

# Sound Gradual Typing

*Only Mostly Dead*

Spenser Bauman  
Mathworks

Carl Friedrich Bolz-Tiereck  
Heinrich-Heine-Universität Düsseldorf

Jeremy Siek  
Indiana University

Sam Tobin-Hochstadt  
Indiana University

# A Murder Mystery



## Gradual Typing for Functional Languages

Jeremy G. Siek  
University of Colorado  
siekb@cs.colorado.edu

Wadid Taha  
Rice University  
taha@cs.rice.edu

### Abstract

Static and dynamic type systems have well-known strengths and weaknesses, and each is better suited for different programming tasks. There have been many efforts to integrate static and dynamic typing and thereby combine the benefits of both typing disciplines in the same language. The flexibility of static typing can be improved by adding a type Dynamic and a type static form. The safety and performance of dynamic typing can be improved by adding optional type annotations or by performing type inference (as in soft typing). However, there has been little formal work on type systems that allow a programmer-controlled migration between dynamic and static typing. We propose Quasi-Static Typing, but it does not statically catch all type errors in completely annotated programs. Anderson and Bronevetsky defined a nominal type system for an object-oriented language with optional type annotations. However, developing a sound, gradual type system for functional languages with structural types is an open problem.

In this paper we present a solution based on the intuition that the structure of a type may be partially known/unknown at compile-time and the fluid of the type system is to catch incompatibilities between the known parts of types. We define the static and dynamic semantics of a  $\lambda$ -calculus with optional type annotations and we prove that its type system is sound with respect to the simply-typed  $\lambda$ -calculus for fully-annotated terms. We prove that this calculus is type safe and that the cost of dynamism is ‘pay-as-you-go’.

**Categories and Subject Descriptors** D.3.1 [Programming Languages]: Formal Definitions and Theory; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type and type

**General Terms** Languages, Performance, Theory

**Keywords** static and dynamic typing, optional type annotation

### 1. Introduction

Static and dynamic typing have different strengths, making them better suited for different tasks. Static typing provides early error detection, more efficient program execution, and better documentation, whereas dynamic typing enables rapid development and fast adaptation to changing requirements.

The focus of this paper is languages that literally provide static and dynamic typing in the same program, with the programmer controlling the fluid of the type system.

bring the degree of static checking by annotating function parameters with types, or not. We use the term *gradual typing* for type systems that provide this capability. Languages that support gradual typing to a large degree include C# [8], Boo [10], extensions to Visual Basic.NET and C# proposed by Meijer and Drayton [26], and extensions to Java proposed by Gray et al. [17], and the Bigloo [6, 36] dialect of Scheme [24]. The purpose of this paper is to provide a type-theoretic foundation for languages such as these with gradual typing.

There are numerous other ways to combine static and dynamic typing that fall outside the scope of gradual typing. Many dynamically typed languages have optional type annotations that are used to improve run-time performance but not to increase the amount of static checking. Common LISP [23] and Dylan [12, 37] are examples of such languages. Similarly the Soft Typing of Camerlingh and Flanagan [7] improves the performance of dynamically typed languages but it does not statically catch type errors. At the other end of the spectrum, statically typed languages can be made more flexible by adding a Dynamic type and a type static form, as in the work by Abdalla et al. [1]. However, such language do not allow for programming in a dynamically typed style because the programmer is required to insert corrections to and from type Dynamic.

A short example serves to demonstrate the idea of gradual typing. Figure 1 shows a call-by-value interpreter for an applied  $\lambda$ -calculus written in Scheme extended with gradual typing and algebraic data types. The version on the left does not have type annotations, and so the type system performs little type checking and instead many tag-checks occur at run time.

As development progresses, the programmer adds type annotations to the parameters of interp, as shown on the right side of Figure 1, and the type system provides more static type checking. We use the notation  $\tau$  for the dynamic type. The type system checks that the uses of  $\text{env}$  and  $\text{e}$  are appropriate: the case analysis on  $\text{e}$  is static and so is the application of  $\text{apply}$  to  $\text{x}$  and  $\text{env}$ . The recursive calls to interp also type check and the call to apply type checks trivially because the parameters of apply are dynamic. Note that we are still using dynamic typing for the value domain of the object language. To obtain a program with complete static checking, we would introduce a datatype for the value domain and use that as the return type of interp.

**Contribution.** We present a formal type system that supports gradual typing for functional languages, providing the flexibility of dynamically-typed languages when type annotations are omitted by the programmer and providing the benefits of static checking when function parameters are annotated. These benefits include both safety and performance: type errors are caught at compile-time and values may be stored in unboxed form. That is, for statically typed portions of the program there is no need for run-time tag and tag checking.

We introduce a calculus named  $\lambda^*$ , and define its type system (Section 2). We show that this type system, when applied to fully an-

## Interlanguage Migration: From Scripts to Programs

Sam Tobin-Hochstadt  
Northeastern University  
Boston, MA  
samth@ccs.neu.edu

Matthias Felleisen  
Northeastern University  
Boston, MA  
matthias@ccs.neu.edu

### ABSTRACT

As scripts grow into full-fledged applications, programmers should want to port portions of their programs from scripting languages to languages with sound and rich type systems. This form of interlanguage migration ensures type-safety and provides minimal guarantees for reuse in other applications, too.

In this paper, we present a framework for expressing this form of interlanguage migration. Given a program that consists of modules in the untyped lambda calculus, we prove that rewriting one of them in a simply-typed lambda calculus produces an equivalent program and adds the expected amount of type safety, i.e., code in typed modules can't go wrong. To ensure these guarantees, the migration process infers constraints from the statically-typed module and imposes them on the dynamically-typed modules in the form of behavioral contracts.

### 1. WHEN SCRIPTS GROW UP

In the beginning, the programmer created a script to mechanize some routine but problematic task. The script consisted of a few dozen lines of code in a dynamically-typed and expressive scripting language. Before long, the programmer discovered that friends were coping with similar problems and with a few changes here and a few hacks there, the script became a useful 1,000-line program for his friends, too. Not surprisingly, the programmers decided to offer this service to some of his favorite mailing lists and modified the program a few more times. At that point, there were actually several related applications, each consisting of a few dozen components, mostly drawn from a common library of modules; and all these applications supported different tasks, only superficially related to the original one. It goes without saying that the application suite became the core of a small, yet exponentially growing business and that the programmers sold it for a lot of money to some big company.

Fortunately, our programmer has employed a test-driven or test-first approach to programming [5], a technique that

originated with dynamically-typed languages such as Lisp, Scheme and Smalltalk.<sup>1</sup> Every bug report has been turned into a test case; the test suites have been maintained in a meticulous manner. Unfortunately the big company isn't satisfied with the result. Because the software deals with people's financial holdings, the company's management team wants to cross all the t's and dot all the i's, at least for the critical modules. The team has set the goal to rewrite the program in a programming language with a sound type system. They believe that this step will eliminate some long-standing bugs from the module and improve the efficiency of the debugging team. After all, "typed programs can't go wrong" [24], i.e., the programme doesn't have to look at code in typed modules when a run-time type check fails.

Currently, this common, realistic situation poses a major difficulty for the (unfortunate) programmers of the company.

Mostly, they should port one module at a time, always leaving the overall product intact and running.

Most foreign-language interfaces, however, support only connections between high-level languages and C-level libraries.

Support for connecting a high-level typed language with a high-level dynamically-typed language rarely exists.

Hence, programmers often re-develop the entire program from scratch and run into Brooks' "second system" syndrome [6].

In this paper, we investigate an alternative to this reimplemention approach. Specifically, we present a framework for porting programs in a gradual manner from an dynamically-typed to a statically-typed and semantically-related, statically-typed programming language. We are making three philosophical assumptions. First, a program is a sequence of modules in a safe, but dynamically-typed programming language. The second assumption is that we have an explicitly, statically-typed programming language that is variant of the dynamically-typed language. Specifically, the two languages share run-time values and differ only in that one has a type system and the other doesn't. While there have been very realistic examples of such systems [7], one can imagine creating such a pair of languages from any dynamically-typed language (say Scheme), possibly based on the decades-old soft typing research. Finally, a third and less realistic assumption is that (at least) one of the modules is "typable," meaning that equipping variables with type doc-

Permitting to make digital or hard copies of all or part of this work for personal use is granted by the copyright holder for the author and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DLS '06 Portland, Oregon  
Copyright © 2006 ACM [to be supplied] . \$5.00.

<sup>1</sup>Although Kent Beck has coined the phrase and popularized test-driven development, the idea of developing tests first had been around for ages in the Scheme community and had been inherited from the Lisp community. For example, see the unpublished manuscript "A Guide to the Metaphysical Universe" by Friedman, Haynes, Kohlbecker, and Wand circa 1984.

# 2017

The screenshot shows a GitHub repository page for 'mvitousek / reticulated'. The repository name is at the top left. Below it are navigation links for Code, Issues (21), Pull requests (0), Projects (0), Wiki, and Insights. The main content area is titled 'Types for Python'. It displays 264 commits, 5 branches, and 0 releases. A pull request button 'New pull request' is visible. Below this is a list of commits from 'mvitousek' with descriptions like 'fixing repl', 'fixes', and 'added entry module'. At the bottom, there's a 'README.md' file. On the right side of the page, a sidebar for 'The Typed Racket Guide' is open. It has a search bar '...search manuals...' and a version indicator 'v.6.10'. The main content of the sidebar lists chapters: 1 Quick Start, 2 Beginning Typed Racket, 3 Specifying Types, 4 Types in Typed Racket, 5 Occurrence Typing, 6 Typed-Untyped Interaction, 7 Optimization in Typed Racket, and 8 Caveats and Limitations. Below the chapters, it says 'ON THIS PAGE: The Typed Racket Guide'.

## The Typed Racket Guide

by Sam Tobin-Hochstadt <samth@racket-lang.org>, Vincent St-Amour <stamourv@racket-lang.org>, Eric Dobson <endobson@racket-lang.org>, and Asumu Takikawa <asumu@racket-lang.org>

Typed Racket is Racket's gradually-typed sister language which allows the incremental addition of statically-checked type annotations. This guide is intended for programmers familiar with Racket. For an introduction to Racket, see [The Racket Guide](#).

For the precise details, also see [The Typed Racket Reference](#).

### 1 Quick Start

[1.1 Using Typed Racket from the Racket REPL](#)

### 2 Beginning Typed Racket

[2.1 Datatypes and Unions](#)

# Sound Gradual Typing

TypeScript lets you write JavaScript the way you really want to.  
TypeScript is a typed superset of JavaScript that compiles to plain  
JavaScript.  
Any browser. Any host. Any OS. Open Source.

Get TypeScript Now

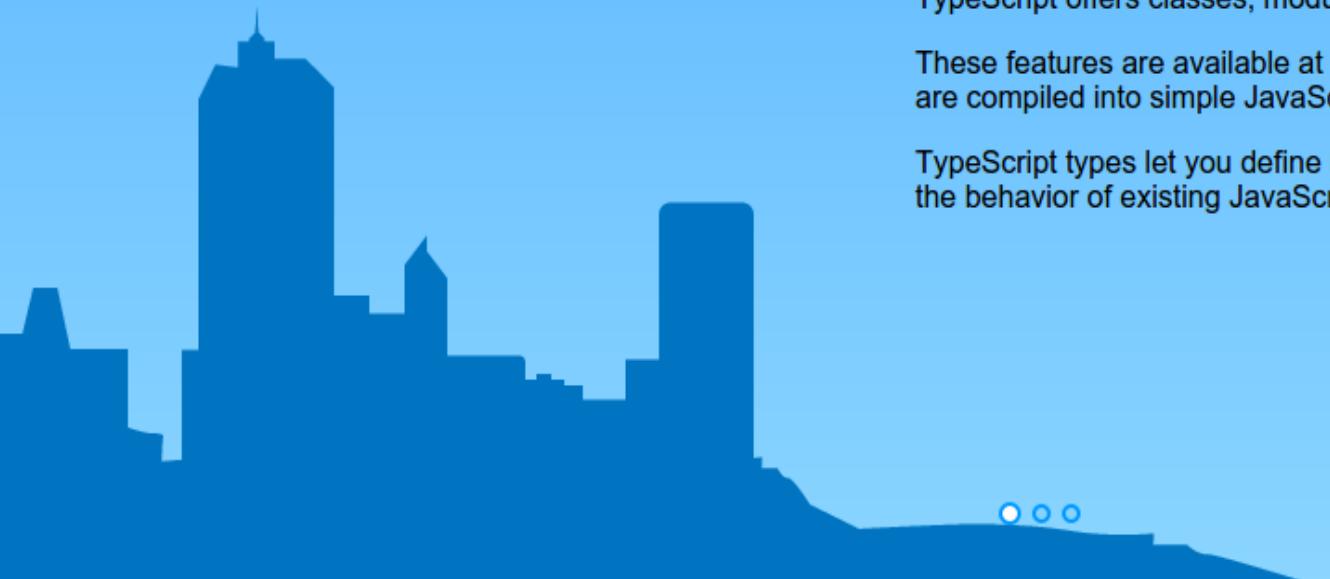


## Scalable

TypeScript offers classes, modules, and interfaces to help you build robust components.

These features are available at development time for high-confidence application development and are compiled into simple JavaScript.

TypeScript types let you define interfaces between software components and to gain insights into the behavior of existing JavaScript libraries.



Node.js

Tools

Open Source

# Programming productivity without breaking things

[INSTALL](#)[TUTORIAL](#)

```
1  <?hh
2  class MyClass {
3      public function alpha(): int {
4          return 1;
5      }
6
7      public function beta(): string {
8          return 'hi test';
9      }
10 }
11
12 function f(MyClass $my_inst): string {
13     // Fix me!
14     return $my_inst->alpha();
15 }
```

## What is Hack?

[Click here to read the official Hack announcement](#)

Hack Developer Day: April 9, 2014



## Community

- [Hack on Twitter](#)
- [HHVM Blog](#)
- [HHVM IRC Chat](#)
- [Hack Dev Day 2014](#)

## Open Source Info

- [GitHub](#)
- [License](#)
- [Contributing](#)
- [Cookbook on Github](#)
- [Docs on Github](#)



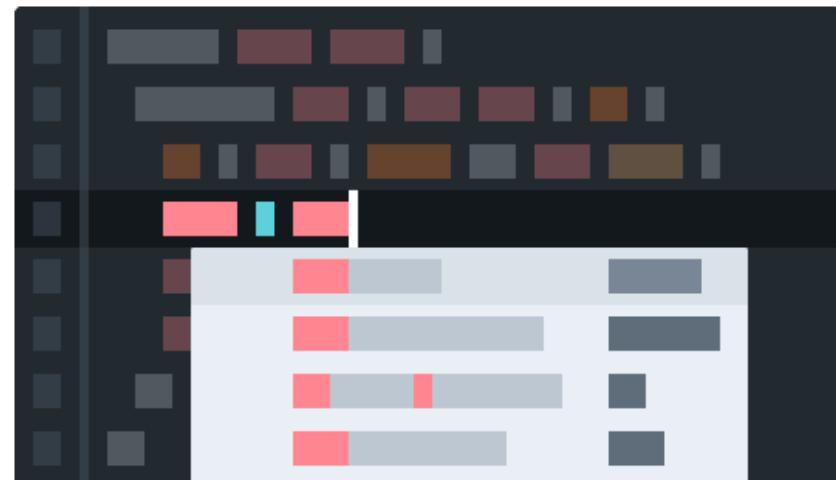
# FLOW IS A STATIC TYPE CHECKER FOR JAVASCRIPT.

[GET STARTED](#)[INSTALL FLOW](#) [Star 13,889](#)

Current Version: [v0.57.3](#)

## CODE FASTER.

Tired of having to run your code to find bugs? Flow identifies problems as you code. Stop wasting your time guessing and checking.



# Titan

[build](#) passing [codecov](#) 81%

Titan is a new programming language, designed to be a statically-typed, ahead-of-time compiled sister language to [Lua](#). It is an application programming language with a focus on performance.

This repository contains the initial prototype of the Titan compiler. It compiles a single Titan module to C code in the [artisanal style](#). The syntax is a subset of Lua syntax, plus types, and is specified in `titan-v0.ebnf`.

## Requirements for running the compiler

1. [Lua](#) >= 5.3.0
2. [LPEGLabel](#) >= 1.0.0
3. [inspect](#) >= 3.1.0
4. [argparse](#) >= 0.5.0

You need to build the Lua interpreter in the `lua` folder with `MYCFLAGS=-fPIC`, or `titanc` will not be able to build any Titan code.

## Install

Titan must be installed in a standard location; [LuaRocks](#) will do this, and will also install all dependencies automatically.

```
$ [install luarocks]
$ luarocks install titan-lang-scm-1.rockspe
```

## Usage

# Titan

[build](#) passing [codecov](#) 81%

Titan is a new programming language, designed to be a statically-typed, ahead-of-time compiled sister language to [Lua](#). It is an application programming language with a focus on performance.

This repository contains the initial prototype of the Titan compiler. It compiles a [Lua module](#) to C code in the [artisanal style](#). The syntax is a subset of Lua syntax, plus types, and is specified in [the grammar](#).

## Requirements for running the compiler

1. [Lua](#) >= 5.3.0
2. [LPegLabel](#) >= 1.0.0
3. [inspect](#) >= 3.1.0
4. [argparse](#) >= 0.5.0

You need to build the Lua interpreter from source. If you build it with `MYCFLAGS=-fPIC`, or `titanc` will not be able to build any Titan code.

All Unsound!

## Install

Titan must be installed in a standard location; [LuaRocks](#) will do this, and will also install all dependencies automatically.

```
$ [install luarocks]
$ luarocks install titan-lang-scm-1.rockspe
```

## Usage



But Why?

Interfacing Typed Racket and Racket code may involve a lot of dynamic checks, which can have significant overhead, and cause that kind of stuttering.

*Vincent St. Amour, Dec. 2015*

It's very slow.

It looks like it has to do with a dc<%>  
instance crossing from untyped to  
typed code.

*Neil Toronto, May 2015*

... the resulting dynamic checks  
will probably cause them to be  
unacceptably slow.

*John Clements, January 2016*



# Is Sound Gradual Typing Dead?

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, Matthias Felleisen

Northeastern University, Boston, MA

## Abstract

Programmers have come to embrace dynamically-typed languages for prototyping and delivering large and complex systems. When it comes to maintaining and evolving these systems, the lack of explicit static typing becomes a bottleneck. In response, researchers have explored the idea of gradually-typed programming languages which allow the incremental addition of type annotations to software written in one of these untyped languages. Some of these new, hybrid languages insert run-time checks at the boundary between typed and untyped code to establish type soundness for the overall system. With sound gradual typing, programmers can rely on the language implementation to provide meaningful error messages when type invariants are violated. While most research on sound gradual typing remains theoretical, the few emerging implementations suffer from performance overheads due to these checks. None of the publications on this topic comes with a comprehensive performance evaluation. Worse, a few report disastrous numbers.

In response, this paper proposes a method for evaluating the performance of gradually-typed programming languages. The method hinges on exploring the space of partial conversions from untyped to typed. For each benchmark, the performance of the different ver-

many cases, the systems start as innocent prototypes. Soon enough, though, they grow into complex, multi-module programs, at which point the engineers realize that they are facing a maintenance nightmare, mostly due to the lack of reliable type information.

Gradual typing [21, 26] proposes a language-based solution to this pressing software engineering problem. The idea is to extend the language so that programmers can incrementally equip programs with types. In contrast to optional typing, gradual typing provides programmers with soundness guarantees.

Realizing type soundness in this world requires run-time checks that watch out for potential impedance mismatches between the typed and untyped portions of the programs. The granularity of these checks determine the performance overhead of gradual typing. To reduce the frequency of checks, *macro-level* gradual typing forces programmers to annotate entire modules with types and relies on behavioral contracts [12] between typed and untyped modules to enforce soundness. In contrast, *micro-level* gradual typing instead assigns an implicit type *Dyn* [1] to all unannotated parts of a program; type annotations can then be added to any declaration. The implementation must insert casts at the appropriate points in the code. Different language designs use slightly different semantics with different associated costs and limitations.



# Is Sound Gradual Typing Dead?

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, Matthias Felleisen

Northeastern University, Boston, MA

The problem is that, according to our measurements, the cost of enforcing soundness is overwhelming.

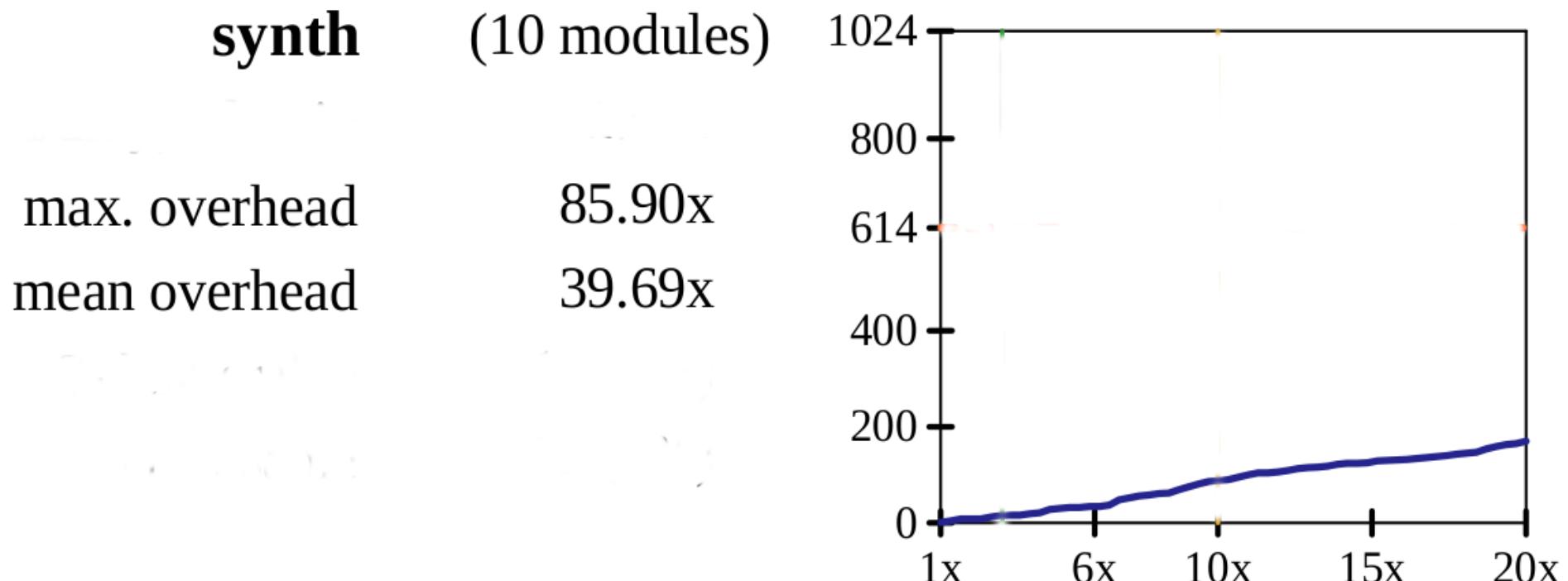
for prototyping and delivering large and complex systems. When it comes to maintaining and evolving these systems, the lack of explicit static typing becomes a bottleneck. In response, researchers have explored the idea of gradually-typed programming languages which allow the incremental addition of type annotations to software written in one of these untyped languages. Some of these new, hybrid languages insert run-time checks at the boundary between typed and untyped code to establish type soundness for the overall system. With sound gradual typing, programmers can rely on the language implementation to provide meaningful error messages when type invariants are violated. While most research on sound gradual typing remains theoretical, the few emerging implementations suffer from performance overheads due to these checks. None of the publications on this topic comes with a comprehensive performance evaluation. Worse, a few report disastrous numbers.

In response, this paper proposes a method for evaluating the performance of gradually-typed programming languages. The method hinges on exploring the space of partial conversions from untyped to typed. For each benchmark, the performance of the different ver-

mare, mostly due to the lack of reliable type information.

Gradual typing [21, 26] proposes a language-based solution to this pressing software engineering problem. The idea is to extend the language so that programmers can incrementally equip programs with types. In contrast to optional typing, gradual typing provide programmers with soundness guarantees.

Realizing type soundness in this world requires run-time checks that watch out for potential impedance mismatches between the typed and untyped portions of the programs. The granularity of these checks determine the peformance overhead of gradual typing. To reduce the frequency of checks, *macro-level* gradual typing forces programmers to annotate entire modules with types and relies on behavioral contracts [12] between typed and untyped modules to enforce soundness. In contrast, *micro-level* gradual typing instead assigns an implicit type Dyn [1] to all unannotated parts of a program; type annotations can then be added to any declaration. The implementation must insert casts at the appropriate points in the code. Different language designs use slightly different semantics with different associated costs and limitations.



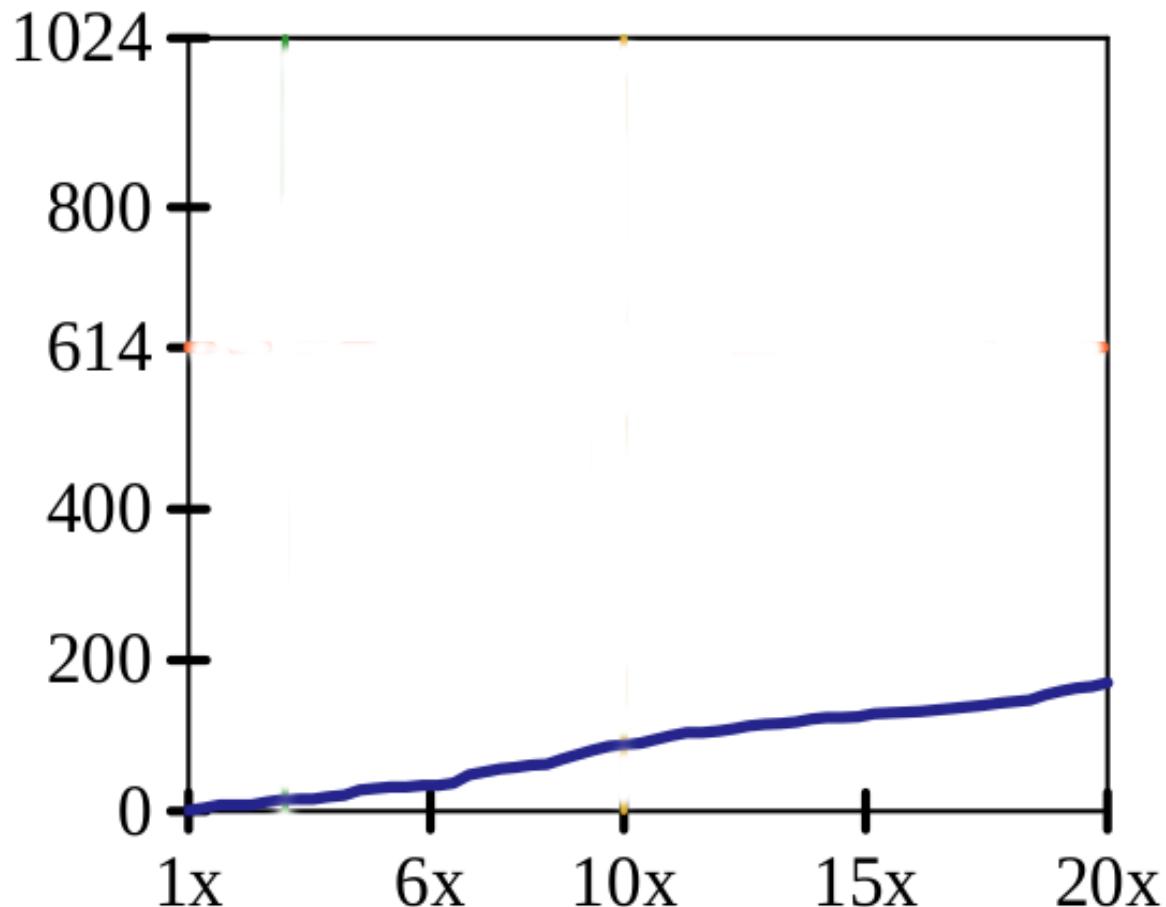
None of the publications on this topic comes with a comprehensive performance evaluation. Worse, a few report disastrous numbers.

In response, this paper proposes a method for evaluating the performance of gradually-typed programming languages. The method hinges on exploring the space of partial conversions from untyped to typed. For each benchmark, the performance of the different ver-

relies on behavioral contracts [12] between typed and untyped modules to enforce soundness. In contrast, *micro-level* gradual typing instead assigns an implicit type *Dyn* [1] to all unannotated parts of a program; type annotations can then be added to any declaration. The implementation must insert casts at the appropriate points in the code. Different language designs use slightly different semantics with different associated costs and limitations.

0 modules)

85.90x  
39.69x



None of the publications on this topic comes with a comprehensive performance evaluation. Worse, a few report disastrous numbers.

In response, this paper proposes a method for evaluating the performance of gradually-typed programming languages. The method hinges on exploring the space of partial conversions from untyped to typed. For each benchmark, the performance of the different ver-

relies on behavioral contracts [12] between typed and untyped modules to enforce soundness. In contrast, *micro-level* gradual typing instead assigns an implicit type *Dyn* [1] to all unannotated parts of a program; type annotations can then be added to any declaration. The implementation must insert casts at the appropriate points in the code. Different language designs use slightly different semantics with different associated costs and limitations.



# Is Sound Gradual Typing Dead?

We find that Typed Racket's cost of soundness is not tolerable. If applying our method to other gradual type system implementations yields similar results, then sound gradual typing is dead.

new, hybrid languages insert run-time checks at the boundary between typed and untyped code to establish type soundness for the overall system. With sound gradual typing, programmers can rely on the language implementation to provide meaningful error messages when type invariants are violated. While most research on sound gradual typing remains theoretical, the few emerging implementations suffer from performance overheads due to these checks. None of the publications on this topic comes with a comprehensive performance evaluation. Worse, a few report disastrous numbers.

In response, this paper proposes a method for evaluating the performance of gradually-typed programming languages. The method hinges on exploring the space of partial conversions from untyped to typed. For each benchmark, the performance of the different ver-

provide programmers with soundness guarantees.

Realizing type soundness in this world requires run-time checks that watch out for potential impedance mismatches between the typed and untyped portions of the programs. The granularity of these checks determine the performance overhead of gradual typing. To reduce the frequency of checks, *macro-level* gradual typing forces programmers to annotate entire modules with types and relies on behavioral contracts [12] between typed and untyped modules to enforce soundness. In contrast, *micro-level* gradual typing instead assigns an implicit type `Dyn` [1] to all unannotated parts of a program; type annotations can then be added to any declaration. The implementation must insert casts at the appropriate points in the code. Different language designs use slightly different semantics with different associated costs and limitations.



# Is Sound Gradual Typing Dead?

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, Matthias Felleisen

Northeastern University, Boston, MA

## Abstract

Programmers have come to embrace dynamically-typed languages for prototyping and delivering large and complex systems. When it comes to maintaining and evolving these systems, the lack of explicit static typing becomes a bottleneck. In response, researchers have explored the idea of gradually-typed programming languages which allow the incremental addition of type annotations to software written in one of these untyped languages. Some of these new, hybrid languages insert run-time checks at the boundary between typed and untyped code to establish type soundness for the overall system. With sound gradual typing, programmers can rely on the language implementation to provide meaningful error messages when type invariants are violated. While most research on sound gradual typing remains theoretical, the few emerging implementations suffer from performance overheads due to these checks. None of the publications on this topic comes with a comprehensive performance evaluation. Worse, a few report disastrous numbers.

In response, this paper proposes a method for evaluating the performance of gradually-typed programming languages. The method hinges on exploring the space of partial conversions from untyped to typed. For each benchmark, the performance of the different ver-

many cases, the systems start as innocent prototypes. Soon enough, though, they grow into complex, multi-module programs, at which point the engineers realize that they are facing a maintenance nightmare, mostly due to the lack of reliable type information.

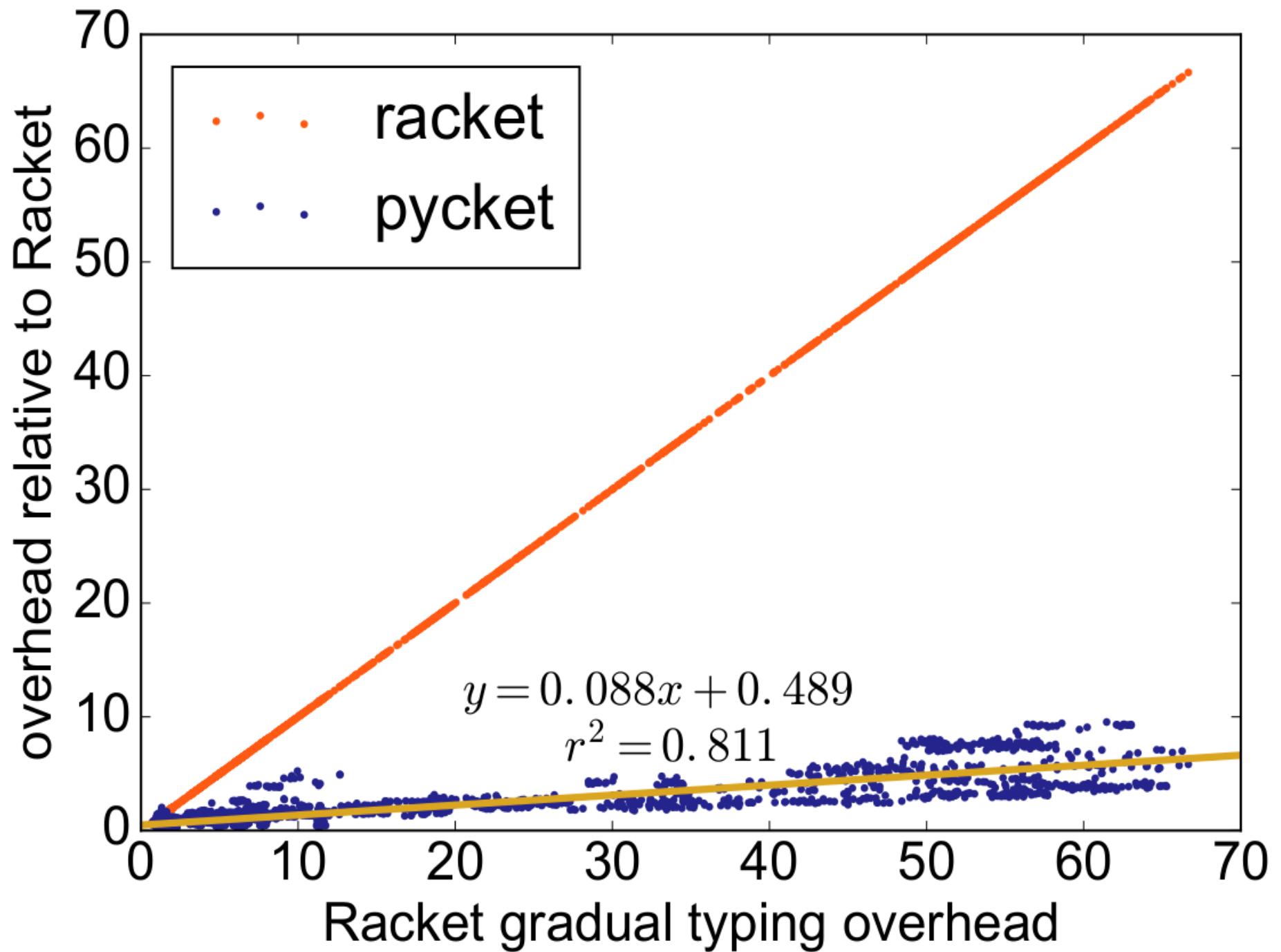
Gradual typing [21, 26] proposes a language-based solution to this pressing software engineering problem. The idea is to extend the language so that programmers can incrementally equip programs with types. In contrast to optional typing, gradual typing provides programmers with soundness guarantees.

Realizing type soundness in this world requires run-time checks that watch out for potential impedance mismatches between the typed and untyped portions of the programs. The granularity of these checks determine the performance overhead of gradual typing. To reduce the frequency of checks, *macro-level* gradual typing forces programmers to annotate entire modules with types and relies on behavioral contracts [12] between typed and untyped modules to enforce soundness. In contrast, *micro-level* gradual typing instead assigns an implicit type *Dyn* [1] to all unannotated parts of a program; type annotations can then be added to any declaration. The implementation must insert casts at the appropriate points in the code. Different language designs use slightly different semantics with different associated costs and limitations.

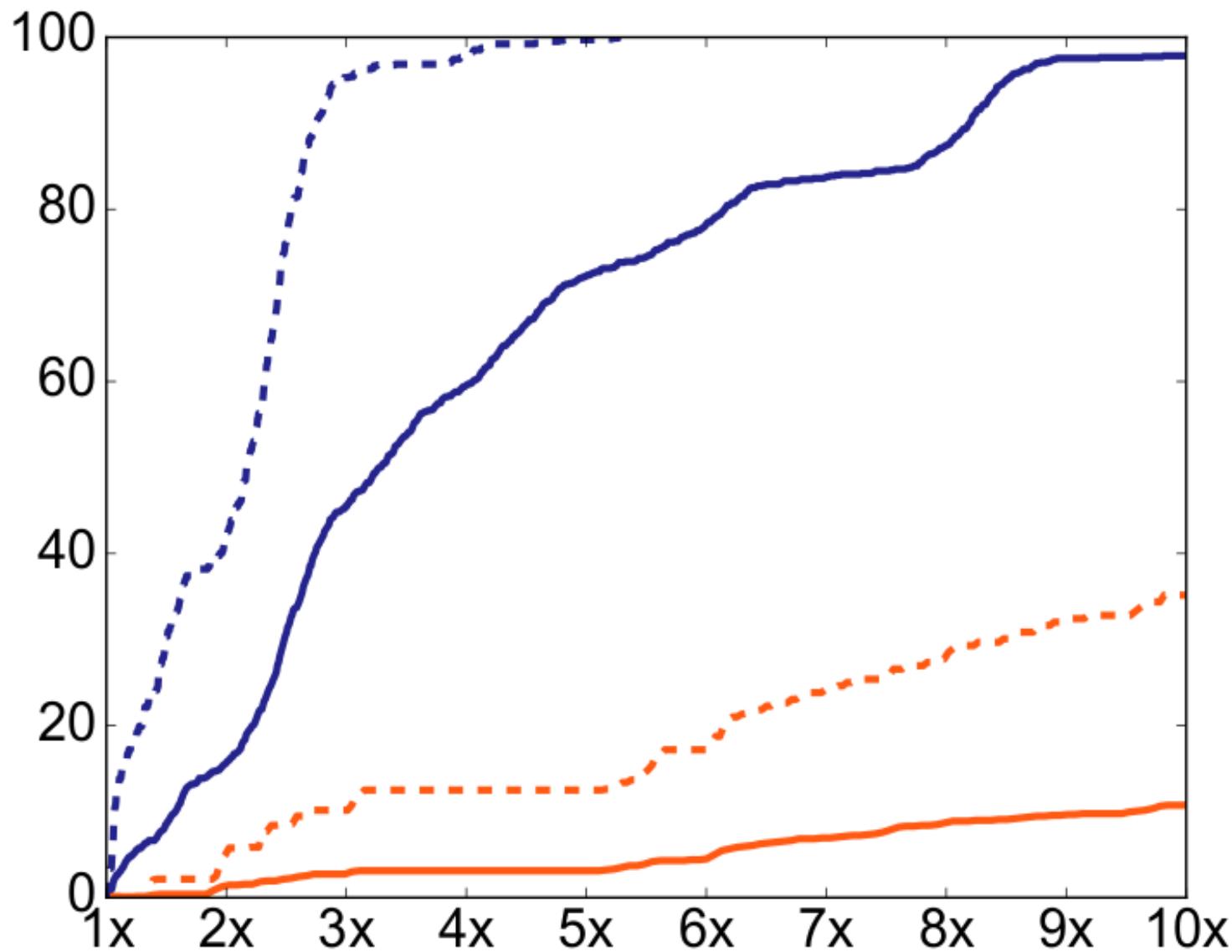


A close-up portrait of a man with long, light-colored hair and a beard, wearing a patterned robe. He has a serious expression and is looking slightly to the right. His hand is raised near his face, with fingers partially hidden in his hair.

Enter Pycket



## Synth, again



# Two Key Ideas

- Tracing JIT Compilation
- Hidden Classes for Chaperones

A close-up portrait of an elderly woman with short, light-colored hair, possibly grey or white, styled in a bun. She has a gentle, slightly smiling expression, looking off-camera to her right. Her skin is wrinkled, and she appears to be wearing a dark green top with a subtle, intricate pattern. The lighting is soft, creating a warm and intimate atmosphere.

Tracing to the rescue

```
(define f (cast (λ (x) (+ x 1))
                  (-> Integer Integer)))
```

```
(define f (cast (λ (x) (+ x 1))  
                  (-> Integer Integer)))
```



```
(define (f x*)  
  (cast ((λ (x) (+ x 1)) (cast x* Integer))  
        Integer))
```

```
(define f (cast (λ (x) (+ x 1))  
                  (-> Integer Integer)))
```



```
(define f* (checked-fn (λ (x) (+ x 1))  
                         Integer Integer))  
(define (f x*)  
  (cast ((checked-fn-op f)  
         (cast x* (checked-fn-domain f)))  
        (checked-fn-range f)))
```

(f x)

```
var f_code = (checked-fn-op f);  
var f_dom = (checked-fn-domain f);  
var f_rng = (checked-fn-rng f);
```

guard (integer? x)

```
var res = x + 1;  
guard (integer? res)
```

return res;

(f x)

```
var f_code = (checked-fn-op f);  
var f_dom = (checked-fn-domain f);  
var f_rng = (checked-fn-rng f);
```

guard (integer? x)

```
var res = x + 1;  
guard (integer? res)
```

return res;

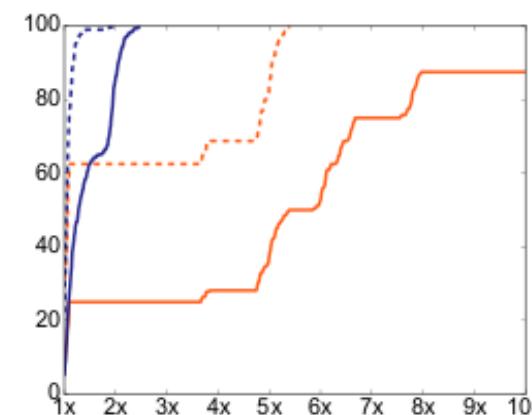
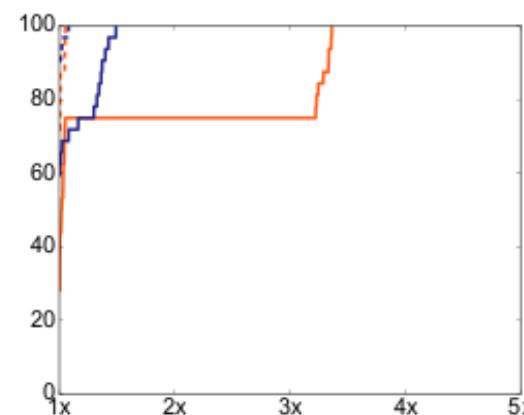
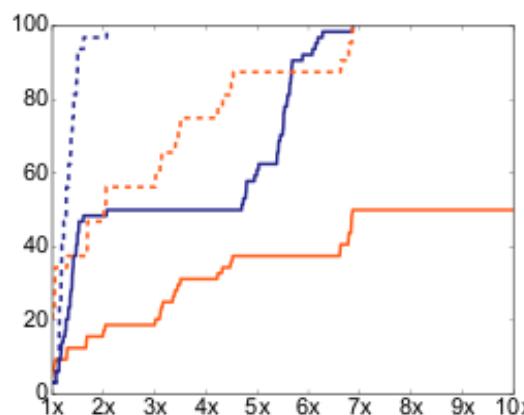
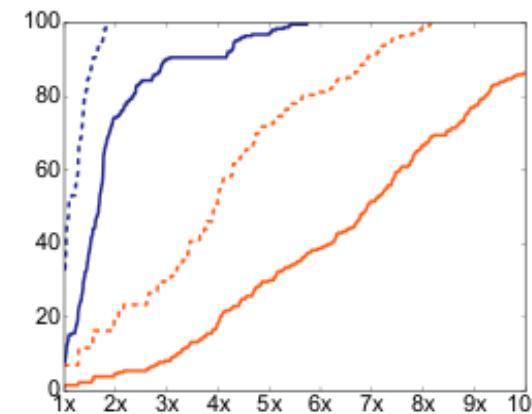
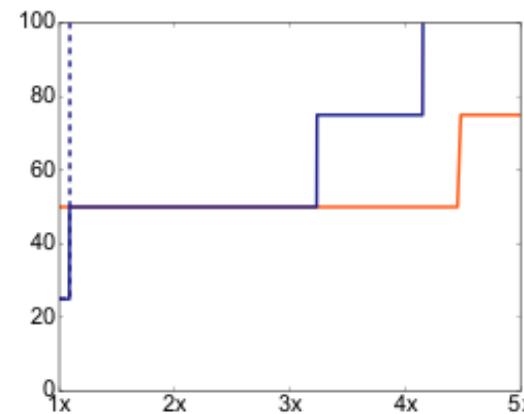
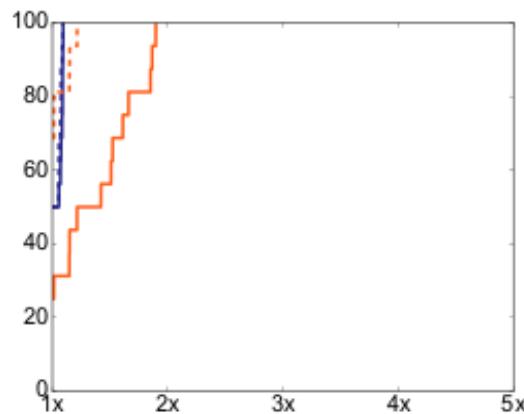
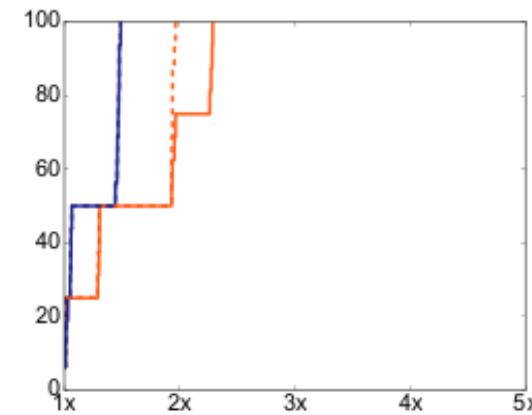
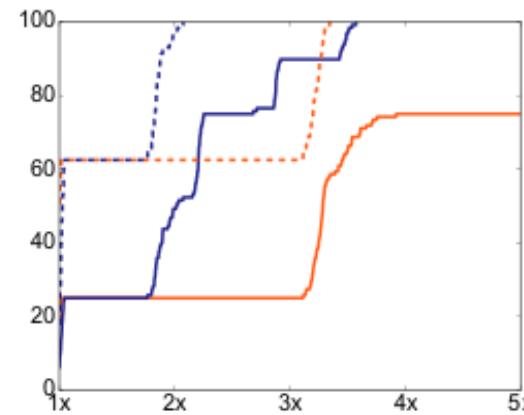
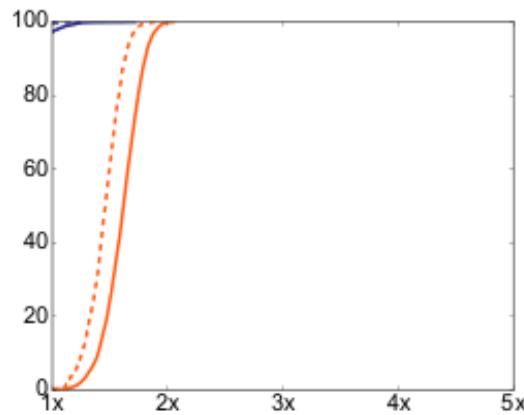
(f x)

```
var f_code = (checked-fn-op f);  
var f_dom = (checked-fn-domain f);  
var f_rng = (checked-fn-rng f);
```

guard (integer? x)

```
var res = x + 1;  
guard (integer? res)
```

return res;





# Sound Gradual Typing: Only Mostly Dead

SPENSER BAUMAN, Indiana University, United States

CARL FRIEDRICH BOLZ-TEREICK, King's College London, United Kingdom

JEREMY SIEK and SAM TOBIN-HOCHSTADT, Indiana University, United States

While gradual typing has proven itself attractive to programmers, many systems have avoided *sound* gradual typing due to the run time overhead of enforcement. In the context of sound gradual typing, both anecdotal and systematic evidence has suggested that run time costs are quite high, and often unacceptable, casting doubt on the viability of soundness as an approach.

We show that these overheads are not fundamental, and that with appropriate improvements, **just-in-time compilers can greatly reduce the overhead of sound gradual typing**. Our study takes benchmarks published in a recent paper on gradual typing performance in Typed Racket ([Takikawa et al., POPL 2016](#)) and evaluates them using an experimental tracing JIT compiler for Racket, called Pycket. On typical benchmarks, Pycket is able to eliminate more than 90% of the gradual typing overhead. While our current results are not the final word in optimizing gradual typing, we show that the situation is not dire, and where more work is needed.

Pycket's performance comes from several sources, which we detail and measure individually. First, we apply a sophisticated tracing JIT compiler and optimizer, automatically generated in Pycket using the RPython framework originally created for PyPy. Second, we focus our optimization efforts on the challenges posed by run time checks, implemented in Racket by *chaperones and impersonators*. We introduce representation improvements, including a novel use of *hidden classes* to optimize these data structures, and measure the performance implications of each optimization.

**CCS Concepts:** • Theory of computation → Program specifications; • Software and its engineering → Just-in-time compilers; Software evolution;

**Additional Key Words and Phrases:** Gradual Typing, Just-in-Time compilation, Performance Evaluation

**ACM Reference Format:**

Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. Sound Gradual



# Sound Gradual Typing: Only Mostly Dead

SPENSER BAUMAN, Indiana University, United States

CARL FRIEDRICH BOLZ-TEREICK, King's College London, United Kingdom

- Hidden Classes for Chaperones
- Benchmark impacts for all optimizations
- Loop finding & wrappers

evaluates them using an experimental tracing JIT compiler for Racket, called Pycket. On typical benchmarks, Pycket is able to eliminate more than 90% of the gradual typing overhead. While our current results are not the final word in optimizing gradual typing, we show that the situation is not dire, and where more work is needed.

Pycket's performance comes from several sources, which we detail and measure individually. First, we apply a sophisticated tracing JIT compiler and optimizer, automatically generated in Pycket using the RPython framework originally created for PyPy. Second, we focus our optimization efforts on the challenges posed by run time checks, implemented in Racket by *chaperones and impersonators*. We introduce representation improvements, including a novel use of *hidden classes* to optimize these data structures, and measure the performance implications of each optimization.

CCS Concepts: • Theory of computation → Program specifications; • Software and its engineering → Just-in-time compilers; Software evolution;

Additional Key Words and Phrases: Gradual Typing, Just-in-Time compilation, Performance Evaluation

ACM Reference Format:

Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. Sound Gradual

# **Sound Gradual Typing**

## **Mostly Dead is Slightly Alive**

[github.com/pycket](https://github.com/pycket)

A photograph of a man and a woman in a forest. The man, on the left, has short blonde hair and is wearing a dark t-shirt. The woman, on the right, has long brown hair and is wearing a white, beaded, off-the-shoulder dress. She is also wearing a large, ornate crown. They are standing in front of a large tree with yellow leaves.

# **Sound Gradual Typing**

# **Mostly Dead is Slightly Alive**

[github.com/pycket](https://github.com/pycket)