ON PERFORMANCE
(of gradual typing, esp. in Racket)
This Talk is NOT About

➤ Horrific performance overhead
➤ The death of gradual typing
➤ Impending doom
"[Typed Racket is] Very nice to work with!"

"The static typechecking is invaluable to me"

"Typed Racket has improved [my Racket prototype] considerably."
"What I find appealing about TR's **gradual typing** is the idea that, like the contract system, there's not One Right Way to use it. For instance, I've been using TR simply as a way of **creating better untyped code**, because the typechecker catches subtle reasoning errors."
We are a coalition of researchers seeking to discover the unifying principles underlying the design of gradual type systems through reproducibility studies, implementations of type systems and tools, plus evaluations covering both the feasibility of gradual typing as well as its long-term value to software engineers.
Get Involved

We are actively seeking talented students and researchers at all levels. Stop by one of our offices if you're in town, or visit our websites to learn how to apply for your Masters, Ph.D, or post.

Visit our websites for information:
- Brown University [https://www.brown.edu/academics/gradschool/apply]
- Indiana University [http://www.soic.indiana.edu/graduate/admissions/how-
- Northeastern University [http://www.ccis.northeastern.edu/academics/phd/
- University of Maryland [https://gradschool.umd.edu/admissions]]}

General Information

For questions or comments about this website, email benjaminlgreenman@gmail.com.
#lang typed/racket

(define-type Year Natural)
(define-type Degree (U 'phd 'me 'bse 'diplom 'ms 'msc 'postdoc 'bs 'bsc))
(define-type Degree* (Listof (List Degree University Year)))
(define-type Email email)
(define-type Position* (Listof (List University Year)))

(struct person (
    [short-name : String]
    [full-name : String]
    [gender  : Symbol]
    [title   : String]
    [mailto  : Email]
    [href    : URL]
    [degree* : Degree*]
) #:transparent)

(struct student person ([university : University]) #:transparent)
(struct pi person ([position* : Position*]) #:transparent)

(struct university (
    [name : String]
    [href : URL]
) #:transparent)
10 modules

6 untyped

"building web pages"

4 typed

"representing data"
Typed Racket is:

**Sound**

All runtime type errors are caught at a boundary between typed and untyped code.
Typed Racket is:

Sound and Expressive

All runtime type errors are caught at a boundary between typed and untyped code

Seamless integration with untyped code
Typed Racket is:

**Sound** and **Expressive**

All runtime type errors are **caught** at a **boundary** between **typed** and **untyped** code

"The static typechecking is invaluable to me"

"I've been using TR [for] creating better *untyped* code"
Performance?
"About twice as slow on common queries"

"From 1 ms to 12 seconds ... I feel like I got a bit burned here"

"The end product appears to be a 50% performance hybrid due to boundary contracts"
"So far Typed Quad is running about 10x slower than regular...it seems that whatever I'm gaining [from the TR optimizer] is more than offset by other factors.

"FWIW, as a practitioner, there are costs associated with using TR, therefore it has to provide equivalent performance improvements to be worthwhile at all.

"'equally good' runtime perf = net loss overall = I can't justify using it."
Research Questions

How to leverage case studies to systematically improve performance?

How to evaluate the performance of a gradual type system?

5 Lessons ➔ 2 Design Criteria ➔ 1 Design
"About twice as slow on common queries"

"From 1ms to 12 seconds ..."

"... a 50% performance hybrid ...

"... about 10x slower than regular ..."
Lesson 1: the problem is 

overhead introduced by gradual typing
"About twice as slow on common queries"

"From 1ms to 12 seconds ..."

"... a 50% performance hybrid ..."

"... about 10x slower than regular ..."
Lesson 2: users have diverse performance requirements
3x is **NOT** "Deliverable"

10x is **NOT** "Usable"
Lesson 2: users have diverse performance requirements
"About twice as slow on common queries"

"From 1ms to 12 seconds ... "

... a 50% performance hybrid ...

"... about 10x slower than regular ..."
Lesson 3: developers may tolerate slowdown between releases
Lesson 1: problem = overhead

Lesson 2: diverse user requirements

Lesson 3: development vs. production

Criteria 1: evaluation must show a range of overhead values
Lesson 4: we don't know why programmers add types
- High security?
- Stable API?
- Tightly coupled?
- Easy to annotate?

Gradual typing promises to support **ANY** use-case
Lesson 5: fully-typed is not the goal
Get Involved

We are actively seeking talented students and researchers at all levels. Stop by one of our offices if you're in town, or visit our websites to learn how to apply for your Masters, Ph.D., or post.

admissions:\`
(,brown-university "https://www.brown.edu/academics/gradschool/apply")
(,indiana-university "http://www.soic.indiana.edu/graduate/admissions/how-
(northeastern-university "http://www.ccis.northeastern.edu/academics/phd/
(university-of-maryland "https://gradschool.umd.edu/admissions"))}

General Information

For questions or comments about this website
Email AT{benjamingreenman} AT{gmail.com}
239 modules

123 typed
116 untyped

"new code"       "legacy code"
239 modules

123 typed  116 untyped

"new code"  "legacy code"
Lesson 4: cannot predict use-cases
Lesson 5: fully-typed is not the goal

Criteria 2: evaluation must consider all possible ways of gradually using types in a program
1024 total configurations

x45

x120

...

x10
 Criteria 1: show a range of overhead values

Criteria 2: consider all possible ways of using types in a program
1024 total configurations

x45

x120

...

x10
% Configs.

Overhead (vs. untyped)
% Configs.

Overhead (vs. untyped)
3x is NOT "Deliverable"

10x is NOT "Usable"
% Configs.

Overhead (vs. untyped)
% Configs.

Overhead (vs. untyped)

"useful"

"curious"
% Configs.

Overhead (vs. untyped)

GC ~1984

"useful"

"curious"
% Configs.

Overhead (vs. untyped)
% Configs.

Overhead (vs. untyped)
% Configs.

Overhead (vs. untyped)
% Configs.

Overhead (vs. untyped)
morsecode
100%
tetris
100%
suffixtree
100%
sieve
100%
zombie
100%
synth
100%
kcf
100%
forth
100%
dungeon
100%
snake
100%
fsm
100%
zordonz
100%
take5
100%
fsmoo
100%
quadBG
100%
acquire
100%
mbta
100%

Lessons:
- problem = overhead
- diverse user requirements
- development vs. production
- cannot predict use-cases
- fully-typed is **not** the goal

Criteria:
- show range of overheads
- consider all possible ways of using types

Implementation:
with Zeina Migeed
➤ Apply methodology to Reticulated
➤ Identify bottlenecks (and bugs)
➤ Compare cast insertion strategies
Take5

FSM

Evolution
<table>
<thead>
<tr>
<th></th>
<th>Modules</th>
<th>Classes</th>
<th>Fields</th>
<th>Functions</th>
<th>Args</th>
</tr>
</thead>
<tbody>
<tr>
<td>Take5</td>
<td>3</td>
<td>2</td>
<td>10</td>
<td>14</td>
<td>30</td>
</tr>
<tr>
<td>FSM</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>17</td>
<td>30</td>
</tr>
<tr>
<td>Evolution</td>
<td>11</td>
<td>10</td>
<td>39</td>
<td>+50</td>
<td>++50</td>
</tr>
</tbody>
</table>

\[ 2^{11} = 2,048 \quad 2^{17} = 131,072 \quad 2^{39} = 549,755,813,888 \]
<table>
<thead>
<tr>
<th>Modules</th>
<th>Classes</th>
<th>Fields</th>
<th>Functions</th>
<th>Args</th>
</tr>
</thead>
<tbody>
<tr>
<td>Take5</td>
<td>3</td>
<td>2</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>FSM</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>Evolution</td>
<td>11</td>
<td>10</td>
<td>39</td>
<td>+50</td>
</tr>
</tbody>
</table>

\[2^{11} = 2,048 \quad 2^{17} = 131,072 \quad 2^{39} = 549,755,813,888\]
from retic import List, Tuple, Void, String, Int

class Player:

    def __init__(self, name: Int, cards: List(Tuple(Int, Int))) -> Void:
        self.name = name
        self.cards = cards

    def discard(self) -> Int:
        ....

    def choose_correct_stack(self, stacks: List(List(Tuple(Int, Int)))) -> Int:
        ....

    def get_index_of_closest_stack(self, cards: List(Tuple(Int, Int)), card: Tuple(Int, Int)) -> Int:
        ....
Can we predict performance for exponentially many configurations given a linear number of measurements?
"The End"
Weaknesses

➤ No absolute runtimes
➤ No map from configs. to overheads
➤ Does not express migration paths
(require "population.rkt")

(define (evolve pop count)
  (if (zero? count)
      null
      (evolve (step pop) (- count 1)))))

(define-type Population
  (Class ....))

(provide
  (step (Population -> Population))
  (create (Natural -> Population)))

(evolve (create 100) 5)
fsm

100%

50
don 0

1 2 20x

16 configurations

100%

50
don 0

1 2 20x
Weaknesses 2

➤ Many ways of typing a program
➤ Many ways of modularizing a program
Data from Reticulated
transient

Take5
FSM guarded

low overheads!

Take5
Finally, it is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous.

What would we think of a sailing enthusiast who wears his lifejacket when training on dry land, but takes it off as soon as he goes to sea?

- C.A.R. Hoare

On the other hand, that sailor isn't so foolish if life vests are extremely expensive and if he is such an excellent swimmer that the chance of needing one is quite small compared with the other risks he is taking.

- Donald Knuth
3x is **NOT** "Deliverable"

10x is **NOT** "Usable"
4.2 Reading the Figures

Our method defines the number of $L$-step $N/M$-usable configurations as the key metric for measuring the quality of a gradual type system. For this experiment we have chosen values of $3x$ and $10x$ for $N$ and $M$, respectively, and allow up to 2 additional type conversion steps. These values are rather liberal, but serve to ground our discussion.

---

7 We would expect that most production contexts would not tolerate anything higher than $2x$, if that much.
Misc. Quotes from Typed Racket users
The end-product appears to be a 50%-performance hybrid due to boundary contracts, but ameliorated runtime-wise by utilizing the typed/racket/no-check language after it's all working in type checked mode. JGC

But user time is limited too. In my case, I'm trying to decide whether TR is a cost-effective upgrade for my Racket program.
needs to be at least 10x faster. This was the original impetus for trying TR — improving performance by avoiding contracts + getting type-optimized operations. But it seems that whatever I'm gaining is more than offset by other factors.

MB

Unfortunately, the prototype worked so well that I'm using it now for real. JGC

I use typed racket in production too, and I also heavily use Scribble on the same source codebase. WG
For me as a programmer, **Typed Racket is a different language** from Racket, because a valid program in one language is not a valid program in the other. Whether or not Typed Racket's hash ends up calling plain Racket's hash is an implementation detail I don't care about, except perhaps when dealing with interfacing modules in the two languages.

From this point of view, Typed Racket is to a large degree an **undocumented** language. Much of the documentation simply points to the one of plain Racket, which doesn't fully apply.

Moreover, there is no simple set of rules that would let me deduce Typed Racket's API (which includes types) from plain Racket's API.

KH

https://groups.google.com/forum/#!searchin/racket-users/typed$20racket/racket-users/-RI1p1Z1ZRE/u2e6ECZ1_LcJ
the recurring tasks that sends me looking for documentation is
instantiating polymorphic functions into appropriate type-specific
forms using `(inst proc args ...)`. 