Pleiad

Gradual Typing

Éric Tanter
University of Chile

https://pleiad.cl/etanter

Gradual Typing

Basics

Static **vs** Dynamic Type Checking

Long-standing divide in programming languages

static	dynamic
early error detection	flexible programming idioms
enforce abstractions	rapid prototyping
checked documentation	no spurious errors
efficiency	simplicity
Java, Scala, C#/…, ML, Haskell, Go, Rust, etc.	Python, JavaScript, Racket, Clojure, PHP, Smalltalk, etc.

why should we have to choose?

can't we have both?

Static **and** Dynamic Checking

many languages (now) try to have both

C# 4.0
Typed Clojure
Dart
ActionScript
Python 3
TypeScript
Typed Racket
Hack
Elixir
Perl 6
Scala
Kotlin

very different flavor & guarantees…

Static **and** Dynamic Checking

many different theories too!

hybrid typing
multi-language programs
soft typing
quasi-static typing
gradual typing
RTTI
optional typing
manifest contracts

very different flavor & guarantees…

Gradual Typing

[Siek & Taha, 2006]

- **Combine** both static and dynamic checking in a single language
- Programmer **controls** which discipline is used where
- Supports **seamless evolution** between static/dynamic
- **Pay-as-you-go**: static regions can be safely optimized

Fully Static & Fully Dynamic

Gradual as superset of static and dynamic

```
def f(x) = x + 2
def h(g) = g(1)
h(f)
```
→ 3 ✓

```
def f(x) = x + 2
def h(g) = g(true)
h(f)
```
→ true + 2 ✗
runtime error

```
def f(x:int) = x + 2
def h(g:int→int) = g(1)
h(f)
```
→ 3 ✓

```
def f(x:int) = x + 2
def h(g:int→int) = g(true)
h(f)
```
✗
static error

Sound Interoperability

Partially-typed programs

```
def f(x:int) = x + 2
def h(g) = g(1)
h(f)
```

⟶ 3 ✓

```
def f(x:int) = x + 2
def h(g) = g(true)
h(f)
```

⟶ f(true) ✗

runtime error
at the boundary

protect assumptions made in static code

Inside Gradual Typing

```
def f(x) = x + 2
def h(g) = g(true)
h(f)
```

=

```
def f(x:?) = x + 2
def h(g:?) = g(true)
h(f)
```

unknown type ?

Inside Gradual Typing

static semantics: consistency

type equality

T = T

type consistency

T ~ T

T ~ ? ? ~ T

$$\frac{S \sim S' \quad T \sim T'}{S \to T \sim S' \to T'}$$

not transitive!

int ~ ? ? ~ bool

int ≁ bool

```
def f(x:int) = x + 2
f(true)
```

✗ **static error**

Inside Gradual Typing

dynamic semantics: casts

```
def f(x:?) = x + 2
```

⟶

```
def f(x:?) = <int⇐?>x + 2
```

check that it's an int

```
f(5) ⟶ <int⇐?>5 + 2 ⟶ 5 + 2 ⟶ 7
```

✓

```
f(true) ⟶ <int⇐?>true + 2 ⟶ runtime error
```

✗

```
def f(x:int) = x + 2
def h(g) = g(true)
h(f)
```

body is safe!
can be compiled efficiently

```
def f(x:int) = x + 2
def h(g:?) = (<?→?⇐?>g)(<?⇐bool>true)
h(<?⇐int→int>f)
```

check it is a func

tagged value

```
⟶ (<?→?⇐?><?⇐int→int>f)(<?⇐bool>true)
⟶ (<?→?⇐int→int>f)(<?⇐bool>true)
⟶ fun(x:?){<?⇐int>f(<int⇐?>x)}(<?⇐bool>true)
⟶ <?⇐int>f(<int⇐?><?⇐bool>true)
⟶ <?⇐int>f(<int⇐bool>true)   ⟶ runtime error
```


Gradual Typing Research Directions

Metatheory & criteria

Foundational methodologies

Languages features

Advanced typing disciplines

Implementation & performance

Beyond Simple Gradual Typing

- Subtyping (structural/nominal, records, objects, classes) [Siek&Taha'07, Ina&Igarashi'11, Takikawa+'12, etc.]
- Parametric polymorphism [Ahmed+'08'11'17'20, Igarashi'17, Toro+'19, etc.]
- Type inference and gradual types [Siek&Vachharajani'08, Garcia&Cimini'15]
- Union and recursive types [Siek&Tobin-Hochstadt'16]
- Algebraic data types [Malewski+'21]
- Delimited continuations [Miyazaki+'16]
- Effect handlers [New+'23]

…

Gradual Typing

Refined

Properties of Gradual Languages

[Siek & Taha, 2006]

type safety — admits runtime type errors

If $\vdash t : T$ then either
t is a value v, or
$t \longmapsto t'$ with $\vdash t' : T$, or
$t \longmapsto \mathbf{error}$

equivalence for static terms — conservative extension

embedding of dynamic terms — expressive

Conservative Extension wrt Static

Static Language → Gradual Language

7 3 → 7 3

"7 applied to 3???"

NOPE!

static error

$\vdash_S t : T$ if and only if $\vdash t : T$ $\quad t \Downarrow_S v$ if and only if $t \Downarrow v$

Dynamic Embedding

Dynamic Language → Gradual Language

7 3 → $(7 :: ?)\ (3 :: ?)$

"7 applied to 3???"

YUP!

dynamic error

If t is closed then $\vdash \lceil t \rceil : ?$ $\quad t \Downarrow_D r$ if and only if $\lceil t \rceil \Downarrow r$

What do you mean "Gradual"?

[Siek *et al.*, 2015]

"meaning has become **diluted** to encompass anything related to the integration of static and dynamic typing"

Refine

Jeremy
John Ta

1 Indian
150 S
jsiek@

2 University of Wisconsin – waukee, Department of EECS
PO Box 784, Milwaukee W 3201, USA
boyland@cs.uwm.edu

Abstract

Siek and Taha [2006] coined the ter *gradual typing* to describe a theory for integrating static and dynamic typing within a single language that 1) puts the programmer in control of which regions of code are statically or dynamically typed and 2) enables the gradual evolution of code between the two typing disciplines. Since 2006, the term *gradual typing* has become quite popular but its meaning has become diluted to encompass anything related to the integration of static and dynamic typing. This dilution is partly the fault of the original paper, which provided an incomplete formal characterization of what it means to be gradually typed. In this paper we draw a crisp line in the sand that includes a new formal property, named the *gradual guarantee*, that relates the behavior of programs that differ only with respect to their type annotations. We argue that the gradual guarantee provides important guidance for designers of gradually typed languages. We survey the gradual typing literature, critiquing designs in light of the gradual guarantee. We also report on a mechanized proof that the gradual guarantee holds for the Gradually Typed Lambda Calculus.

1998 ACM Subject Classification F.3.3 Studies of Program Constructs – Type structure

Gradual Typing, refined

[Siek *et al.*, 2015]

it's all about (**im**)**precision**

some gradual types convey more information than others

$\text{Int} \to \text{Int} \sqsubseteq \text{Int} \to ? \sqsubseteq ? \to ? \sqsubseteq ?$

Gradual Typing
best-effort static checking
backed by dynamic checking

Precision

type precision extends to **term precision**

```
def f(x) = x + 2
def h(g) = g(1)
h(f)
```

```
def f(x:int) = x + 2
def h(g) = g(1)
h(f)
```

```
def f(x:int) = x + 2
def h(g:int→int) = g(1)
h(f)
```

no explicit checks
evolution is completely driven by type annotations

Properties of Gradual Languages (ctd)

[Siek *et al*, 2015]

```
def f(x) = x + 2
def h(g) = g(1)
h(f)
```

```
def f(x:int) = x + 2
def h(g) = g(1)
h(f)
```

```
def f(x:int) = x + 2
def h(g:i
h(f)
```

Gradual Guarantee

losing precision

1) preserves typing

2) preserves reduction

adding type information can only introduce new static/dynamic errors

TS* is not "Gradual"

Gradual Typing Embedded Securely in JavaScript

Nikhil Swamy[1] Cédric Fournet[1] Aseem Rastogi[2] Karthikeyan Bhargavan[3]
Juan Chen[1] Pierre-Yves Strub[4] Gavin Bierman[1]

Microsoft Research[1] University of Maryland[2] INRIA[3] IMDEA Software Institute[4]
{nswamy, fournet, juanchen, gmb}@microsoft.com, aseem@cs.umd.edu, karthikeyan.bhargavan@inria.fr, pierre-yves@strub.nu

[Swamy et al., POPL'14]

$(\lambda f : \star.\ f\ \texttt{true})\ (\lambda x : \star.\ x)$ ok

$(\lambda f : \texttt{bool}\to\texttt{bool}.\ f\ \texttt{true})\ (\lambda x : \star.\ x)$ runtime error!

$(\lambda f : \texttt{bool}\to\texttt{bool}.\ f\ \texttt{true})\ (\lambda x : \texttt{bool}.\ x)$ ok

the gradual guarantee avoids such hiccups

"Coe ner-order values. For example, trying to coerce the identity function to the type $\texttt{bool} \to \texttt{bool}$ using `setTag` will fail, since $\star \not<: \texttt{bool}$." [57]

Optional or Gradual?

protect assumptions made in static code

```
def f(x:int→int) = x(2)
def h(g) = g(true)
h(f)
```

→ f(true) ✗ → true(2) ✗

runtime error at the boundary

runtime error not safe otherwise!

“ok” if adding gradual types to a safe dynamic language
(eg. JS, Python, Elixir, Racket)

even if safe, could be to
(side-effects, type for secu

```
>>> x = True
>>> x(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bool' object is not callable
```


Optional or Gradual or Sound Gradual or ?

Pluggable Type Systems

Gilad Bracha

October 27, 2004

Is Sound Gradual Typing Dead?

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

Northeastern University, Boston, MA

Sound Gradual Typing is Nominally Alive and Well

FABIAN MUEHLBOECK, Cornell University, USA

ROSS TATE, Cornell University, USA

Abstract

Recent research has identified significant performance hurdles that sound gradual typing needs to overcome. These performance hurdles stem from the fact that the run-time checks gradual type systems insert into code can cause significant overhead. We propose that designing a type system for a gradually typed language hand in hand with its implementation from scratch is a possible way around these and several other hurdles on the way to efficient sound gradual typing. Such a design process also highlights the type-system restrictions required for efficient composition with gradual typing. We formalize the core of a nominal object-oriented language that fulfills a variety of desirable properties for gradually typed languages, and present evidence that

“optional type systems are neither syntactically nor semantically required, no effect on the dynamic

Gradual typing is a way to combine static and dynamic typing. Type-annotated Python allows opting in to static type checking at a fine level of granularity, so that some type errors can be caught statically,

A **gradual** type system is one in which a special “unknown” or “dynamic” type is used to describe names or expressions whose types are not known statically. In Python, this type is spelled Any.

These operations are still dynamically checked, via the Python runtime’s usual dynamic checking.

What do programmers want?

[OOPSLA’18]

A Spectrum of Type Soundne

BEN GREENMAN, PLT @ Northeastern Univer

MATTHIAS FELLEISEN, PLT @ Northeastern

“Our most important finding is that our respondents prefer a runtime semantics that fully enforces statically declared types”

The Behavior of Gradual Types: A User Study

Preston Tunnell Wilson
Brown University
Providence, Rhode Island, USA
ptwilson@brown.edu

Ben Greenman
Northeastern University
Boston, Massachusetts, USA
benjaminlgreenman@gmail.com

Justin Pombrio
Brown University
Providence, Rhode Island, USA
jpombrio@cs.brown.edu

Shriram Krishnamurthi
Brown University
Providence, Rhode Island, USA
sk@cs.brown.edu

Abstract

There are several different gradual typing semantics, reflecting different trade-offs between performance and type soundness guarantees. Notably absent, however, are any data on which of these semantics developers actually prefer.

We begin to rectify this shortcoming by surveying professional developers, computer science students, and Mechanical Turk workers on their preferences between three gradual typing semantics. These semantics reflect important points in the design space, corresponding to the behaviors of Typed Racket, TypeScript, and Reticulated Python. Our most important finding is that our respondents prefer a runtime

typing [27, 31]. In a gradually typed language, programmers are free to mix typed and untyped code. Some of the early gradually typed languages were created by retrofitting a type system on a (sublanguage of a) dynamic language (e.g., Typed Racket [32, 33], TypeScript [3], and Reticulated Python [36]); more recently, new languages are being made gradually typed from the outset, such as Pyret (pyret.org) and Dart 1 (v1-dartlang-org.firebaseapp.com).

But what should the semantics of a gradually-typed program be? In particular, when typed and untyped regions of code interact, what sort of runtime checks should protect the invariants of typed code? The answer to this question

[DLS’18]

What do programmers want?

are they willing to pay for it?

Science of Computer Programming

Volume 96, Part 1, 15 December 2014, Pag

Gradual typing for Smallt

Esteban Allende [a], Oscar Callaú [a], Johan Fab

Marcus Denker [b]

Racket

6.3.4 When to Use Deep, Shallow, or Optional?

- Deep types maximize the benefits of static checking and type-driven optimiz them for tightly-connected groups of typed modules. Avoid them when unty higher-order values frequently cross boundaries into typed code. Expensive t types include `Vectorof`, `->`, and `Object`.
- Shallow types are best for small typed modules that frequently interact with code. This is because Shallow shape checks run quickly: constant-time for m and linear time (in the size of the type, not the value) for a few exceptions s and `case->`. Avoid Shallow types in large typed modules that frequently call or access data structures because these operations may incur shape checks a cost may be significant.
- Optional types enable the typechecker and nothing else. Use them when you want types enforced at run-time.

Dart

On this page > Customizing static analysis

Customizing st

Tools > Static analysis

Static analysis allows you to find problems be powerful tool used to prevent bugs and ensu

With the help of the analyzer, you can find simple typos. For example, perhaps an accidental semicolon made its way into an `if` statement:

More Criteria for Gradual Typing

it's not over yet ;)

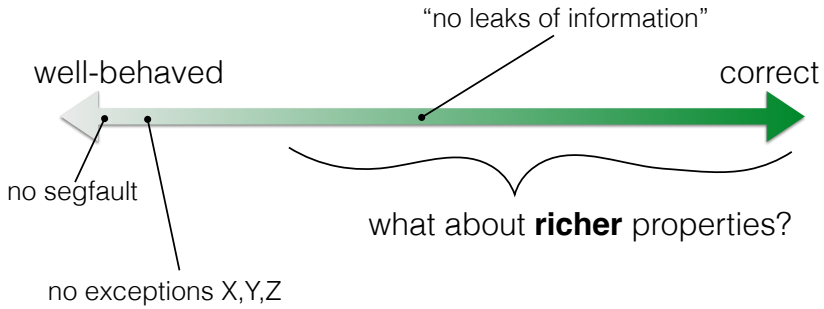
- Blame assignment [ESOP'09,TOPLAS'23]
- Open world soundness [POPL'17]
- Complete monitoring [OOPSLA'19]
- Graduality (semantic account of DGG) [ICFP'18]
- Fully-abstract embedding from static to gradual [POPL'21]
- Vigilance [OOPSLA'24]
- Robust dynamic embedding [ICFP'25]

Gradual Typing

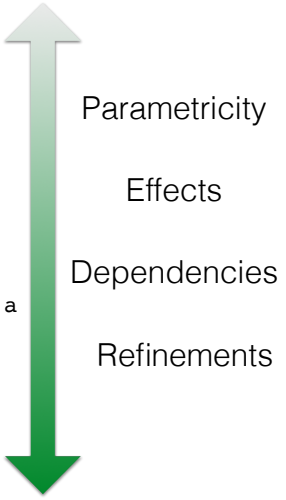
Extended

What are types useful for?

define & enforce **properties** about program behavior




```
list → list → list

list a → list a → list a

list a → list a ⟶IO list a

list a n → list a m → list a (n+m)

n:int{n>=0} → l:list a{length l > n} → Tot a

l1:list a n → l2:list a m → l:list a (n+m)
  {∀i i∈{0..n-1}⇒l[i]=l1[i]
    ∧ i∈{n..n+m}⇒l[i]=l2[i]}
```



Rich Types: Bestiary

Parametricity

Affine types

Effects

Typestates

Dependencies

Ownership types

Linear types

Security types

Refinements

Energy types

Session types

Sensitivity types

Advanced Gradual Types

some examples

- Gradual **typestate** [ECOOP'11, TOPLAS'14]
- Gradual **effects** [ICFP'14, OOPSLA'15, JFP'16]
- Gradual **refinement types** [POPL'17, OOPSLA'18]
- Gradual **security types** [TOPLAS'18]
- Gradual **sensitivity types** [CSF'25]
- Gradual **parametricity** [POPL'19, OOPSLA'22, JACM'22]
- Gradual **dependent types** [ICFP'19, TOPLAS'22, ICFP'22 (x2), ICFP'24]

Gradual Effects

[ICFP'14, OOPSLA'15, JFP'16]

Effects

[Marino & Millstein, 2009]

performing an **effectful operation**
requires the corresponding **privilege**

effect domains	**effect privileges**	**effectful operations**
I/O	in, out, err	println, File.read(), …
memory	alloc, read, write	new, x[i], x[i]=y, …
exceptions	raise[T]	throw e
…	…	…

Effect Systems

$T_1 \xrightarrow{\Phi} T_2$ — set of latent effects

```
// Int -{io}-> Int
def f(x: Int): Int @{io} =
  println("hola")
  x + 1
```

✓

```
// Int -{}-> Int
def f(x: Int): Int @{} =
  println("hola")
  x + 1
```

✗ static error

Gradual Effects

```
def f(x: Int): Int =
  println("hola")
  x + 1
```

→

```
def f(x: Int): Int @{¿} =
  has(print); println("hola")
  x + 1
```

✗ runtime error

dynamic check: **has print privilege?**

"untyped" ≡ has unknown effect ¿

```
def run(callback: Int -{}-> Int) =
  v = …
  callback(v)
```

restrict current context to **no privileges**

```
run(f)
```

→

```
run(fun(x:Int){ restrict {} f(x) })
```

✓

has/restrict play the role of "effect casts"

Gradual Refinement Types

[POPL'17, OOPSLA'18]

Refinement Types

```
type Nat = {ν:Int | ν ≥ 0}

def fib(x: Nat): Nat

def isNat(x: Int):{ν: Bool | ν=true ⇔ x ≥ 0}

def bar(x: Int): Int
  if isNat(x)
  then fib(x)   ✓ ✗ static error
  else fib(-x)  ✓ ✗ static error
```

Gradual Refinement Types

```
type Nat = {ν:Int | ν ≥ 0}

def fib(x: Nat): Nat

def isNat(x: Int):{ν: Bool | ν = true ⇒ x ≥ 0 ∧ ?}

def bar(x: Int): Int
  if isNat(x)
  then fib(x)  ✓ + dynamic check
  else fib(-x) ✓ + dynamic check
```


Gradual Refinement Types

```
type Nat = {ν:Int | ν ≥ 0}

def fib(x: Nat): Nat

def isNat(x: Int):{ν: Bool | ν = true ⇒ x ≥ 0 ∧ ?}

def bar(x: Int): Int
  if isNat(x)
  then fib(-x) ✗ static error
  else fib(x)  ✓ + dynamic check
```


Gradual Parametricity

[POPL'19, JACM'22, OOPSLA'22]

Parametricity, Intuitively

Polymorphic types dictate **uniformity of behavior**

[Reynolds83]

```
let f : ∀X.X→X =
  …
in f [Int] 10
```

f ≃ id

should get back 10

strong **type-based** reasoning principle

"free theorems" [Wadler89]

Gradual Parametricity, Intuitively

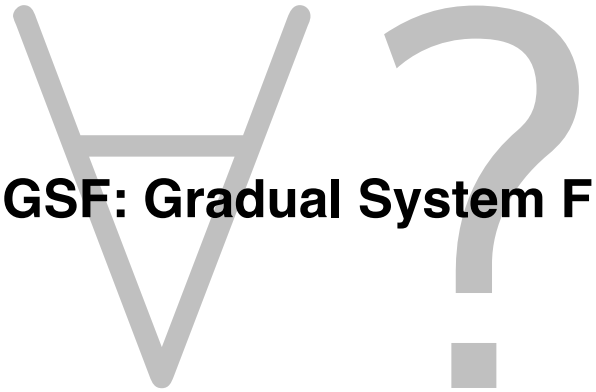
```
let g : ? = λa.λb.if b then a else a+1
in
let f : ∀X.X→X =
    ΛX.λx:X.g x false

in f [Int] 10 ⟼* error ✓
```

"half parametric"

is f really parametric?

tracking type safety at runtime is not enough

need some form of runtime **sealing** [Morris73]



GSF: Gradual System F

GSF in Action

			GSF
conservative extension of System F			
∀X.X→X ∼ Int→Bool	✓	X	X
∀X.X→?	✓	X	✓
natural precision & consistency			
`let f : ? = ΛX.λx:X.x in f [Int] true`	true	true	error
respect explicit type instantiations			
`let f : ∀X.X→? = ΛX.λx:X.x in (f [Int] 1) + 1`	error	error	2
type-driven sealing			

GSF: Type-Driven Sealing

```
let f : ∀X.X→? = ΛX.λx:X.x in …
                 X → X
```

→ justifies both sealing and unsealing

```
let f : ∀X.X→? = ΛX.λx:X.(x::?) in …
                 X → ?
```

→ only justifies sealing

```
let p = ΛX.<λx:X.(x::?),λx:?.(x::X)> in …
          X → ?      ? → X
```

→ pair of seal and unseal functions à la λ_{seal}!

[Sumii&Pierce'04]

Dynamic Sealing in **GSF**

$$\text{gen-seal} = \Lambda X.\, \langle \lambda x{:}X.(x :: ?), \lambda x{:}?.(x :: X) \rangle$$

let $\langle s, u \rangle$ = gen-seal [Int] in
let $v{:}? = s(10)$ in
let $r{:}? = \ldots(v)\ldots$ in
let $w{:}\texttt{Int} = u(r)$

value is sealed

any use of v will fail

unsealing succeeds only for values sealed with *s*

Gradual Security Types

[TOPLAS'18]

Security Typing

```
let age = 31_L
let salary = 58000_H
let intToString : Int_α → String_α = ...
let print : String_L → Unit_L = ...
print(intToString(salary))
```

✖ **static error**

private salary goes to public channel

Gradual Security Typing

```
let age = 31?
let salary = 58000H
let intToString : Int? → String? = ...
let print : StringL → Unit? = ...
print(intToString(salary))
```

✓ + dynamic check (runtime error)

Security Types & Free Theorems

```
let mix : IntL → IntH → IntL =
  fun pub priv => ...
```

theorem
result does not leak 2nd argument

```
let foo : (IntL → IntH → IntL) → BoolH =
  fun f => ... f x y ...
```

can assume theorem is **not** violated

Gradual Security Typing, with Theorems

```
let mix : IntL → IntH → IntL =
  fun pub priv => if pub < priv then 1L else 2L
```

✗ static error

```
let mix : IntL → Int? → IntL =
  fun pub priv => if pub < priv then 1L else 2L
```

✓

```
mix 1L 2H
```

✗ runtime error

```
let mix' : IntL → IntH → IntL =
  fun pub priv => mix pub priv
```

the types tell the theorems!

```
mix' 1L 2L
```

✗ runtime error

Ranges of Precision

fully precise ⟵ ⟶ fully imprecise

gradual effects

$$A \xrightarrow{\{\text{io,alloc}\}} B \sqsubseteq A \xrightarrow{\{\text{io,?}\}} B \sqsubseteq A \xrightarrow{\{?\}} B$$

gradual refinements

$$\{\,\texttt{Int} \mid 0 < \nu < 10\,\} \sqsubseteq \{\,\texttt{Int} \mid 0 < \nu \wedge ?\,\} \sqsubseteq \{\,\texttt{Int} \mid ?\,\}$$

gradual security

$$\texttt{Int}_\texttt{H} \sqsubseteq \texttt{Int}_?$$

Many different soundness properties

- Gradual **typestate**: *respect resource protocols*
- Gradual **effects**: *no unauthorized effectful operations*
- Gradual **refinement types**: *result satisfies predicate*
- Gradual **parametricity**: *relational parametricity*
- Gradual **security types**: *noninterference*
- Gradual **sensitivity types**: *metric preservation*

it's not all about type safety!

Gradual Typing

Designed

can't we define runtime semantics directly?

where come

what's the connection to the static language?

high cost
renegotiation of foundations
ingenious "tricks"
ad hoc justifications

gradual langu… …ulus

TYPE SYSTEM | RUNTIME SEMANTIC… | …UNTIME …MANTICS

how to deal with imprecision?

"right" definitions?

… gradual guarantees?

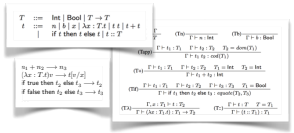
eg. equality, subtyping, containment, implication, etc.

Designing Gradual Languages

without the guesswork

Abstracting Gradual Typing

[POPL'16]



static type system & type safety proof

$$\mathrm{GType} \underset{\alpha}{\overset{\gamma}{\rightleftarrows}} \mathcal{P}(\mathrm{Type})$$

syntax & interpretation of gradual types

AGT

systematic, **not** automatic

gradual language

TYPE SYSTEM | RUNTIME SEMANTICS

by construction

AGT

I - Static Semantics

$$(T+)\ \frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad T_1 = \mathsf{Int} \quad T_2 = \mathsf{Int}}{\Gamma \vdash t_1 + t_2 : \mathsf{Int}}$$

explicit side conditions

$$(T\mathrm{app})\ \frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad T_2 = dom(T_1)}{\Gamma \vdash t_1\ t_2 : cod(T_1)}$$

AGT

syntax-directed rules

partial functions

$$(T\mathrm{if})\ \frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad \Gamma \vdash t_3 : T_3 \quad T_1 = \mathsf{Bool}}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : equate(T_2, T_3)}$$

$cod : \mathrm{Type} \rightharpoonup \mathrm{Type}$
$cod(T_1 \to T_2) = T_2$
$cod(T)$ undefined otherwise

[Garcia & Cimini, 2015]

$$(\widetilde{T}+)\ \frac{\Gamma \vdash \widetilde{t}_1 : \widetilde{T}_1 \quad \Gamma \vdash \widetilde{t}_2 : \widetilde{T}_2 \quad \widetilde{T}_1 \sim \mathsf{Int} \quad \widetilde{T}_2 \sim \mathsf{Int}}{\Gamma \vdash \widetilde{t}_1 + \widetilde{t}_2 : \mathsf{Int}}$$

consistent side conditions

$$(\widetilde{T}\mathrm{app})\ \frac{\Gamma \vdash \widetilde{t}_1 : \widetilde{T}_1 \quad \Gamma \vdash \widetilde{t}_2 : \widetilde{T}_2 \quad \widetilde{T}_2 \sim \widetilde{dom}(\widetilde{T}_1)}{\Gamma \vdash \widetilde{t}_1\ \widetilde{t}_2 : \widetilde{cod}(\widetilde{T}_1)}$$

compositional lifting

lifted partial functions

$$(\widetilde{T}\mathrm{if})\ \frac{\Gamma \vdash \widetilde{t}_1 : \widetilde{T}_1 \quad \Gamma \vdash \widetilde{t}_2 : \widetilde{T}_2 \quad \Gamma \vdash \widetilde{t}_3 : \widetilde{T}_3 \quad \widetilde{T}_1 \sim \mathsf{Bool}}{\Gamma \vdash \mathsf{if}\ \widetilde{t}_1\ \mathsf{then}\ \widetilde{t}_2\ \mathsf{else}\ \widetilde{t}_3 : \widetilde{T}_2 \sqcap \widetilde{T}_3}$$

gradual meet

we now need to define and justify all of this!

Syntax of Gradual Types

static types $\textsc{Type}$

$$T ::= \mathsf{Int} \mid \mathsf{Bool} \mid T \to T$$

"represents"

gradual types $\textsc{GType}$

$$\widetilde{T} ::= \mathsf{Int} \mid \mathsf{Bool} \mid \widetilde{T} \to \widetilde{T} \mid ?$$

Concretization

$$\gamma : \textsc{GType} \to \mathcal{P}(\textsc{Type})$$

$$\gamma(\mathsf{Int}) = \{\, \mathsf{Int} \,\}$$
$$\gamma(\mathsf{Bool}) = \{\, \mathsf{Bool} \,\}$$
$$\gamma(\widetilde{T}_1 \to \widetilde{T}_2) = \{\, T_1 \to T_2 \mid T_1 \in \gamma(\widetilde{T}_1), T_2 \in \gamma(\widetilde{T}_2) \,\}$$
$$\gamma(?) = \textsc{Type}$$

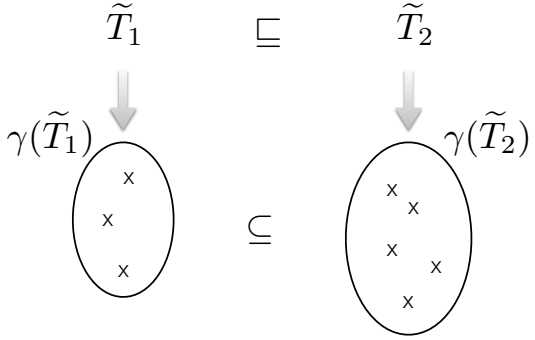
e.g.

$$\gamma(\mathsf{Int} \to ?) = \{\, \mathsf{Int} \to T \mid T \in \textsc{Type} \,\}$$

Type Precision

$$\mathsf{Int} \to \mathsf{Int} \;\sqsubseteq\; \mathsf{Int} \to ? \;\sqsubseteq\; ? \to ? \;\sqsubseteq\; ?$$

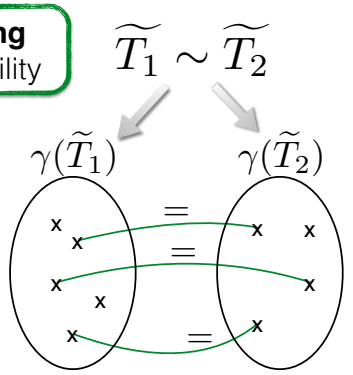
directly induced by concretization



Consistency

lifting equality

existential lifting
captures plausibility



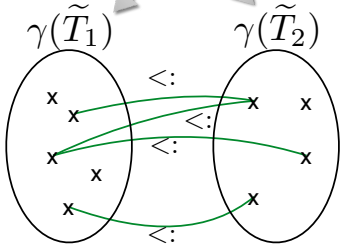
not transitive!

coincides with [Siek & Taha, 2006]

Consistent Subtyping

lifting subtyping

$$\widetilde{T_1} \lesssim \widetilde{T_2}$$



coincides with [Siek & Taha, 2007]

Consistent Join

NEW

Galois connection (sound & optimal)

$\widetilde{T_1} \;\widetilde{\vee}\; \widetilde{T_2}$

γ γ

$\widetilde{\vee}$

α

$\widetilde{T}$

$\alpha(\{\,\text{Int}\,\}) = \text{Int}$

$\alpha(\{\,\text{Bool}\,\}) = \text{Bool}$

$\alpha(\{\,\overline{T_{i1} \to T_{i2}}\,\}) = \alpha(\{\,\overline{T_{i1}}\,\}) \to \alpha(\{\,\overline{T_{i2}}\,\})$

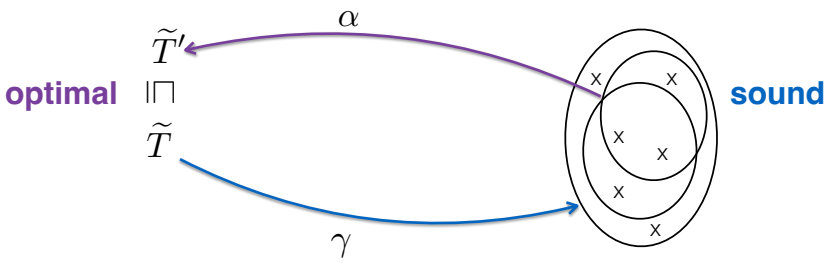
$\alpha(\emptyset)$ is undefined

$\alpha(\widehat{T}) = ?$ otherwise

Reminder

Galois Connection

abstract domain GTYPE $\sqsubseteq$ $\quad\langle\alpha,\gamma\rangle\quad$ concrete domain $\mathcal{P}(\text{TYPE})$ $\subseteq$



AGT

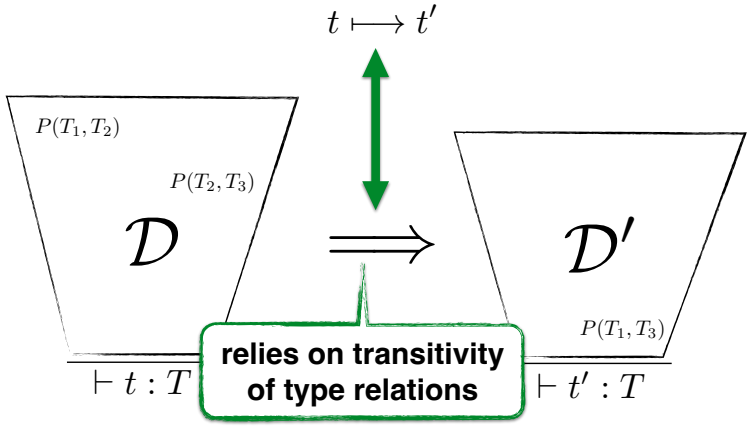
II - Dynamic Semantics

Curry-Howard

Logic	**PL**
Propositions	Types
Proofs	Programs
Proof reduction	Program evaluation

Reminder

Type Safety as Proof Reduction



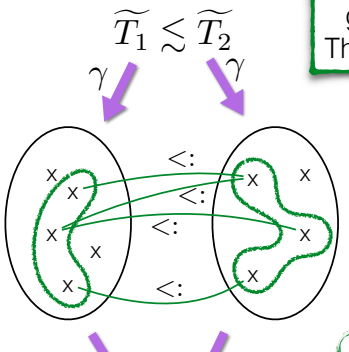
$$P(T_1, T_2) \wedge P(T_2, T_3) \Rightarrow P(T_1, T_3)$$

Evidence

generalization of the Threesome middle type

[Siek & Wadler, POPL'10]

holds... but why?

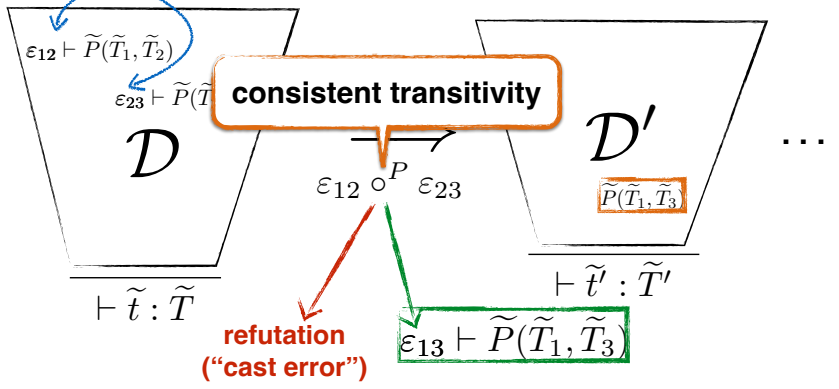


local justification (sound & optimal)

$$? \to \texttt{Bool} \lesssim \texttt{Int} \to ?$$

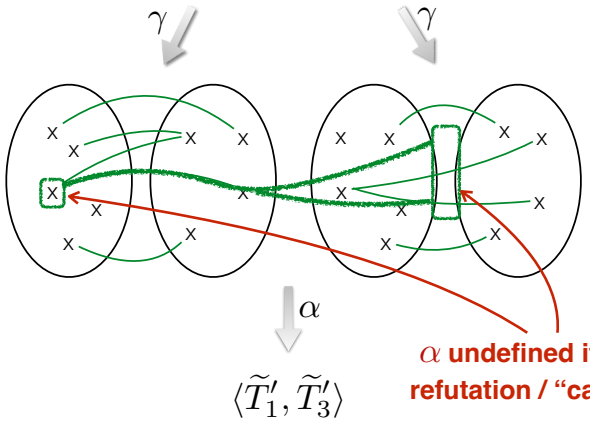
Reduction: Combining Evidence

typing derivations with evidence



Consistent Transitivity

$\langle \widetilde{T}_1, \widetilde{T}_{21} \rangle \circ^P \langle \widetilde{T}_{22}, \widetilde{T}_3 \rangle$



$\alpha^2(\{\langle T_1, T_3 \rangle \in \gamma^2(\widetilde{T}_1, \widetilde{T}_3) \mid \exists T_2 \in \gamma(\widetilde{T}_{21}) \cap \gamma(\widetilde{T}_{22}).\, P(T_1, T_2) \wedge P(T_2, T_3)\})$

$$\frac{\dfrac{\dfrac{x{:}? \vdash x : ? \quad x{:}? \vdash 1 : \mathrm{Int}}{\langle \mathrm{Int} \rangle \vdash ? \sim \mathrm{Int}}}{x{:}? \vdash x + 1 : \mathrm{Int}} \qquad \dfrac{\vdash \mathrm{false} : \mathrm{Bool}}{\langle \mathrm{Bool} \rangle \vdash \mathrm{Bool} \sim ?}}{\vdash (\lambda\, x{:}?.\ x + 1)\ \mathrm{false} : \mathrm{Int}}$$

$$\Longrightarrow \quad \frac{\vdash \mathrm{false} : \mathrm{Bool} \quad \vdash 1 : \mathrm{Int}}{??? \vdash \mathrm{Bool} \sim \mathrm{Int}} \qquad \frac{}{\langle \mathrm{Bool} \rangle \circ \langle \mathrm{Int} \rangle \vdash \mathrm{Bool} \sim \mathrm{Int}}$$

$\vdash \mathrm{false} + 1 : \mathrm{Int}$

undefined

⇒ runtime error

AGT

Gredex: AGT in Action

https://pleiad.cl/gredex

Designing Gradual Languages

$$\mathrm{GType} \underset{\alpha}{\overset{\gamma}{\rightleftarrows}} \mathcal{P}(\mathrm{Type})$$

- **Galois connection**
  - defining γ is the **central design decision**
  - α is **uniquely determined** by γ ("just" find it!)
- given the Galois connection, **lifting the statics is direct**
- Galois connection also **central in the dynamics**

Designing Gradual Languages

$$\text{GType} \underset{\alpha}{\overset{\gamma}{\rightleftarrows}} \mathcal{P}(\text{Type})$$

- AGT also **pays off for the runtime semantics**
  - justifies runtime errors, threesomes
  - dynamic gradual guarantee “for free” (monotonicity of consistent transitivity)
- Direct evidence-based semantics: **canonical**
  - not a cast calculus
  - can prove translation+cc equivalent [SAS’17]

Perspectives

- Dynamics driven by **type safety** argument
  - can involve more operators (eg. substitution [POPL’17])
  - ensures type safety (+ gradual guarantee)
- **type soundness ≠ type safety**
  - eg. parametricity, noninterference, metric preservation, etc.
  - can need a **more precise GC** for dynamics than for statics
  - tension with gradual [illegible]3, P

semantic property enforcement

programming flexibility

Applications of AGT (so far…)

- records with subtyping
- gradual rows

POPL’16

- effect typing

ICFP’14 / JFP’16 (statics)

- refinement types

POPL’17

- set-theoretic types

ICFP’17 (statics)

- union types

SAS’17

- security typing

TOPLAS’18

- parametricity

POPL’19, JACM’22, OOPSLA’22

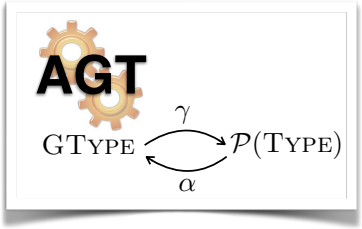
- Hoare-style verification

VMCAI’18, OOPSLA’20

- dependent types

ICFP’19, ICFP’22

need not be mostly guesswork & intuition



focus on key issues
streamline what can be

Galois connection(s)
algorithmic definitions

optimizations

semantic properties

richer types

Gradual Typing

Implemented

A First Warning

Space-Efficient Gradual Typing

David Herman[1], Aaron Tomb[2], and Cormac Flanagan[2]

[1] Northeastern University
[2] University of California, Santa Cruz

Abstract

Gradual type systems offer a smooth continuum between static and dynamic typing by permitting the free mixture of typed and untyped code. The runtime systems for these languages–and other languages with hybrid type checking–typically enforce function types by dynamically generating function proxies. This approach can result in unbounded growth in the number of proxies, however, which drastically impacts space efficiency and destroys tail recursion.
We present an implementation strategy for gradual typing that is based on *coercions* instead of function proxies, and which combines adjacent coercions to limit their space consumption. We prove bounds on the space consumed by coercions as well as sound-

[TFP'07]

use coercions instead of casts
(avoid higher-order proxies)

- GT breaks tail call optimizations

$$even = \lambda n{:}\texttt{Int}.\ \lambda k{:}(? \to ?).\quad \text{if } (n = 0) \text{ then } (k\ \texttt{true}) \text{ else } odd\ (n-1)\ k$$
$$odd = \lambda n{:}\texttt{Int}.\ \lambda k{:}(\texttt{Bool} \to \texttt{Bool}).\quad \text{if } (n = 0) \text{ then } (k\ \texttt{false}) \text{ else } even\ (n-1)\ k$$

Here, the recursive calls to *odd* and *even* quietly cast the continuation argument k with higher-order casts $\langle \texttt{Bool} \to \texttt{Bool} \rangle$ and $\langle ? \to ? \rangle$, respectively. This means that the function argument k is wrapped in an additional function proxy at each recursive call!

A Second (Hard) Warning

Is Sound Gradual Typing Dead?

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, Matthias Felleisen
Northeastern University, Boston, MA

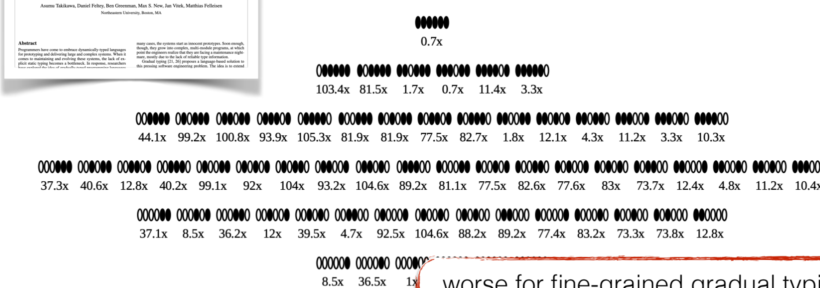
[POPL'16] Racket

- observed slowdowns of up to 105x 😭
- (Typed Racket is implemented with contracts)

Benchmarking Gradual Typing

Is Sound Gradual Typing Dead?

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



worse for fine-grained gradual typing
(n is # of type annotations)

For n modules there are 2^n possible configurations

⇒ only measure a linear sample of configurations

An Extensive Research Agenda

Is Sound Gradual Typing Dead?

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

Toward Efficient Gradual Typing for Structural Types via Coercions

Andre Kuhlenschmidt Deyaaeldeen Almahallawi Jeremy G. Siek

Sound Gradual Typing is Nominally Alive and Well

FABIAN MUEHLBOECK, Cornell University, USA

Collapsible Contracts: Fixing a Pathology of Gradual Typing

DANIEL FELTEY, Northwestern University, USA
BEN GREENMAN, Northeastern University, USA
CHRISTOPHE SCHOLLIERS, Ghent University, Belgium
ROBERT BRUCE FINDLER, Northwestern University, USA

Transitioning from Structural to Nominal Code with Efficient Gradual Typing

FABIAN MUEHLBOECK, IST Austria, Austria
ROSS TATE, Cornell University, United States of America

GTP Benchmarks for Gradual Typing Performance

Ben Greenman
Brown University
Providence, Rhode Island, USA
benjaminlgreenman@gmail.com

ABSTRACT

The Easy Way Out

- If targeting a dynamic language, just fall back to optional typing (or lightweight checks)

TypeScript Python Racket Dart

- Not an option if target language is static
- Not an option for advanced typing disciplines

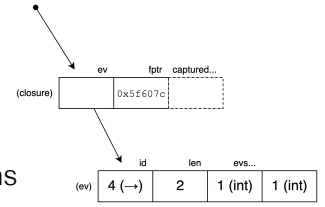
Implementation approaches

- casts
- coercions
- contracts
- … evidence? [on-going]

A Compiler for Evidence-Based Gradual Typing

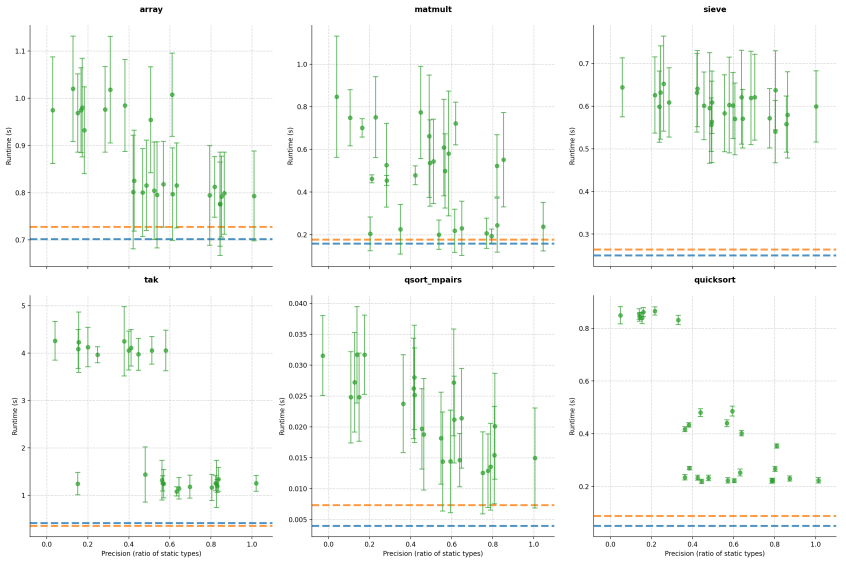
- int and bools are unboxed and tagged
- boxed values point to heap blocks
- evidence as a tree of tags
- initial ascriptions (opt: remove safe ones)
- consistent transitivity to reduce ascriptions

Value	Block
Tuple	[ev ; len ; vals ...]
Closure	[ev ; fptr ; captured vals ...]
Vec	[ev ; vals ...]
Variant	[ev ; id ; vals ...]

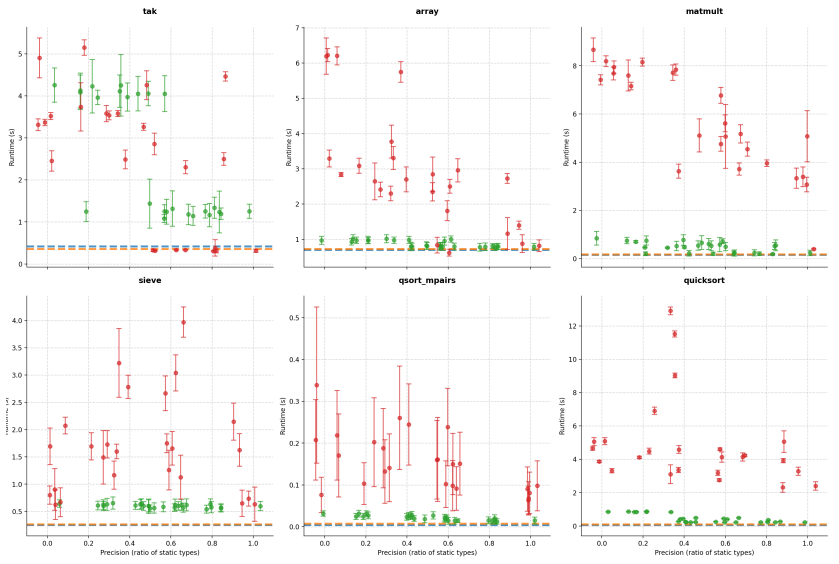


use the benchmarks of Grift [PLDI'19]

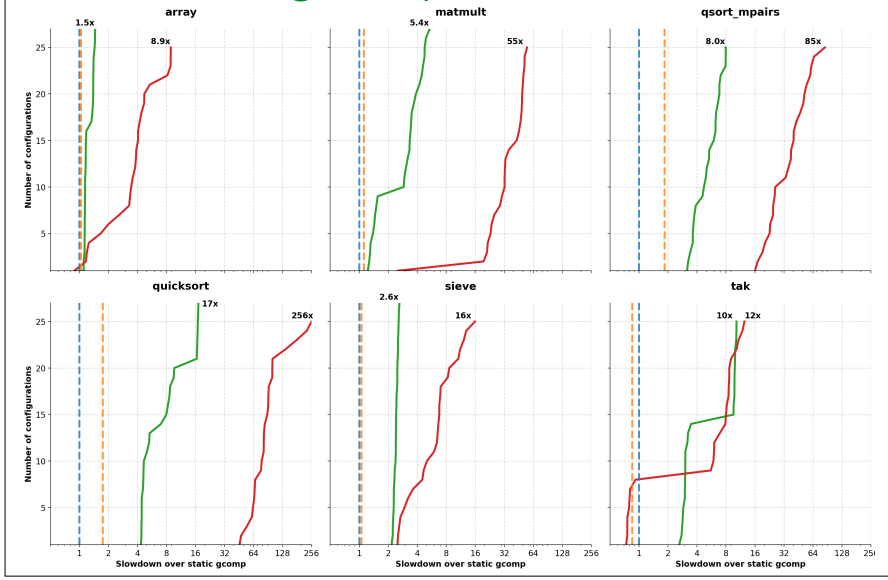
gradual vs. static vs. dynamic



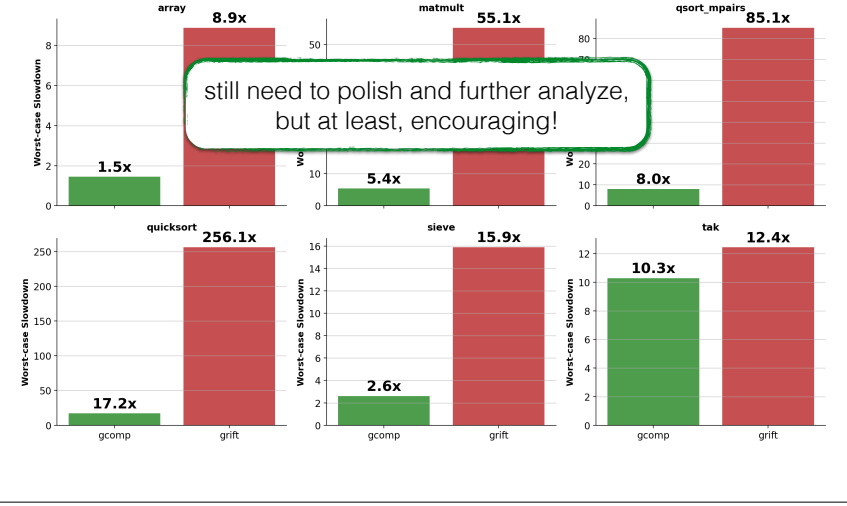
gcomp vs Grift



gcomp vs Grift



gcomp vs Grift



Implementation & performance

Tyger

- evidence-based gradual typing framework
- functional on a subset of Python, with IDE support
- early stage, on-going work
- highlights new issues (esp. inheritance)
- no performance evaluation yet

Conclusions

Gradual Typing

precision-driven type checking

Metatheory & criteria

Foundational methodologies

Languages features

Advanced typing disciplines

Implementation & performance

… and Beyond!

precision-driven reasoning

core concepts are transferrable to other reasoning frameworks

- Hoare-style program verification [VMCAI'18, OOPSLA'20, TOPLAS'24]
- Program analysis:
  nulls [ECOOP'21], abstract interpretation [draft@IFL'25]

🙏 collaborators 🙏

Ron Garcia
Jonathan Aldrich
Esteban Allende
Johan Fabry
Oscar Callaú
Felipe Bañados
Nicolas Tabareau
Matías Toro
Nico Lehmann
Niki Vazou
Elizabeth Labrada
Joey Eremondi
Meven Lennon-Bertrand
Kenji Maillard
Johannes Bader
Jenna (Wise) DiVincenzo
Damián Árquez
Federico Olmedo
Mara (Stefan) Malewski
José Luis Romero
Cristobal Ardiles
and more…

Metatheory & criteria

Foundational methodologies

Languages features

Advanced typing disciplines

Implementation & performance

www.dcc.uchile.cl

Pleiad

pleiad.cl

contact me if interested in MSc / PhD / postdoc