Abstracting Gradual Typing

Ronald Garcia Alison Clark
University of British Columbia

Éric Tanter
University of Chile

Gradual Typing

In a Nutshell

“traditional way”

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

dynamic 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
```

$\texttt{int} \nsim \texttt{bool}$

type consistency

$T \sim T$

$T \sim ? \qquad ? \sim T$

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

```
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 it is an int

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

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

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

body is safe!
can be compiled efficiently

check it is a func

tagged value

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

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)  ⟶  cast error
```

The End

?

Beyond Simple Gradual Typing

- **Subtyping** (structural, nominal, objects)
- **Parametric polymorphism** ("blame for all")
- **Type inference** and gradual types
- **Union and recursive types**

[Siek&Taha'07, Ina&Igarashi'11]

[Ahmed et al '08 '11]

[Siek&Vachharajani'08, Garcia&Cimini'15]

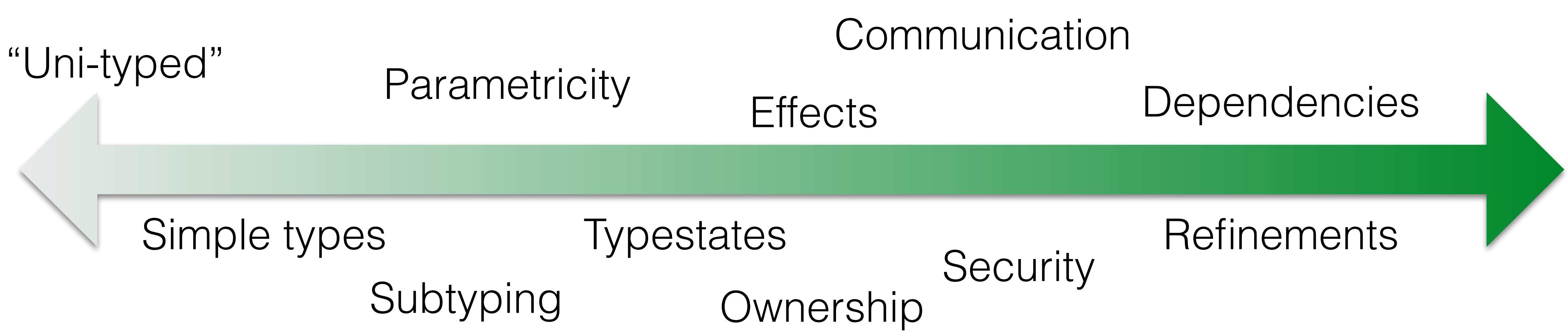
[Siek&Tobin-Hochstadt'16]

Gradual Typing

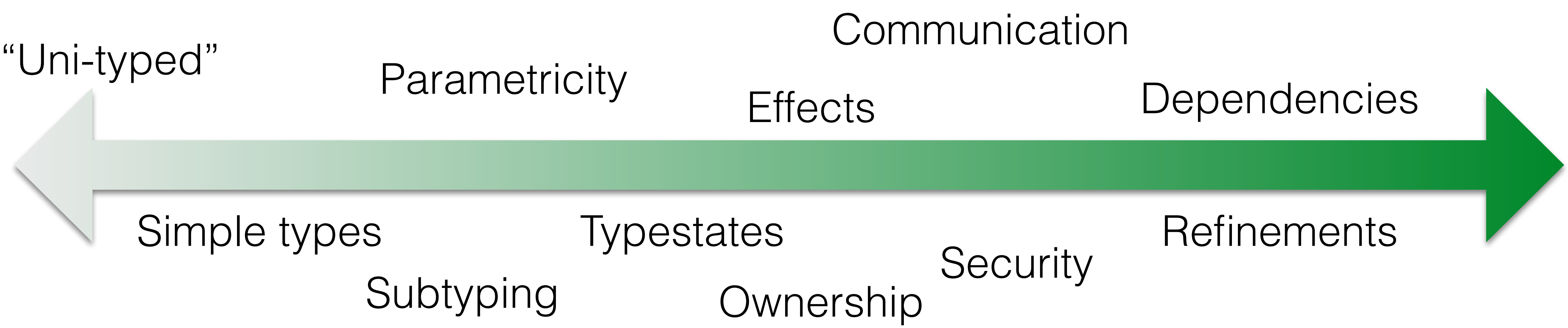
=

~~reconciling static and dynamic typing~~

reconciling **type disciplines of different strength**



Gradualized Type Disciplines



- **typestate**
- **effects** [ICF
- **refinemen** n-going]
- **security typing** [arXiv'15, on-going]

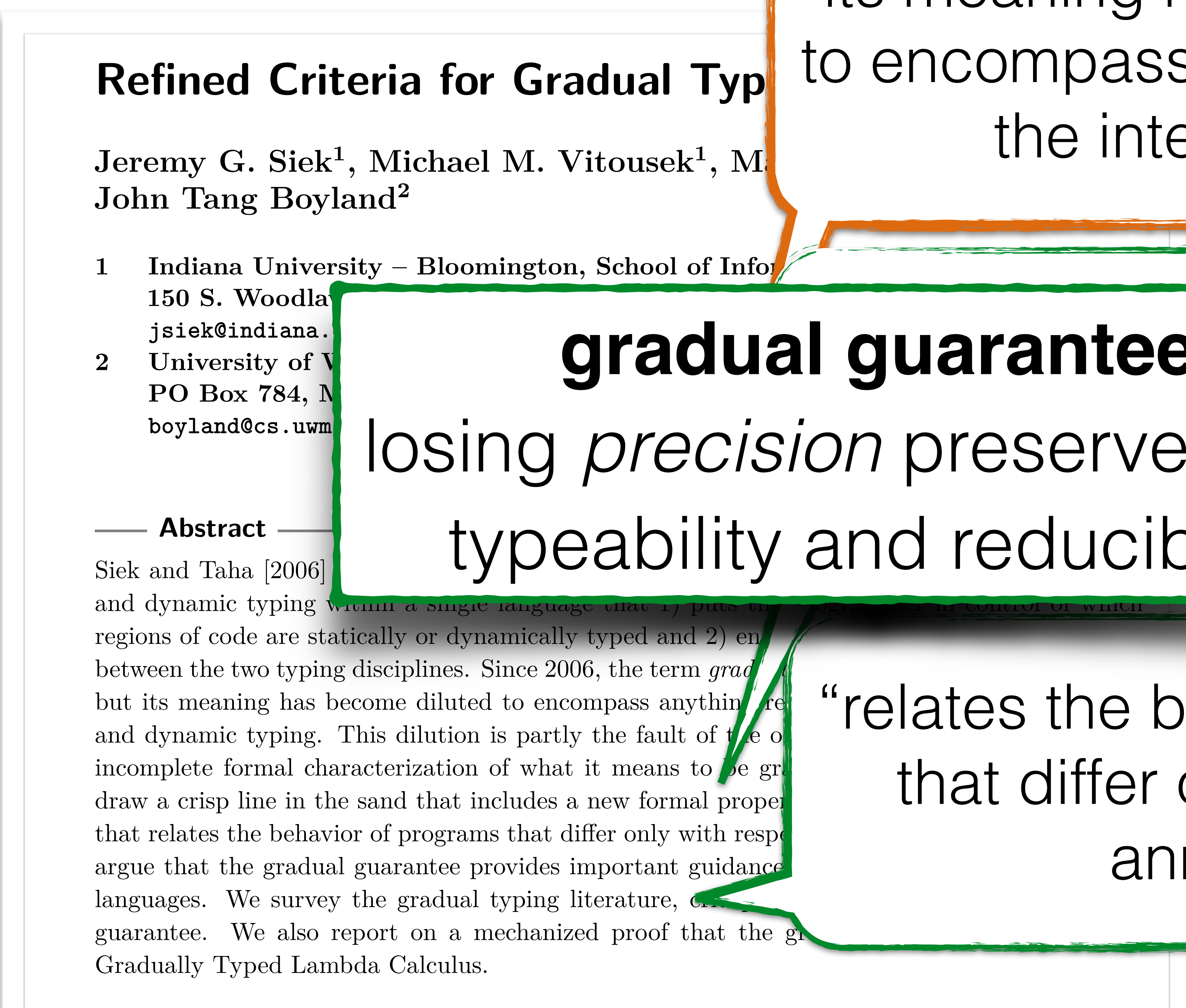
high cost

renegotiation of foundations

ingenious "tricks"

ad hoc justifications

What do you mean “Gradual”?



Refined Criteria for Gradual Typ

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

1 Indiana University – Bloomington, School of Infor
150 S. Woodla
jsiek@indiana.
2 University of W
PO Box 784, M
boyland@cs.uwm

— Abstract —

Siek and Taha [2006]
and dynamic typing
regions of code are statically or dynamically typed and 2) en
between the two typing disciplines. Since 2006, the term *grad*
but its meaning has become diluted to encompass anythin
and dynamic typing. This dilution is partly the fault of t
incomplete formal characterization of what it means to be gr
draw a crisp line in the sand that includes a new formal prope
that relates the behavior of programs that differ only with respe
argue that the gradual guarantee provides important guidance
languages. We survey the gradual typing literature,
guarantee. We also report on a mechanized proof that the gr
Gradually Typed Lambda Calculus.

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

l property,
uarantee”

gradual guarantee
losing *precision* preserves both typeability and reducibility

“relates the behavior of programs that differ only wrt their type annotations”

Stepping back…

Traditional Approach to Gradual Typing

type safe

inserts casts at boundaries

type safe

gradual language

cast calculus

TYPE SYSTEM | RUNTIME SEMANTICS

TYPE SYSTEM' | RUNTIME SEMANTICS

pre

enforces static assumptions

accounts for unknown type information

Challenges of Traditional Approach

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

RUNTIME SEMANTICS

gradual guarantees?

what are the "right" definitions?

how should unknown information be dealt with?

scale to advanced type disciplines?

WANTED

general foundations

systematic design principles

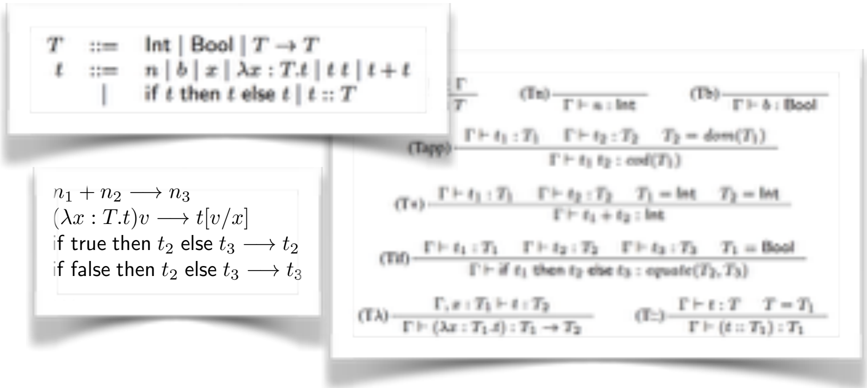
crisp connection to static language

formal justification

Abstracting Gradual Typing

AGT

[POPL 2016]



static type system
& type safety proof

?

interpretation of
gradual types

AGT

gradual language

type safe

gradual guarantee

TYPE SYSTEM	RUNTIME SEMANTICS

by construction

?

interpretation of gradual types

AGT

What *is* a gradual type?

static types $\textsc{Type}$

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

$$\Big\uparrow \text{“represents”}$$

gradual types $\textsc{GType}$

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

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

$$\mathsf{Int} \xrightarrow{\text{“represents”}} \mathsf{Int}$$

$$\mathsf{Int} \rightarrow ? \longrightarrow \begin{array}{l} \mathsf{Int} \rightarrow \mathsf{Int} \\ \mathsf{Int} \rightarrow \mathsf{Bool} \\ \mathsf{Int} \rightarrow \mathsf{Int} \rightarrow \mathsf{Int} \\ \mathsf{Int} \rightarrow \mathsf{Bool} \rightarrow \mathsf{Int} \\ \mathsf{Int} \rightarrow \ldots \end{array}$$

$$? \longrightarrow \text{any static type!}$$

Concretization

$$\gamma : \text{GT}\textsc{ype} \to \mathcal{P}(\text{T}\textsc{ype})$$

$$\begin{aligned}
\gamma(\mathsf{Int}) &= \{\, \mathsf{Int} \,\} \\
\gamma(\mathsf{Bool}) &= \{\, \mathsf{Bool} \,\} \\
\gamma(\widetilde{T}_1 \to \widetilde{T}_2) &= \{\, T_1 \to T_2 \mid T_1 \in \gamma(\widetilde{T}_1), T_2 \in \gamma(\widetilde{T}_2) \,\} \\
\boldsymbol{\gamma(?) = \text{T}\textsc{ype}}
\end{aligned}$$

e.g.

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

Design Space of Gradual Types

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

$$\gamma(?) = \text{Type}$$

$$\gamma(?) = \{\, \mathsf{Int}, \mathsf{Bool} \,\}$$

$$\gamma(?^T) = \{\, T' \in \text{Type} \mid T' <: T \,\}$$

$$\gamma(T_?) = \{\, T_\ell, \ell \in \text{Label} \,\}$$

This is the only design decision!
all the rest follows by AGT

Precision of Gradual Types

$$\widetilde{T}_1 \sqsubseteq \widetilde{T}_2$$

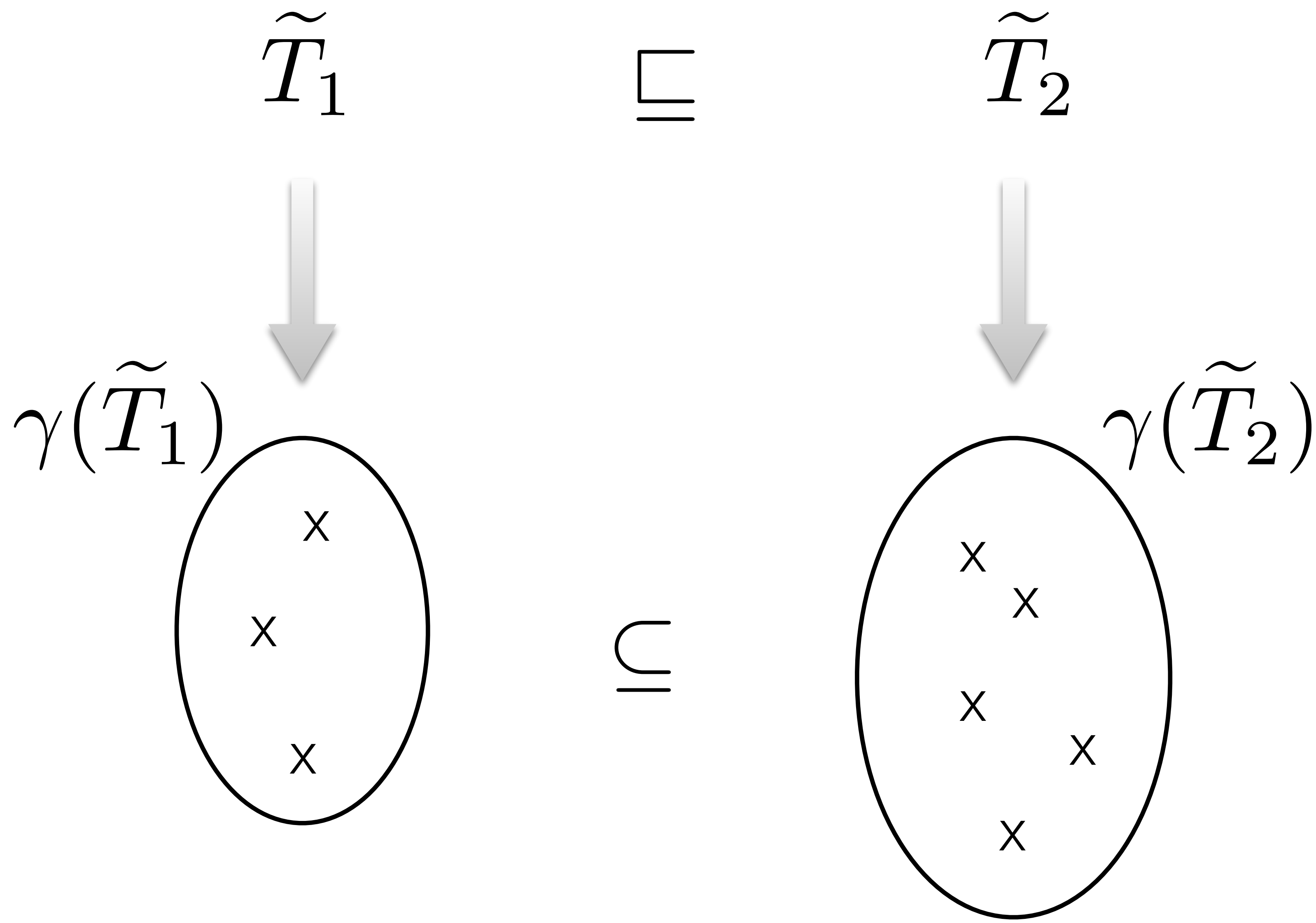
less unknown
aka.
more precise

$$\mathsf{Int} \rightarrow \mathsf{Int} \quad \sqsubseteq \quad \mathsf{Int} \rightarrow ? \quad \sqsubseteq \quad ? \rightarrow ? \quad \sqsubseteq \quad ?$$

a.k.a. “naive subtyping”

[Wadler & Findler, 2009]

Precision of Gradual Types



concretization induces the classic notion

AGT

I - Static Semantics

Gradualizing the Type System

0. start from a static typing discipline

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

explicit side conditions

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

AGT

syntax-directed rules

partial functions

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

$cod : \textsc{Type} \rightharpoonup \textsc{Type}$

$cod(T_1 \to T_2) = T_2$

$cod(T)$ undefined otherwise

[Garcia & Cimini, 2015]

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

consistent side conditions

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

compositional lifting

lifted partial functions

gradual meet

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

we now need to define and justify all of this!

Gradualizing the Type System

1. lift type predicates
2. lift type functions

Gradualizing the Type System

1. **lift type predicates**
2. lift type functions

Lifting Type Predicates

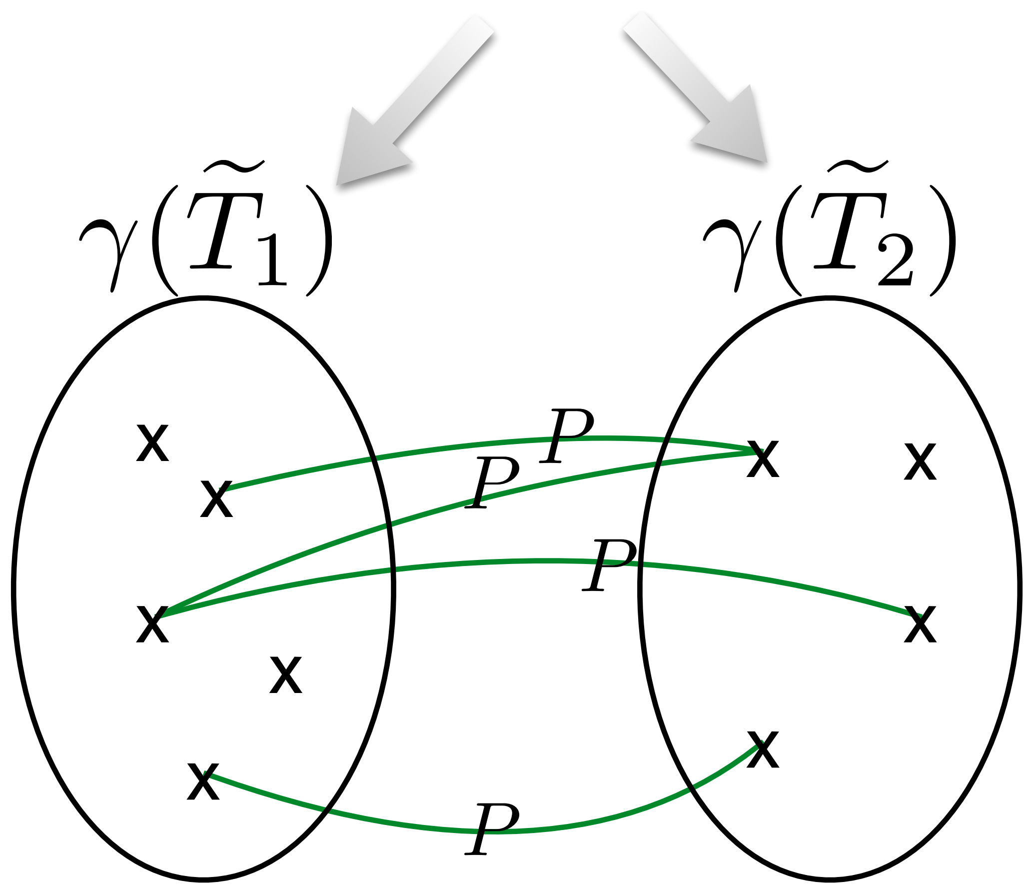
$$P \subseteq \textsc{Type} \times \textsc{Type} \longrightarrow \widetilde{P} \subseteq \textsc{GType} \times \textsc{GType}$$

$$T_1 = T_2 \longrightarrow \widetilde{T_1} \sim \widetilde{T_2}$$

$$T_1 <: T_2 \longrightarrow \widetilde{T_1} \lesssim \widetilde{T_2}$$

Lifting Type Predicates

$$\widetilde{P}(\widetilde{T}_1, \widetilde{T}_2)$$

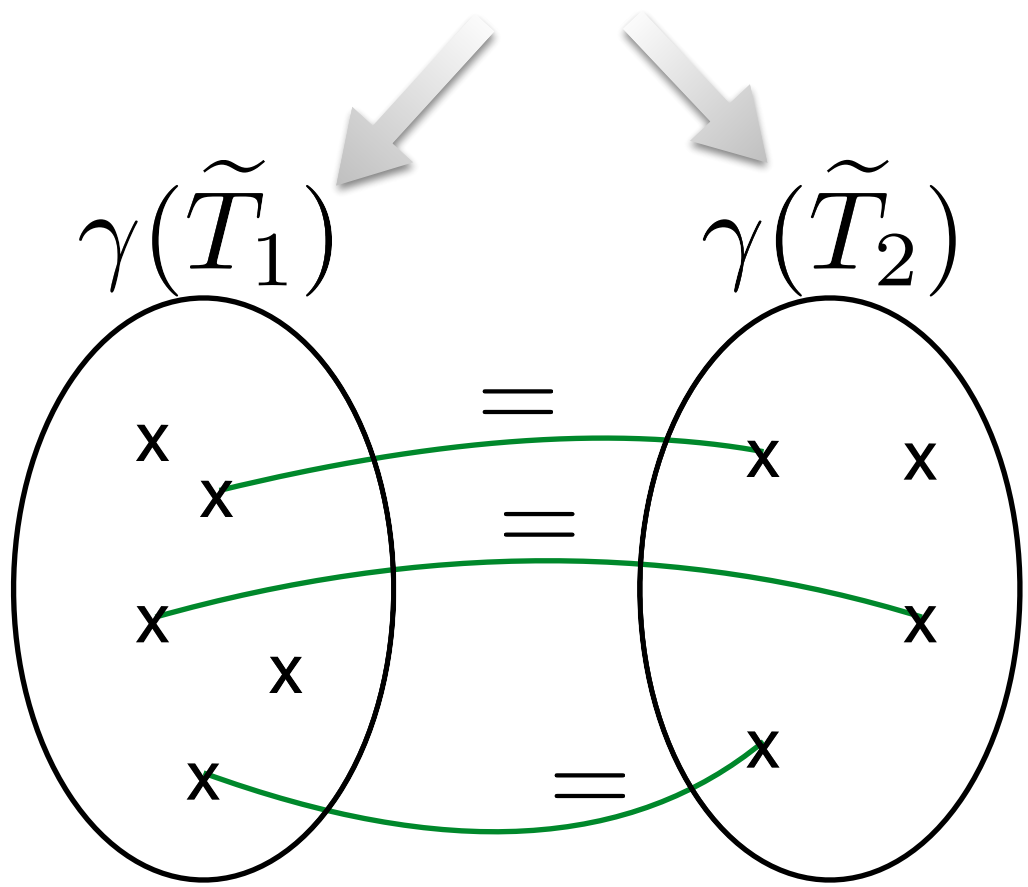


$$\text{iff } P(T_1, T_2) \text{ for } \textit{some } \langle T_1, T_2 \rangle \in \gamma(\widetilde{T}_1) \times \gamma(\widetilde{T}_2)$$

plausibility

Lifting Equality

$$\widetilde{T_1} \sim \widetilde{T_2}$$

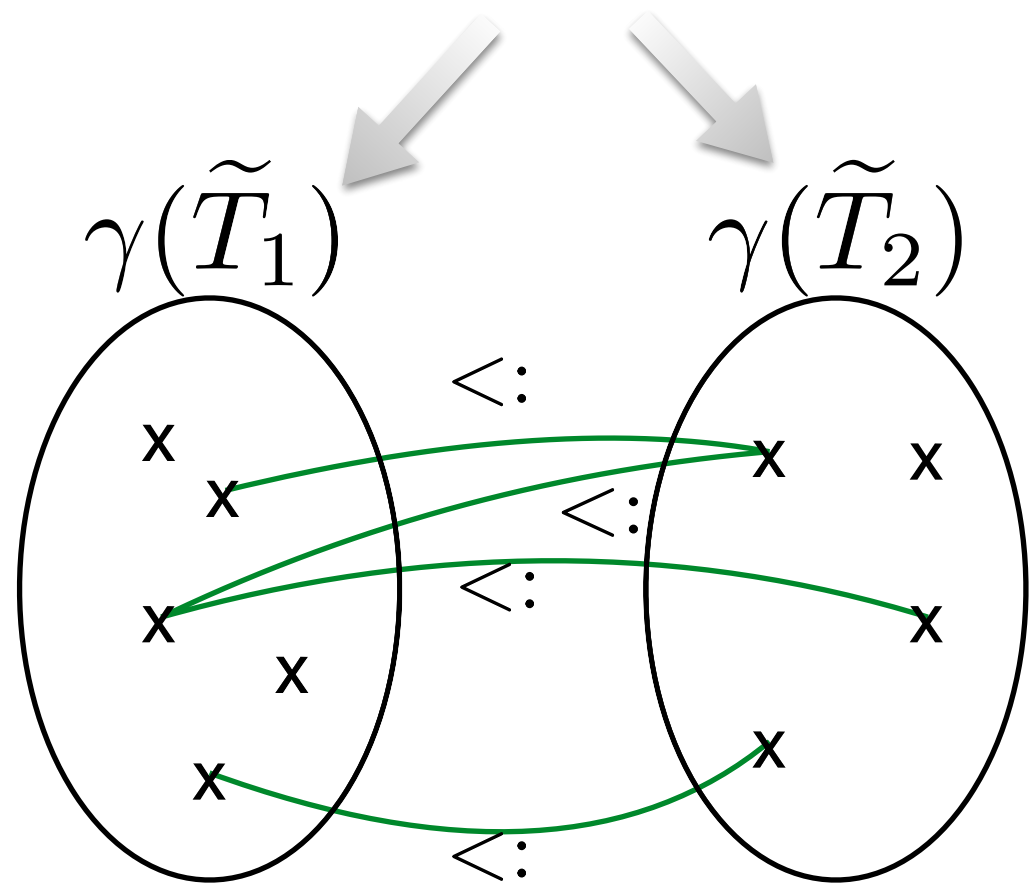


iff $T_1 = T_2$ for *some* $\langle T_1, T_2 \rangle \in \gamma(\widetilde{T}_1) \times \gamma(\widetilde{T}_2)$.

coincides with [Siek & Taha, 2006]

Lifting Subtyping

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



iff $T_1 <: T_2$ for *some* $\langle T_1, T_2 \rangle \in \gamma(\widetilde{T}_1) \times \gamma(\widetilde{T}_2)$.

coincides with [Siek & Taha, 2007]

Lifting Subtyping

$$\text{iff } T_1 <: T_2 \text{ for } \textit{some } \langle T_1, T_2 \rangle \in \gamma(\widetilde{T}_1) \times \gamma(\widetilde{T}_2).$$

subtyping on *gradual* types

$$\sigma \lesssim \tau \equiv \sigma|_\tau <: \tau|_\sigma$$

[Siek & Taha, 2007]

gradual type *masking*

$$
\begin{array}{ll}
\sigma|_\tau = \textsf{case } (\sigma, \tau) \textsf{ of} & \\
\quad (-, ?) \Rightarrow ? & \\
\quad \mid ([l_1 : s_1, \ldots, l_n : s_n], [l_1 : t_1, \ldots, l_m : t_m]) \textsf{ where } \tau & \\
\quad\quad [l_1 : s_1|_{t_1}, \ldots, l_n : s_n|_{t_n}] & \\
\quad \mid ([l_1 : s_1, \ldots, l_n : s_n], [l_1 : t_1, \ldots, l_m : t_m]) \textsf{ where } \tau & \\
\quad\quad [l_1 : s_1|_{t_1}, \ldots, l_m : s_m|_{t_m}, l_{m+1} : s_{m+1}, \ldots, l_n & \\
\quad \mid (-, -) \Rightarrow \sigma &
\end{array}
$$

$$(\sigma_1 \to \sigma_2)|_{(\tau_1 \to \tau_2)} = (\sigma_1|_{\tau_1}) \to (\sigma_2|_{\tau_2})$$

$$
\begin{tikzcd}
\sigma' \arrow[rr, dash, bend left, "\sim"] & \tau|_\sigma \arrow[r, dash, "\sim"'] & \tau \\
\sigma \arrow[u, "<:"] \arrow[r, dash, "\sim"] \arrow[rr, dash, bend right, "\sim"'] & \sigma|_\tau \arrow[u, "<:"] & \sigma'' \arrow[u, "<:"]
\end{tikzcd}
$$

for some σ', and
for some σ''.

Gradualizing the Type System

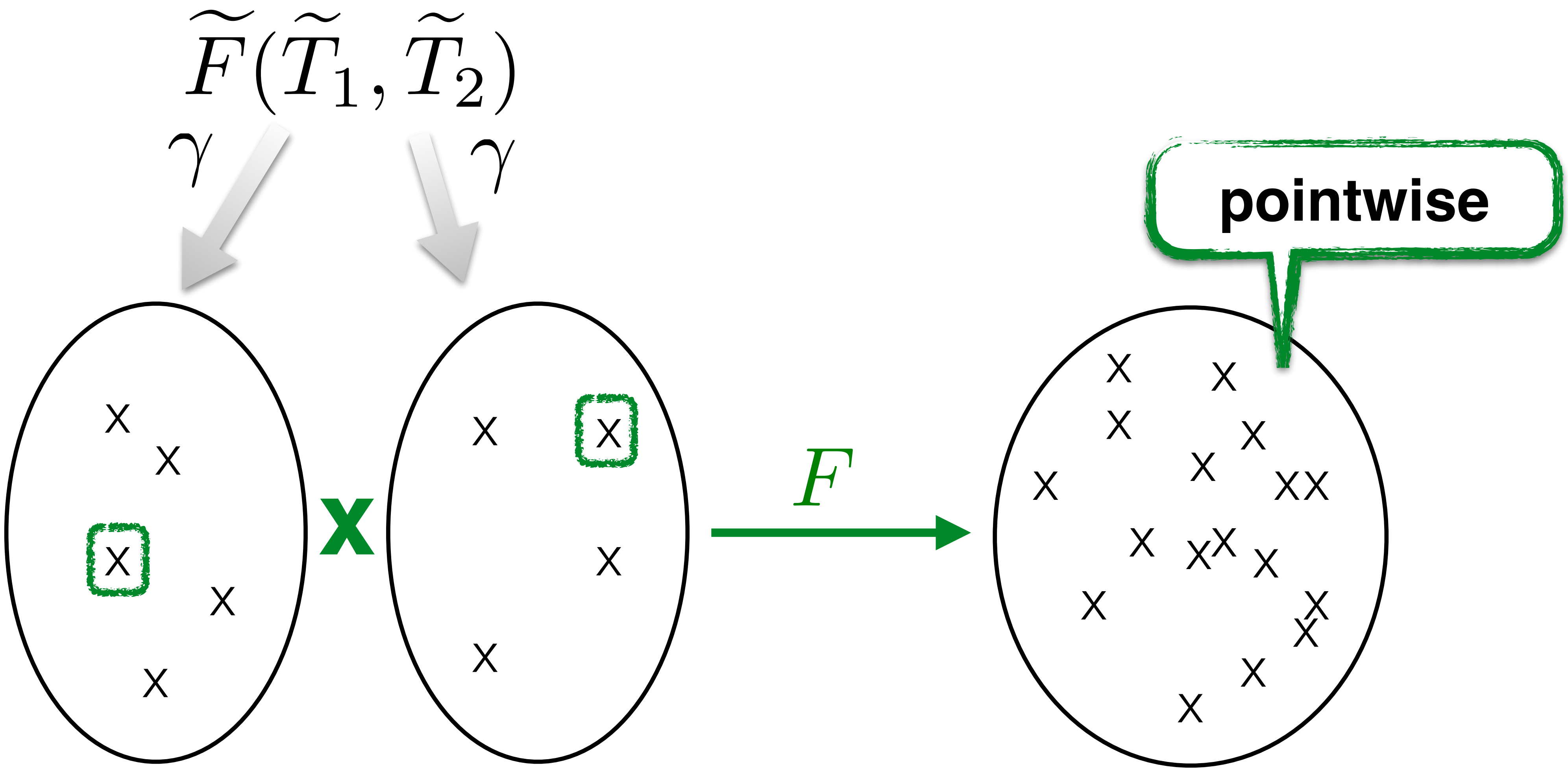
1. lift type predicates
2. **lift type functions**

Lifting Type Functions

$$F : \textsc{Type}^n \rightharpoonup \textsc{Type} \Longrightarrow \widetilde{F} : \textsc{GType}^n \rightharpoonup \textsc{GType}$$

$$cod(T) \Longrightarrow \widetilde{cod}(\widetilde{T})$$

$$T_1 \mathbin{\ddot{\vee}} T_2 \Longrightarrow \widetilde{T}_1 \mathbin{\widetilde{\ddot{\vee}}} \widetilde{T}_2$$

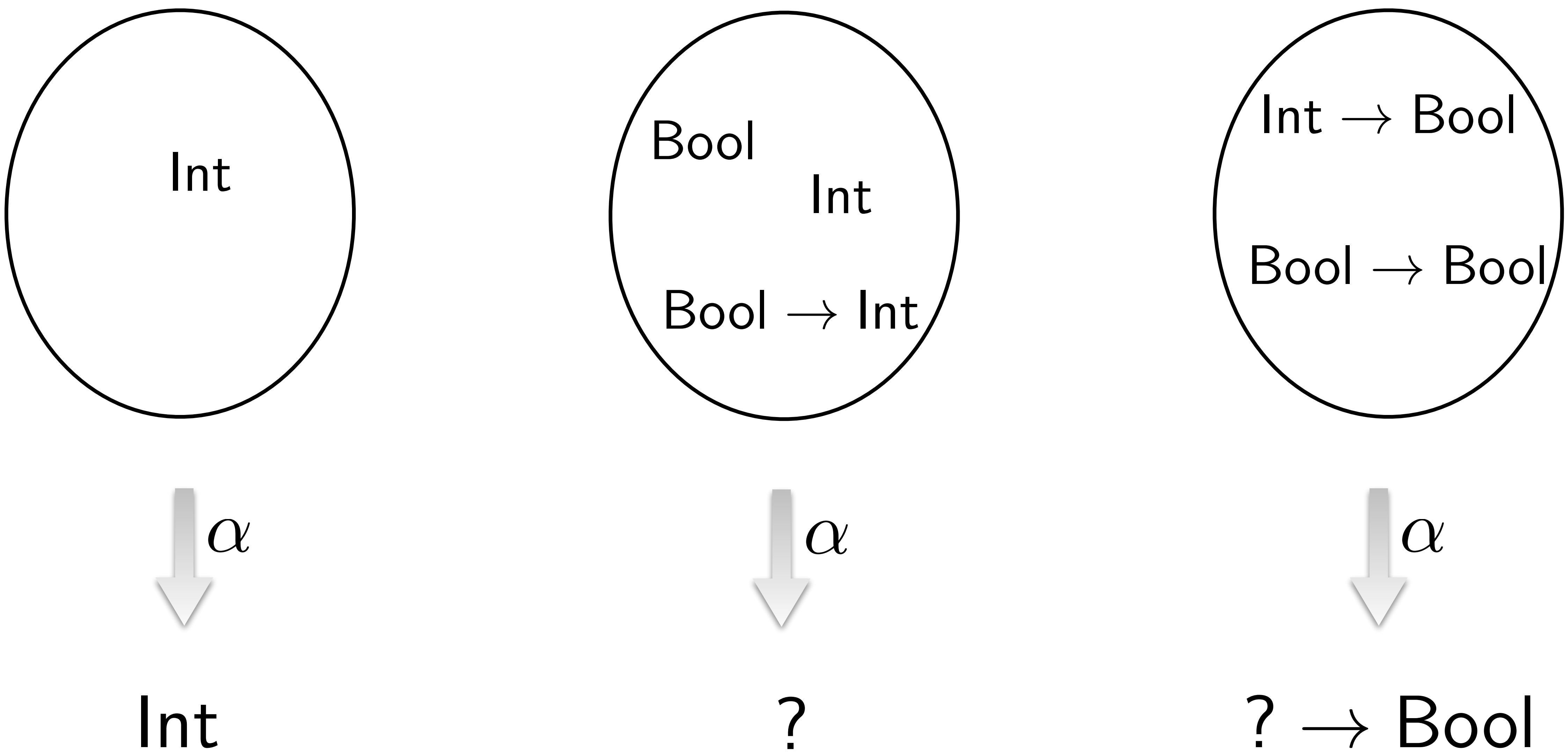


what's the corresponding gradual type?

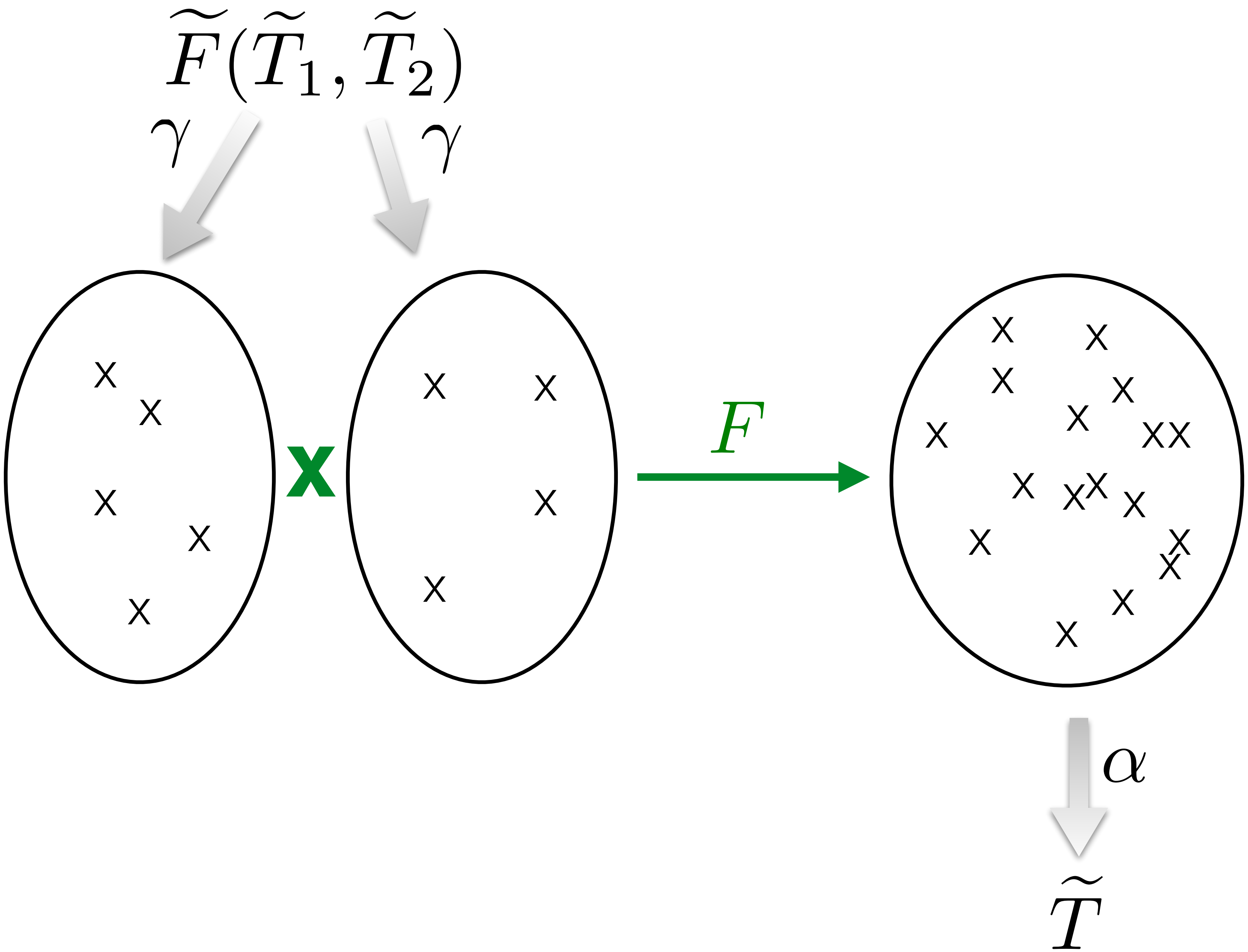
we need a notion of ***abstraction***

Abstraction

$$\alpha : \mathcal{P}(\textsc{Type}) \rightharpoonup \textsc{GType}$$



Galois connection (sound & optimal)



$$\widetilde{F}(\widetilde{T}_1, \widetilde{T}_2) = \alpha(\{\, F(T_1, T_2) \mid \langle T_1, T_2 \rangle \in \gamma(\widetilde{T}_1) \times \gamma(\widetilde{T}_2) \,\})$$

Lifting *cod*

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

$$\widetilde{cod} : \text{G}\textsc{Type} \rightharpoonup \text{G}\textsc{Type}$$
$$\widetilde{cod}(\widetilde{T}_1 \to \widetilde{T}_2) = \widetilde{T}_2$$
$$\boxed{\widetilde{cod}(?) = ?}$$
$$\widetilde{cod}(\widetilde{T}) \text{ undefined otherwise}$$

Lifting *equate*

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

$$\mathit{equate} : \textsc{Type} \times \textsc{Type} \rightarrow \textsc{Type}$$

"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

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

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

Lifting *subtyping join*

$$\frac{\Gamma \vdash t_1 : \mathsf{Bool} \qquad \Gamma \vdash t_2 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T_1 \mathbin{\ddot{\vee}} T_2}$$

$$\frac{\Gamma \vdash \widetilde{t}_1 : \widetilde{T}_0 \qquad \widetilde{T}_0 \lesssim \mathsf{Bool} \qquad \Gamma \vdash \widetilde{t}_2 : \widetilde{T}_1 \qquad \Gamma \vdash \widetilde{t}_2 : \widetilde{T}_2}{\Gamma \vdash \mathsf{if}\ \widetilde{t}_1\ \mathsf{then}\ \widetilde{t}_2\ \mathsf{else}\ \widetilde{t}_3 : \widetilde{T}_1 \mathbin{\widetilde{\ddot{\vee}}} \widetilde{T}_2}$$

Properties of Gradual Languages

(part 1: static semantics)

by construction

equivalence for static terms [Siek & Taha, 2006]

$\vdash_S t : T$ if and only if $\vdash t : T$

embedding of dynamic terms [Siek & Taha, 2006]

If $\check{t}$ is closed then $\vdash \lceil \check{t} \rceil : ?$

losing precision preserves typing [Siek *et al*, 2015]

If $\vdash \widetilde{t}_1 : \widetilde{T}_1$ and $\widetilde{t}_1 \sqsubseteq \widetilde{t}_2$, then $\vdash \widetilde{t}_2 : \widetilde{T}_2$ and $\widetilde{T}_1 \sqsubseteq \widetilde{T}_2$

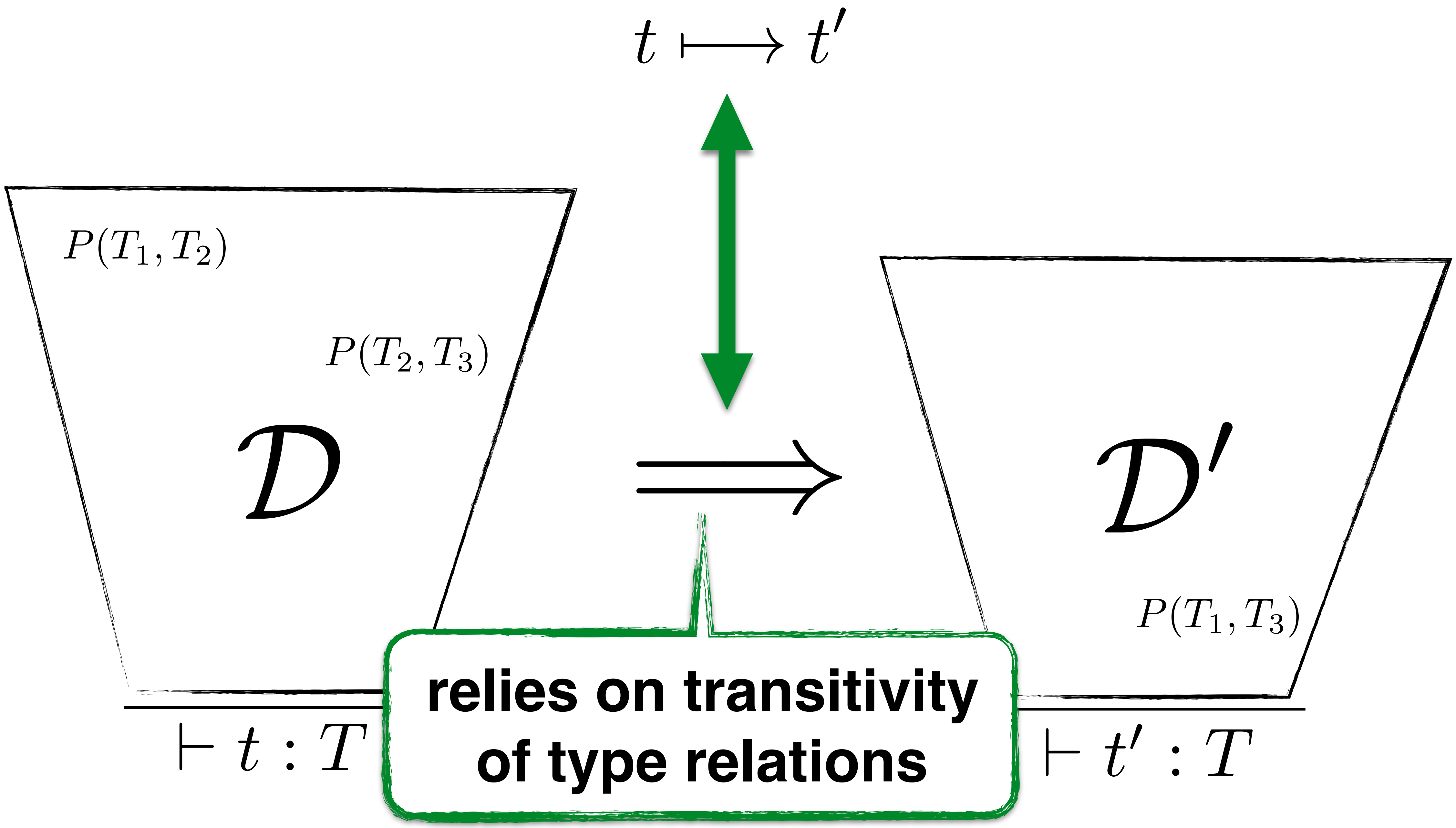
AGT

II - Dynamic Semantics

Curry-Howard

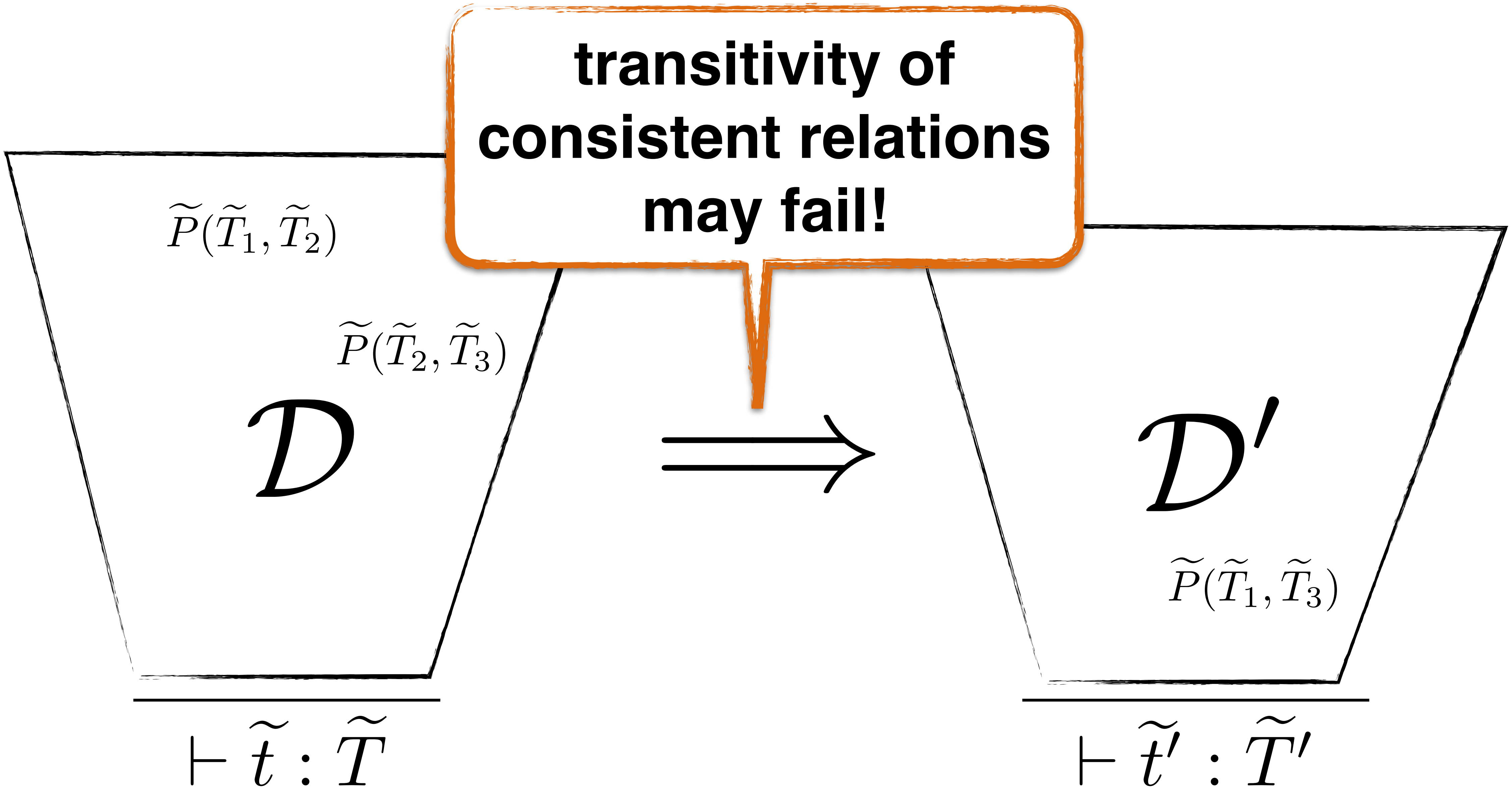
Logic	**PL**
Propositions	Types
Proofs	Programs
Proof reduction	Program evaluation

Type Safety Proof as Reduction



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

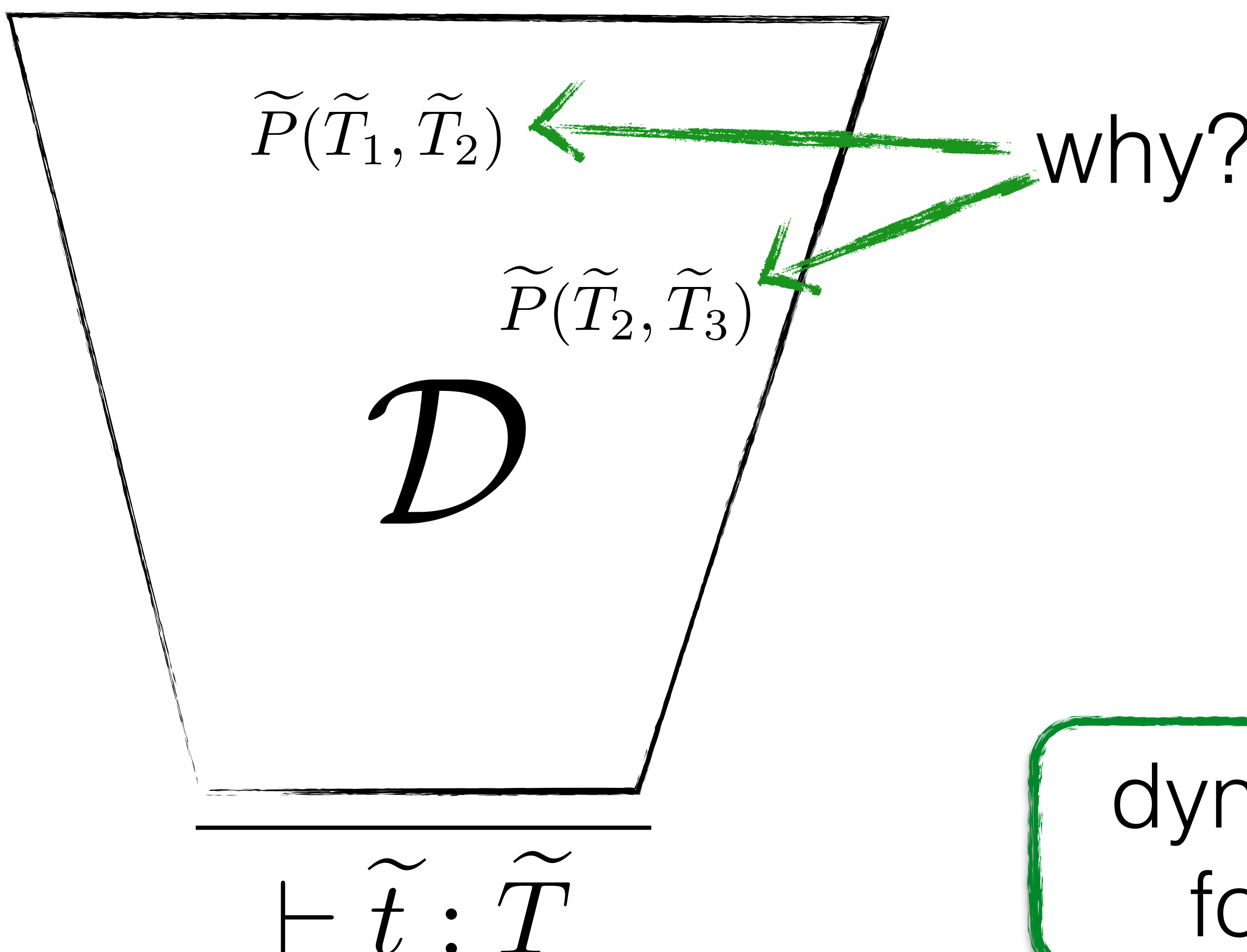
Reduction of Gradual Derivations



$$\widetilde{P}(\widetilde{T}_1, \widetilde{T}_2) \wedge \widetilde{P}(\widetilde{T}_2, \widetilde{T}_3) \Rightarrow^{?} \widetilde{P}(\widetilde{T}_1, \widetilde{T}_3)$$

$\mathsf{Int} \sim ?$ $\quad ? \sim \mathsf{Bool}$ $\quad \mathsf{Int} \not\sim \mathsf{Bool}$

Evidence of Consistent Judgments



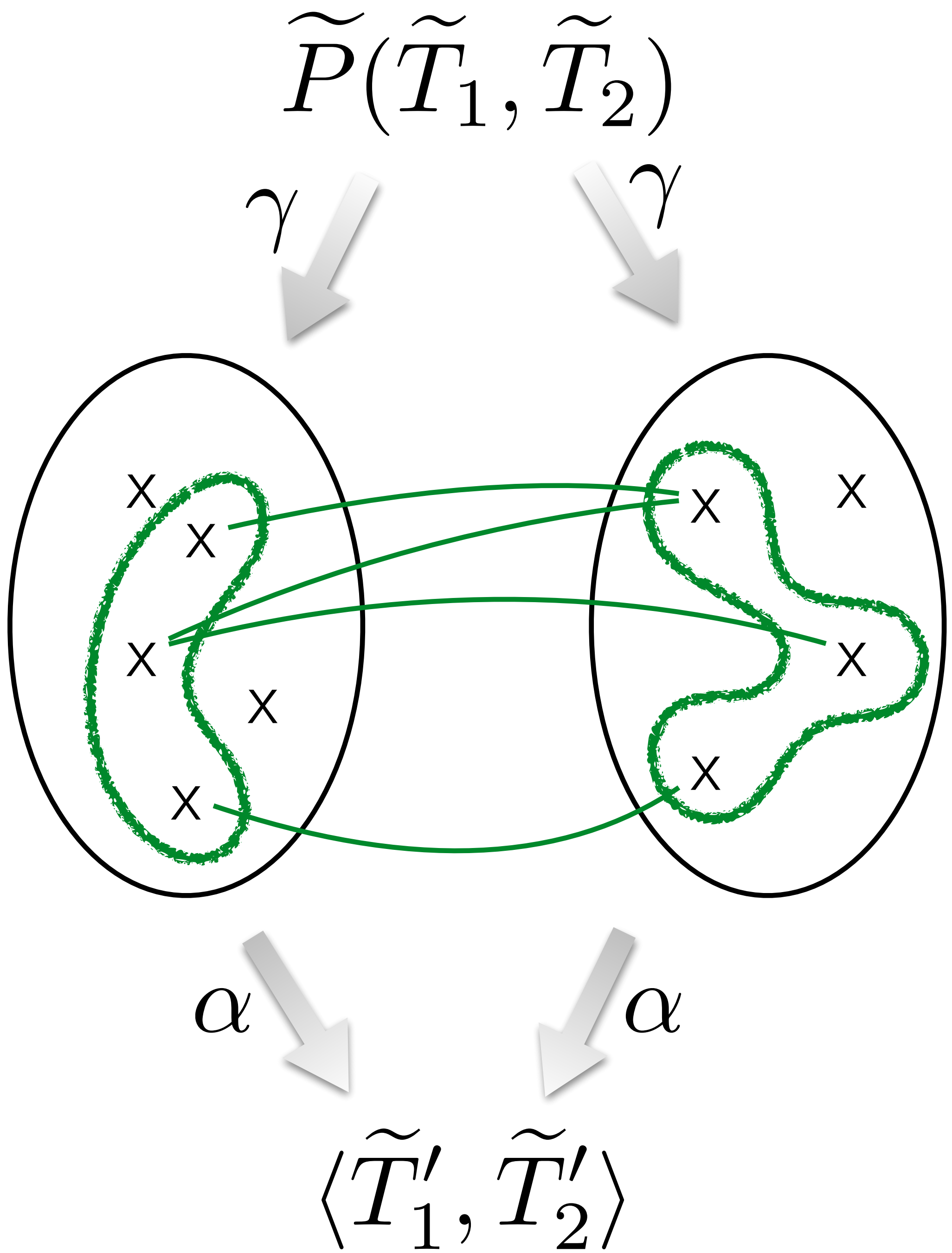
introduce **evidence**

$$\varepsilon_{12} \vdash \widetilde{P}(\widetilde{T}_1, \widetilde{T}_2)$$

dynamic “local justification”
for consistent judgment

Initial Evidence

$$\varepsilon_{12} \vdash \widetilde{P}(\widetilde{T}_1, \widetilde{T}_2)$$

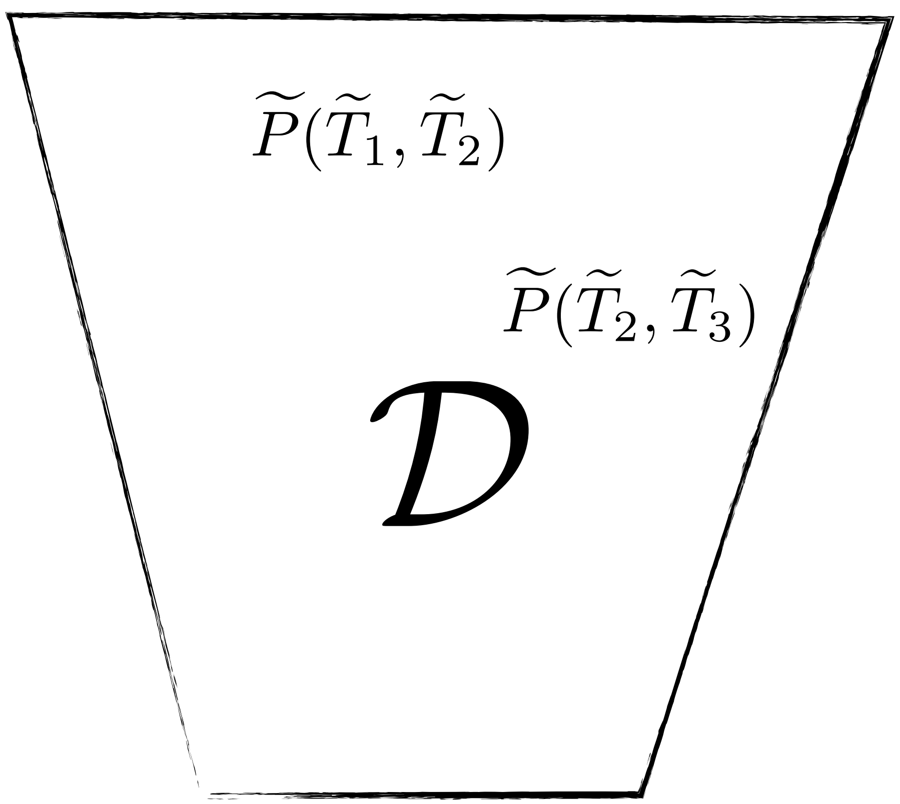


$$\begin{array}{ccc} [x : \mathsf{Int} \to ?, y : ?] & \lesssim & [x : ? \to \mathsf{Bool}] \\ \Downarrow \sqsupseteq & & \Downarrow \sqsupseteq \\ [x : \mathsf{Int} \to \mathsf{Bool}, y : ?] & & [x : \mathsf{Int} \to \mathsf{Bool}] \end{array}$$

corresponds to
Threesome middle type

[Siek & Wadler, 2010]

Evidence

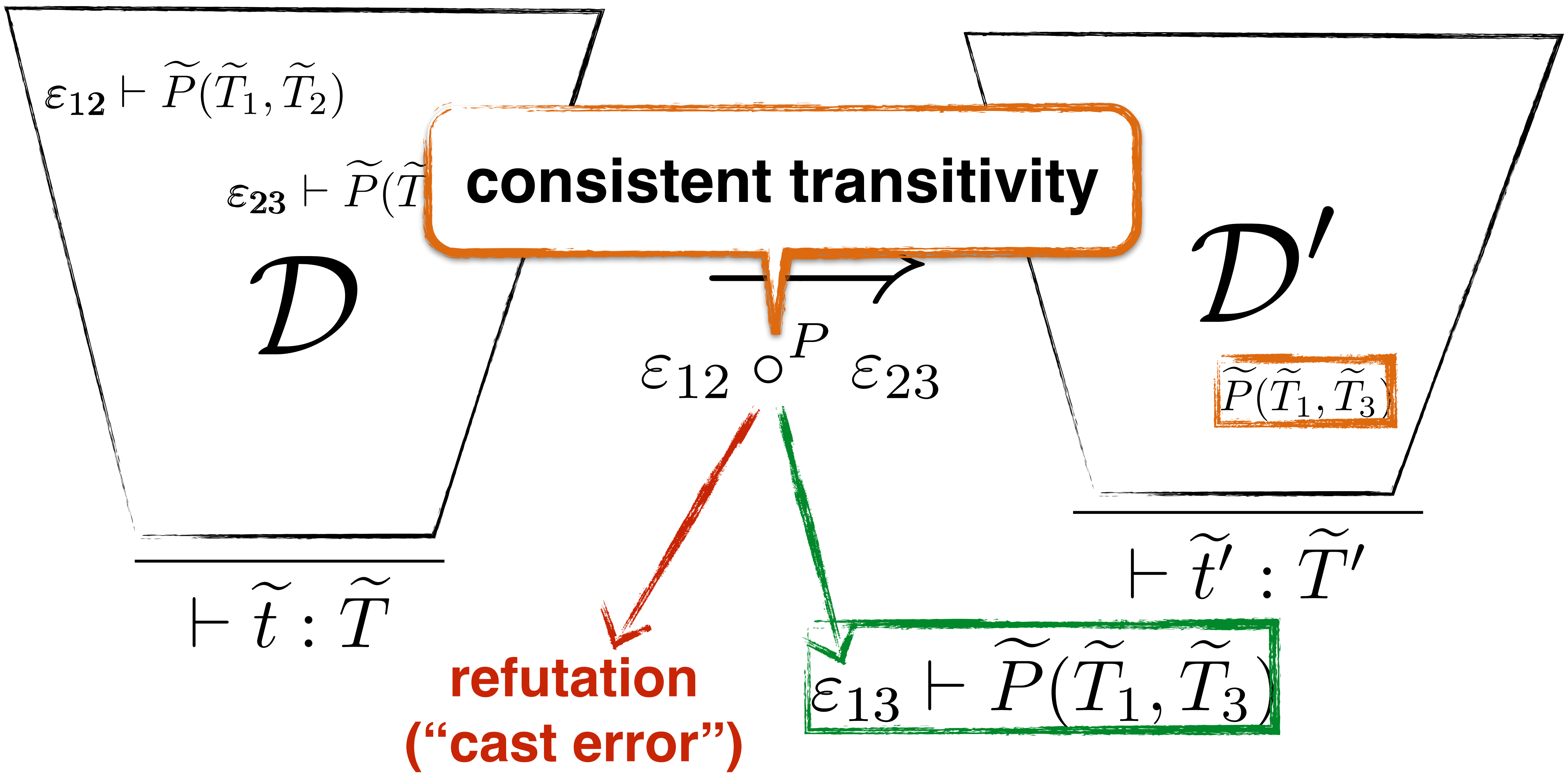


$$\vdash \widetilde{t} : \widetilde{T}$$

$$\langle \widetilde{T}_1', \widetilde{T}_2' \rangle \vdash \widetilde{P}(\widetilde{T}_1, \widetilde{T}_2)$$

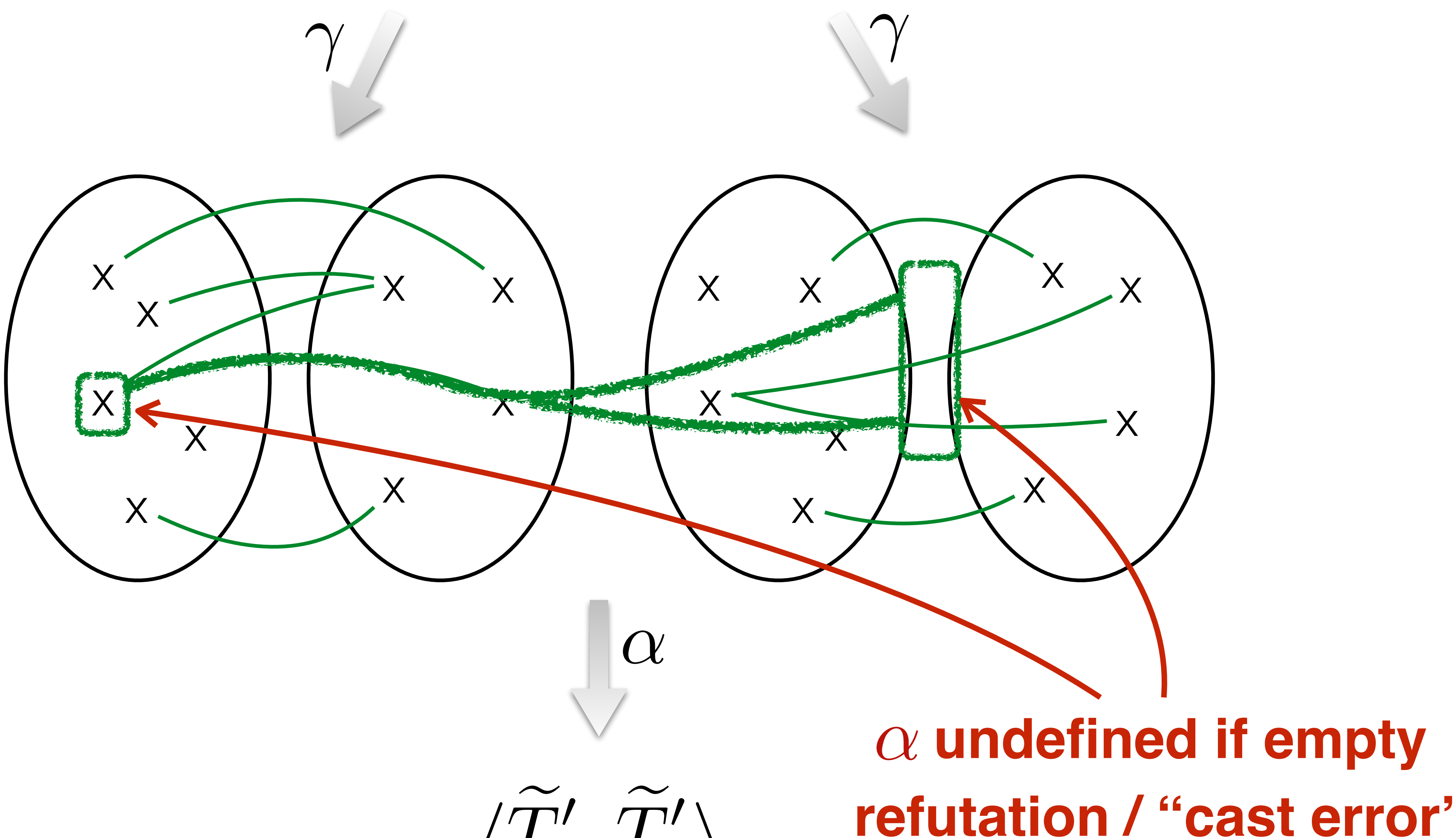
$$\varepsilon_{23} \vdash \widetilde{P}(\widetilde{T}_2, \widetilde{T}_3)$$

Consistent Transitivity



Consistent Transitivity

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



$$\langle \widetilde{T}_1', \widetilde{T}_3' \rangle$$

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

Properties of Gradual Languages

(part 2: dynamic semantics)

by construction

type safety

If $\vdash \widetilde{t} : \widetilde{T}$ then either

$\widetilde{t}$ is a value v, or $\widetilde{t} \longmapsto \widetilde{t}'$ with $\vdash \widetilde{t}' : \widetilde{T}$

or $\widetilde{t} \longmapsto \mathbf{error}$

losing precision preserves reduction [Siek *et al*, 2015]

Suppose $\widetilde{t}_1 \sqsubseteq \widetilde{t}_2$ with $\vdash \widetilde{t}_1 : \widetilde{T}_1$ and $\vdash \widetilde{t}_2 : \widetilde{T}_2$

If $\widetilde{t}_1 \longmapsto \widetilde{t}_1'$ then $\widetilde{t}_2 \longmapsto \widetilde{t}_2'$ and $\widetilde{t}_1' \sqsubseteq \widetilde{t}_2'$

AGT

Conclusions

RECOVERED

precision
consistency
consistent subtyping
gradual meet
threesomes
runtime semantics
cast errors
gradual guarantees

Breadth of AGT

- Applications of AGT so far
  - records with subtyping
  - gradual rows (*à la* row polymorphism)
  - gradual effects
  - gradual references
  - gradual security typing
  - gradual refinements
  - (quite) some more ;-)

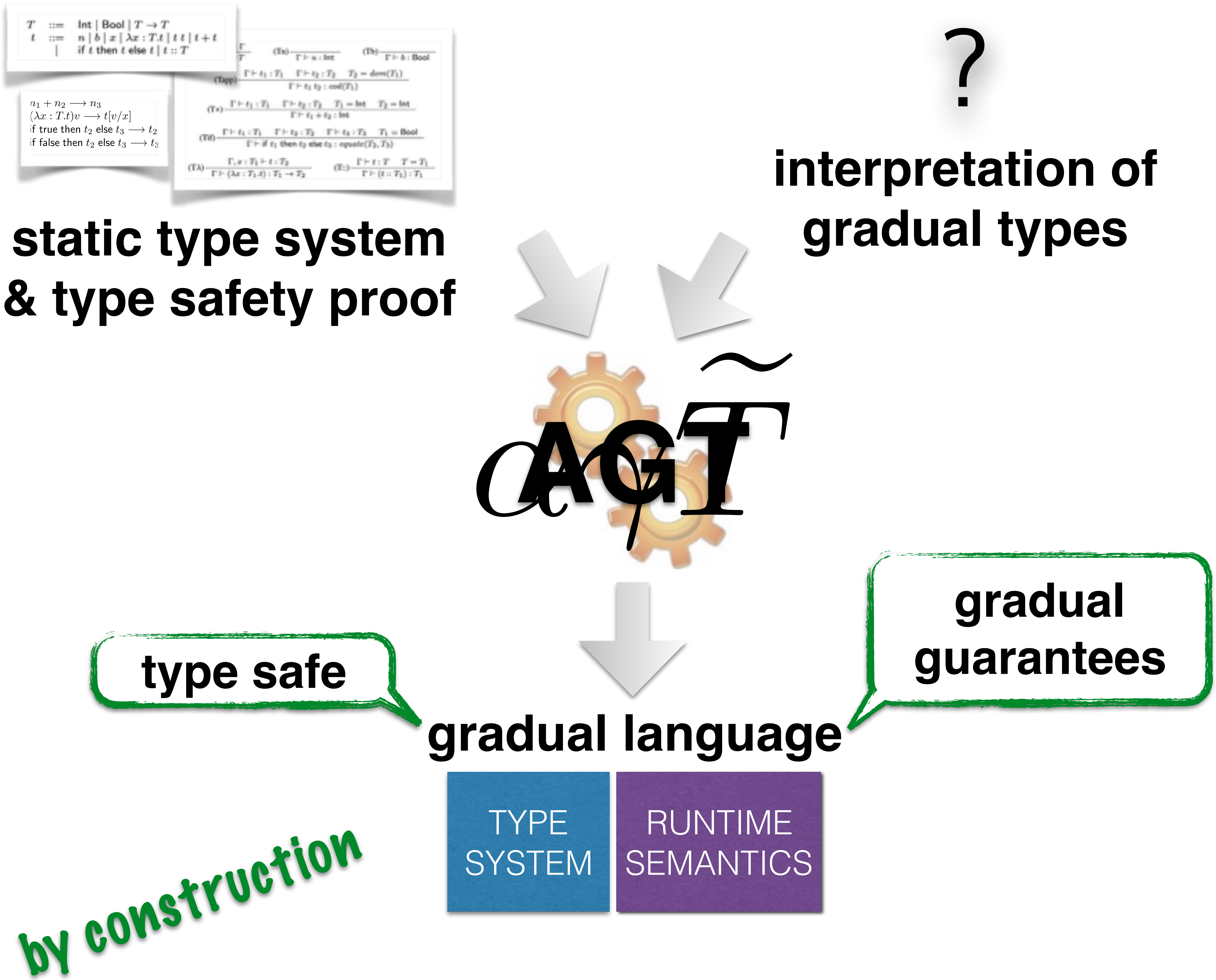
POPL'16

ICFP'14 & co
(statics based on AGT)

on going!

Future work on AGT

- Cast calculi
    - representation of computational content of derivation trees
    - validate existing cast calculi wrt “reference semantics”
    - space and time efficiency (eliminate useless evidence)
- Blame tracking
- Relational soundness properties (eg. non-interference)
    - Static vs dynamic abstractions


static type system
& type safety proof
?
interpretation of
gradual types
AGT
type safe
gradual
guarantees
gradual language
TYPE
SYSTEM
RUNTIME
SEMANTICS
by construction