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

2006


Gradual Typing for Functional Languages

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

Walid Taha
Rice University
taha@rice.edu

Abstract

1. Introduction


Interlanguage Migration: From Scripts to Programs

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

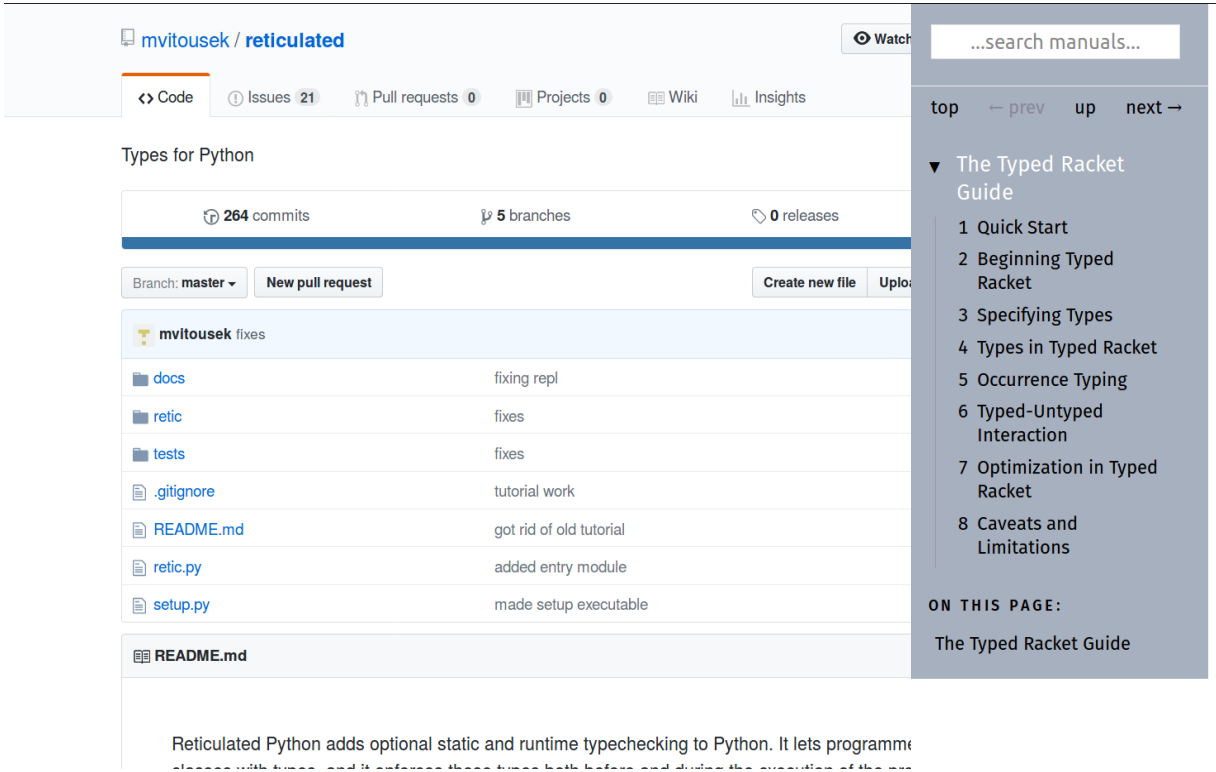
Matthias Felleisen
Northeastern University
Boston, MA
matthias@ccs.neu.edu

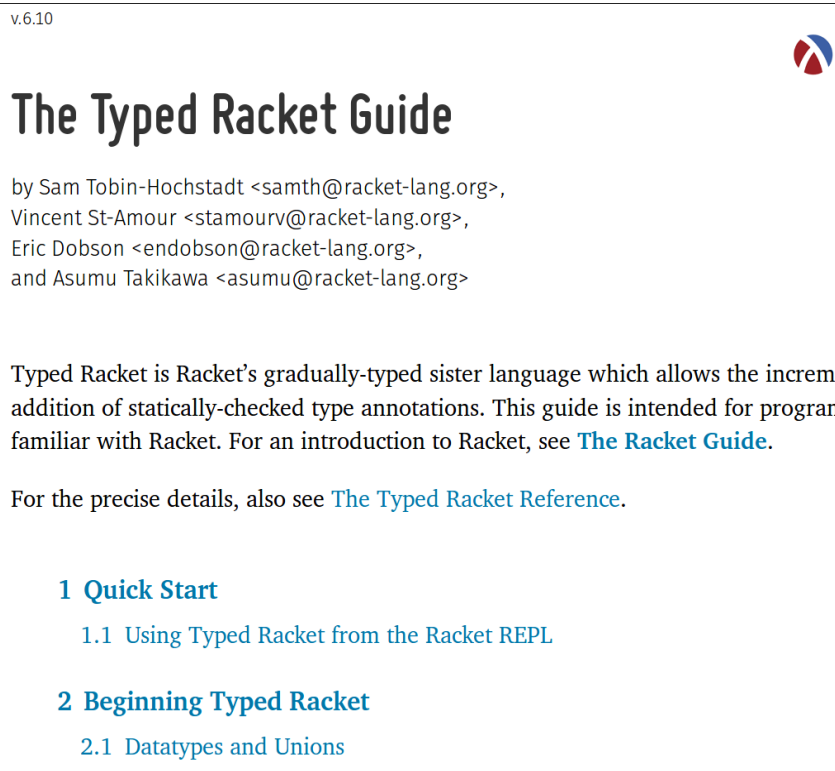
ABSTRACT

1. WHEN SCRIPTS GROW UP

Sound Gradual Typing

2017





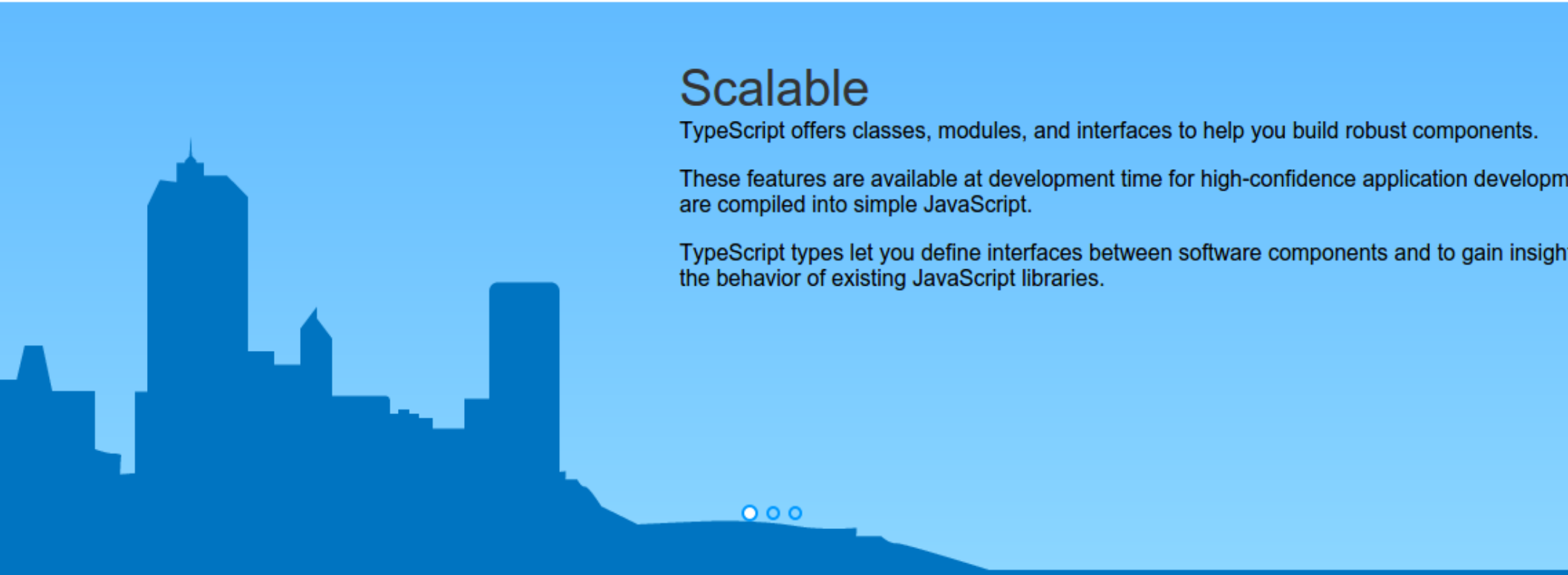
Sound Gradual Typing

TypeScript

learn play download interact

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 developm
are compiled into simple JavaScript.

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

Node.js

he command-line TypeScript compiler can be

Tools

Visual Studio includes TypeScript in the box.

Open Source

TypeScript is being developed on GitHub. The

Programming productivity without breaking things

INSTALL TUTORIAL

```hack
<?hh
class MyClass {
  public function alpha(): int {
    return 1;
  }

  public function beta(): string {
    return 'hi test';
  }
}

function f(MyClass $my_inst): string {
  // Fix me!
  return $my_inst->alpha();
}
```

What is Hack?

Click here to read the official Hack announcement

Hack Developer Day: April 9, 2014

Hack is a programming language for **HHVM** that interoperates seamlessly with PHP. Hack

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.

README.md

Titan

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.rockspec
```

Usage

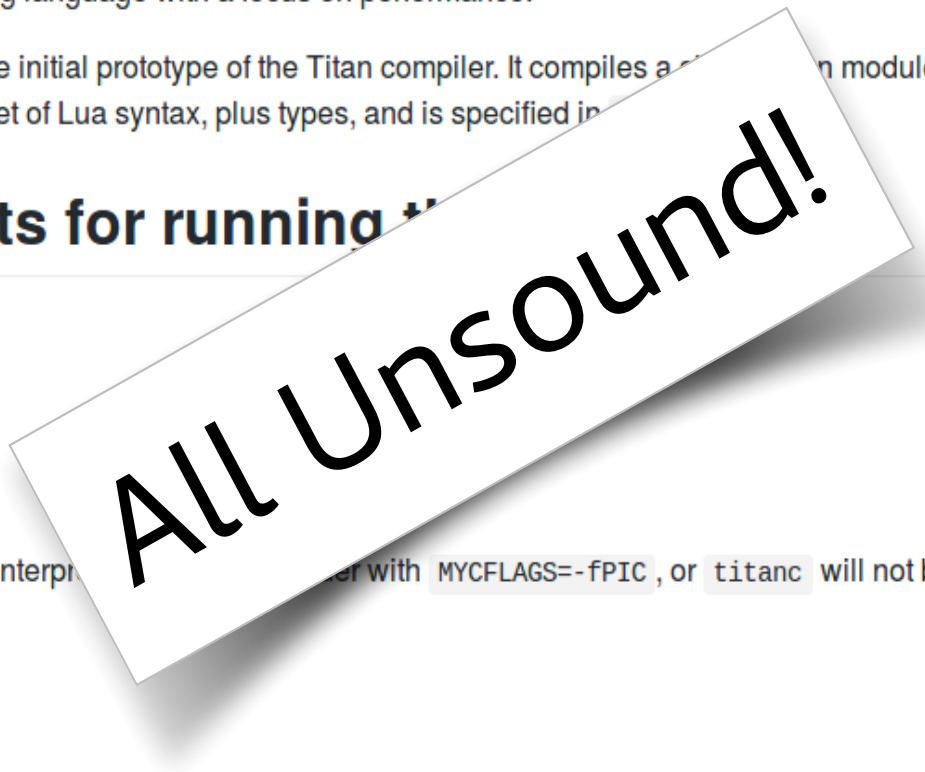
README.md

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 [illegible] module to C code in the artisanal style. The syntax is a subset of Lua syntax, plus types, and is specified in [illegible]



Requirements for running [illegible]

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 interpr[illegible]er 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.rockspec
```

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 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

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-

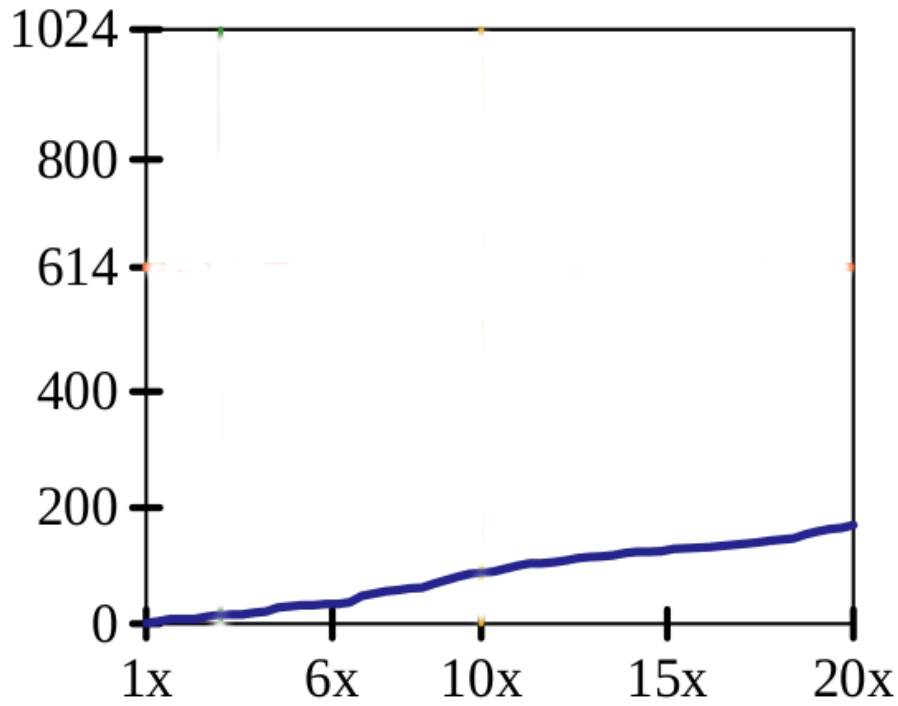
mare, mostly due to the lack of reliable type information.

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

synth (10 modules)

max. overhead	85.90x
mean overhead	39.69x



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

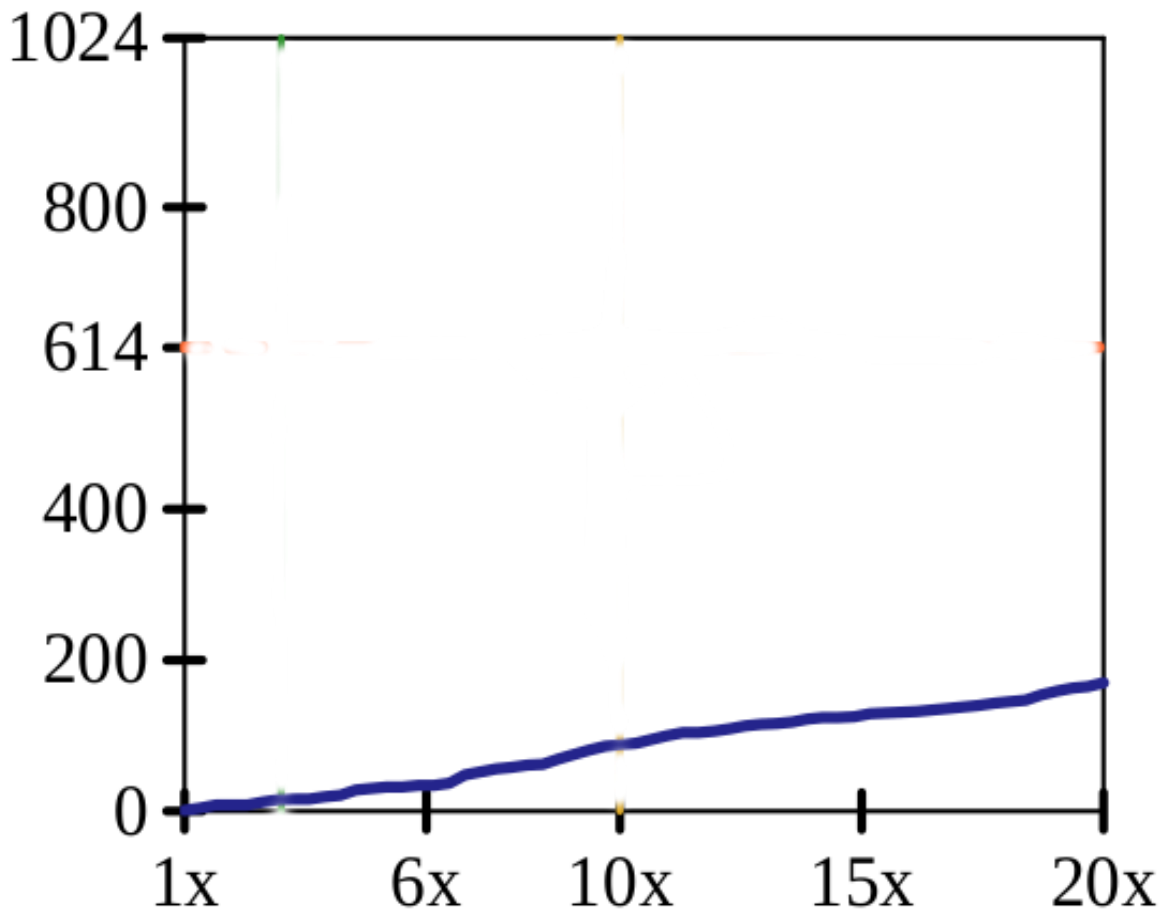
In response, this paper proposes a method for evaluating the performance of gradually-typed programming languages. The method hinges on exploring the space of partial conversions from untyped to typed. For each benchmark, the performance of the different ver-

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

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

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 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

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.

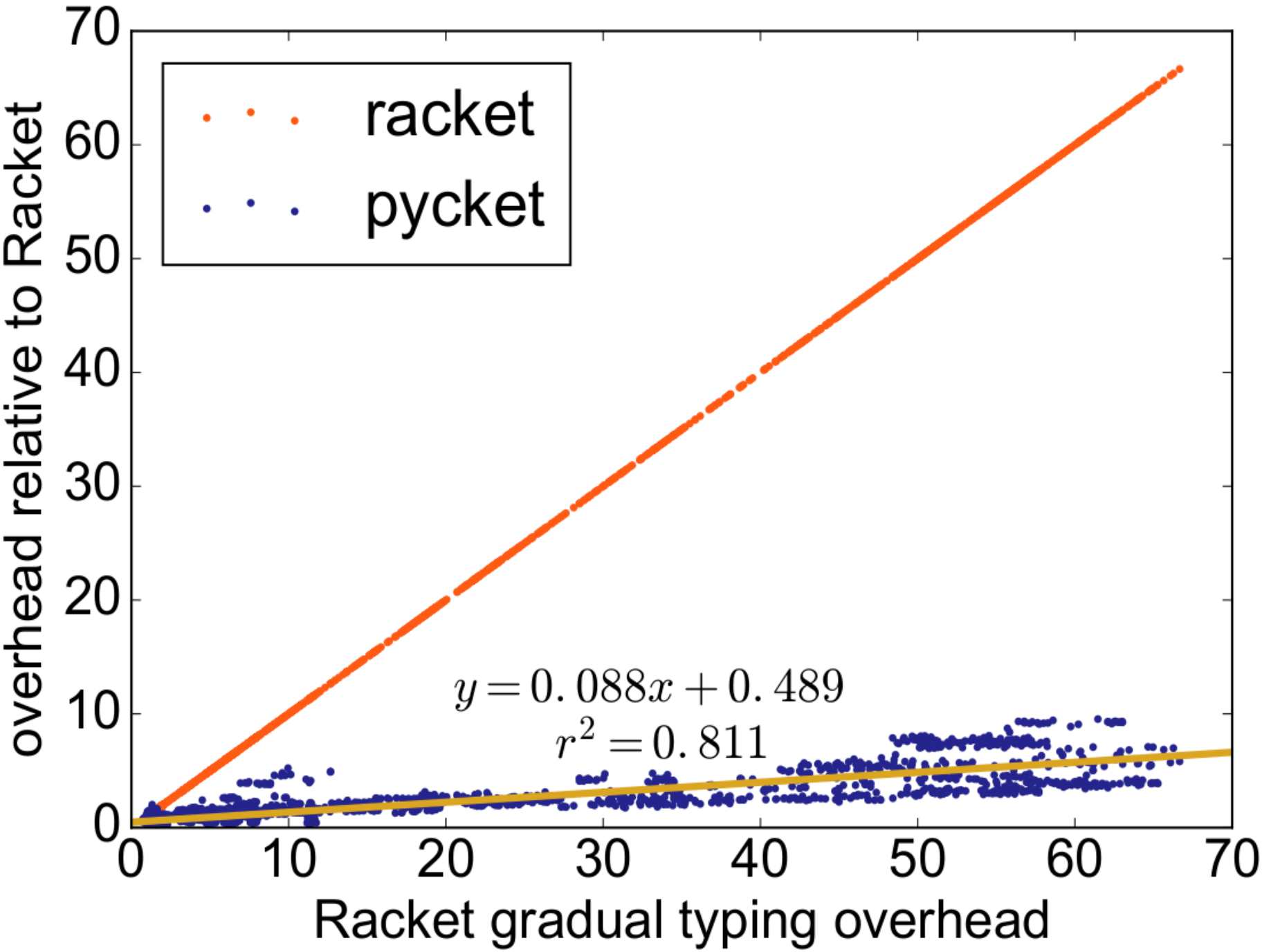
In response, this paper proposes a method for evaluating the performance of gradually-typed programming languages. The method hinges on exploring the space of partial conversions from untyped to typed. For each benchmark, the performance of the different ver-

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.

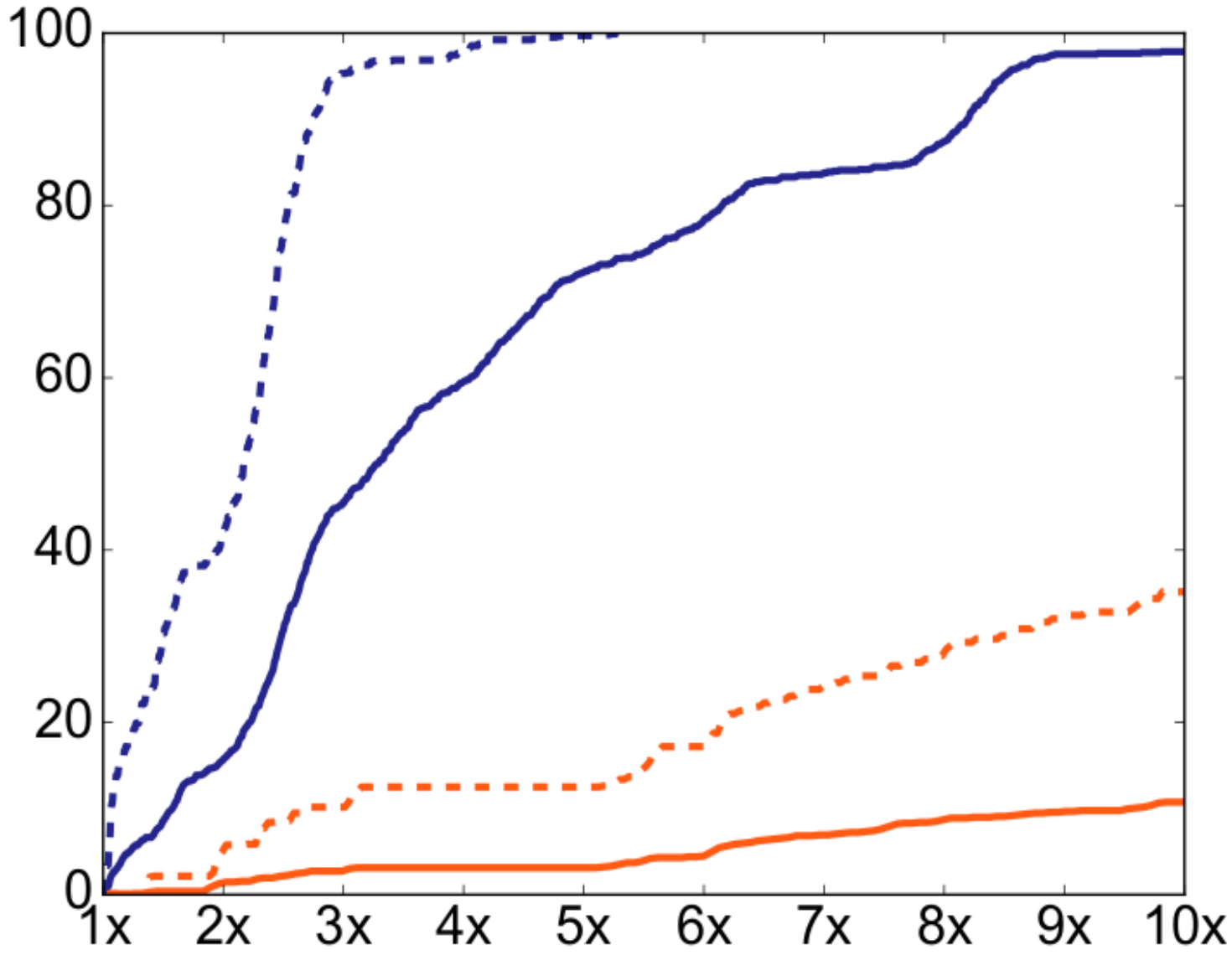
Gradual typing [21, 26] proposes a language-based solution to this pressing software engineering problem. The idea is to extend the language so that programmers can incrementally equip programs with types. In contrast to optional typing, gradual typing provide programmers with soundness guarantees.

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


Enter Pycket


racket
pycket
$y = 0.088x + 0.489$
$r^2 = 0.811$
overhead relative to Racket
Racket gradual typing overhead
0
10
20
30
40
50
60
70

Synth, again



Two Key Ideas

- Tracing JIT Compilation
- Hidden Classes for Chaperones

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;
```

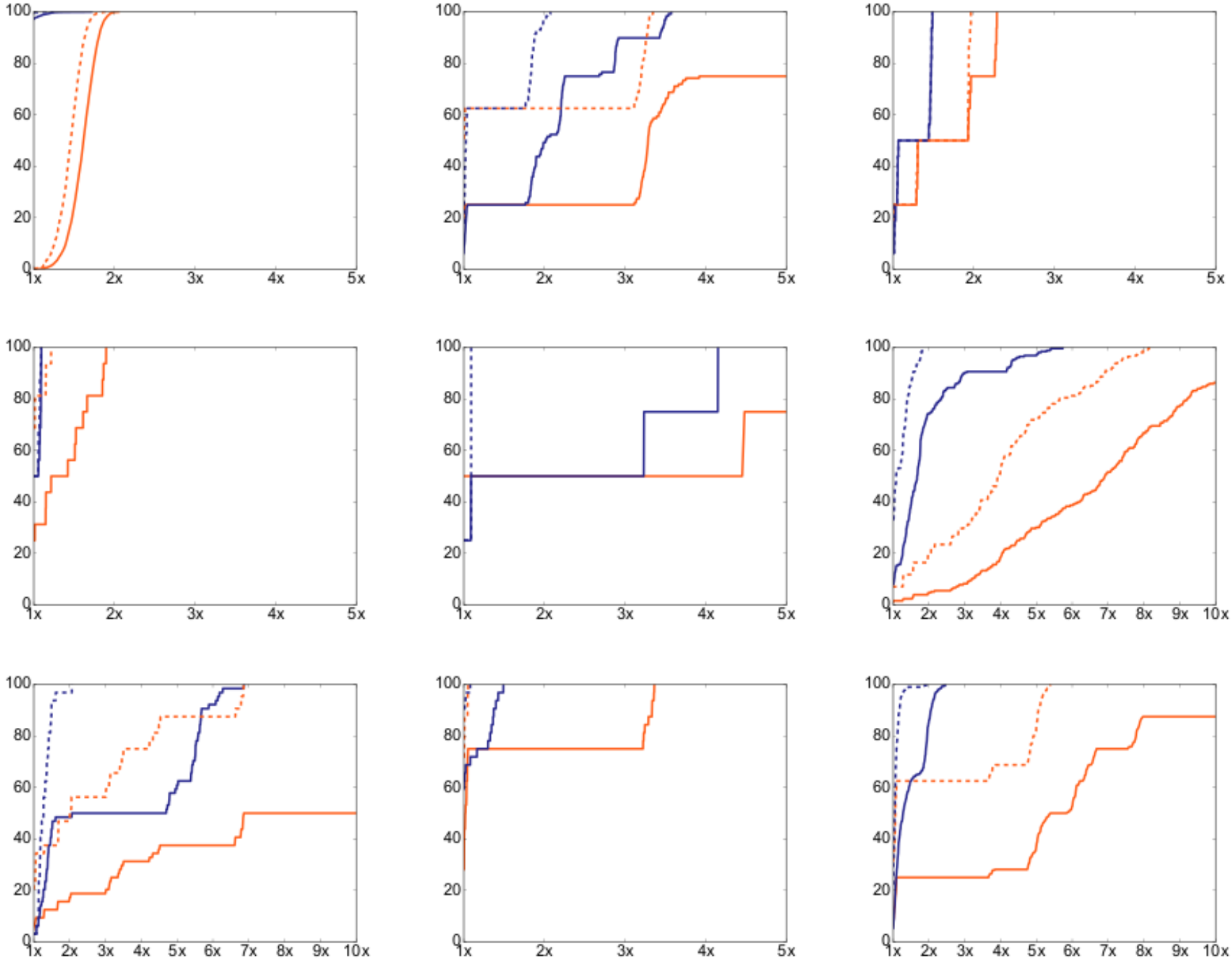
```
(f x)
```

```
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;
```


100
80
60
40
20
0
1x
2x
3x
4x
5x
6x
7x
8x
9x
10x

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`


Sound Gradual Typing
Mostly Dead is Slightly Alive
github.com/pycket