The Essence of Gradual Typing

Éric Tanter
University of Chile

joint work with
Ron Garcia, Felipe Bañados, Alison Clark,
Nico Lehmann, Matías Toro, Elizabeth Labrada

Gradual Typing

The 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 recent languages try to have both

C# 4.0
Typed Clojure
Dart
ActionScript
TypeScript
Typed Racket
Hack
Perl 6
Scala

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 checking disciplines 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

not transitive!

int ~ ? ? ~ bool

int ≁ bool

type consistency

T ~ T

T ~ ? ? ~ T

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

```
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)
```

check it is a func | tagged value

```
h(<?⇐int→int>f)
⟶ (<?→?⇐?><?⇐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
```

Properties of Gradual Languages

[Siek & Taha, 2006]

type safety — admits runtime type errors

equivalence for static terms — conservative extension

embedding of dynamic terms — expressive

The End

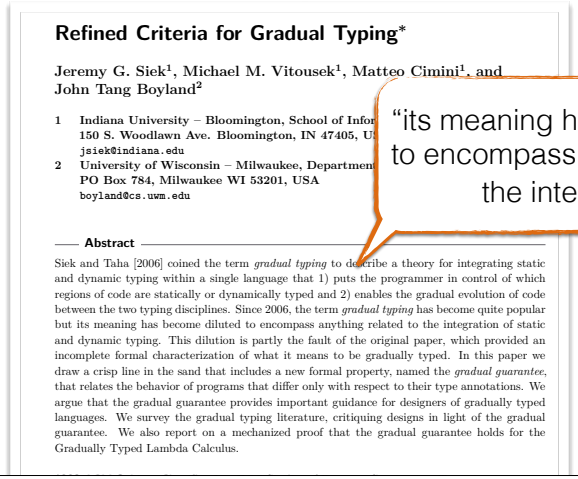
?

Gradual Typing

Refined

What do you mean “Gradual”?

[Siek *et al.*, 2015]



Refined Criteria for Gradual Typing*

Jeremy G. Siek[1], Michael M. Vitousek[1], Matteo Cimini[1], and John Tang Boyland[2]

1 Indiana University – Bloomington, School of Infor
150 S. Woodlawn Ave. Bloomington, IN 47405, U
jsiek@indiana.edu
2 University of Wisconsin – Milwaukee, Departmen
PO Box 784, Milwaukee WI 53201, USA
boyland@cs.uwm.edu

Abstract

Siek and Taha [2006] coined the term *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.

“its meaning has become **diluted** to encompass anything related to the integration […]”

Gradual Typing, refined

[Siek *et al.*, 2015]

it’s all about **precision**

some gradual types convey more information than others

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

Gradual Typing
best-effort static checking
backed by dynamic checking

Precision

type precision extends to **term precision**

$$(\lambda y : ?.y)\ ((\lambda x : ?.x)\ 42)$$
$$\sqsubseteq$$
$$(\lambda y : \mathtt{Int}.y)\ ((\lambda x : ?.x)\ 42)$$
$$\sqsubseteq$$
$$(\lambda y : \mathtt{Int}.y)\ ((\lambda x : \mathtt{Int}.x)\ 42)$$

no explicit checks
evolution is completely driven by type annotations

Properties of Gradual Languages (ctd)

[Siek *et al*, 2015]

Gradual Guarantee

$$(\lambda y : ?.y)\ ((\lambda x : ?.x)\ 42)$$
$$\sqsubseteq$$
$$(\lambda y : \mathtt{Int}.y)\ ((\lambda x : ?.x)\ 42)$$
$$\sqsubseteq$$
$$(\lambda y : \mathtt{Int}.y)\ ((\lambda x : \mathtt{Int}.x)\ 42)$$

losing precision

1) preserves typing

2) preserves reduction

adding type information can only introduce new static/dynamic errors

Gradual Typing

Extended

Beyond Simple Gradual Typing

- **Subtyping** (structural, nominal, objects)
- **Parametric polymorphism**
- **Type inference** and gradual types
- Union and recursive types
- etc.

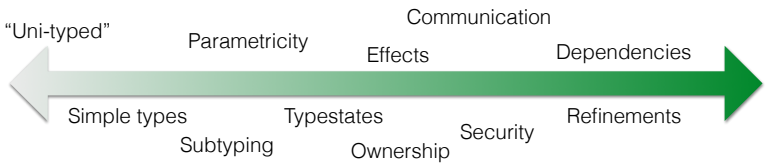
[Siek&Taha'07, Ina&Igarashi'11]
[Ahmed et al 08/11/17, Igarashi'17]
[Siek&Vachharajani'08, Garcia&Cimini'15]
[Siek&Tobin-Hochstadt'16]

Gradual Typing

=

~~reconciling static and dynamic typing~~

combining **type disciplines of different strength**



Advanced Gradual Types

let's look at some examples

- Gradual **effects** [ICFP'14, OOPSLA'15, JFP'16]
- Gradual **refinement types** [POPL'17]
- Gradual **security types** [TOPLAS in progress]

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]

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

Gradual Security Types

[TOPLAS in progress]

Security Typing

```
let age = 31_L
let salary = 58000_H
let intToString : Int_a → String_a = ...
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 = 58000_H
let intToString : Int_? → String_? = ...
let print : String_L → Unit_? = ...
print(intToString(salary))
```

✓ **+ dynamic check (runtime error)**

Security Types & Free Theorems

```
let mix : Int_L → Int_H → Int_L =
  fun pub priv => …
```

theorem
result does not leak 2nd argument

```
let foo : (Int_L → Int_H → Int_L) → Bool_H =
  fun f => … f x y …
```

can assume theorem is **not** violated

Gradual Security Typing, with Theorems

```
let mix : Int_L → Int_H → Int_L =
  fun pub priv => if pub < priv then 1_L else 2_L
```
✗ static error

```
let mix : Int_L → Int ? → Int_L =
  fun pub priv => if pub < priv then 1_L else 2_L
```
✓

```
mix 1_L 2_H
```
✗ runtime error

```
let mix' : Int_L → Int_H → Int_L =
  fun pub priv => mix pub priv
```

the types tell the theorems!

```
mix' 1_L 2_L
```
✗ runtime error

Properties of Gradual Languages

type safety

equivalence for ~~static~~ terms — fully-precise

embedding of ~~dynamic~~ terms — fully-imprecise

losing precision preserves typing

losing precision preserves reduction

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

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

Gradual Typing

- More than static & dynamic typing
- **Precision-driven** type checking
- Applicable to **wide range** of typing disciplines
- Important to be clear about the **guarantees**

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

How to Design Gradual Languages?

can't we define runtime semantics directly?

where come

what's the connection to the static language?

gradual language

TYPE SYSTEM | RUNTIME SEMANTICS

translation

cast calculus

TYPE SYSTEM' | RUNTIME SEMANTICS

how to deal with imprecision?

what are the "right" definitions?

… gradual guarantee?

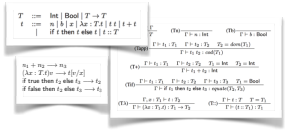
eg. equality, subtyping, containment, implication, etc.

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

$\mathrm{GTYPE} \underset{\alpha}{\overset{\gamma}{\rightleftarrows}} \mathcal{P}(\mathrm{TYPE})$

syntax & interpretation of gradual types

Syntax of Gradual Types

static types TYPE

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

"represents"

gradual types GTYPE

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

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

"represents"

Int	→	Int
$\mathsf{Int} \to ?$	→	$\mathsf{Int} \to \mathsf{Int}$ $\mathsf{Int} \to \mathsf{Bool}$ $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$ $\mathsf{Int} \to \mathsf{Bool} \to \mathsf{Int}$ $\mathsf{Int} \to \ldots$
$?$	→	any static type!

Concretization

$$\gamma : \mathrm{GTYPE} \to \mathcal{P}(\mathrm{TYPE})$$

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

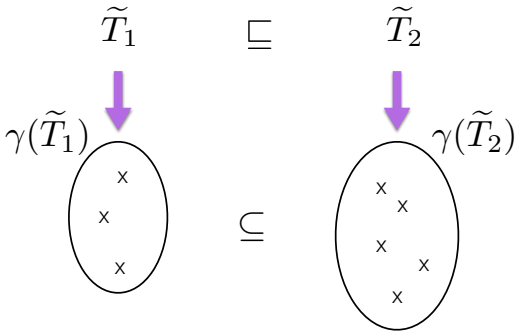
e.g.

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

Type Precision

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

directly induced by concretization

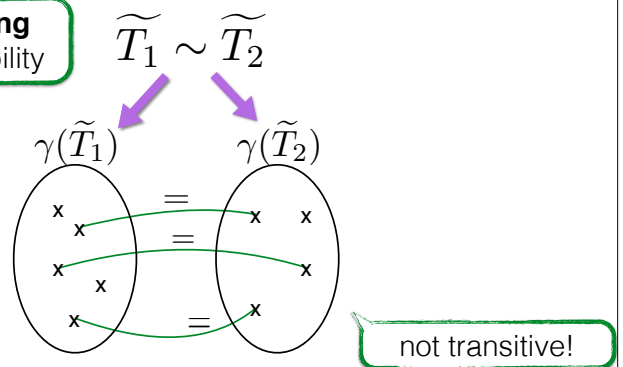


I - Static Semantics

Consistency

lifting equality

existential lifting
captures plausibility

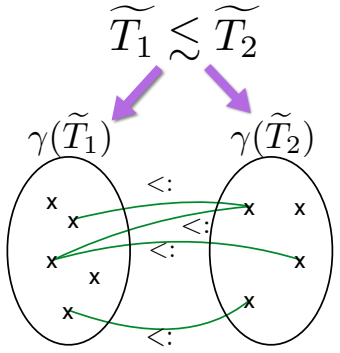


not transitive!

coincides with [Siek & Taha, 2006]

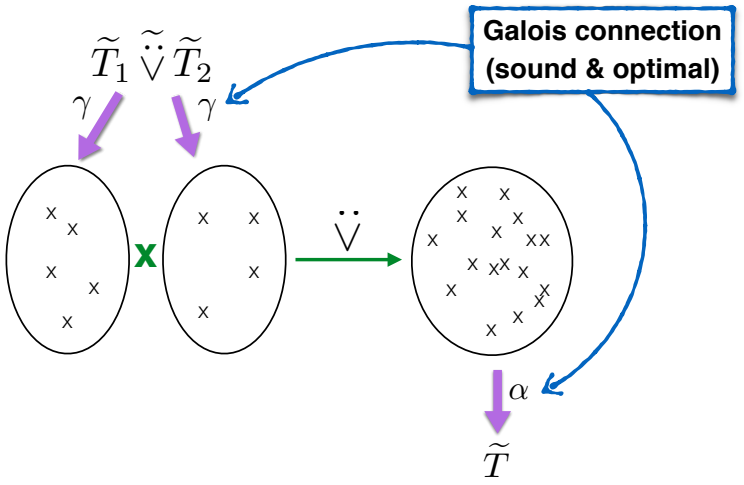
Consistent Subtyping

lifting subtyping

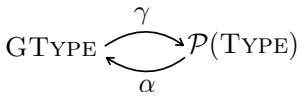


coincides with [Siek & Taha, 2007]

Consistent Join



Soundness and Optimality



why does it matter?

- can break **conservative extension** (both ways)
- can break **safety** (stuck)
- can obtain "**forgetful semantics**" [Greenberg] (inadequate for type-based reasoning)

Lifting *equate*

$$(\text{Tif})\ \frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad \Gamma \vdash t_3 : T_3 \qquad T_1 = \text{Bool}}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \mathit{equate}(T_2, T_3)}$$

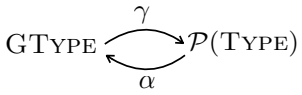
"It was interesting to see how it justifies using meet for conditional expressions… before that I had always thought that I was making an arbitrary choice to prefer meet over join."

- J. Siek

$$\widetilde{\mathit{equate}}(\widetilde{T}_1, \widetilde{T}_2) = \widetilde{T}_1 \sqcap \widetilde{T}_2$$

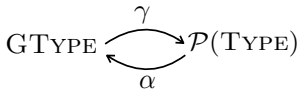
$$\widetilde{T}_1 \sqcap \widetilde{T}_2 = \alpha(\gamma(\widetilde{T}_1) \cap \gamma(\widetilde{T}_2))$$

Designing Gradual Languages



- **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



- **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**

let's see some examples

Defining Gradual Types

[ICFP'14, JFP'16]

$$\text{read}, \text{write}, \text{alloc} \in \text{Priv} \qquad \phi \in \text{PrivSet}$$
$$\text{read}, \text{write}, \text{alloc}, ? \in \text{CPriv} \qquad \Xi \in \text{CPrivSet}$$

$$\gamma : \text{CPrivSet} \to \mathcal{P}(\text{PrivSet})$$
$$\gamma(\{\,\text{read}\,\}) = \{\,\{\,\text{read}\,\}\,\}$$
$$\gamma(\{\,\text{read}, ?\,\}) = \{\,\{\,\text{read}\,\} \cup \phi \mid \phi \in \text{PrivSet}\,\}$$

naturally induces the definition of
consistent containment

```
def f(x: Int): Int = …
def run(callback: Int ⟶{} Int) = …
```

{¿} ⊆ {}

```
run(f)
```


Defining Gradual Types [POPL'17]

is not always trivial!

$$\widetilde{T} ::= \{\nu : B \mid \widetilde{p}\} \mid x{:}\widetilde{T} \to \widetilde{T}$$
$$\widetilde{p} ::= p \mid p \wedge ?$$

$\gamma(p \wedge ?) = \{\, p \wedge q \mid q \in \text{FORMULA} \,\}$ — can pick contradiction

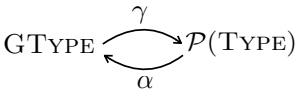
$\gamma(p \wedge ?) = \{\, p \wedge q \mid \text{SAT}(p) \implies \text{SAT}(p \wedge q) \,\}$ — purely syntactic (precision)

$\gamma(p \wedge ?) = \{\, q \mid \{\, q \,\} \models p \,\}$ — admits contextual contradiction

$\gamma(p \wedge ?) = \{\, q \mid \{\, q \,\} \models p \text{ and } \text{local}(q) \,\}$ ✓

note: any definition would yield a "valid" gradual language

Designing Gradual Languages

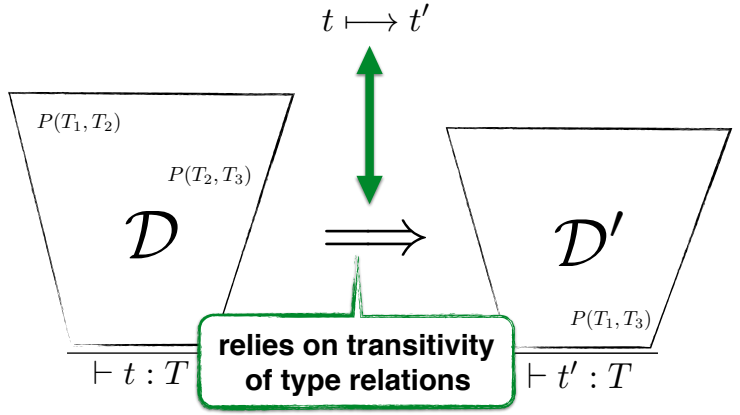


- **Galois connection**
  - defining γ is the **central design decision**
  - α is **uniquely determined** by γ ("just" find it!)
- given the Galois connection, **lifting the statics is direct**
- Galois connection also **central in the dynamics**

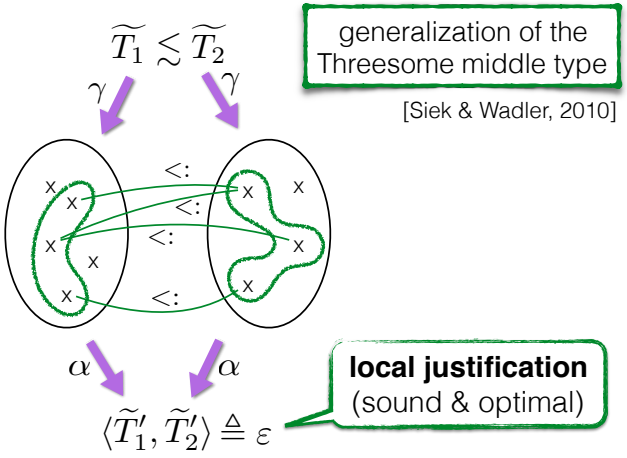
II - Dynamic Semantics

Reminder

Type Safety as Proof Reduction



Evidence



generalization of the Threesome middle type

[Siek & Wadler, 2010]

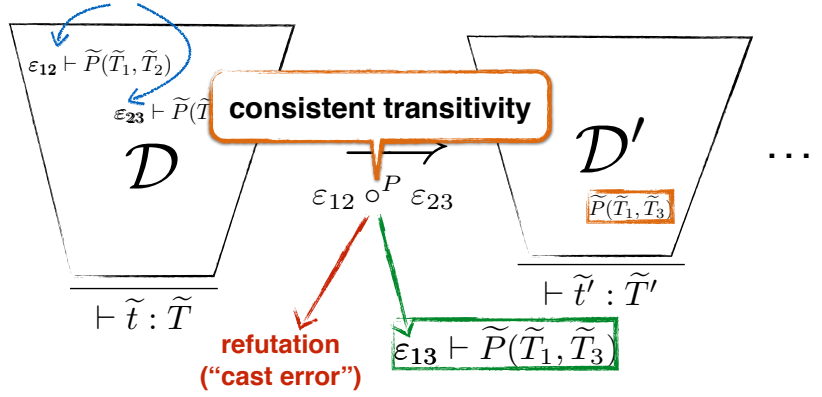
holds... but why?

local justification (sound & optimal)

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

Reduction: Combining Evidence

typing derivations with evidence



Example

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

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

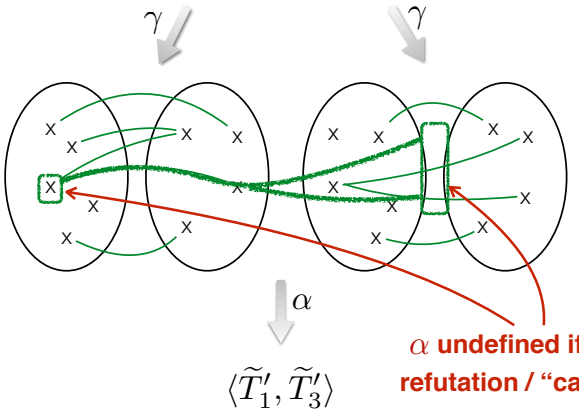
$$\vdash \mathsf{false} + 1 : \mathsf{Int}$$

undefined

⇒ runtime error

Consistent Transitivity

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



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

Recent Developments & Perspectives

- Dynamics driven by **type safety** argument
  - can involve more operators (eg. dependencies [POPL'17])
  - ensures type safety (+ gradual guarantee)
- **type soundness ≠ type safety**
  - eg. parametricity, noninterference
  - **security typing** with references:
    needs a **more precise GC** for dynamics than for statics
  - tension with gradual

semantic property enforcement

programming flexibility

AGT

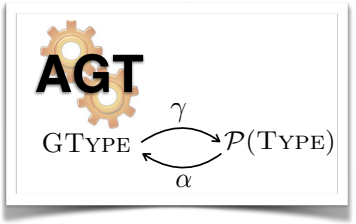
Conclusions

Gradual Typing

- **Precision-driven** type checking
- Applicable to **wide range** of typing disciplines
- Needs **solid** foundations
- Full of **open** challenges, very **active** area

check recent and upcoming
POPL, PLDI, ICFP, OOPSLA, ECOOP, etc.

need not be mostly
guesswork & intuition



focus on key issues
streamline what can be

Galois connection(s)
algorithmic definitions

optimizations

semantic
properties

richer types