power is similar to find any of the several existing true relationships. The pre-study probability of a relationship being true is $R/(R+1)$. The probability of a study finding a true relationship reflects the power $1-\beta$ (one minus the Type II error rate). The probability of claiming a relationship when none truly exists reflects the Type I error

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?"
by Walter F. Tichy, Computer 31:5, May 1998

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

Cybersquare

Walter F. Tichy
University of Karlsruhe

Should Computer Scientists Experiment More?

Computer scientists and practitioners defend their lack of experimentation with a wide range of arguments. Some arguments suggest that experimentation is inappropriate, too difficult, useless, and even harmful. This article discusses several such arguments to illustrate the importance of experimentation for computer science.

o computer scientists need to experiment at all? Only if we answer "yes" does it make sense to ask whether

in the brains of programmers and computer users. The phenomena studied in computer science are much

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

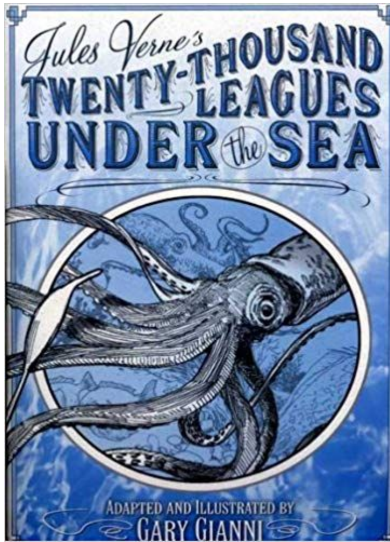
- Good for tenure
- Bad for science

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

Undergraduate

Undergraduate

Undergraduate

I got my start because faculty took a gamble on me

CRA-E Undergraduate Research Faculty Mentoring Award, 2018

Laura Dean, Adam Czeisler, Michael Harder, Alex Rolfe, Ben Morse, Jeremy Nimmer, Nii Dodoo, Lee Lin, Gustavo Santos, Arjun Narayanswamy, Emily Marcus, Jeff Mellen, Cemal Akcaba, Samir Meghani, Adrian Birka, Toh Ne Win, Yuriy Brun, Faisal Anwar, Stanley Cheung, James Anderson, Deepali Garg, Matthew Tschantz, Jonathan Grall, Aaron Iba, Benjamin Wang, Vikash Mansinghka, Jelani Nelson, Punyashloka Biswal, Alan Dunn, Joseph Sikoscow, Meng Mao, Galen Pickard, Kathryn Shih, Kevin Chevalier, Pramook Khungurn, Eric Fellheimer, Philip J. Guo, Michael Gebauer, Sanjukta Pal, Chen Xiao, David Glasser, Matt Papi, Jaime Quinonez, Mahmood Ali, Arjun Dayal, Jeff Yuan, Stephie Wu, Charles Tam, Telmo Luis Correa~Jr., John Marrero, David Harvison, Paley Li, Sigurd Schneider, Robert Rudd, Slava Chernyak, Matt Mullen, David Koenig, Gareth Snow, Tim Vega, Eric Spishak, Stephanie Dietzel, Laure Thompson, Peter Kalauskas, David Lazar, Artem Melentyev, Asumu Takikawa, Naomi Bancroft, Michael Sloan, Zachary Stein, Jeff Gertler, Yoong Woo Kim, Donovan Hunt, Kevin Thai, Allen Liu, William Mason Remy, Mark Davis, Haochen Wei, Andrew Davies, Brian Walker, Jenny Abrahamson, Timothy Vega, Roykrong Sukkerd, Stefan Heule, Wing Lam, Nat Mote, Kellen Donohue, Philip Lai, Jake Bailey, Tyler Rigsby, Forrest Coward, Rafael Vertido, Riley Klinger, Yuxuan (Shawn) Zhang, Rafael Vertido, Gene Kim, Katie Madonna, Siwakorn Ping Srisakaokul, Pingyang He, Dominic Langenegger, Luke Swart, Alain Orbino, Christopher Wei-Chieh Chen, Max Han, Paulo Barros, Patty Wang, Akshay Chalana, Kevin Bi, Hiep Can, Deric Pang, Steve Anton, Haoming Liu, Christopher Mackie, Sergio Delgado Castellanos, Omar Alhadlaq, Waylon Huang, Anmol Jammu, Abhishek Sangameswaran, Joe Santino, Kevin Vu, Arianna Blasi, Justin Kotalik, Adam Geller, David Grant.

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

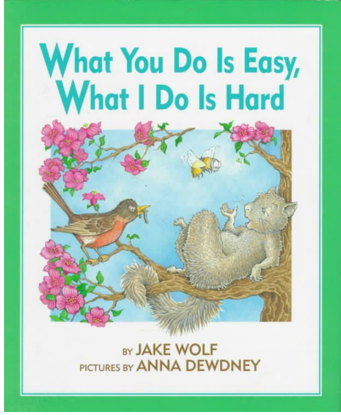
Reusable frameworks

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”

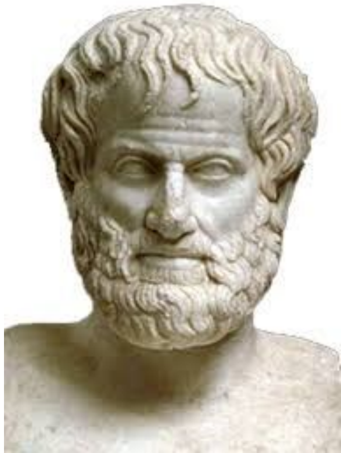


“It would be easy to …”
If it's so easy, why don't you do it?

“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/