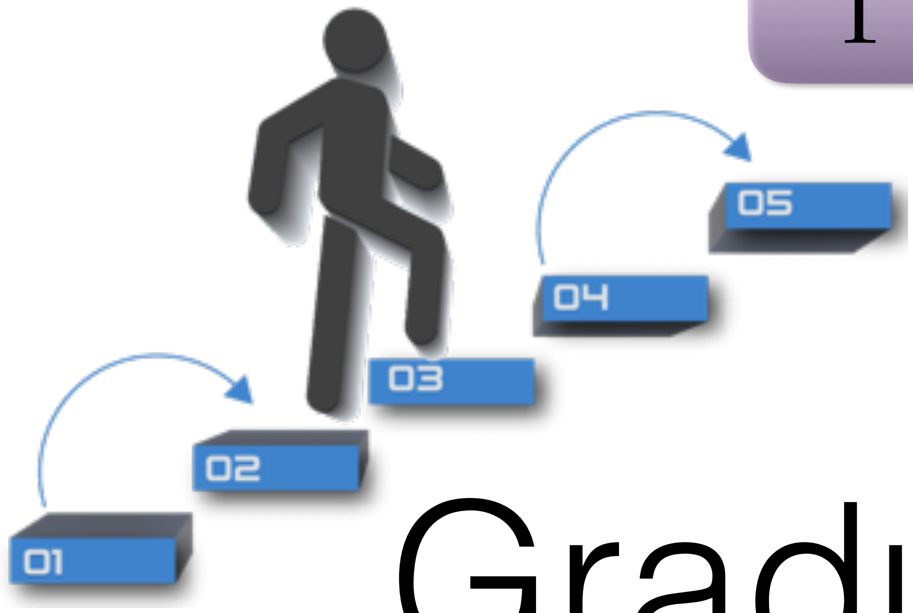
$\Gamma \vdash e : T$



$\Gamma \vdash e \checkmark$

Gradual Typing

Ronald Garcia
University of British Columbia

Static ***vs.*** Dynamic?

static

early error detection
enforced discipline

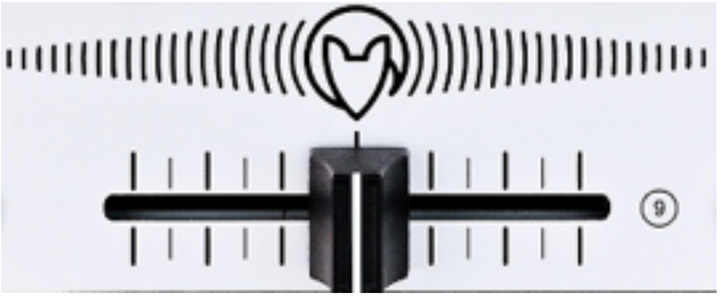
dynamic

rapid prototyping
flexible idioms

Gradual Typing!

early error detection
enforced discipline

rapid prototyping
flexible idioms



programmer-controlled!

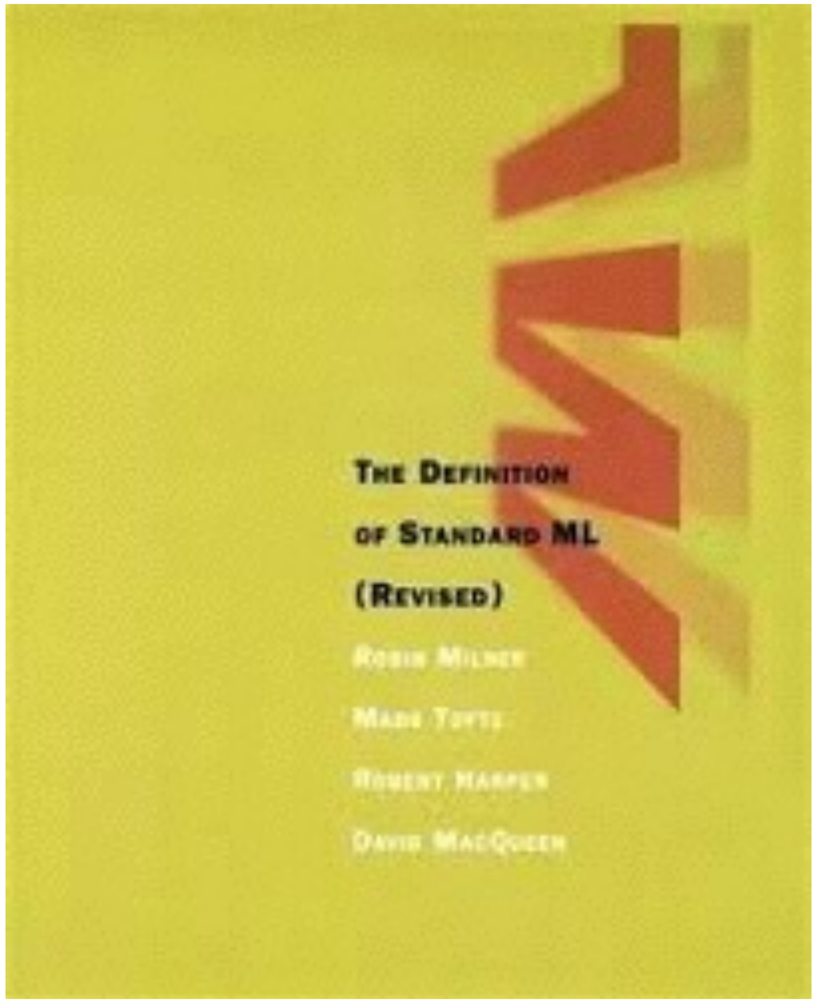
Outline

- Motivating Example (In Two Acts)
- Gradual Typing For All!
- Typing in Small Pieces
- Meat
- Strands and Related Works

Motivating Example
Act 1: A New Type

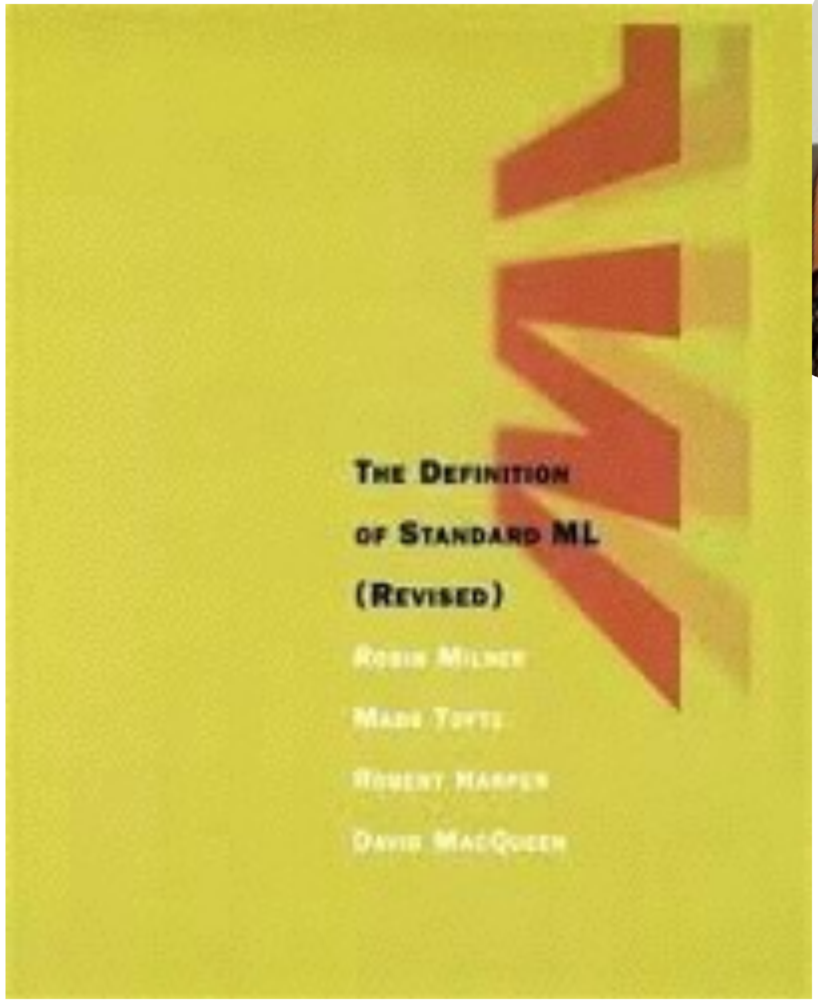
A Dynamic Language

A Dynamic Language



Standard ML

A Dynamic Language




Robin Milner

Standard ML

Standard ML: dynamically typed?

$$\textsf{datatype}\ \textsf{nat}\ =\ \texttt{Zero} \mid \texttt{Succ}\ \textsf{of}\ \textsf{nat}$$

$$\begin{array}{l} \textsf{case}\ x : \textsf{nat}\ \textsf{of} \\ \quad\ \texttt{Zero} \Rightarrow \texttt{...} \\ \ \mid \texttt{Succ}\ y \Rightarrow \texttt{...} \end{array}$$

Adapted from Jafery and Dunfield [POPL17]

Standard ML: dynamically typed?

```
datatype nat  =  Zero | Succ of nat
```

```
 case x : nat of
    Zero ⇒ ...
  | Succ y ⇒ ...
```

But the Definition requires compilers to accept **nonexhaustive** matches:

```
 case x : nat of
   Succ y ⇒ ...
```

Adapted from Jafery and Dunfield [POPL17]

Standard ML: dynamically typed?

```
datatype nat  =  Zero | Succ of nat
```

```
 case x : nat of
    Zero ⇒ ...
  | Succ y ⇒ ...
```

But the Definition requires compilers to accept **nonexhaustive** matches:

```
 case x : nat of
   Succ y ⇒ ...
```

If x = `Zero`, then the exception `Match` is raised.

This nonexhaustive match is fine,
if we know that x will never be `Zero`.

Adapted from Jafery and Dunfield [POPL17]


MOAR
TYPES!!!!!

Standard ML: dynamically typed?

```
datatype nat  =  Zero | Succ of nat
```

```
 case x : nat of
     Zero ⇒ ...
   | Succ y ⇒ ...
```

But the Definition requires compilers to accept **nonexhaustive** matches:

```
 case x : nat of
    Succ y ⇒ ...
```

If x = `Zero`, then the exception `Match` is raised.

This nonexhaustive match is fine,
if we know that x will never be `Zero`.

Adapted from Jafery and Dunfield [POPL17]

Standard ML: dynamically typed?

```
datatype nat  =  Zero | Succ of nat
```

```
case x : nat of
    Zero ⇒ ...
  | Succ y ⇒ ...
```

Frank Pfenning

But the Definition requires compilers to accept
nonexhaustive matches:

```
case x : nat of
   Succ y ⇒ ...
```

If x = `Zero`, then the exception `Match` is raised.

This nonexhaustive match is fine,
if we know that x will never be `Zero`.

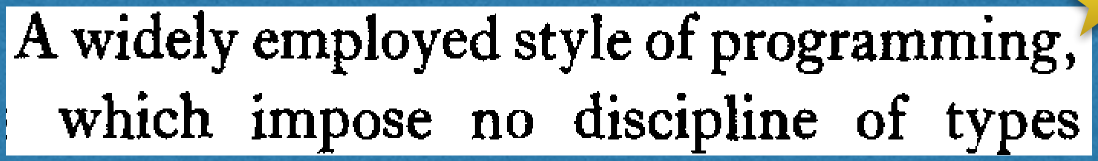
Adapted from Jafery and Dunfield [POPL17]

Standard ML: dynamically typed?

```
datatype nat = Zero | Succ of nat
```

```
case x : nat of
   Zero ⇒ ...
 | Succ y ⇒ ...
```

Frank Pfenning


A widely employed style of programming, which impose no discipline of types ★

lers to accept

```
case x : nat of
   Succ y ⇒ ...
```

If x = Zero, then the exception Match is raised.

This nonexhaustive match is fine,
if we know that x will never be Zero.

★ **Well, actually Milner [1978] said that (about LISP).**

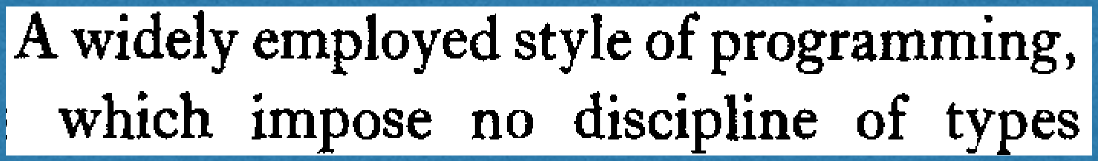
Adapted from Jafery and Dunfield [POPL17]

Standard ML: dynamically typed?

```
datatype nat  =  Zero | Succ of nat

 case x : nat of
    Zero ⇒ ...
  | Succ y ⇒ ...
```

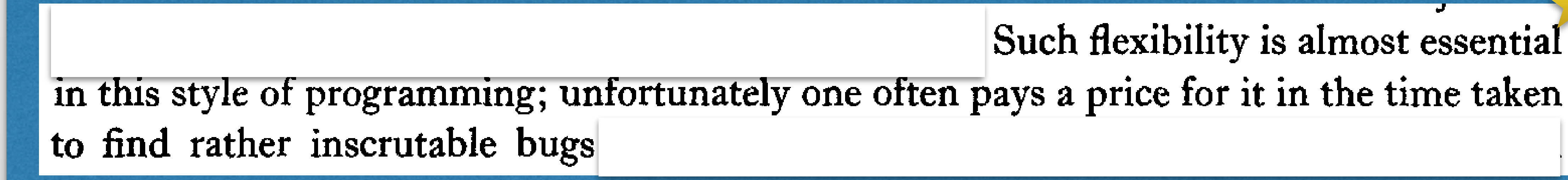
Frank Pfenning



A widely employed style of programming, which impose no discipline of types

lers to accept

case x : nat of



Such flexibility is almost essential in this style of programming; unfortunately one often pays a price for it in the time taken to find rather inscrutable bugs

if we know that x will never be Zero.

★ **Well, actually Milner [1978] said that (about LISP) too**

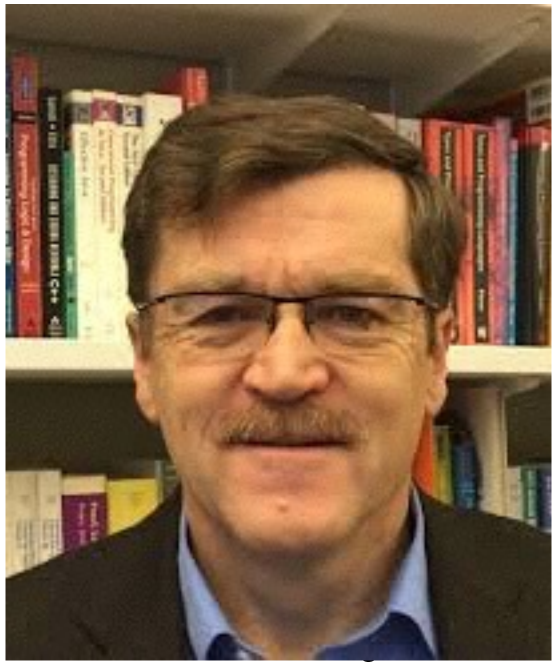
Adapted from Jafery and Dunfield [POPL17]

Refined Standard ML

Datasort refinements [Freeman & Pfenning 1991, Davies 2005, . . .] push the knowledge that x is not `Zero` into the type system.

```
case x : nonzero of
   Succ y ⇒ . . .
```

This **is** exhaustive, because x has **datasort** nonzero.



Frank Pfenning

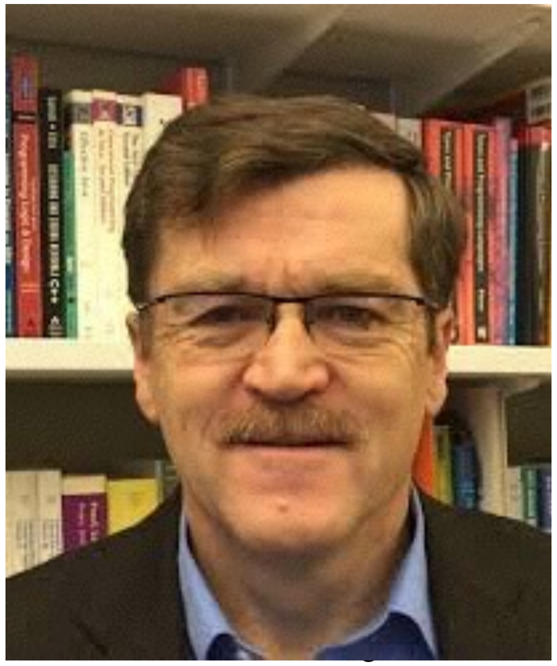
Adapted from Jafery and Dunfield [POPL17]

Refined Standard ML

Datasort refinements [Freeman & Pfenning 1991, Davies 2005, . . .] push the knowledge that x is not `Zero` into the type system.

```
case x : nonzero of
  Succ y ⇒ ...
```

This **is** exhaustive, because x has **datasort** nonzero.



Frank Pfenning

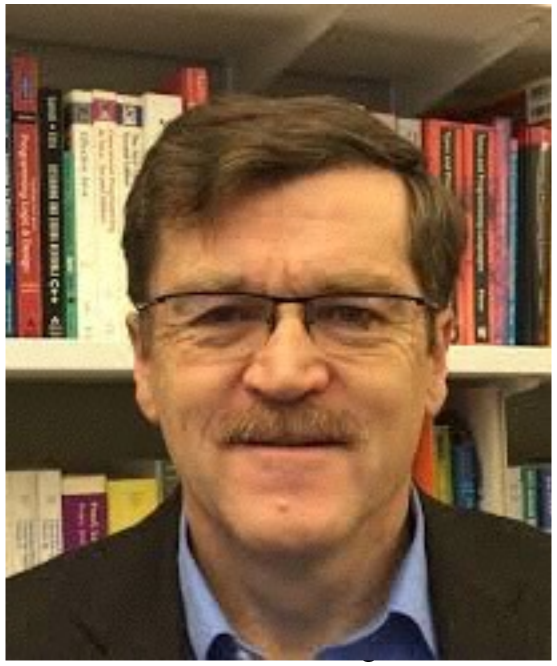
Adapted from Jafery and Dunfield [POPL17]

Refined Standard ML

Datasort refinements [Freeman & Pfenning 1991, Davies 2005, . . .] push the knowledge that x is not `Zero` into the type system.

```
case x : nonzero of
  Succ y ⇒ . . .
```

This **is** exhaustive, because x has **datasort** nonzero.



Frank Pfenning

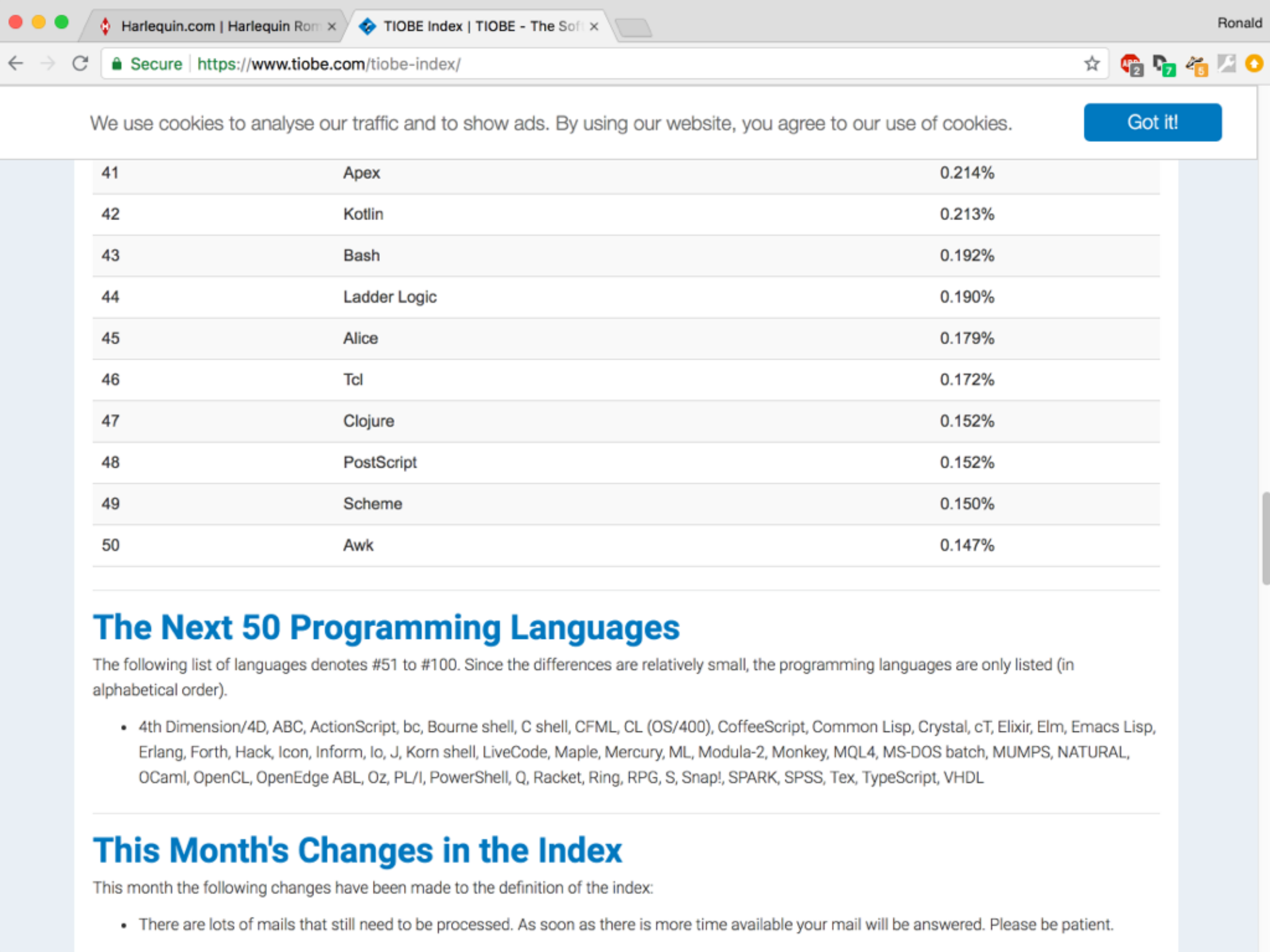
Adapted from Jafery and Dunfield [POPL17]

Outline

STILL

- Motivating Example (In Two Acts)
- Gradual Typing For All!
- Typing in Small Pieces
- Meat
- Strands and Related Works

Motivating Example Act 2: Adoption



We use cookies to analyse our traffic and to show ads. By using our website, you agree to our use of cookies.

41	Apex	0.214%
42	Kotlin	0.213%
43	Bash	0.192%
44	Ladder Logic	0.190%
45	Alice	0.179%
46	Tcl	0.172%
47	Clojure	0.152%
48	PostScript	0.152%
49	Scheme	0.150%
50	Awk	0.147%

The Next 50 Programming Languages

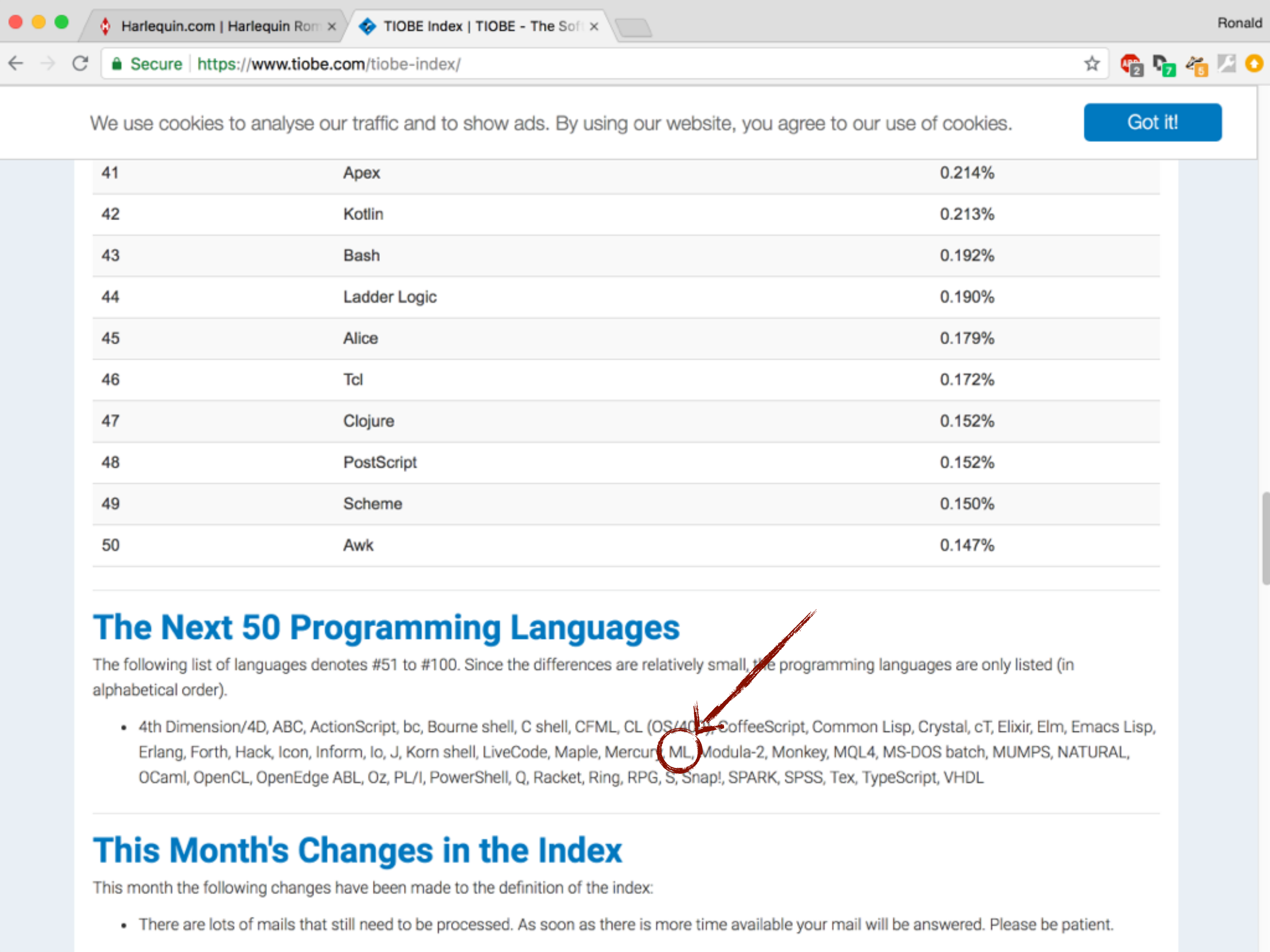
The following list of languages denotes #51 to #100. Since the differences are relatively small, the programming languages are only listed (in alphabetical order).

- 4th Dimension/4D, ABC, ActionScript, bc, Bourne shell, C shell, CFML, CL (OS/400), CoffeeScript, Common Lisp, Crystal, cT, Elixir, Elm, Emacs Lisp, Erlang, Forth, Hack, Icon, Inform, Io, J, Korn shell, LiveCode, Maple, Mercury, ML, Modula-2, Monkey, MQL4, MS-DOS batch, MUMPS, NATURAL, OCaml, OpenCL, OpenEdge ABL, Oz, PL/I, PowerShell, Q, Racket, Ring, RPG, S, Snap!, SPARK, SPSS, Tex, TypeScript, VHDL

This Month's Changes in the Index

This month the following changes have been made to the definition of the index:

- There are lots of mails that still need to be processed. As soon as there is more time available your mail will be answered. Please be patient.



We use cookies to analyse our traffic and to show ads. By using our website, you agree to our use of cookies.

41	Apex	0.214%
42	Kotlin	0.213%
43	Bash	0.192%
44	Ladder Logic	0.190%
45	Alice	0.179%
46	Tcl	0.172%
47	Clojure	0.152%
48	PostScript	0.152%
49	Scheme	0.150%
50	Awk	0.147%

The Next 50 Programming Languages

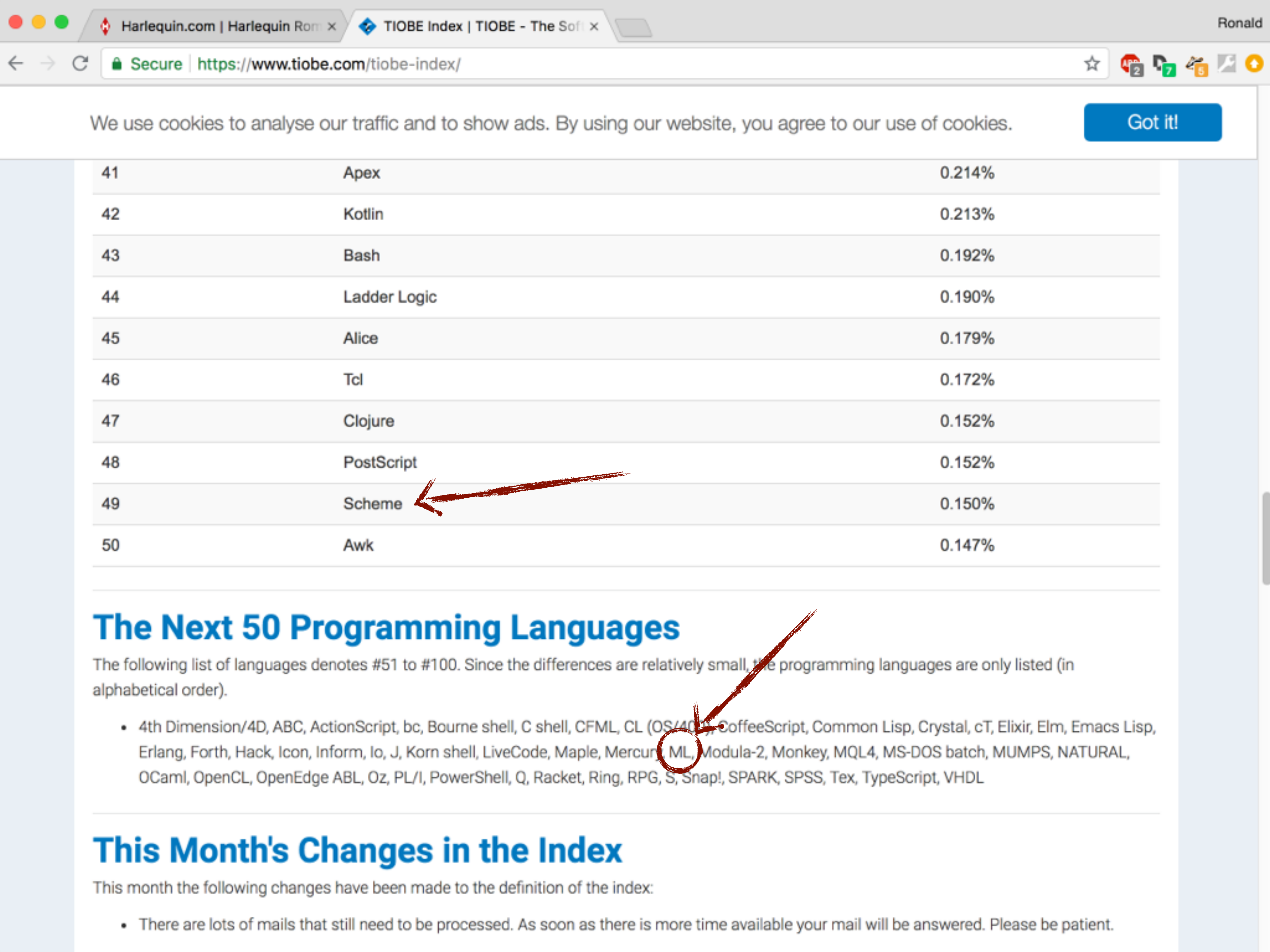
The following list of languages denotes #51 to #100. Since the differences are relatively small, the programming languages are only listed (in alphabetical order).

- 4th Dimension/4D, ABC, ActionScript, bc, Bourne shell, C shell, CFML, CL (OS/400), CoffeeScript, Common Lisp, Crystal, cT, Elixir, Elm, Emacs Lisp, Erlang, Forth, Hack, Icon, Inform, Io, J, Korn shell, LiveCode, Maple, Mercury, ML, Modula-2, Monkey, MQL4, MS-DOS batch, MUMPS, NATURAL, OCaml, OpenCL, OpenEdge ABL, Oz, PL/I, PowerShell, Q, Racket, Ring, RPG, S, Snap!, SPARK, SPSS, Tex, TypeScript, VHDL

This Month's Changes in the Index

This month the following changes have been made to the definition of the index:

- There are lots of mails that still need to be processed. As soon as there is more time available your mail will be answered. Please be patient.



We use cookies to analyse our traffic and to show ads. By using our website, you agree to our use of cookies.

41	Apex	0.214%
42	Kotlin	0.213%
43	Bash	0.192%
44	Ladder Logic	0.190%
45	Alice	0.179%
46	Tcl	0.172%
47	Clojure	0.152%
48	PostScript	0.152%
49	Scheme	0.150%
50	Awk	0.147%

The Next 50 Programming Languages

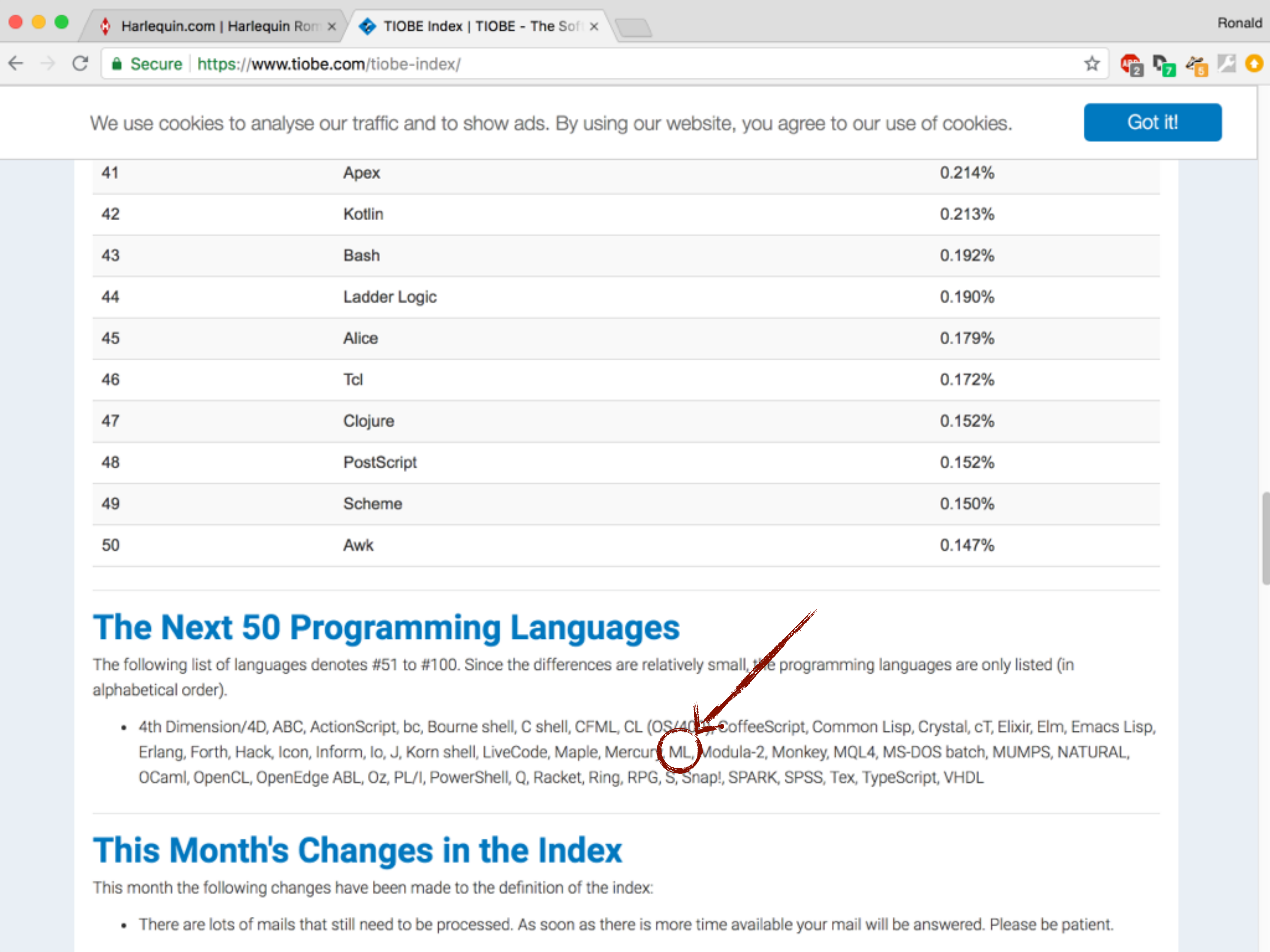
The following list of languages denotes #51 to #100. Since the differences are relatively small, the programming languages are only listed (in alphabetical order).

- 4th Dimension/4D, ABC, ActionScript, bc, Bourne shell, C shell, CFML, CL (OS/400), CoffeeScript, Common Lisp, Crystal, cT, Elixir, Elm, Emacs Lisp, Erlang, Forth, Hack, Icon, Inform, Io, J, Korn shell, LiveCode, Maple, Mercury, ML, Modula-2, Monkey, MQL4, MS-DOS batch, MUMPS, NATURAL, OCaml, OpenCL, OpenEdge ABL, Oz, PL/I, PowerShell, Q, Racket, Ring, RPG, S, Snap!, SPARK, SPSS, Tex, TypeScript, VHDL

This Month's Changes in the Index

This month the following changes have been made to the definition of the index:

- There are lots of mails that still need to be processed. As soon as there is more time available your mail will be answered. Please be patient.



We use cookies to analyse our traffic and to show ads. By using our website, you agree to our use of cookies.

41	Apex	0.214%
42	Kotlin	0.213%
43	Bash	0.192%
44	Ladder Logic	0.190%
45	Alice	0.179%
46	Tcl	0.172%
47	Clojure	0.152%
48	PostScript	0.152%
49	Scheme	0.150%
50	Awk	0.147%

The Next 50 Programming Languages

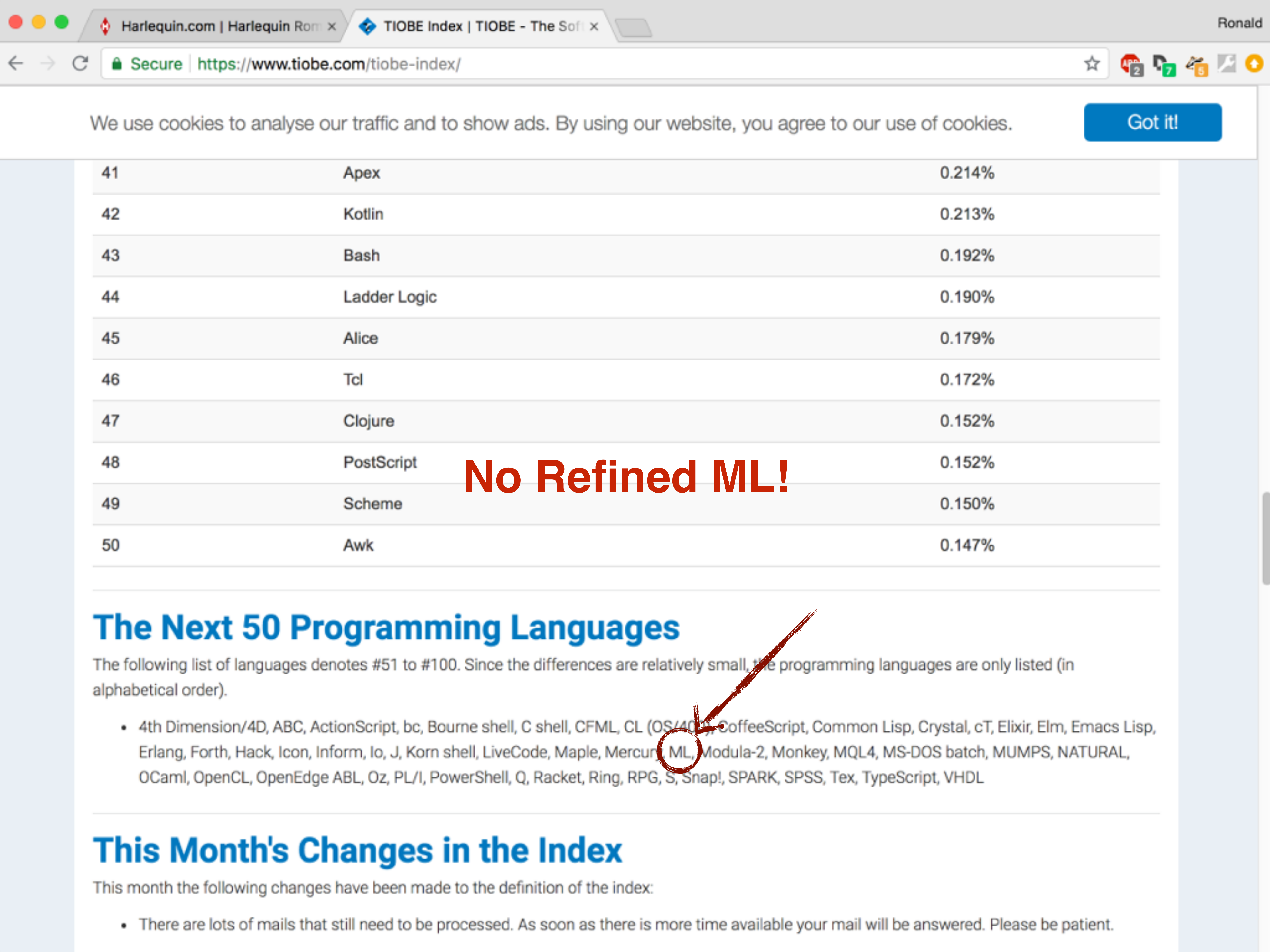
The following list of languages denotes #51 to #100. Since the differences are relatively small, the programming languages are only listed (in alphabetical order).

- 4th Dimension/4D, ABC, ActionScript, bc, Bourne shell, C shell, CFML, CL (OS/400), CoffeeScript, Common Lisp, Crystal, cT, Elixir, Elm, Emacs Lisp, Erlang, Forth, Hack, Icon, Inform, Io, J, Korn shell, LiveCode, Maple, Mercury, ML, Modula-2, Monkey, MQL4, MS-DOS batch, MUMPS, NATURAL, OCaml, OpenCL, OpenEdge ABL, Oz, PL/I, PowerShell, Q, Racket, Ring, RPG, S, Snap!, SPARK, SPSS, Tex, TypeScript, VHDL

This Month's Changes in the Index

This month the following changes have been made to the definition of the index:

- There are lots of mails that still need to be processed. As soon as there is more time available your mail will be answered. Please be patient.



We use cookies to analyse our traffic and to show ads. By using our website, you agree to our use of cookies.

41	Apex	0.214%
42	Kotlin	0.213%
43	Bash	0.192%
44	Ladder Logic	0.190%
45	Alice	0.179%
46	Tcl	0.172%
47	Clojure	0.152%
48	PostScript	0.152%
49	Scheme	0.150%
50	Awk	0.147%

No Refined ML!

The Next 50 Programming Languages

The following list of languages denotes #51 to #100. Since the differences are relatively small, the programming languages are only listed (in alphabetical order).

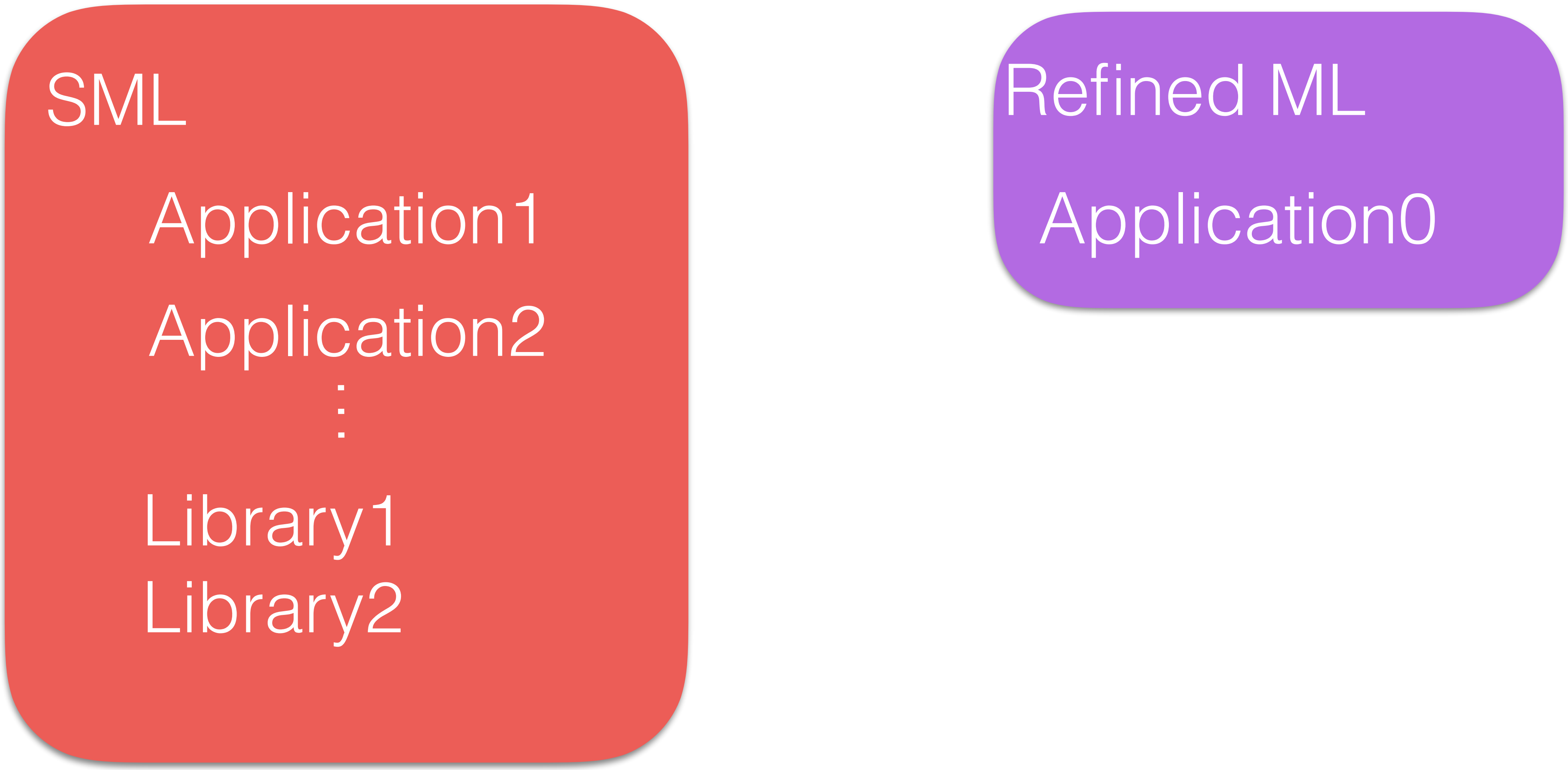
- 4th Dimension/4D, ABC, ActionScript, bc, Bourne shell, C shell, CFML, CL (OS/400), CoffeeScript, Common Lisp, Crystal, cT, Elixir, Elm, Emacs Lisp, Erlang, Forth, Hack, Icon, Inform, Io, J, Korn shell, LiveCode, Maple, Mercury, ML, Modula-2, Monkey, MQL4, MS-DOS batch, MUMPS, NATURAL, OCaml, OpenCL, OpenEdge ABL, Oz, PL/I, PowerShell, Q, Racket, Ring, RPG, S, Snap!, SPARK, SPSS, Tex, TypeScript, VHDL

This Month's Changes in the Index

This month the following changes have been made to the definition of the index:

- There are lots of mails that still need to be processed. As soon as there is more time available your mail will be answered. Please be patient.

Paucity of RML Code



SML

Application1

Application2

⋮

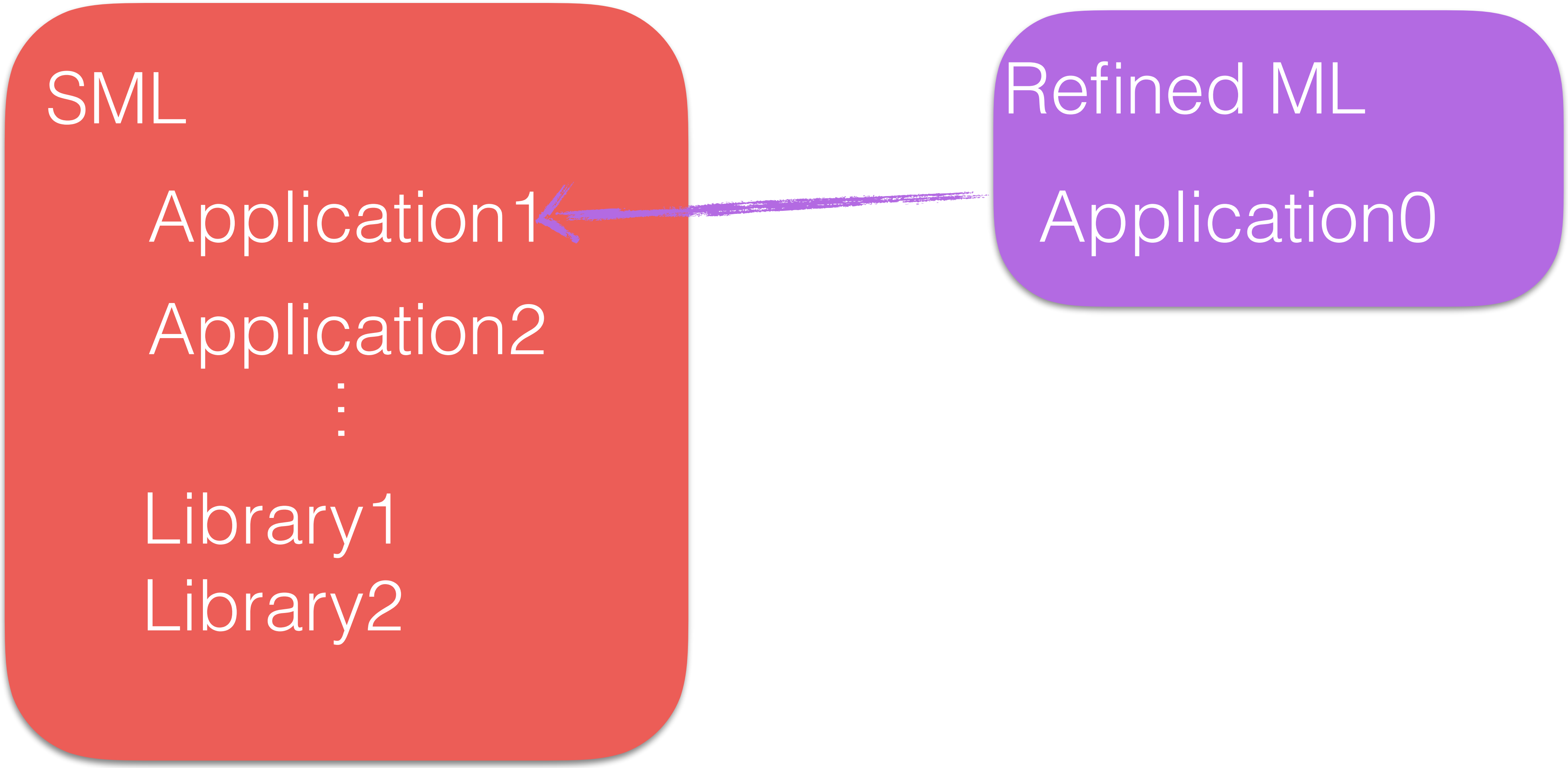
Library1

Library2

Refined ML

Application0

*Figures not drawn to scale



SML

Application1

Application2

⋮

Library1

Library2

Refined ML

Application0

*Figures not drawn to scale

SML

Application1

Application2

⋮

Library1

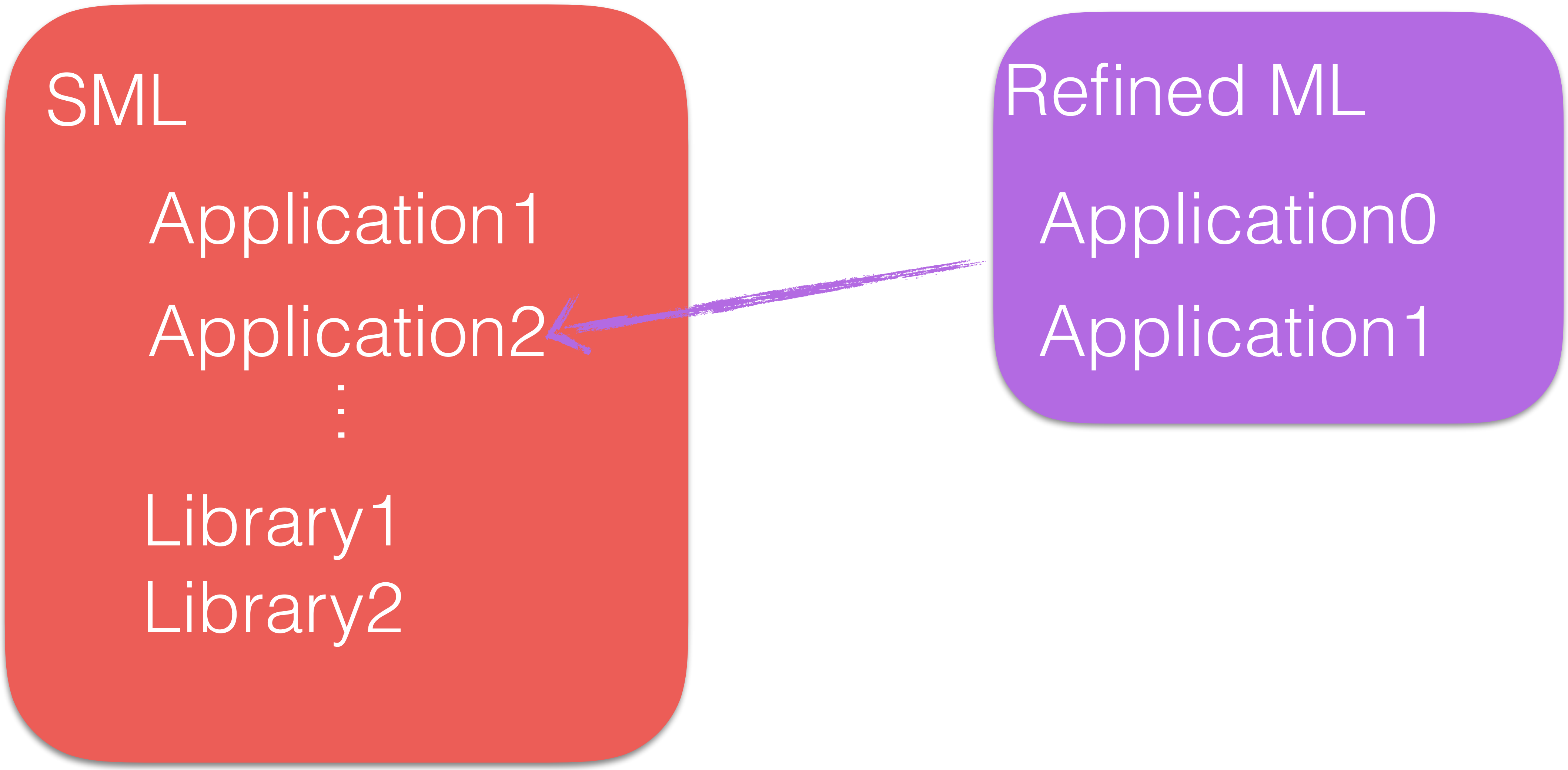
Library2

Refined ML

Application0

Application1

*Figures not drawn to scale



SML

Application1

Application2

⋮

Library1

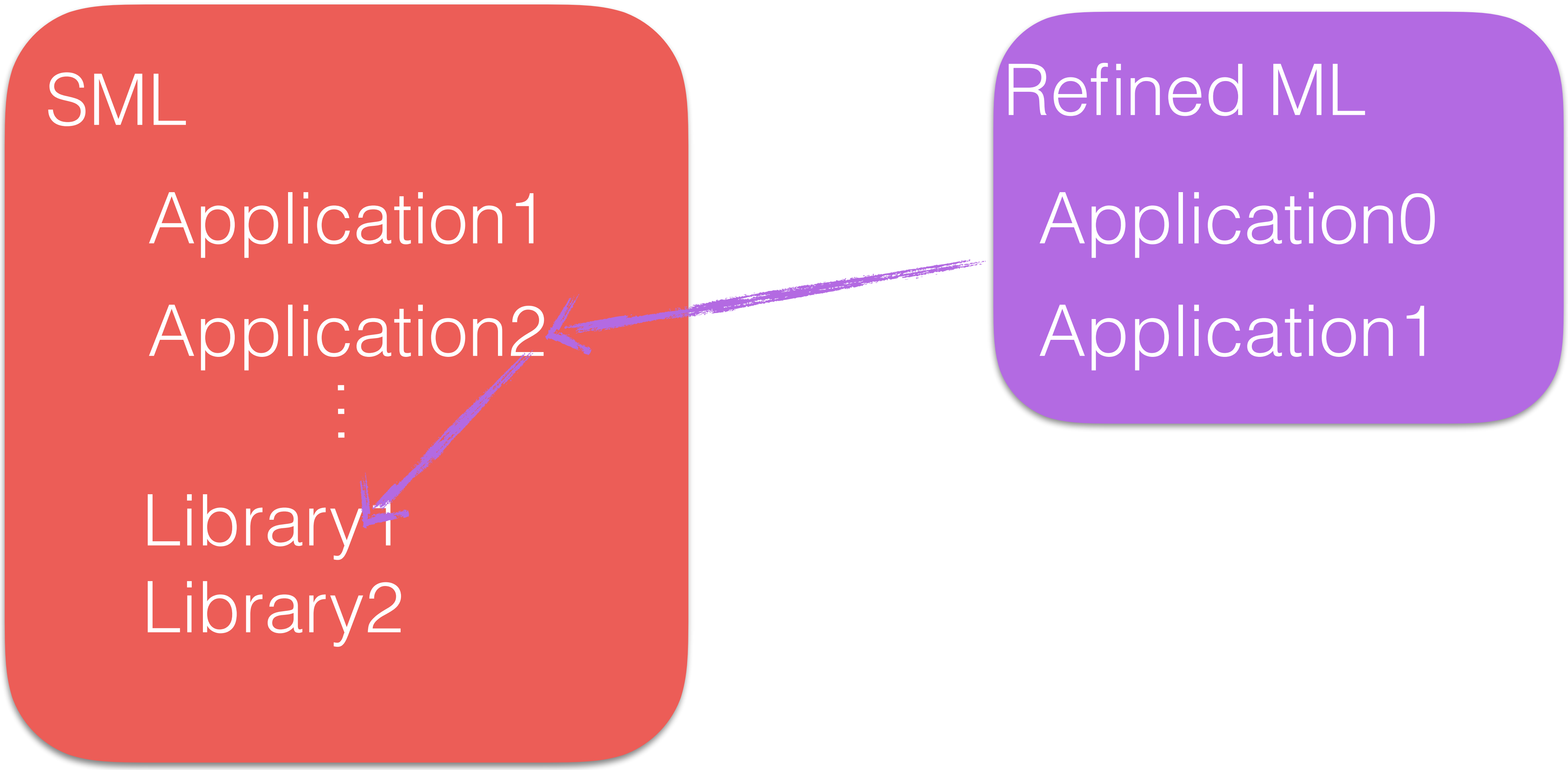
Library2

Refined ML

Application0

Application1

*Figures not drawn to scale



SML

Application1

Application2

⋮

Library1

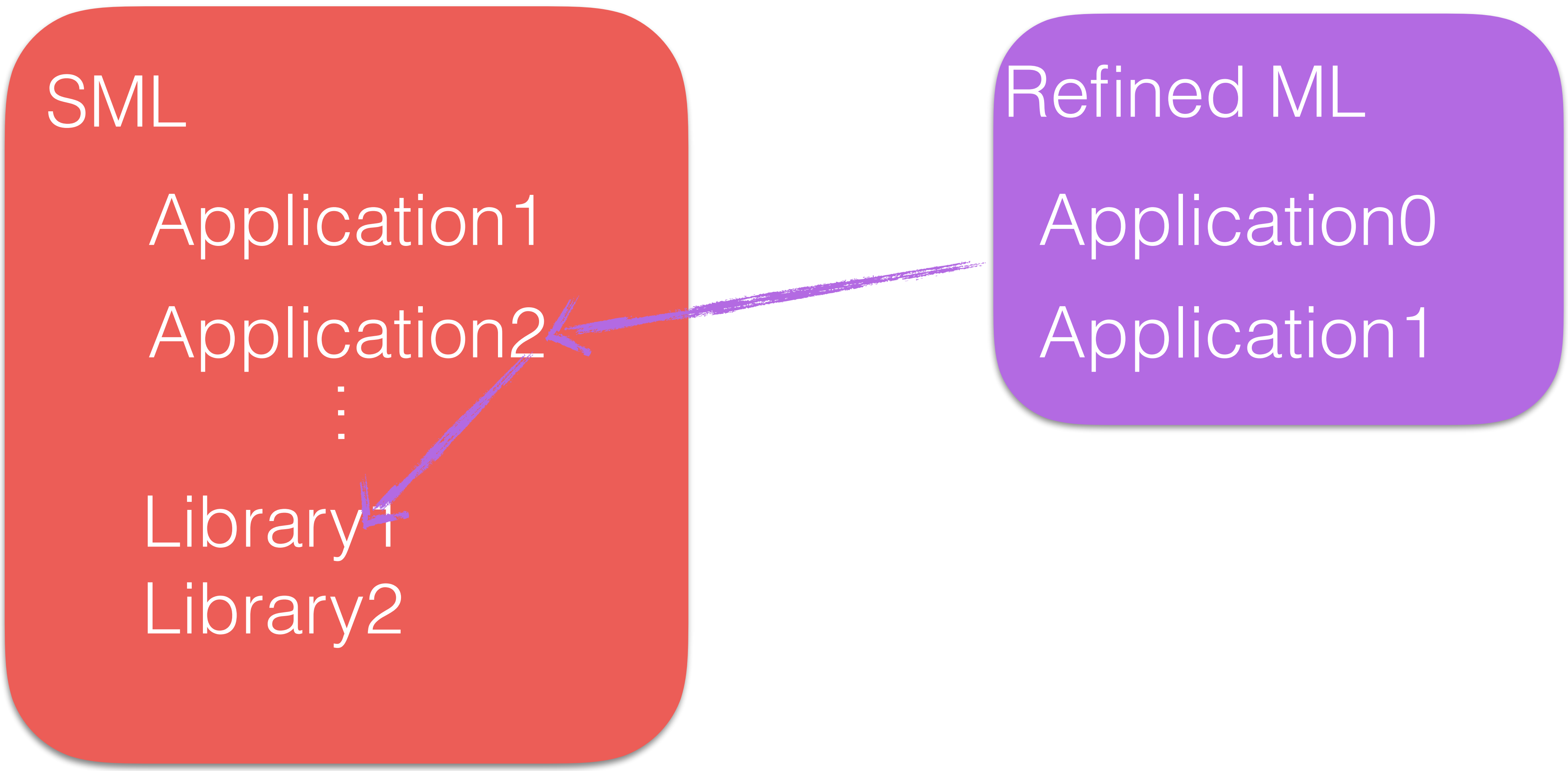
Library2

Refined ML

Application0

Application1

*Figures not drawn to scale



SML

Application1

Application2

⋮

Library1

Library2

Refined ML

Application0

Application1

*Figures not drawn to scale

Wholesale Migration?!?

SML

App

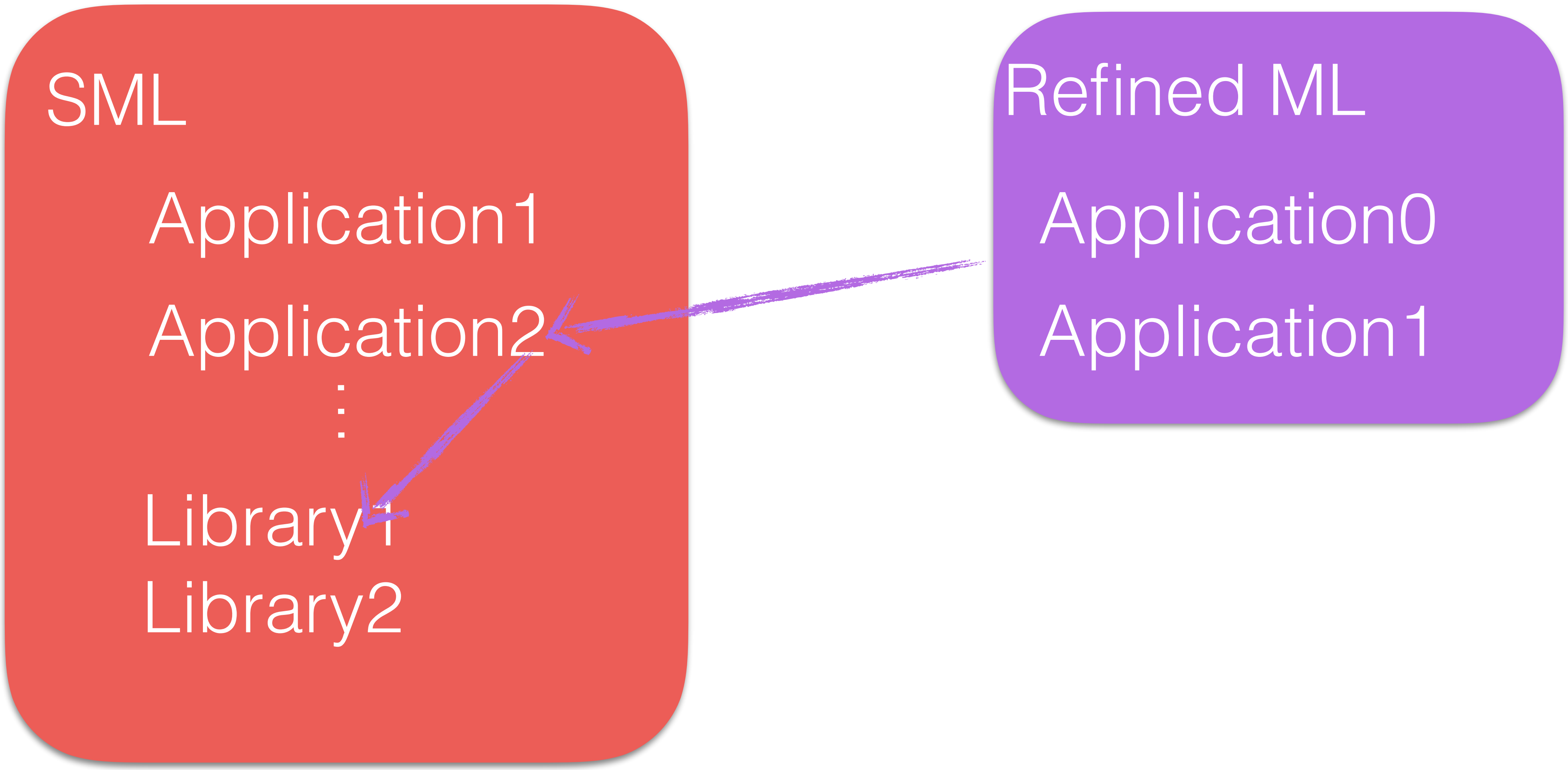
App

Libr

Libr

*Figures not drawn to scale

Wholesale Migration?!?



SML

Application1

Application2

⋮

Library1

Library2

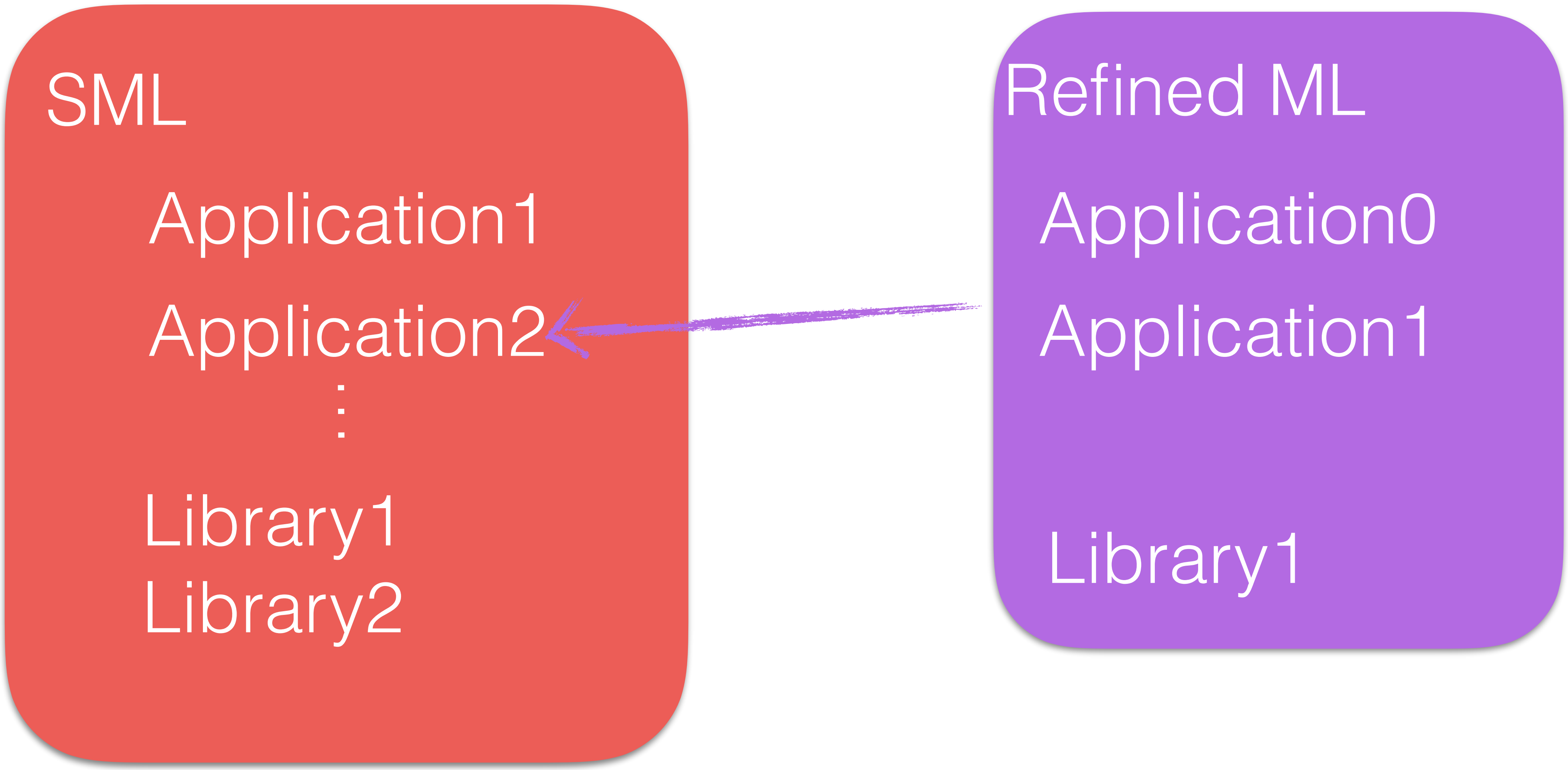
Refined ML

Application0

Application1

*Figures not drawn to scale

Wholesale Migration?!?



SML

Application1

Application2

⋮

Library1

Library2

Refined ML

Application0

Application1

Library1

*Figures not drawn to scale

Wholesale Migration?!?

SML

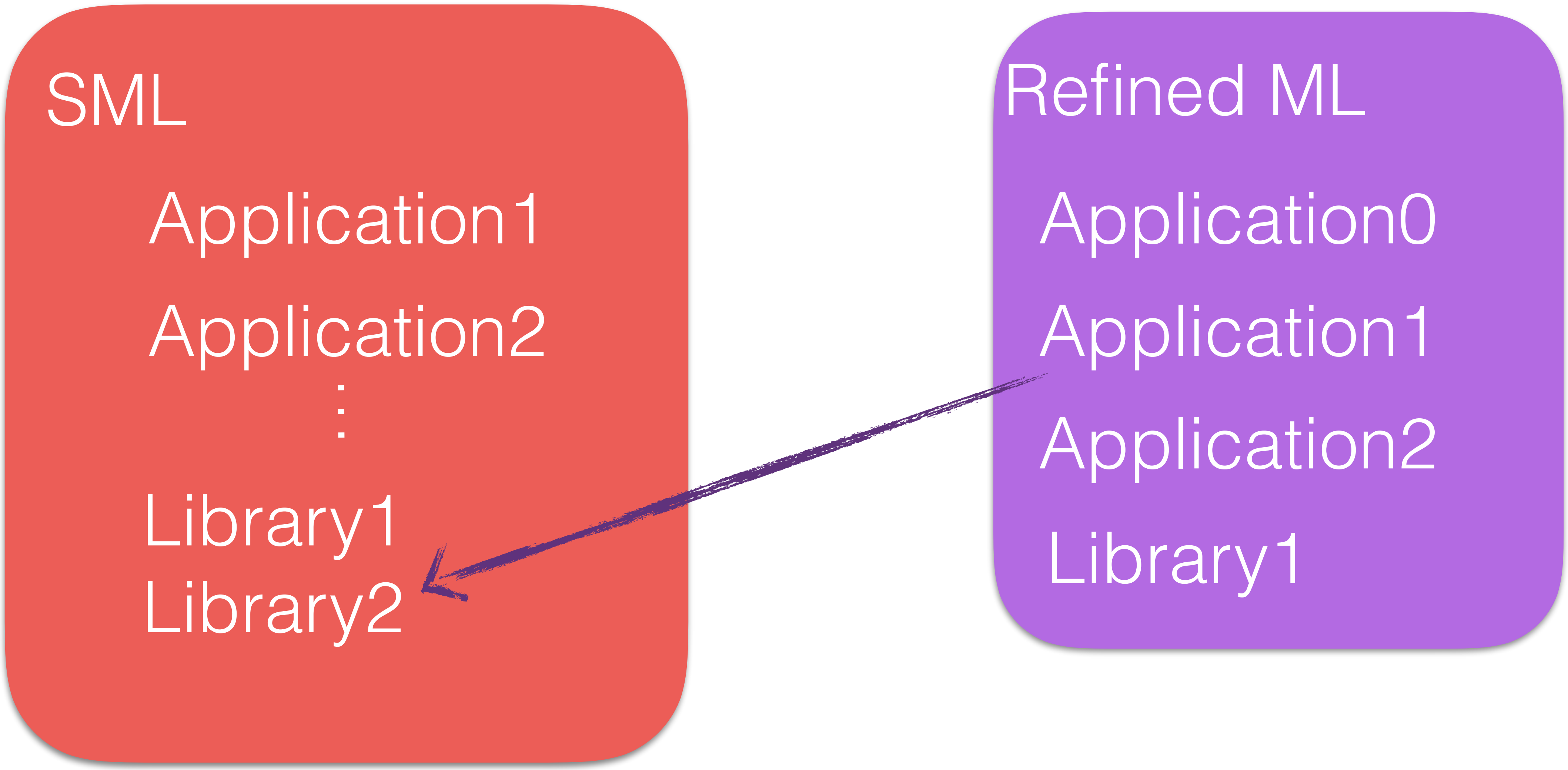
- Application1
- Application2
- ⋮
- Library1
- Library2

Refined ML

- Application0
- Application1
- Application2
- Library1

*Figures not drawn to scale

Wholesale Migration?!?



SML

Application1

Application2

⋮

Library1

Library2

Refined ML

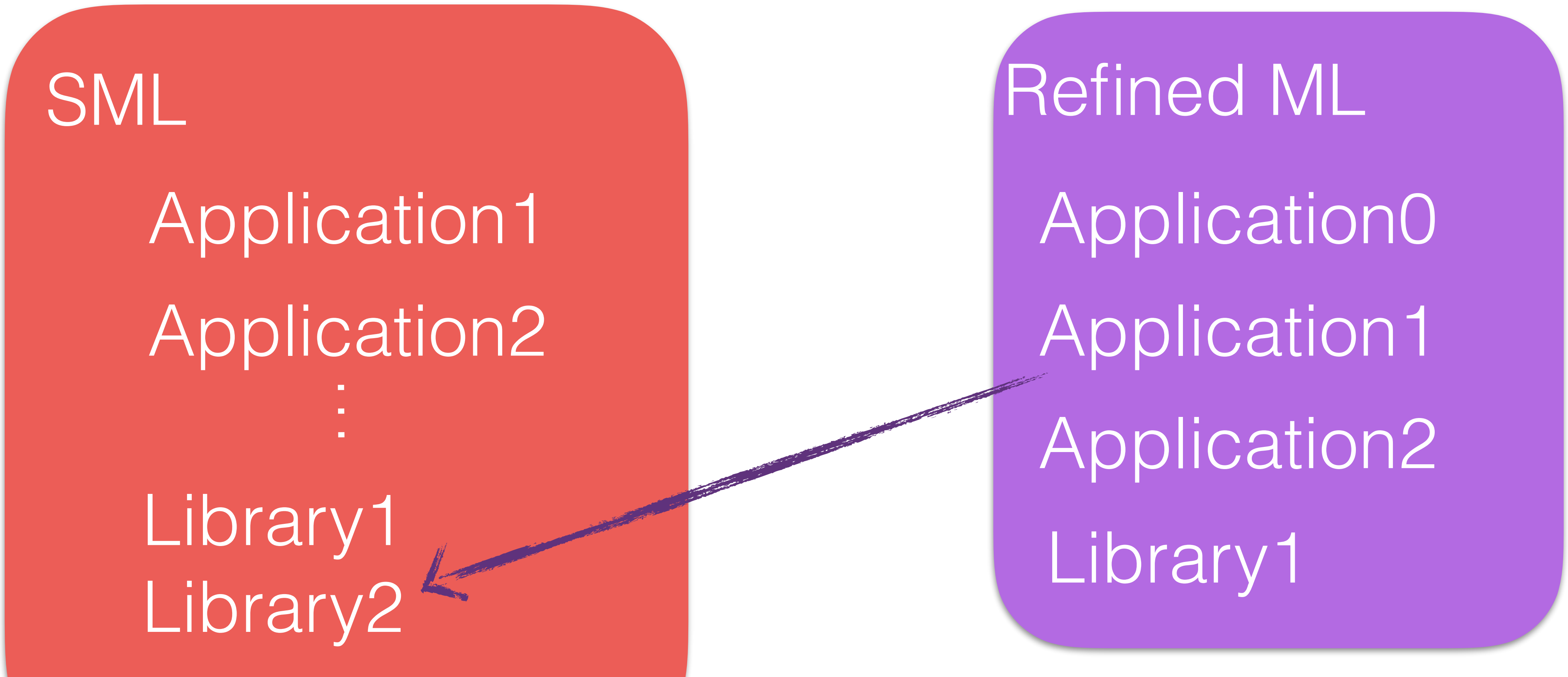
Application0

Application1

Application2

Library1

*Figures not drawn to scale



SML

Application1

Application2

⋮

Library1

Library2

Refined ML

Application0

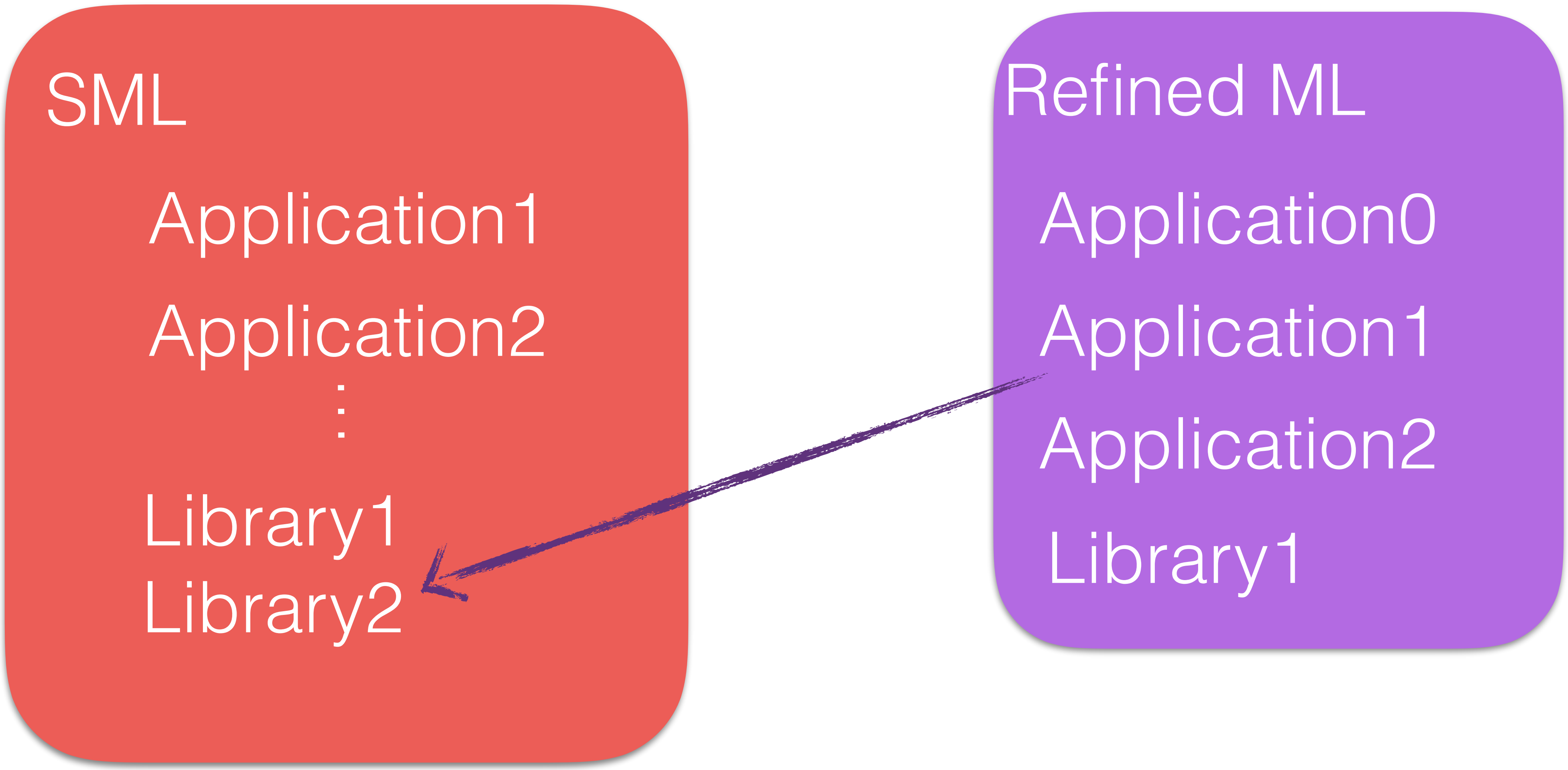
Application1

Application2

Library1

*Figures not drawn to scale

Must We Assimilate?



SML

Application1

Application2

⋮

Library1

Library2

Refined ML

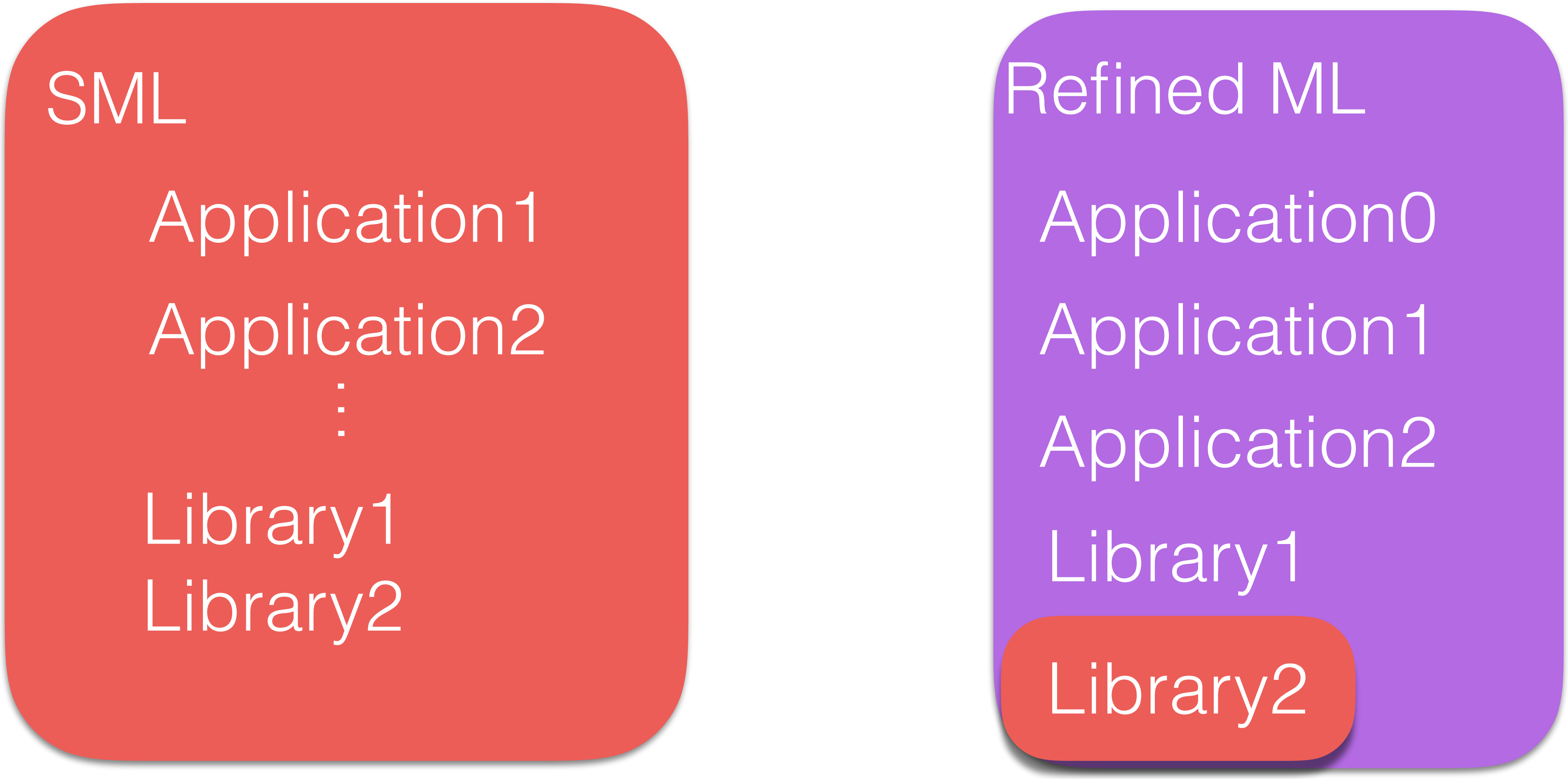
Application0

Application1

Application2

Library1

*Figures not drawn to scale



SML

Application1

Application2

⋮

Library1

Library2

Refined ML

Application0

Application1

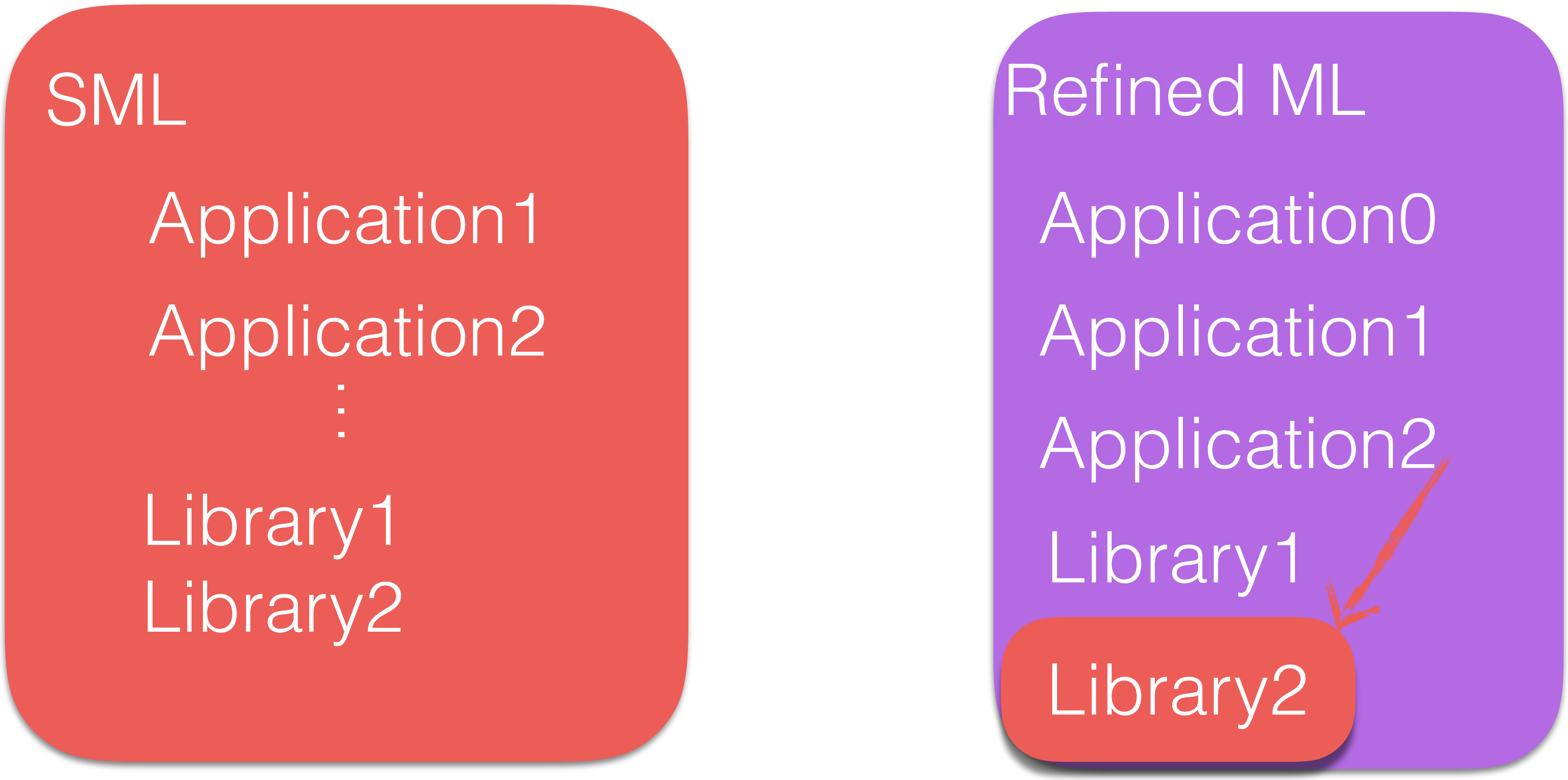
Application2

Library1

Library2

Gradual Migration

*Figures not drawn to scale



SML

Application1

Application2

⋮

Library1

Library2

Refined ML

Application0

Application1

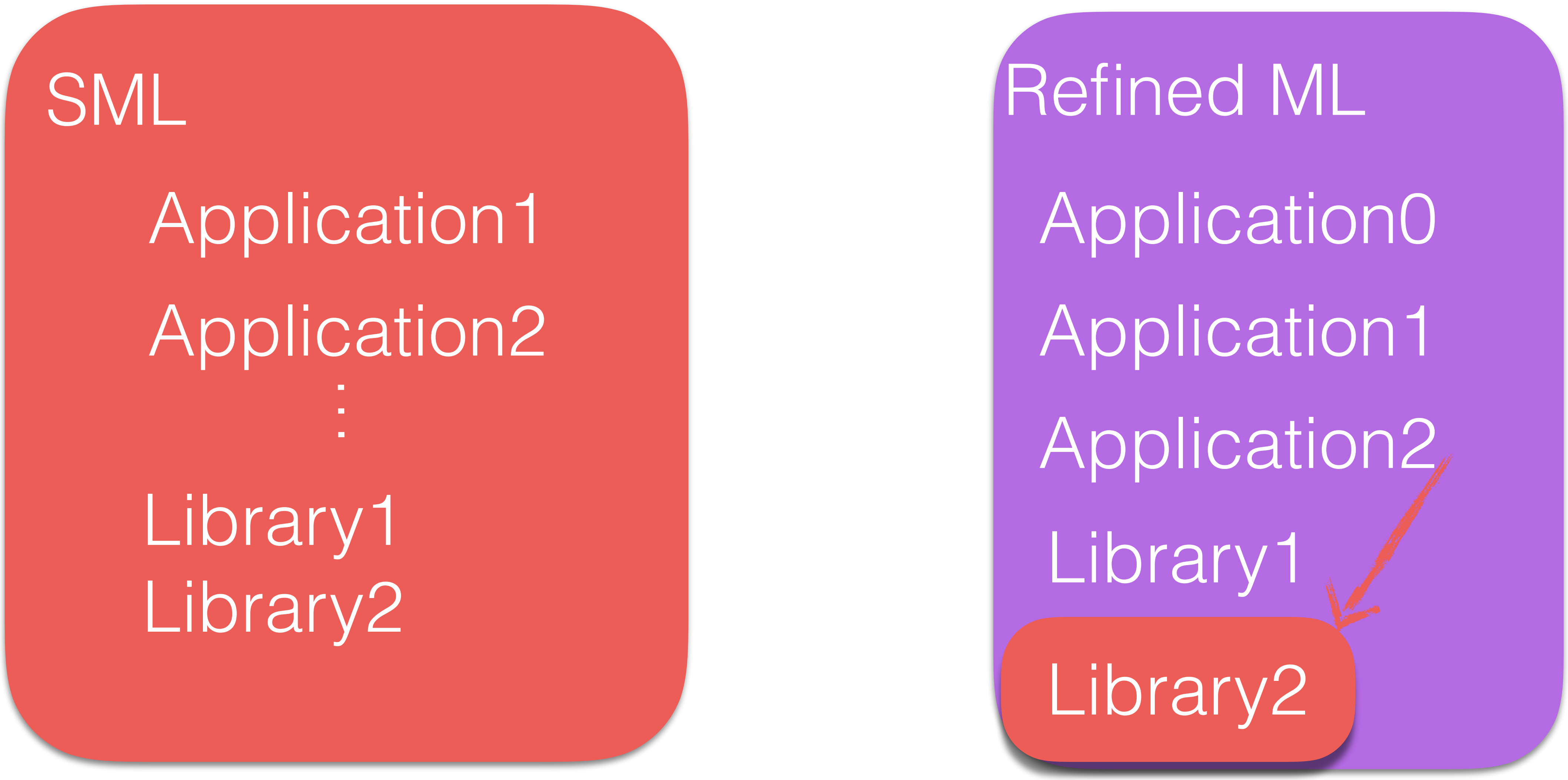
Application2

Library1

Library2

Gradual Migration

*Figures not drawn to scale



SML

Application1

Application2

⋮

Library1

Library2

Refined ML

Application0

Application1

Application2

Library1

Library2

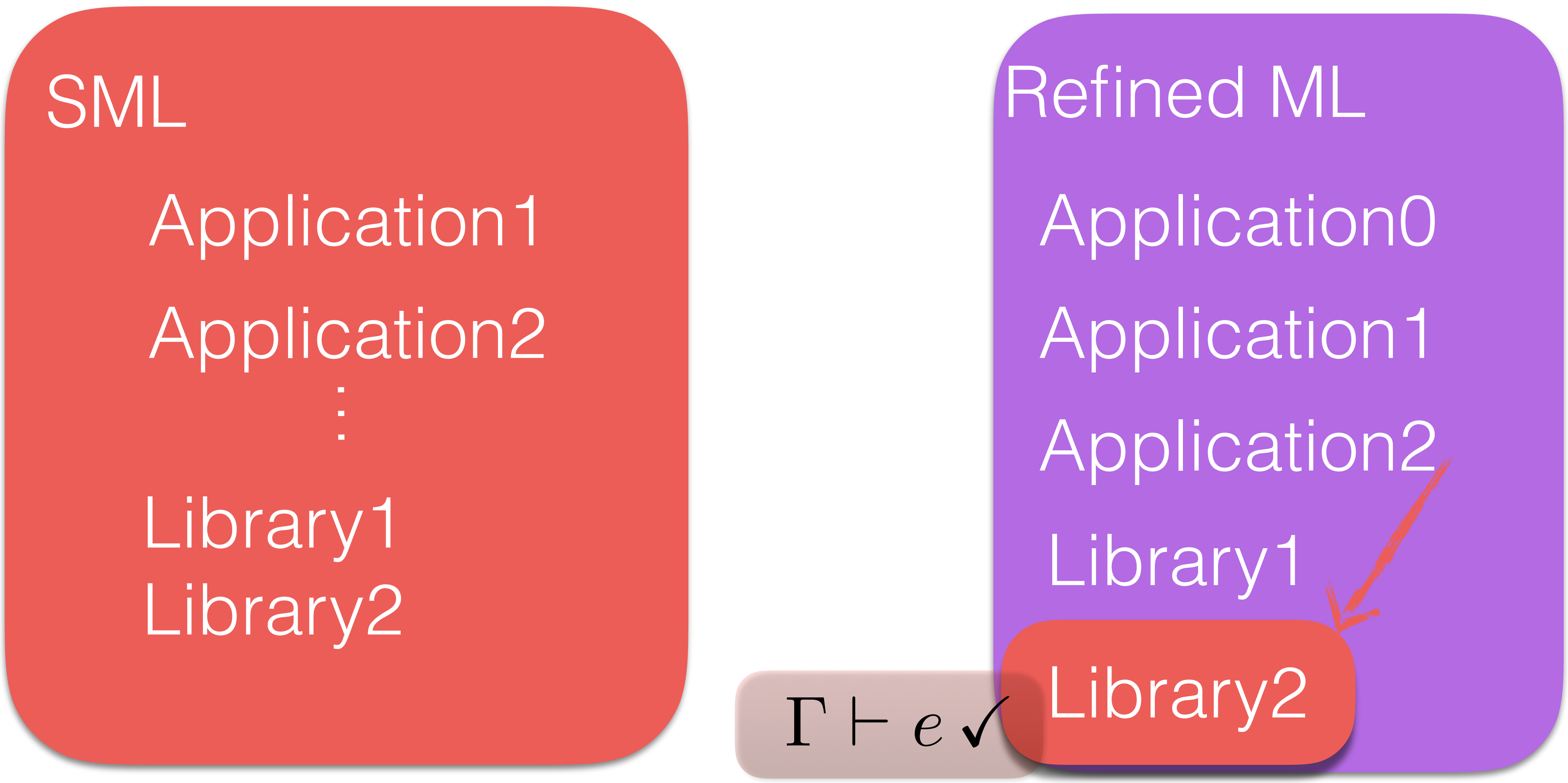
Gradual Migration

SML Code (& Guarantees)

Refined ML Code (& Guarantees)

Interoperating!

*Figures not drawn to scale



SML

Application1

Application2

⋮

Library1

Library2

Refined ML

Application0

Application1

Application2

Library1

Library2

$\Gamma \vdash e$ ✓

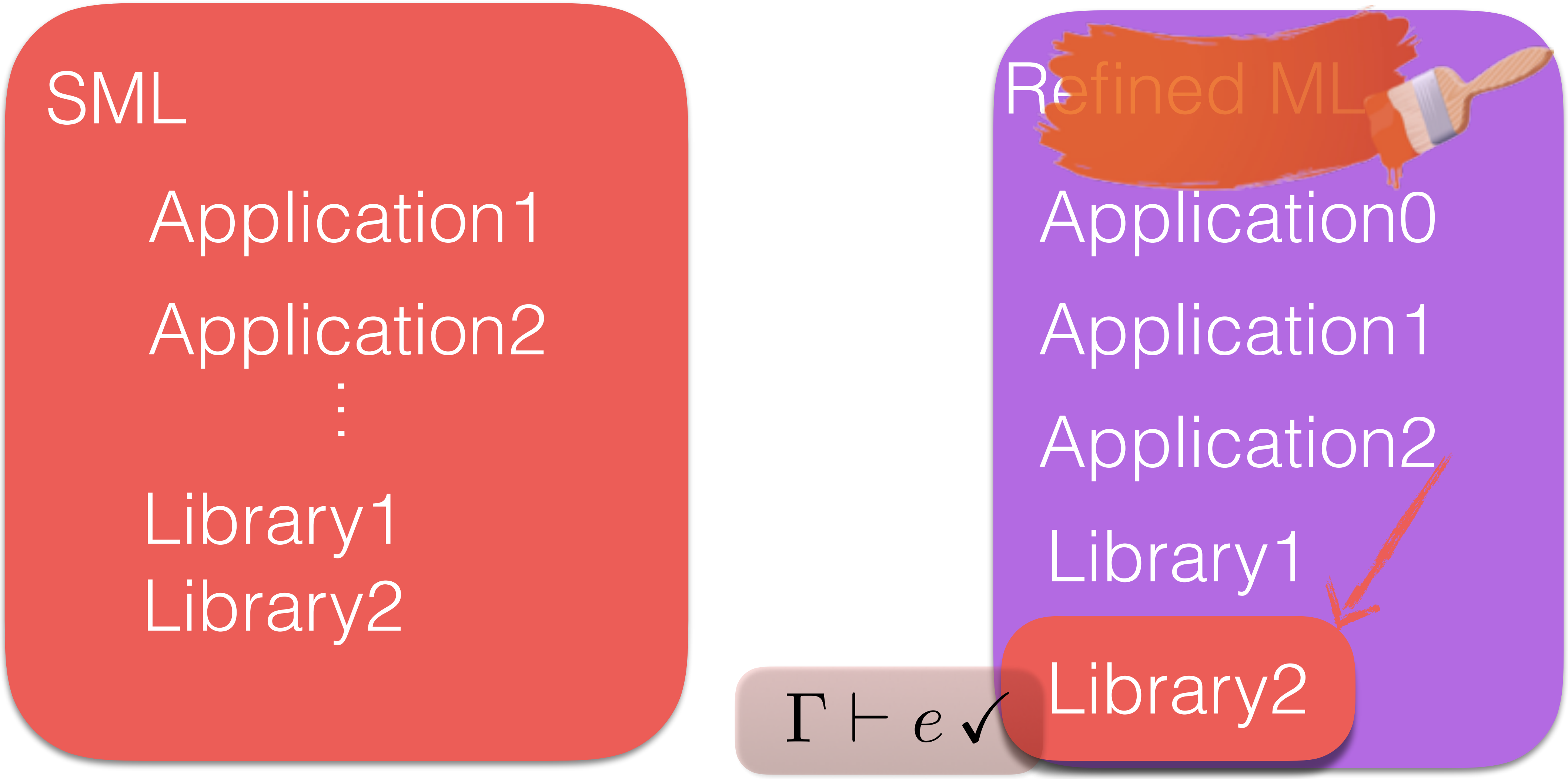
Gradual Migration

SML Code (& Guarantees)

Refined ML Code (& Guarantees)

Interoperating!

*Figures not drawn to scale



SML

Application1

Application2

⋮

Library1

Library2

Refined ML

Application0

Application1

Application2

Library1

Library2

$\Gamma \vdash e$ ✓

Gradual Migration

SML Code (& Guarantees)

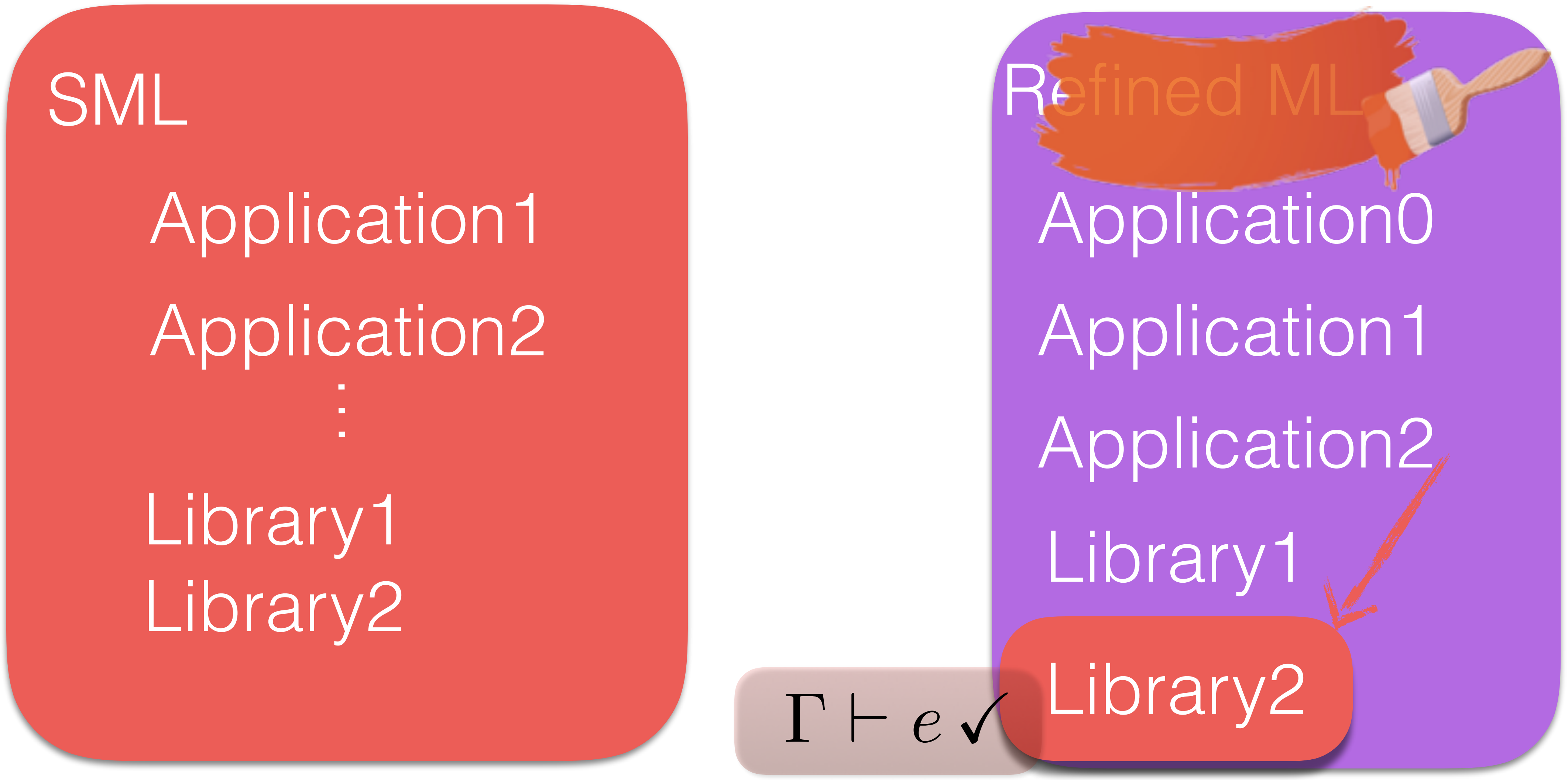
Refined ML Code (& Guarantees)

Interoperating!

Free!

*Figures not drawn to scale

“Optional Typing”



SML

Application1

Application2

⋮

Library1

Library2

Refined ML

Application0

Application1

Application2

Library1

Library2

$\Gamma \vdash e$ ✓

Gradual Migration

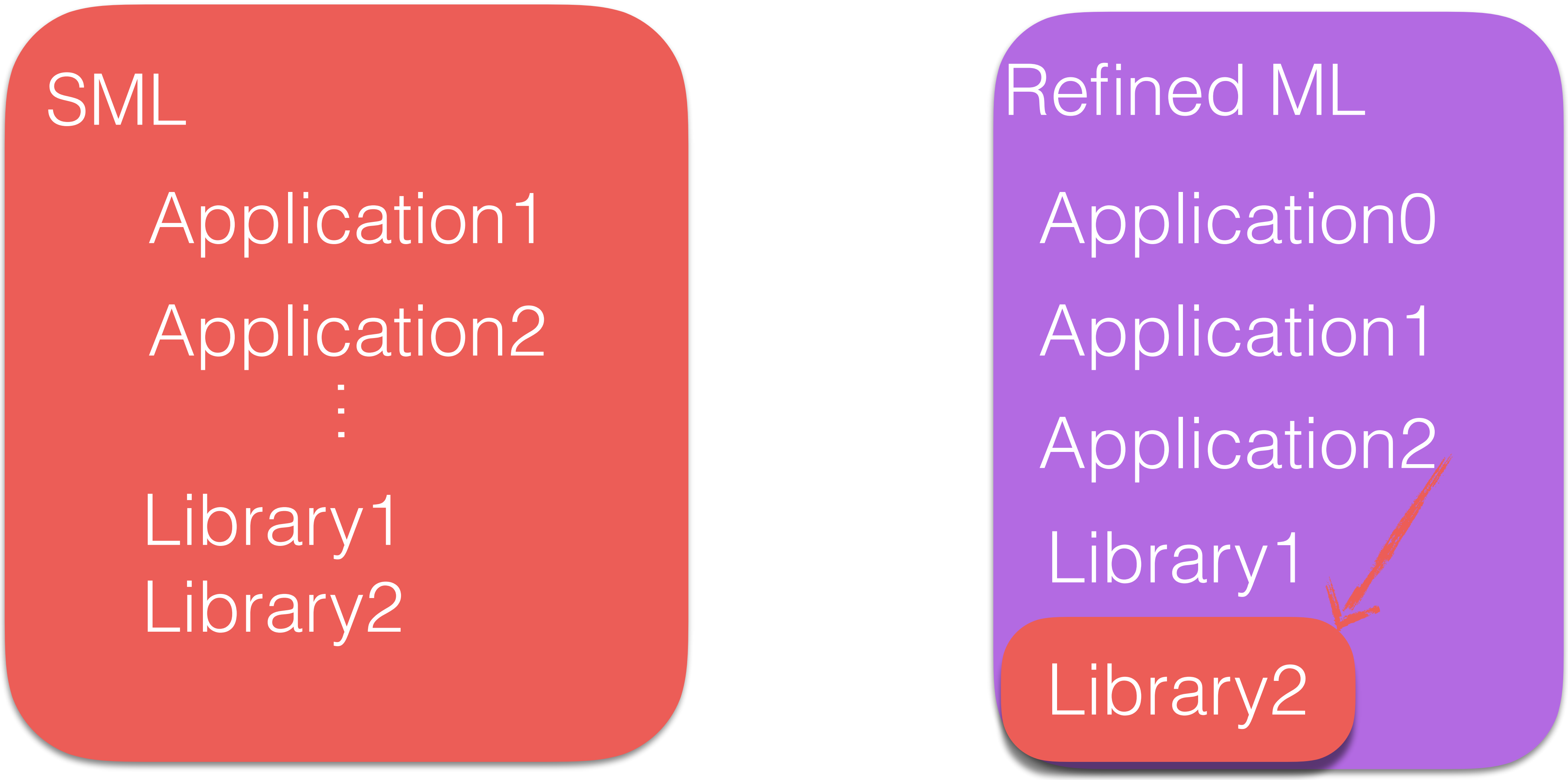
SML Code (& Guarantees)

Free!

Refined ML Code (& Guarantees)

Interoperating!

*Figures not drawn to scale



SML

Application1

Application2

⋮

Library1

Library2

Refined ML

Application0

Application1

Application2

Library1

Library2

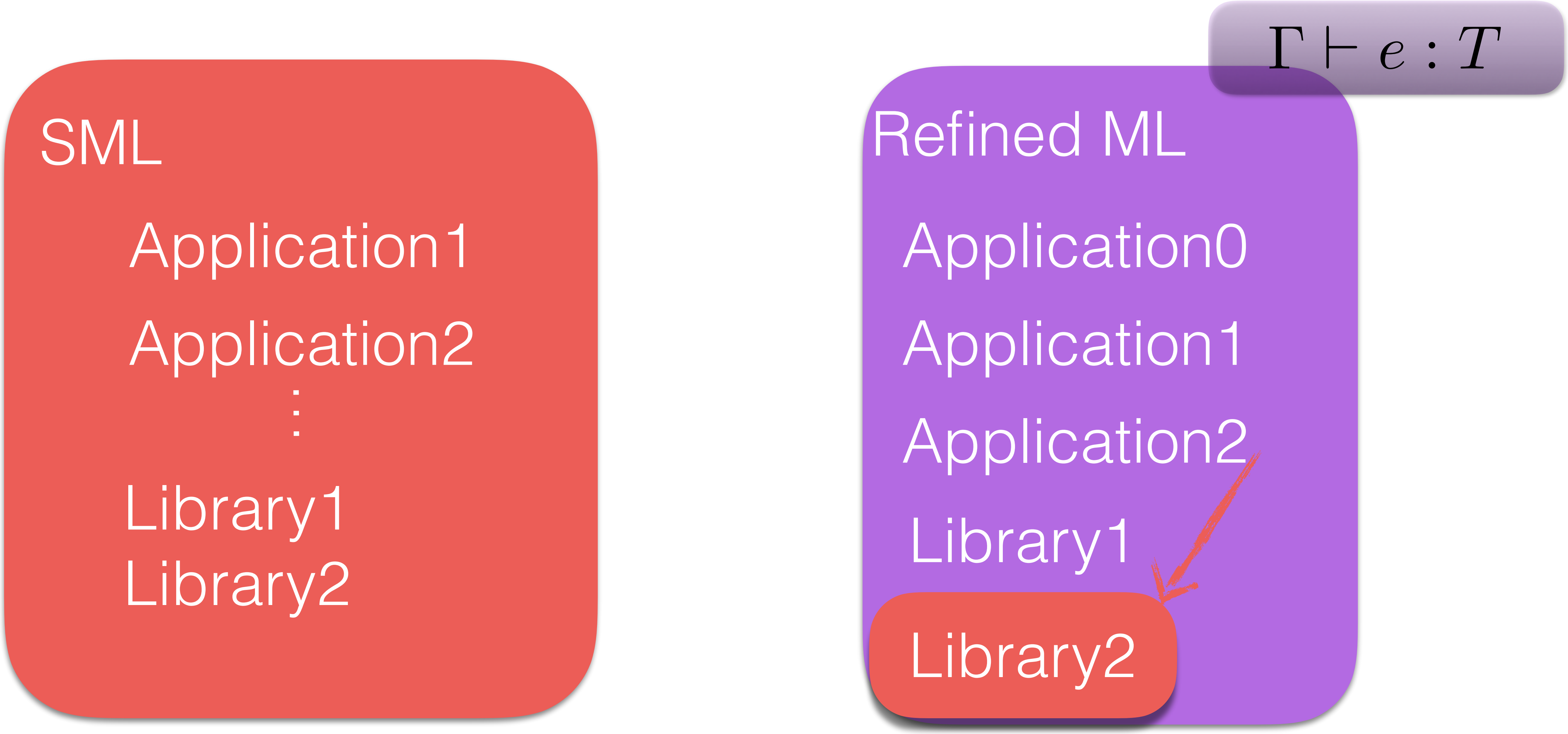
Gradual Migration

SML Code (& Guarantees)

Refined ML Code (& Guarantees)

Interoperating!

*Figures not drawn to scale



SML

Application1

Application2

⋮

Library1

Library2

$\Gamma \vdash e : T$

Refined ML

Application0

Application1

Application2

Library1

Library2

Gradual Migration

SML Code (& Guarantees)

Refined ML Code (& Guarantees)

Interoperating!

*Figures not drawn to scale

SML

Application1

Application2

⋮

Library1

Library2

$\Gamma \vdash e : T$

Refined ML

Application0

Application1

Application2

Library1

Library2

Gradual Migration

SML Code (& Guarantees)

Refined ML Code (& Guarantees)

Interoperating!

*Figures not drawn to scale

SML

Application1

Application2

⋮

Library1

Library2

$\Gamma \vdash e : T$

Refined ML

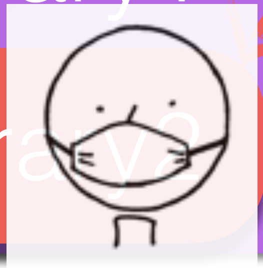
Application0

Application1

Application2

Library1

Library2



Gradual Migration

SML Code (& Guarantees)

Refined ML Code (& Guarantees)

Interoperating!

*Figures not drawn to scale

$\Gamma \vdash e : T$

SML

Application1

Application2

⋮

Library1

Library2

Refined ML

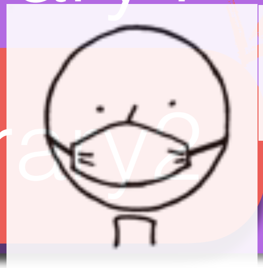
Application0

Application1

Application2

Library1

Library2



???

Gradual Migration

SML Code (& Guarantees)

The Challenge!

Refined ML Code (& Guarantees)

Interoperating!

*Figures not drawn to scale

$\Gamma \vdash e : T$

SML

Ap

Applica

⋮

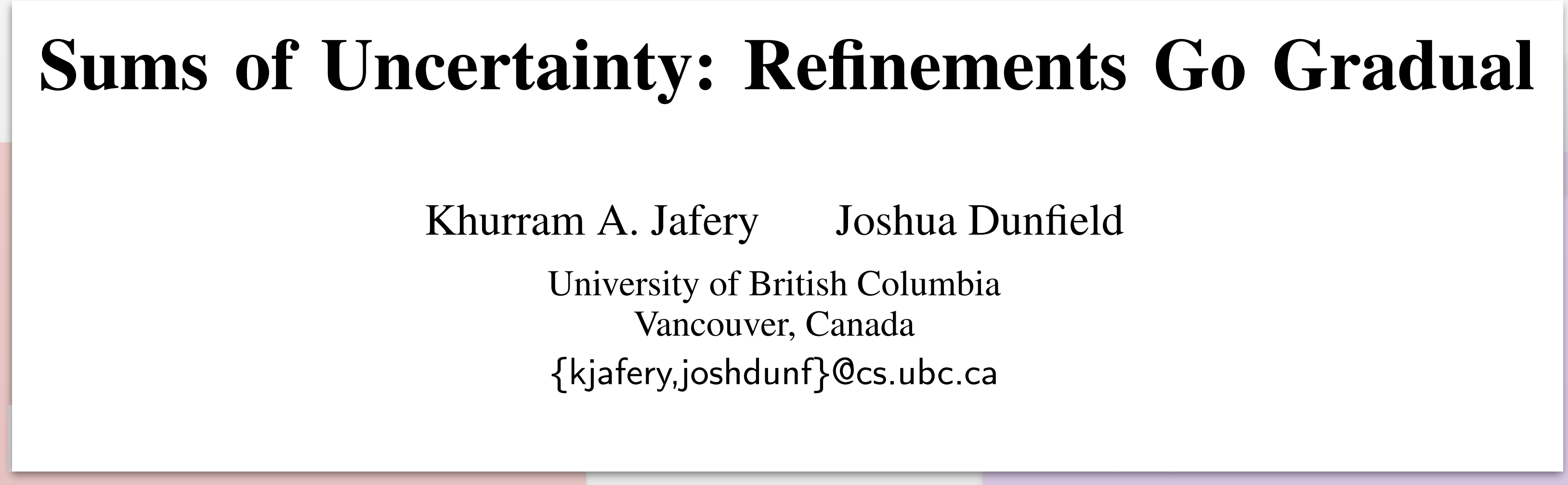
Library1

Library2

tion1

tion2

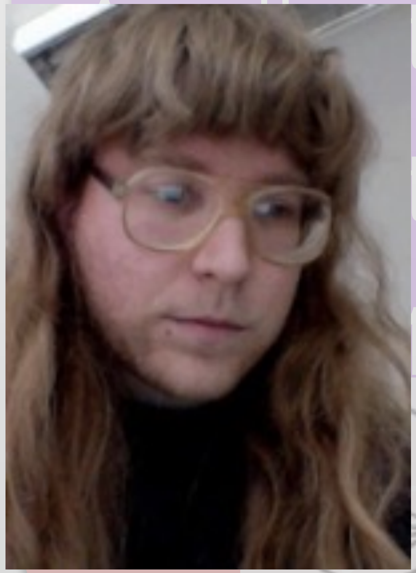
???



Sums of Uncertainty: Refinements Go Gradual

Khurram A. Jafery   Joshua Dunfield

University of British Columbia
Vancouver, Canada
{kjafery,joshdunf}@cs.ubc.ca



POPL17 Gradual Migration

The Challenge!

SML Code (& Guarantees)

Refined ML Code (& Guarantees)

Interoperating!

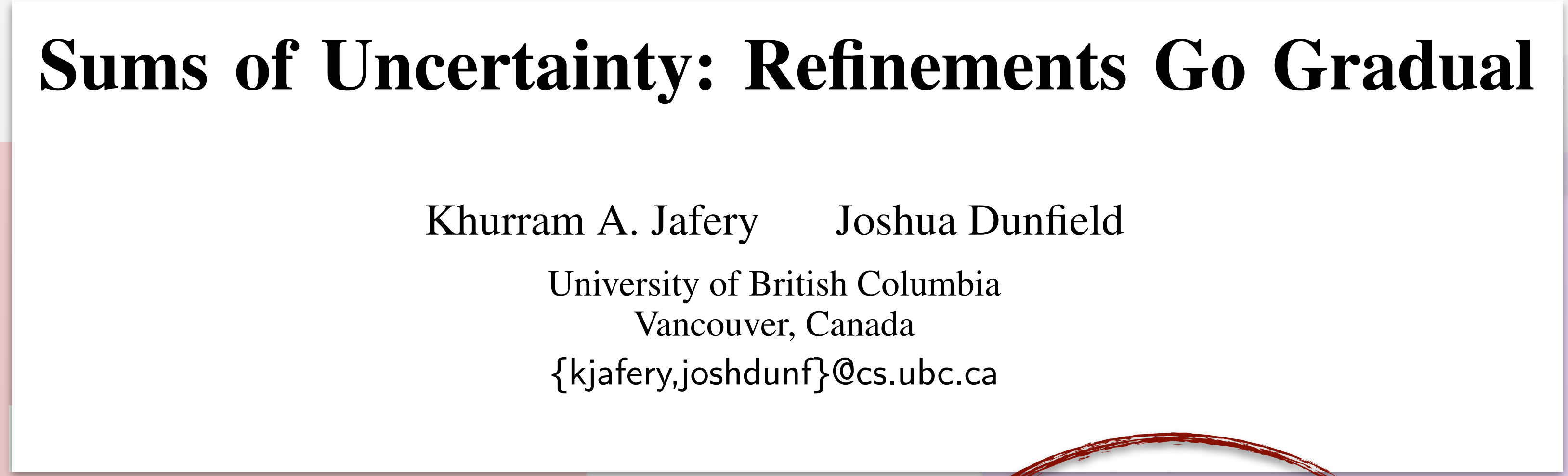
*Figures not drawn to scale

$\Gamma \vdash e : T$

SML

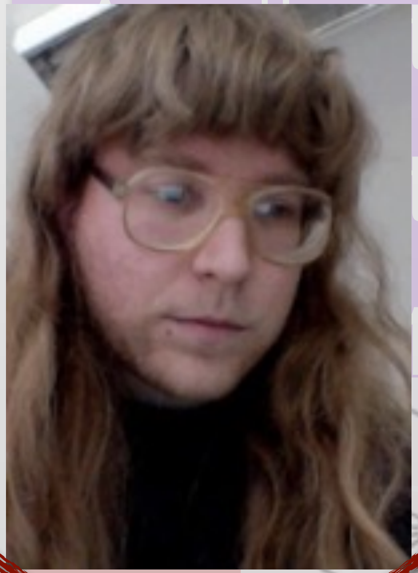
Library1
Library2

???



Sums of Uncertainty: Refinements Go Gradual

Khurram A. Jafery Joshua Dunfield
University of British Columbia
Vancouver, Canada
{kjafery,joshdunf}@cs.ubc.ca



POPL17 Gradual Migration

The Challenge!

SML Code (& Guarantees)

Refined ML Code (& Guarantees)

Interoperating!

*Figures not drawn to scale

Outline

- Motivating Example (In Acts)

- Gradual Typing For All!
- Typing in Small Pieces
- Meat
- Strands and Related Works

Type Spectrum


ML Types

Refinement Types

Type Spectrum

Untyped

ML Types

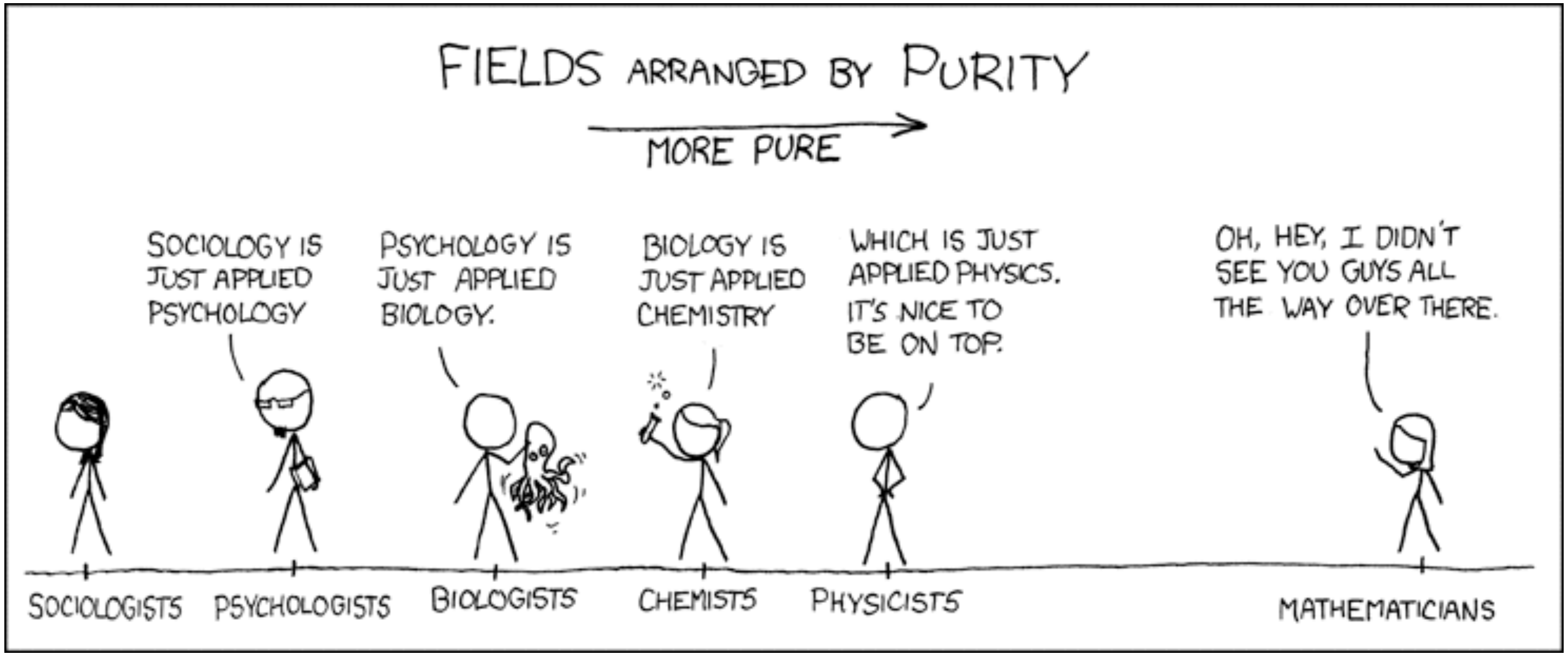
Refinement Types

Type Spectrum

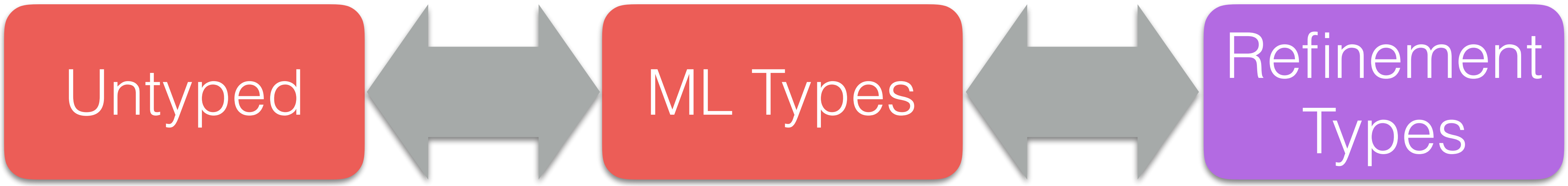

Untyped

ML Types

Refinement Types



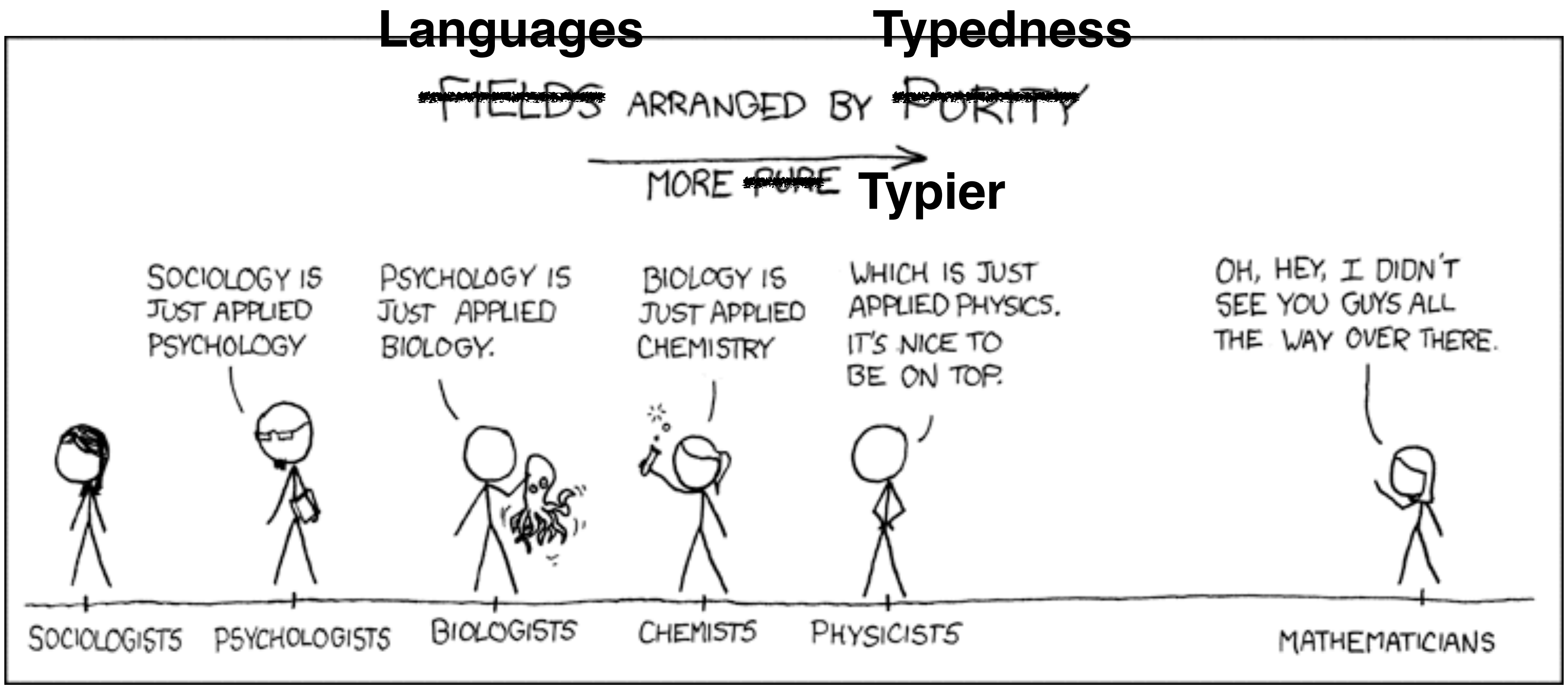
Type Spectrum



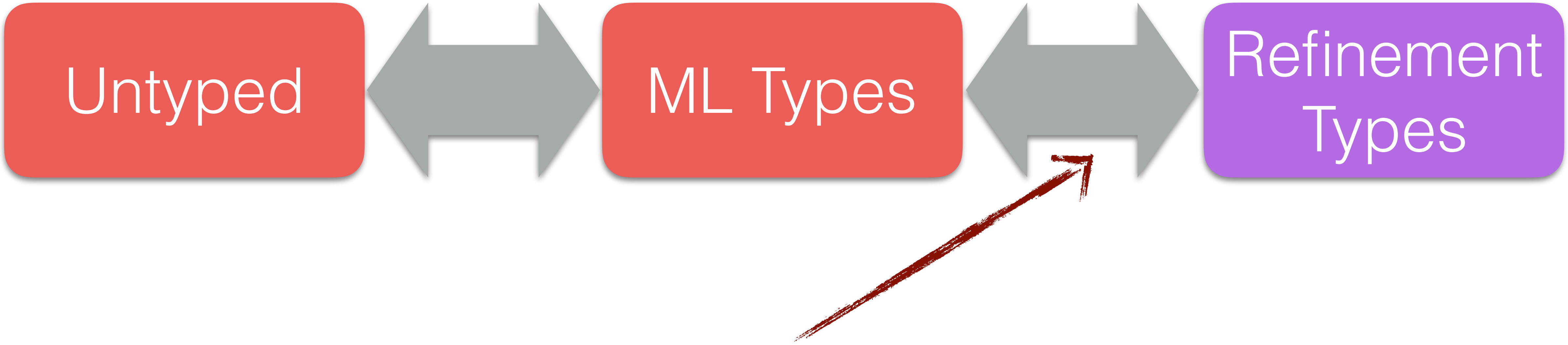
Untyped

ML Types

Refinement Types



Type Spectrum



Untyped

ML Types

Refinement Types

Gradual Typing

Type Spectrum

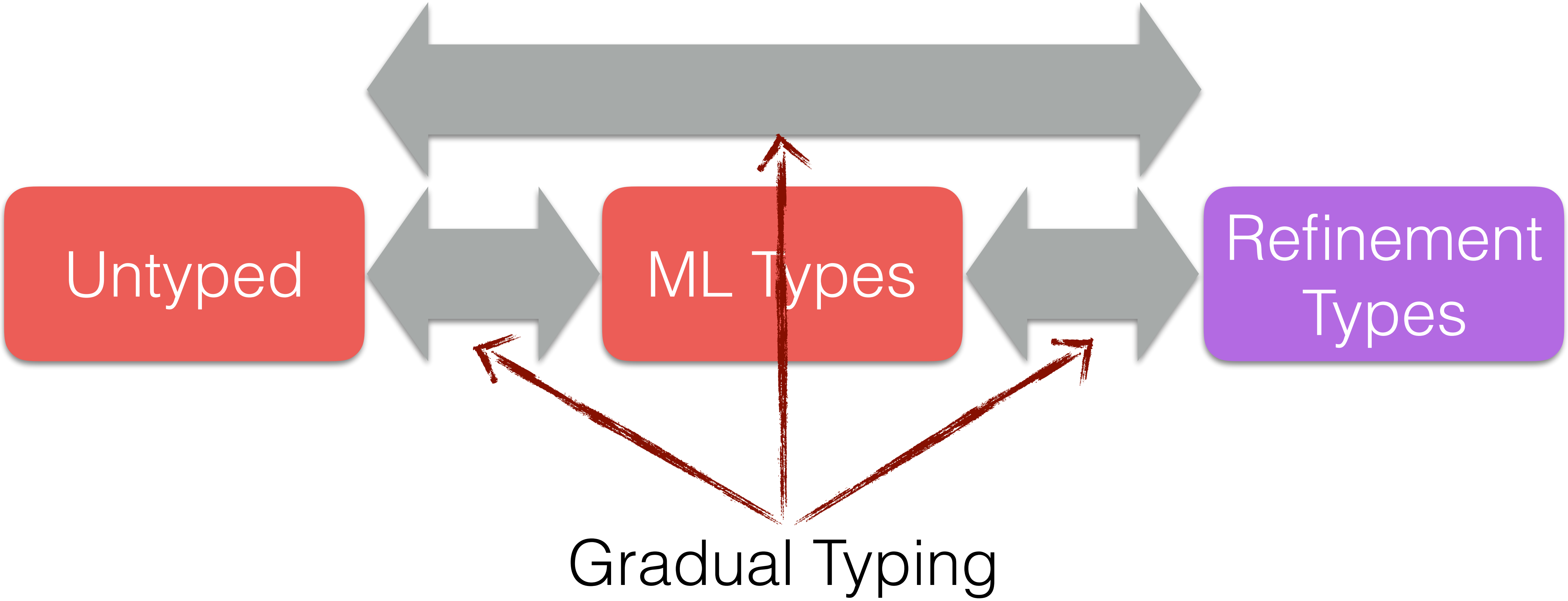

Untyped

ML Types

Refinement Types

Gradual Typing

Type Spectrum



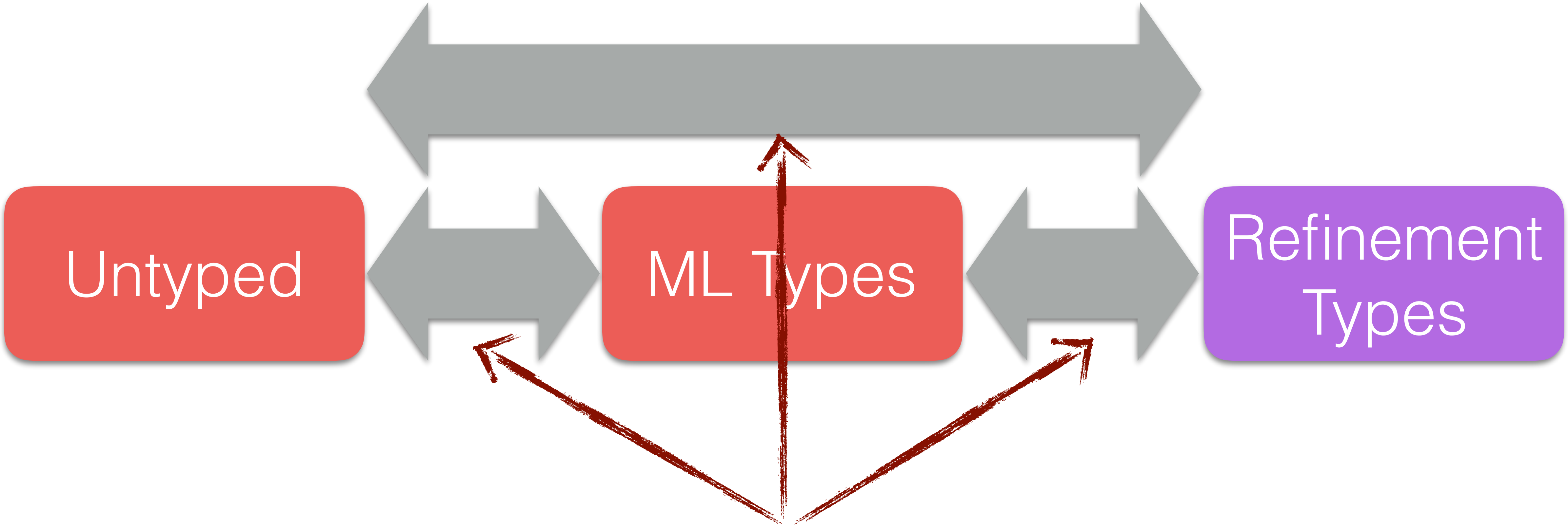
Untyped

ML Types

Refinement Types

Gradual Typing

Type Spectrum



Untyped

ML Types

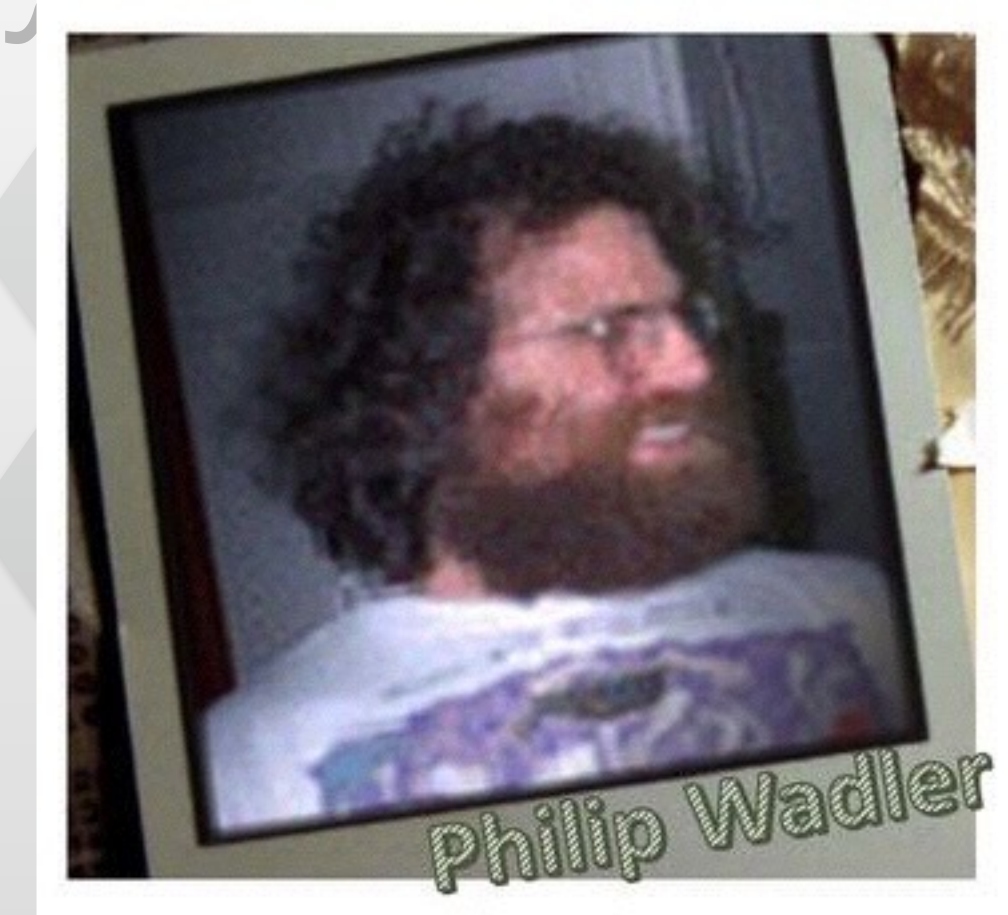
Refinement Types

Gradual Typing

Gradual Typing is a Relative Concept!

Type Spectrum

Untyped

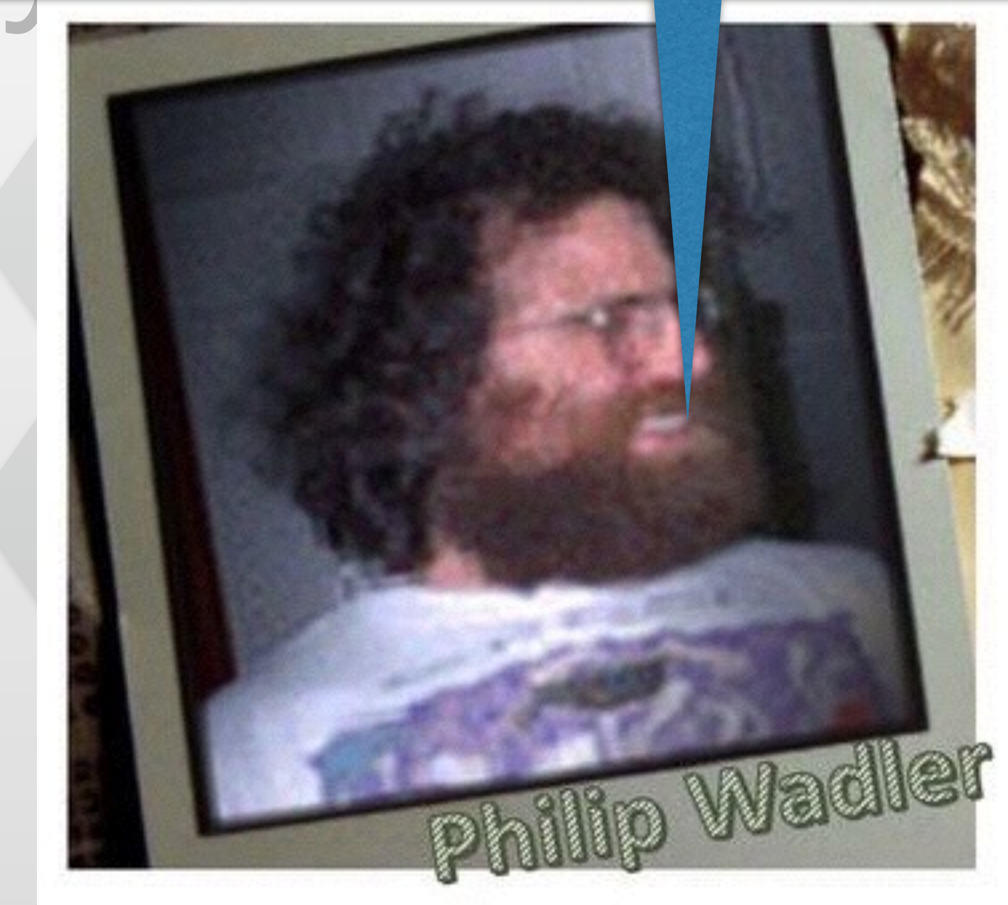


Refinement Types

Gradual Typing

Gradual Typing is a Relative Concept!

> I always assumed gradual types were to help those poor schmucks using untyped languages to migrate to typed languages. I now realise that I am one of the poor schmucks.



Untyped

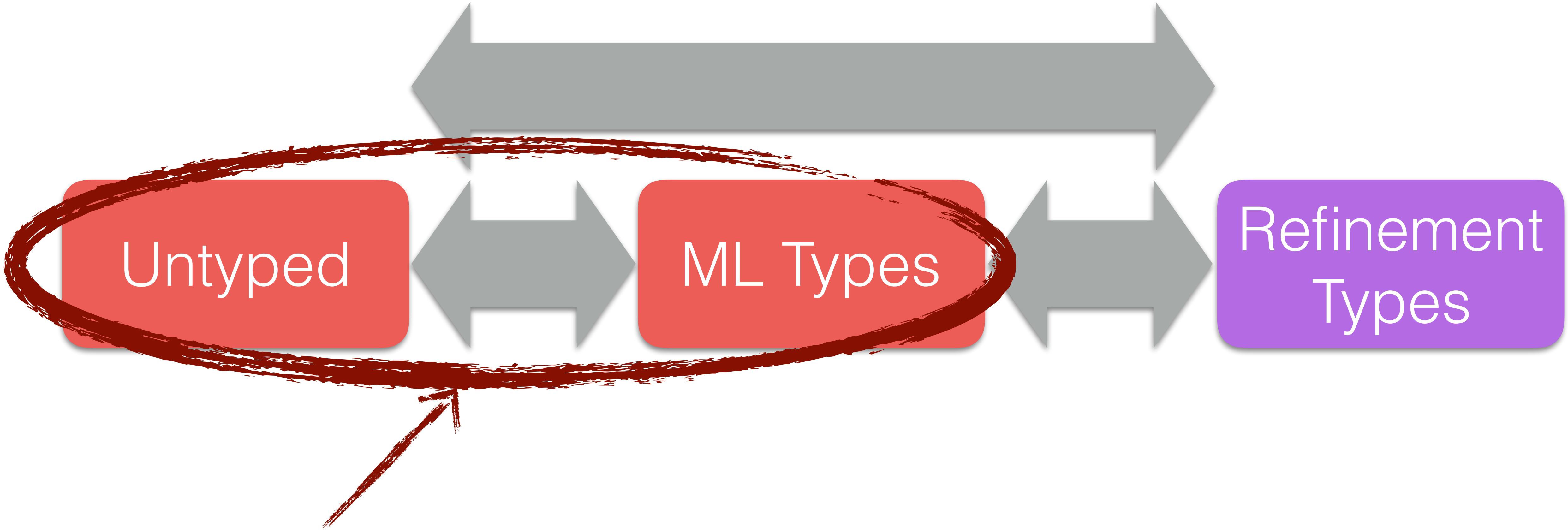
Refinement Types

Gradual Typing

Gradual Typing is a Relative Concept!

★ **[Wadler SNAPL2015]**

Type Spectrum



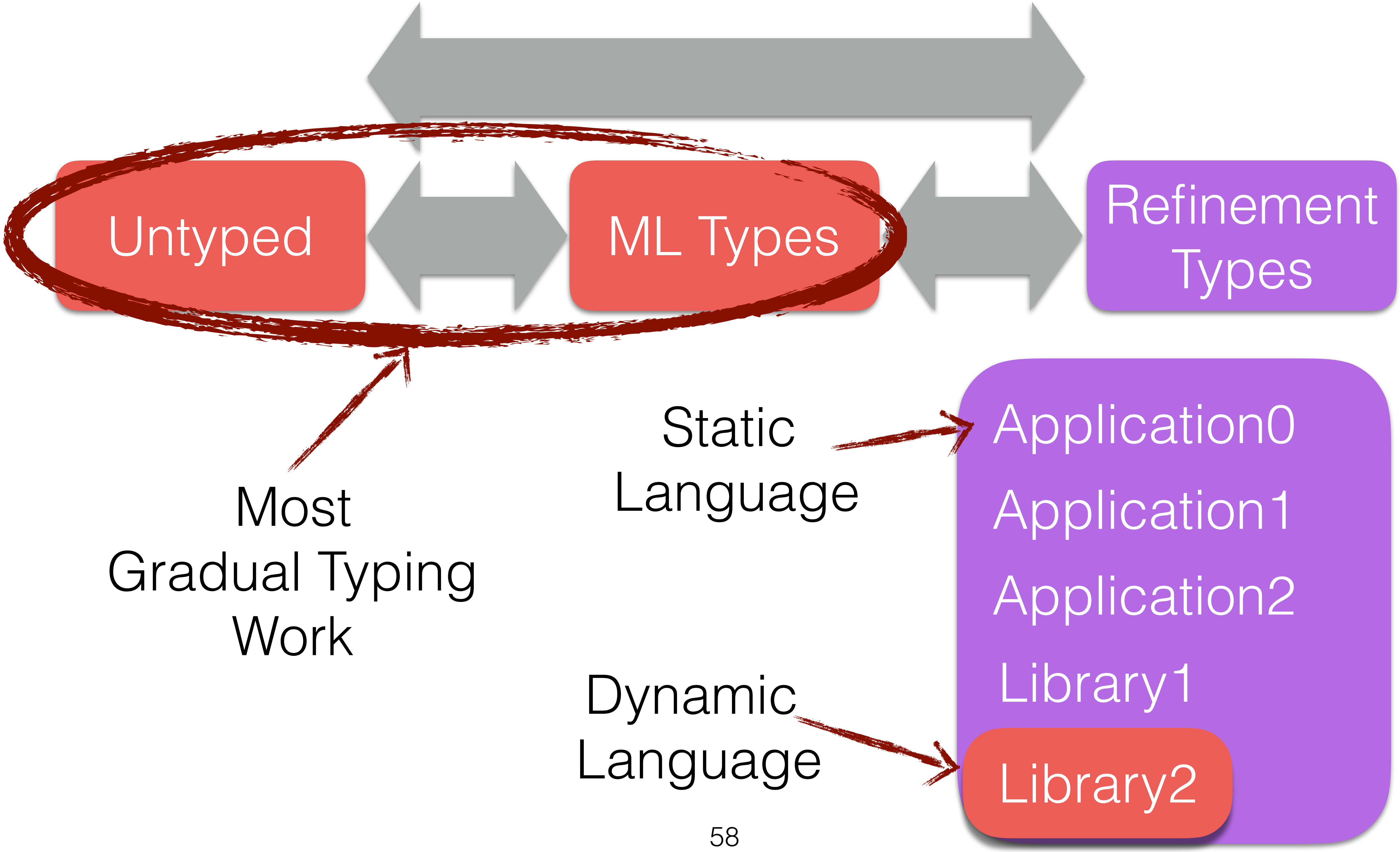
Untyped

ML Types

Refinement Types

Most Gradual Typing Work

Type Spectrum



Untyped

ML Types

Refinement Types

Most Gradual Typing Work

Static Language

Dynamic Language

Application0

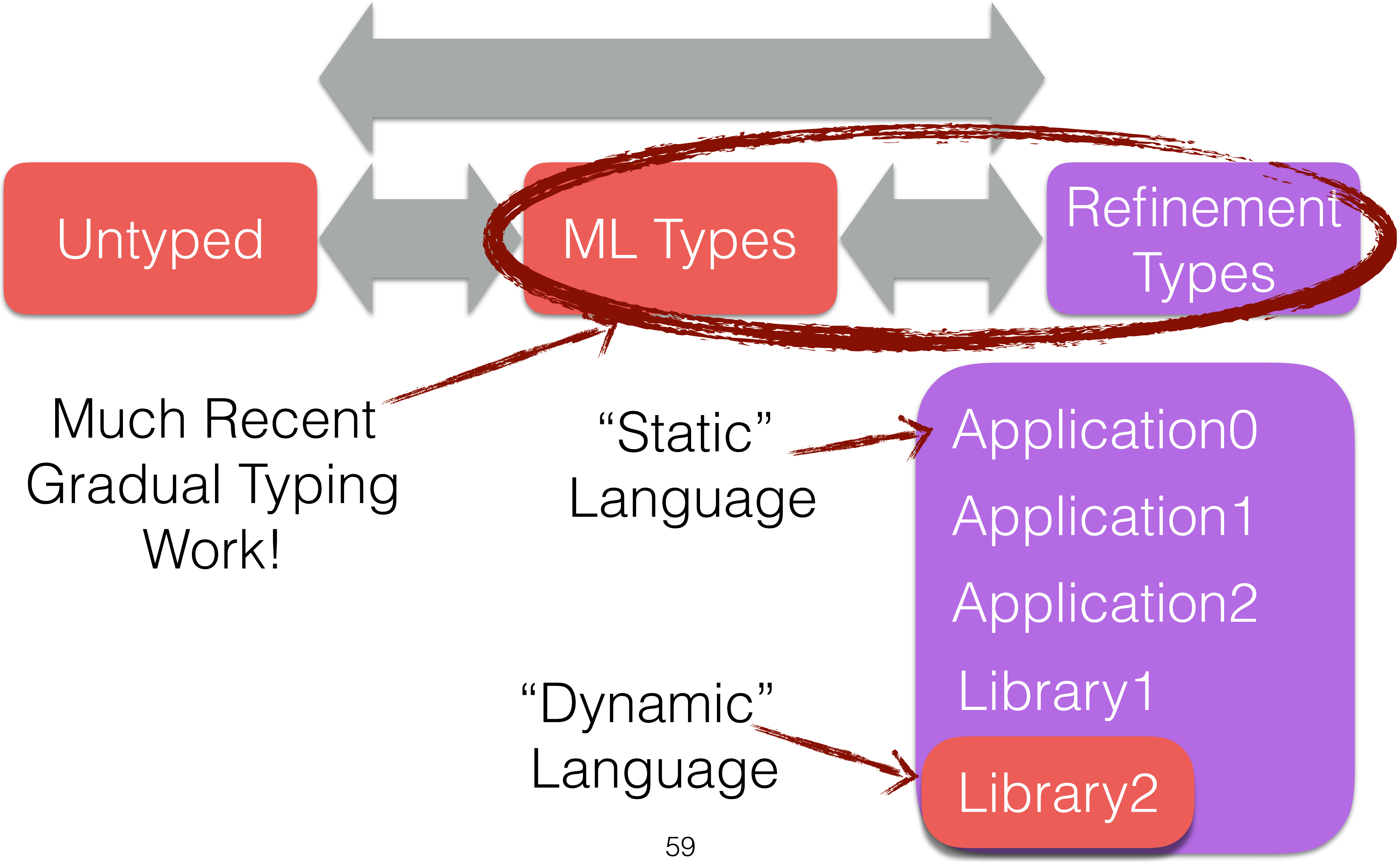
Application1

Application2

Library1

Library2

Type Spectrum



Untyped

ML Types

Refinement Types

Much Recent Gradual Typing Work!

“Static” Language

“Dynamic” Language

Application0

Application1

Application2

Library1

Library2

Outline

- Motivating Example (In Two Acts)
- Gradual Typing For All!

- Typing in Small Pieces
- Meat
- Strands and Related Works

Gradual Types?

Gradual Types?

What does Gradual Typing
have to do with Types?

POSTPONED

~~*Gradual*~~ Types

What are Types *About*?

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1{\to}\mathtt{T}_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11}{\to}\mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-APP)}$$

Typing $\boxed{\Gamma \vdash \texttt{t} : \texttt{T}}$

$$\frac{\texttt{x}:\texttt{T} \in \Gamma}{\Gamma \vdash \texttt{x} : \texttt{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, \texttt{x}:\texttt{T}_1 \vdash \texttt{t}_2 : \texttt{T}_2}{\Gamma \vdash \lambda\texttt{x}:\texttt{T}_1.\texttt{t}_2 : \texttt{T}_1 \to \texttt{T}_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_{11} \to \texttt{T}_{12} \qquad \Gamma \vdash \texttt{t}_2 : \texttt{T}_{11}}{\Gamma \vdash \texttt{t}_1\ \texttt{t}_2 : \texttt{T}_{12}} \qquad \text{(T-APP)}$$

Inductive Definition

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}:\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda\mathtt{x}:\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-APP)}$$

Inductive Definition

Grammar on Steroids

(i.e., data structure spec)

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-}\textsc{Var}\text{)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-}\textsc{Abs}\text{)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-}\textsc{App}\text{)}$$

Inductive Definition

Grammar on Steroids

(i.e., data structure spec)

Informally ascribe ***behavioural*** meaning to it

Typing $\boxed{\Gamma \vdash \texttt{t} : \texttt{T}}$

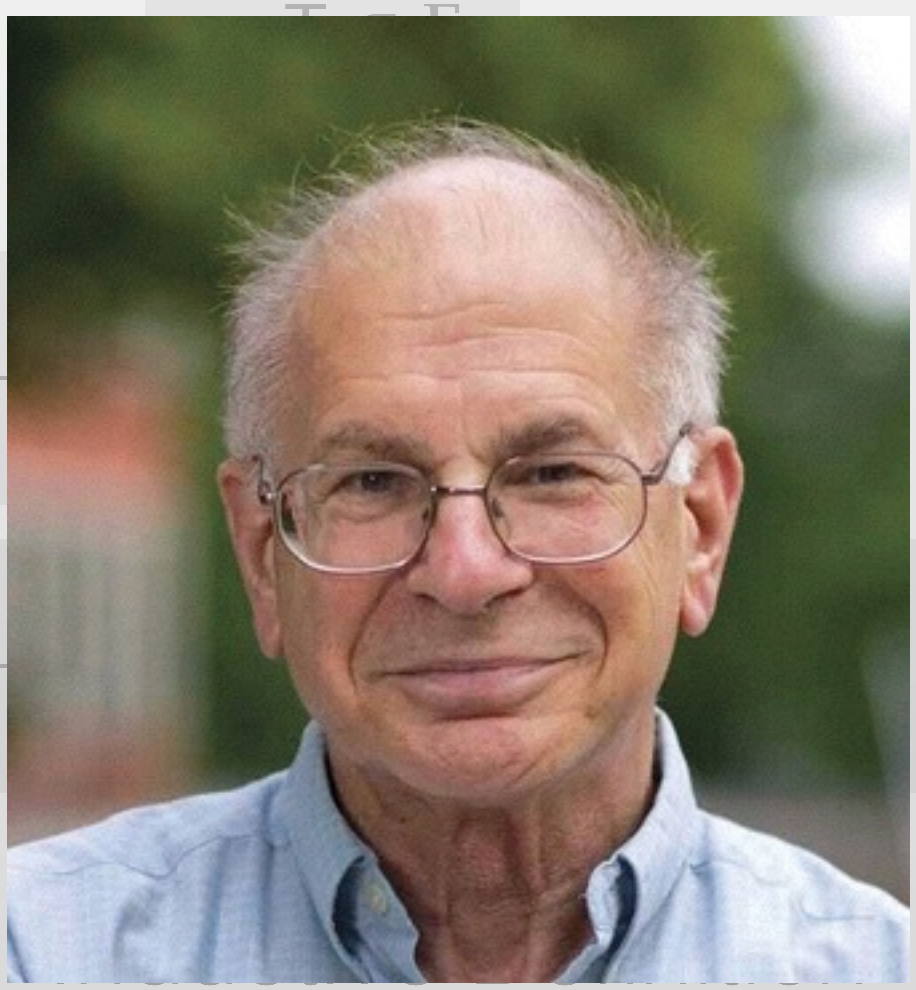
(T-VAR)

Γ

(T-ABS)

$\Gamma \vdash \texttt{t}_1$

(T-APP)



Daniel Kahneman

Grammar on Steroids
(i.e., data structure spec)

Informally ascribe ***behavioural*** meaning to it

Typing

Γ

$\Gamma \vdash t_1$

(T-ABS)

(T-APP)

If you are shown a word on a screen in a language you know, you **will** read it….

Daniel Kahneman

Grammar on Steroids

(i.e., data structure spec)

Thinking, Fast and Slow

Informally ascribe ***behavioural*** meaning to it

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-APP)}$$

Inductive Definition

Grammar on Steroids

(i.e., data structure spec)

Informally ascribe ***behavioural*** meaning to it

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1 . \mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-ABS)}$$

If **t1** turns **T11**s into **T12**s…

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-APP)}$$

Inductive Definition

Grammar on Steroids

(i.e., data structure spec)

Informally ascribe ***behavioural*** meaning to it

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathsf{T}}$

$$\frac{\mathtt{x}{:}\mathsf{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathsf{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathsf{T}_1 \vdash \mathtt{t}_2 : \mathsf{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathsf{T}_1 . \mathtt{t}_2 : \mathsf{T}_1 \to \mathsf{T}_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathsf{T}_{11} \to \mathsf{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathsf{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathsf{T}_{12}} \qquad \text{(T-APP)}$$

If **t1** turns **T11**s into **T12**s…

…and **t2** yields **T11**s…

Inductive Definition

Grammar on Steroids

(i.e., data structure spec)

Informally ascribe ***behavioural*** meaning to it

…and **t2** yields **T11**s…

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

If **t1** turns **T11**s into **T12**s…

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-APP)}$$

…then ***appropriately linking them*** yields **T12s**

Inductive Definition

Grammar on Steroids

(i.e., data structure spec)

Informally ascribe ***behavioural*** meaning to it

Type Safety

THEOREM [PROGRESS]: Suppose `t` is a closed, well-typed term (that is, $\vdash \texttt{t} : \mathsf{T}$ for some T). Then either `t` is a value or else there is some $\texttt{t}'$ with $\texttt{t} \longrightarrow \texttt{t}'$. □

THEOREM [PRESERVATION]: If $\Gamma \vdash \texttt{t} : \mathsf{T}$ and $\texttt{t} \longrightarrow \texttt{t}'$, then $\Gamma \vdash \texttt{t}' : \mathsf{T}$.

“Well-typed programs don’t go wrong”

Gradual Type Safety?

Application0

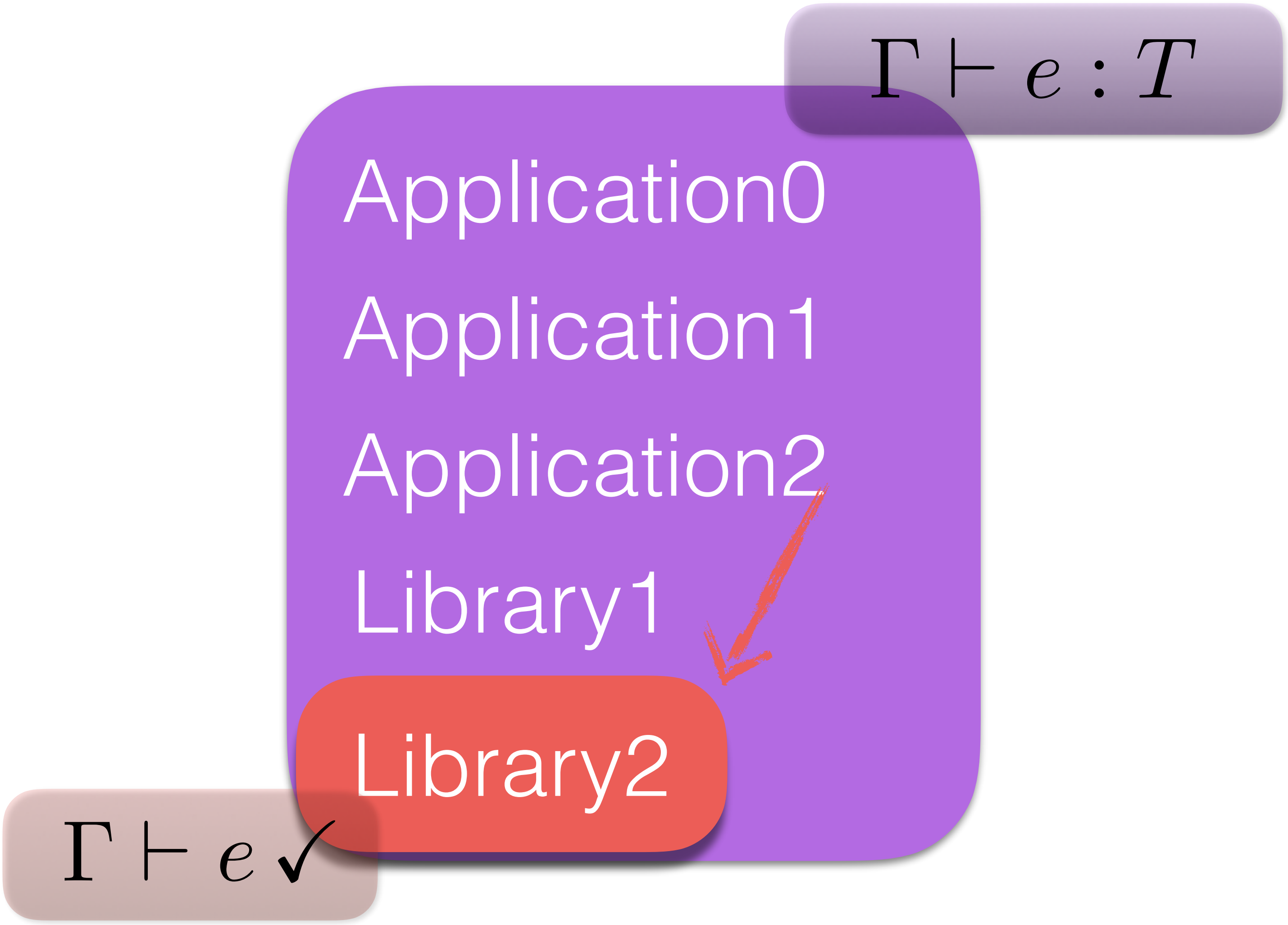
Application1

Application2

Library1

Library2

Now You’ve Got Two Problems!



$\Gamma \vdash e : T$

Application0

Application1

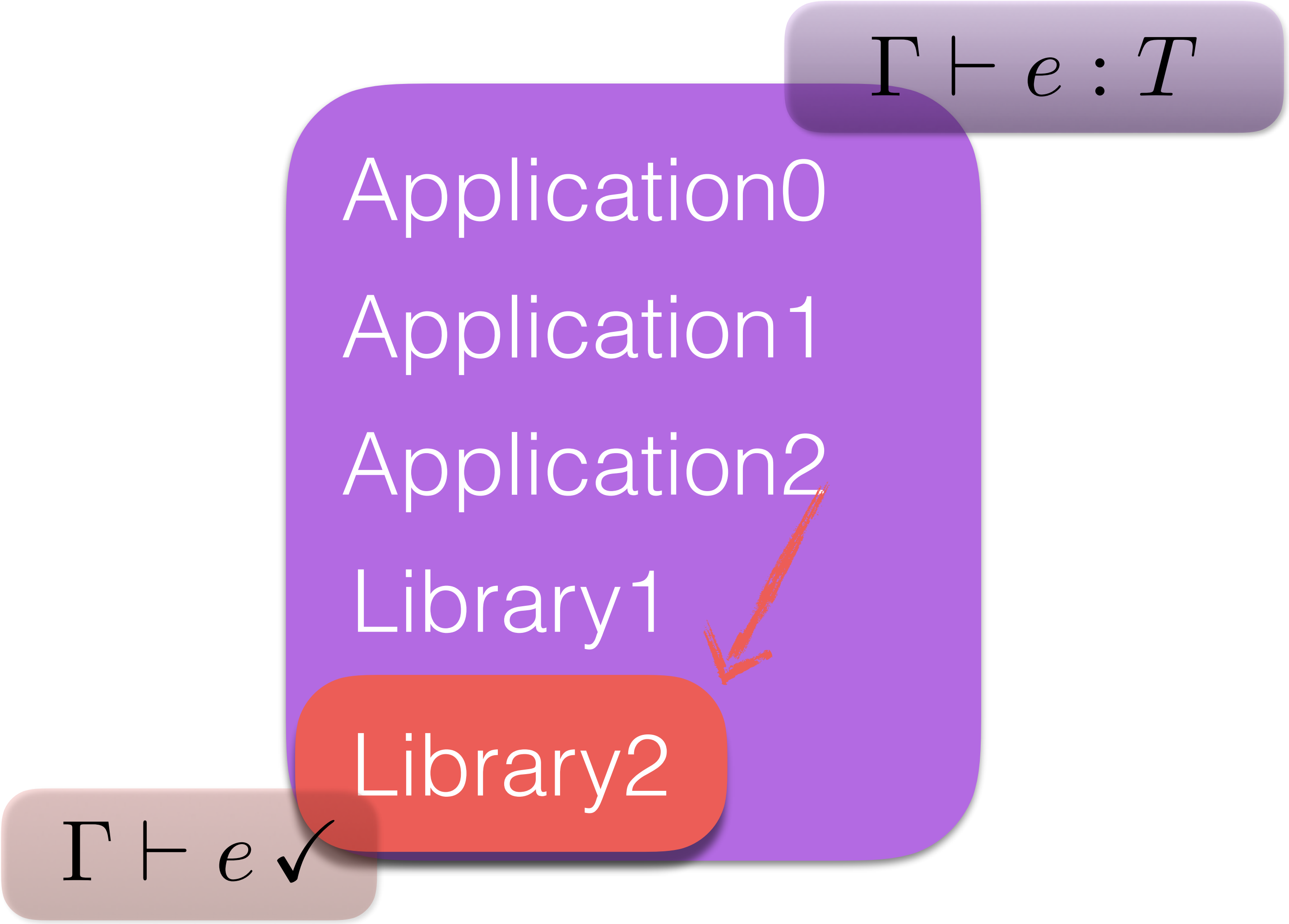
Application2

Library1

Library2

$\Gamma \vdash e \checkmark$

Now You’ve Got Two Problems!



$\Gamma \vdash e : T$

Application0

Application1

Application2

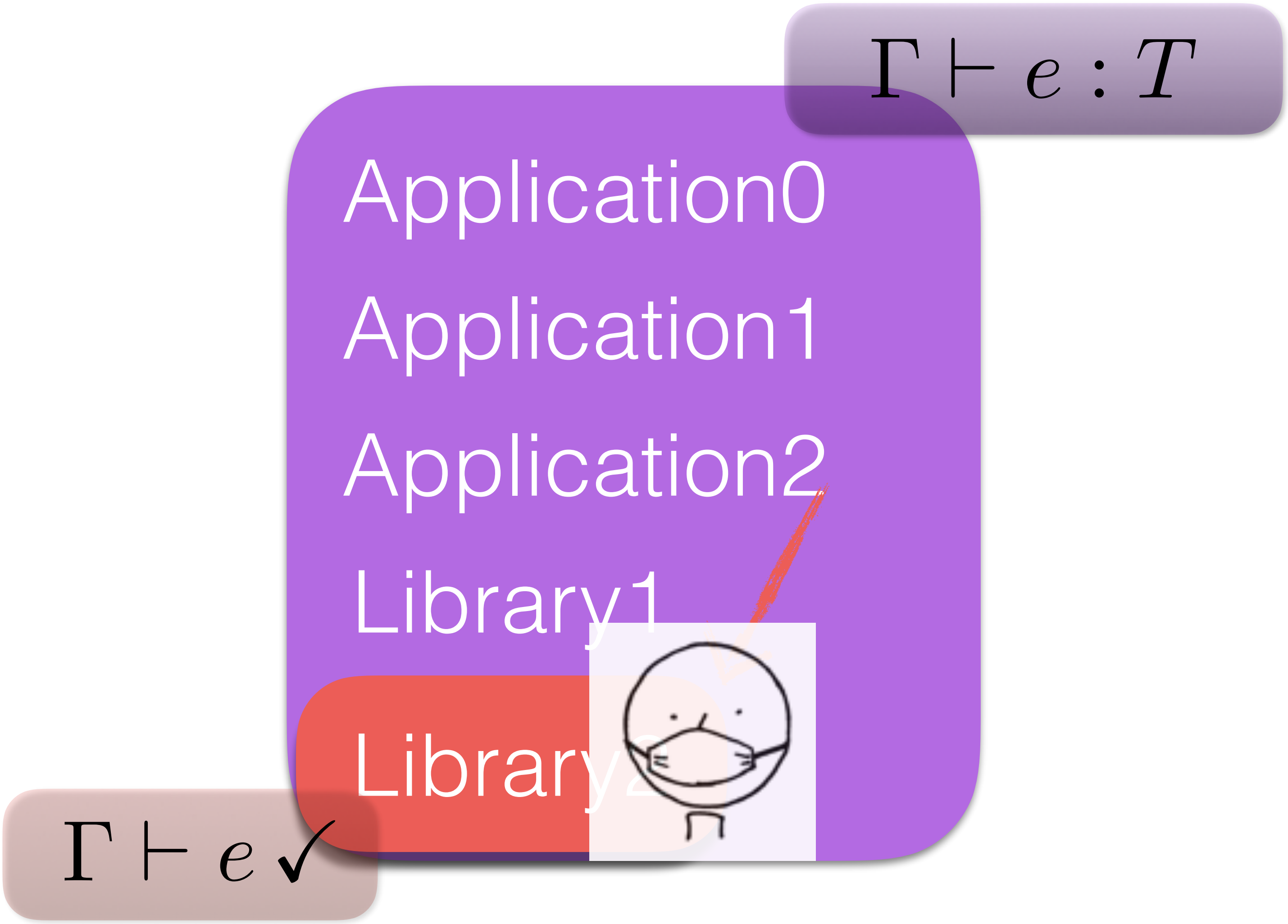
Library1

Library2

$\Gamma \vdash e \checkmark$

Two Typing Judgments = Two “Behavioural Contracts”

Now You've Got Two Problems!



$\Gamma \vdash e : T$

Application0

Application1

Application2

Library1

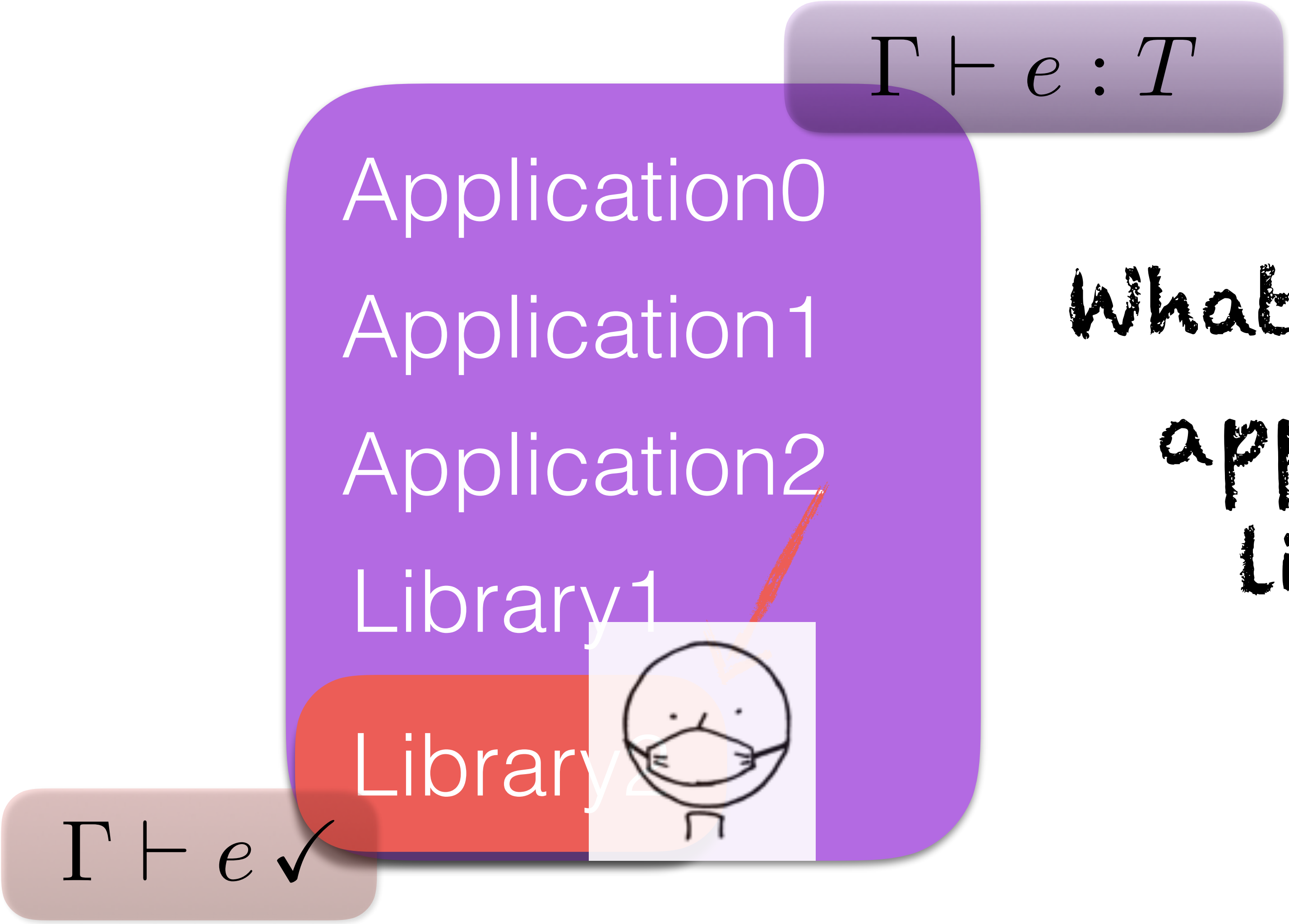
Library2

$\Gamma \vdash e$ ✓

Two Typing Judgments = Two "Behavioural Contracts"

Conflicts Signal Runtime Errors (is that wrong?)

Now You’ve Got Two Problems!



$\Gamma \vdash e : T$

Application0

Application1

Application2

Library1

Library2

$\Gamma \vdash e \checkmark$

What counts as appropriate linking?

Two Typing Judgments = Two “Behavioural Contracts”

Conflicts Signal Runtime Errors (is that wrong?)

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1 . \mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-APP)}$$

If **t1** turns **T11**s into **T12**s…

…and **t2** yields **T11**s…

…then ***appropriately linking them*** yields **T12**s

Typing

$$\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1 . \mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-App)}$$

If the surrounding context behaves as ***Γ*** says…

…and **t1** turns **T11**s into **T12**s…

…and **t2** yields **T11**s…

…then ***appropriately linking them*** yields **T12**s

Typing

$$\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-}\textsc{Var}\text{)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-}\textsc{Abs}\text{)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-}\textsc{App}\text{)}$$

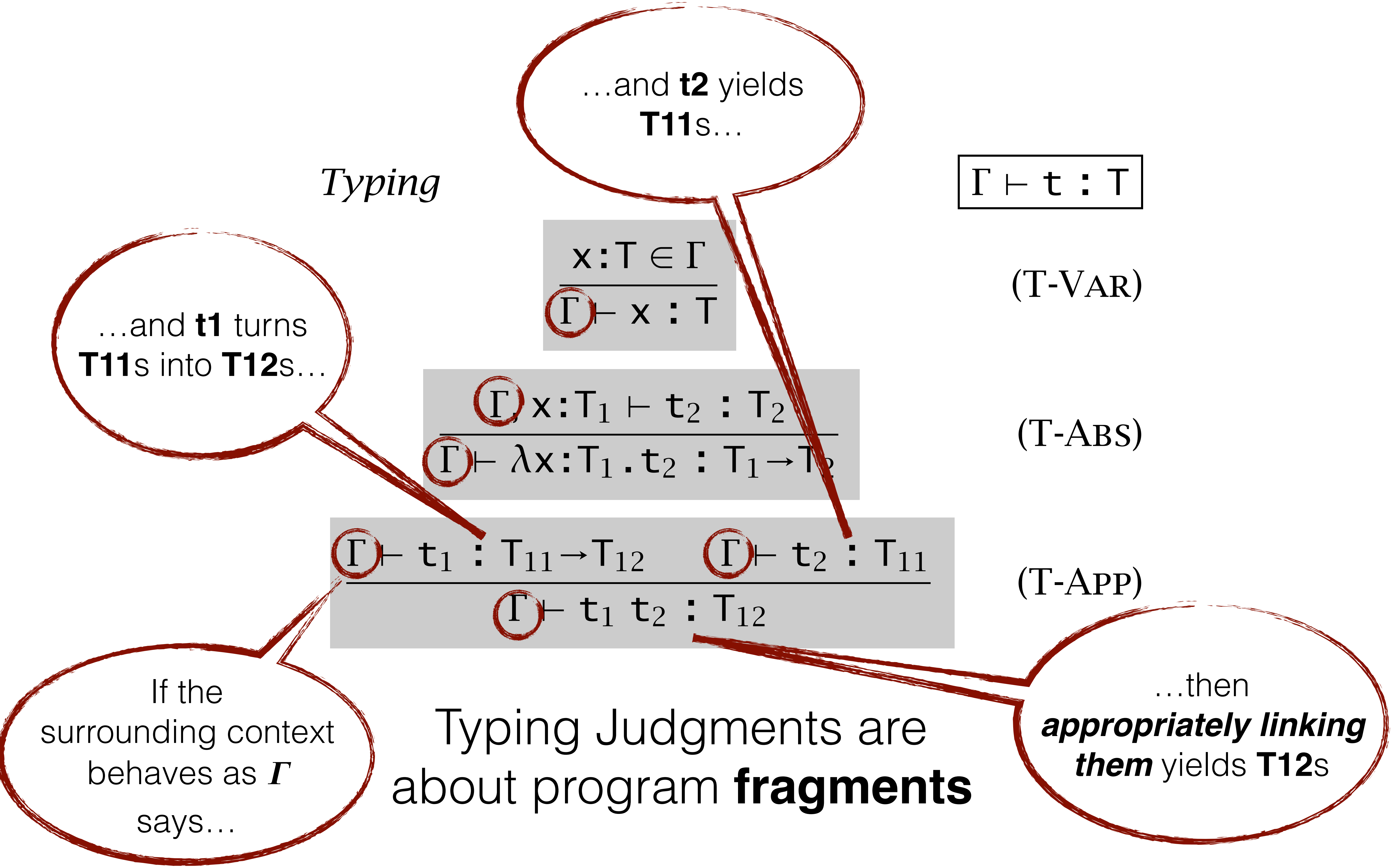
If the surrounding context behaves as ***Γ*** says…

…and **t1** turns **T11**s into **T12**s…

…and **t2** yields **T11**s…

…then ***appropriately linking them*** yields **T12**s

Typing Judgments are about program **fragments**



Typing

$$\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1 . \mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-APP)}$$

If the surrounding context behaves as ***Γ*** says…

…and **t1** turns **T11**s into **T12**s…

…and **t2** yields **T11**s…

…then ***appropriately linking them*** yields **T12**s

Typing Judgments are about program **fragments**

Context Matters, at least intuitively

Type Safety

THEOREM [PROGRESS]: Suppose `t` is a closed, well-typed term (that is, $\vdash \texttt{t} : \texttt{T}$ for some T). Then either `t` is a value or else there is some $\texttt{t}'$ with $\texttt{t} \longrightarrow \texttt{t}'$. □

THEOREM [PRESERVATION]: If $\Gamma \vdash \texttt{t} : \texttt{T}$ and $\texttt{t} \longrightarrow \texttt{t}'$, then $\Gamma \vdash \texttt{t}' : \texttt{T}$.

“Well-typed programs don’t go wrong”

Type Safety

THEOREM [PROGRESS]: Suppose $\mathtt{t}$ is a closed, well-typed term (that is, $\vdash \mathtt{t} : \mathsf{T}$ for some T). Then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$. □

THEOREM [PRESERVATION]: If $\Gamma \vdash \mathtt{t} : \mathsf{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\Gamma \vdash \mathtt{t}' : \mathsf{T}$.

“Well-typed (whole) programs don’t go wrong”

Type Safety

Theorem [Progress]: Suppose $\mathtt{t}$ is a closed, well-typed term (that is, $\vdash \mathtt{t} : \mathsf{T}$ for some T). Then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$. □

Theorem [Preservation]: If $\Gamma \vdash \mathtt{t} : \mathsf{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\Gamma \vdash \mathtt{t}' : \mathsf{T}$.

“Well-typed (whole) programs don’t go wrong”

Do program *fragments* go *right*?

Outline

- Motivating Example (In Two Acts)
- Gradual Typing For All!
- Typing in Small Pieces
- Meat
- Strands and Related Works

Type Safety

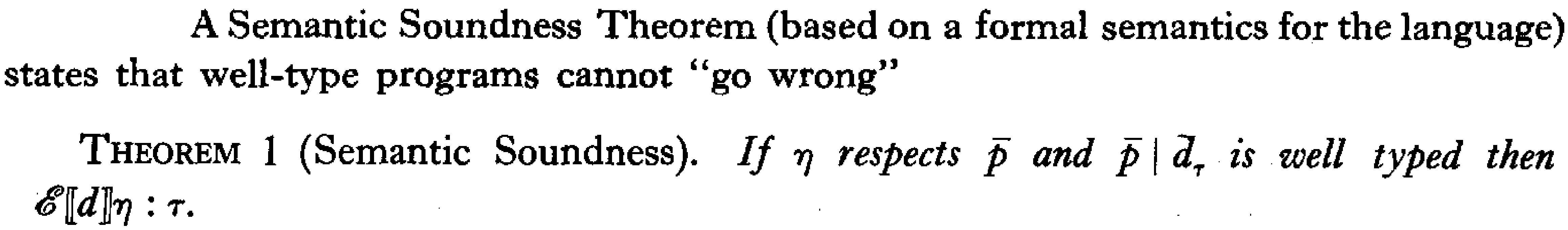
THEOREM [PROGRESS]: Suppose $\mathtt{t}$ is a closed, well-typed term (that is, $\vdash \mathtt{t} : \mathsf{T}$ for some T). Then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$. □

THEOREM [PRESERVATION]: If $\Gamma \vdash \mathtt{t} : \mathsf{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\Gamma \vdash \mathtt{t}' : \mathsf{T}$.

“Well-typed (whole) programs don’t go wrong”

Do program *fragments* go *right*?

Semantic Soundness [Milner '78]



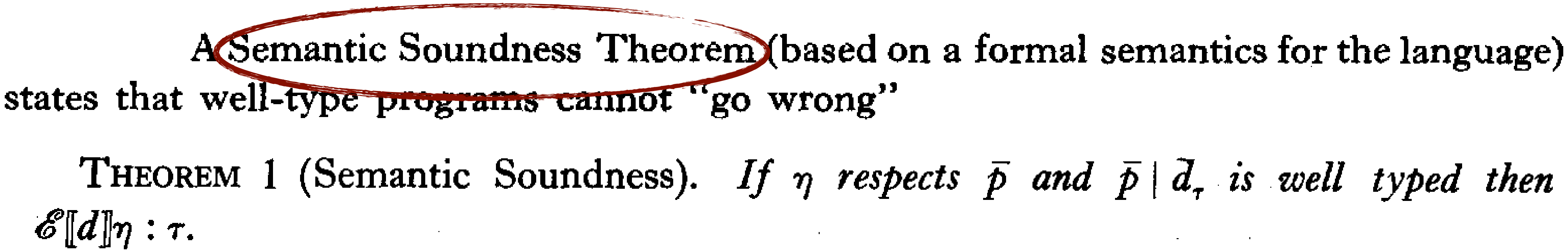
A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot "go wrong"

THEOREM 1 (Semantic Soundness). *If* η *respects* $\bar{p}$ *and* $\bar{p} \mid \bar{d}_\tau$ *is well typed then* $\mathscr{E}[\![d]\!]\eta : \tau$.

i.e., if $\Gamma \vdash e : T$ then $\Gamma \models e : T$.

[Milner 1978]

Semantic Soundness [Milner '78]



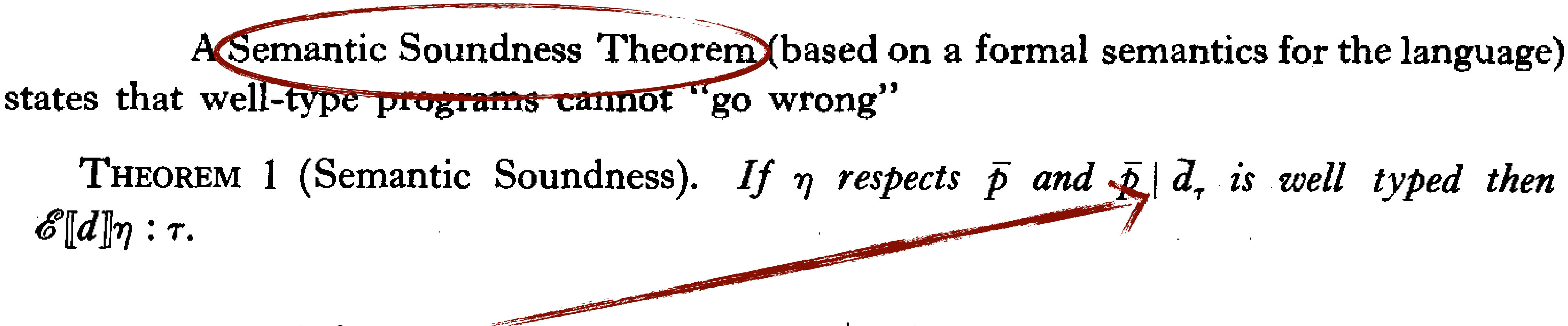
A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot "go wrong"

THEOREM 1 (Semantic Soundness). *If* η *respects* $\bar{p}$ *and* $\bar{p} \mid \bar{d}_\tau$ *is well typed then* $\mathscr{E}[\![d]\!]\eta : \tau$.

i.e., if $\Gamma \vdash e : T$ then $\Gamma \models e : T$.

[Milner 1978]

Semantic Soundness
[Milner '78]



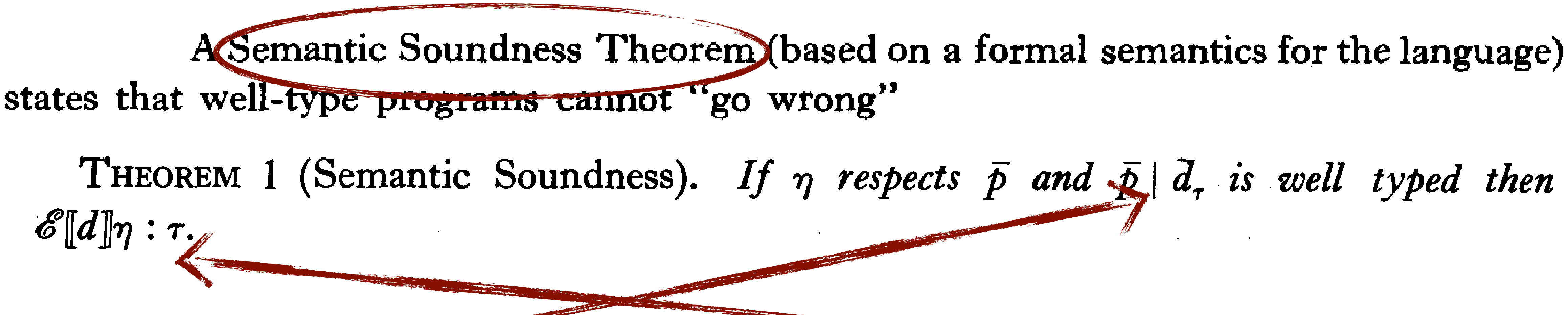
A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot "go wrong"

Theorem 1 (Semantic Soundness). *If* η *respects* $\bar{p}$ *and* $\bar{p} \mid \bar{d}_\tau$ *is well typed then* $\mathscr{E}[\![d]\!]\eta : \tau$.

i.e., if $\Gamma \vdash e : T$ then $\Gamma \models e : T$.

[Milner 1978]

Semantic Soundness [Milner '78]



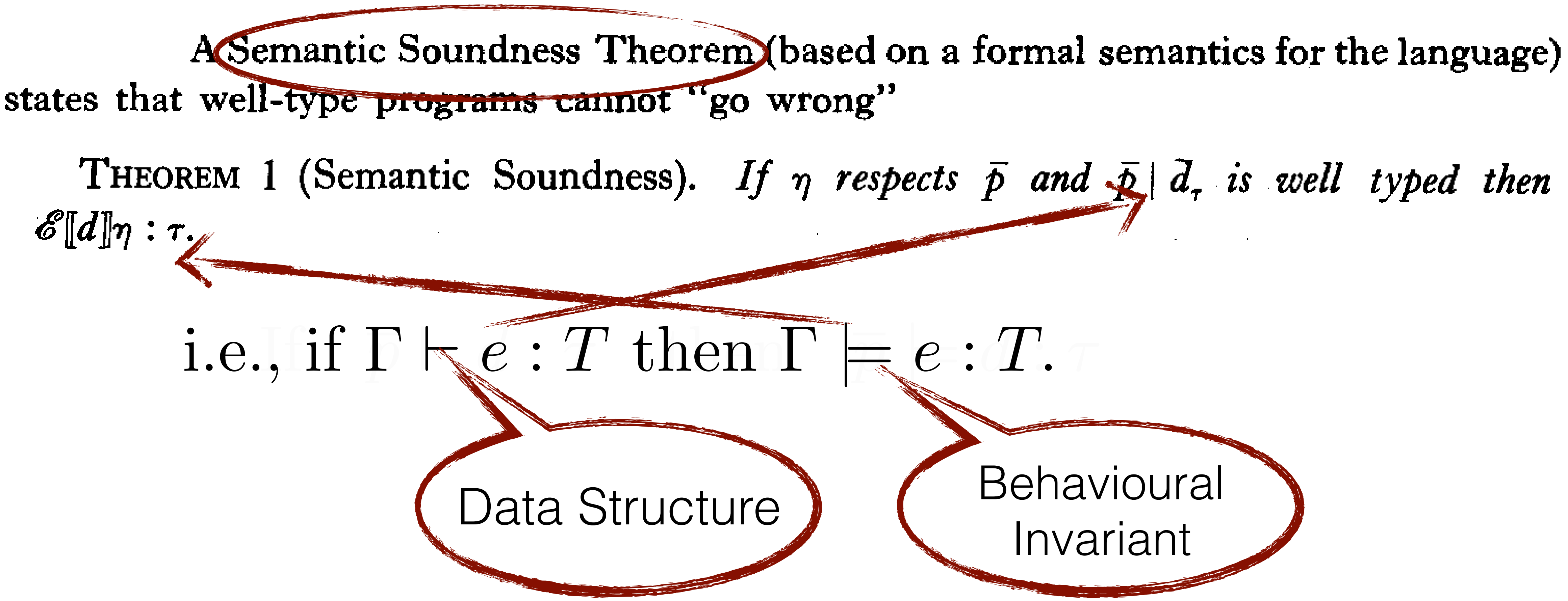
A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot "go wrong"

THEOREM 1 (Semantic Soundness). *If* η *respects* $\bar{p}$ *and* $\bar{p} \mid \bar{d}_\tau$ *is well typed then* $\mathscr{E}[\![d]\!]\eta : \tau$.

i.e., if $\Gamma \vdash e : T$ then $\Gamma \models e : T$.

[Milner 1978]

Semantic Soundness [Milner '78]



A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot "go wrong"

THEOREM 1 (Semantic Soundness). *If η respects $\bar{p}$ and $\bar{p} \mid \bar{d}_\tau$ is well typed then* $\mathscr{E}[\![d]\!]\eta : \tau$.

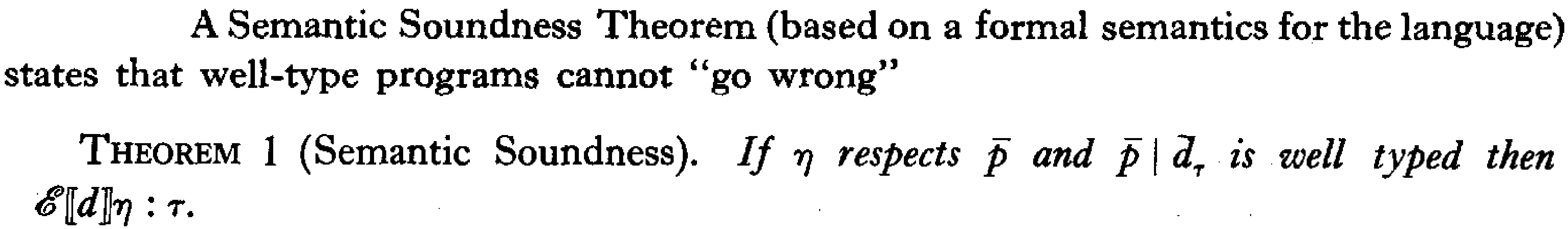
i.e., if $\Gamma \vdash e : T$ then $\Gamma \models e : T$.

Data Structure

Behavioural Invariant

[Milner 1978]

Semantic Soundness



A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot "go wrong"

THEOREM 1 (Semantic Soundness). *If* η *respects* $\bar{p}$ *and* $\bar{p} \mid \bar{d}_\tau$ *is well typed then* $\mathscr{E}[\![d]\!]\eta : \tau$.

i.e., if $\Gamma \vdash e : T$ then $\Gamma \models e : T$.

Syntax

Semantics

Every proof of type assignment
says something meaningful about code

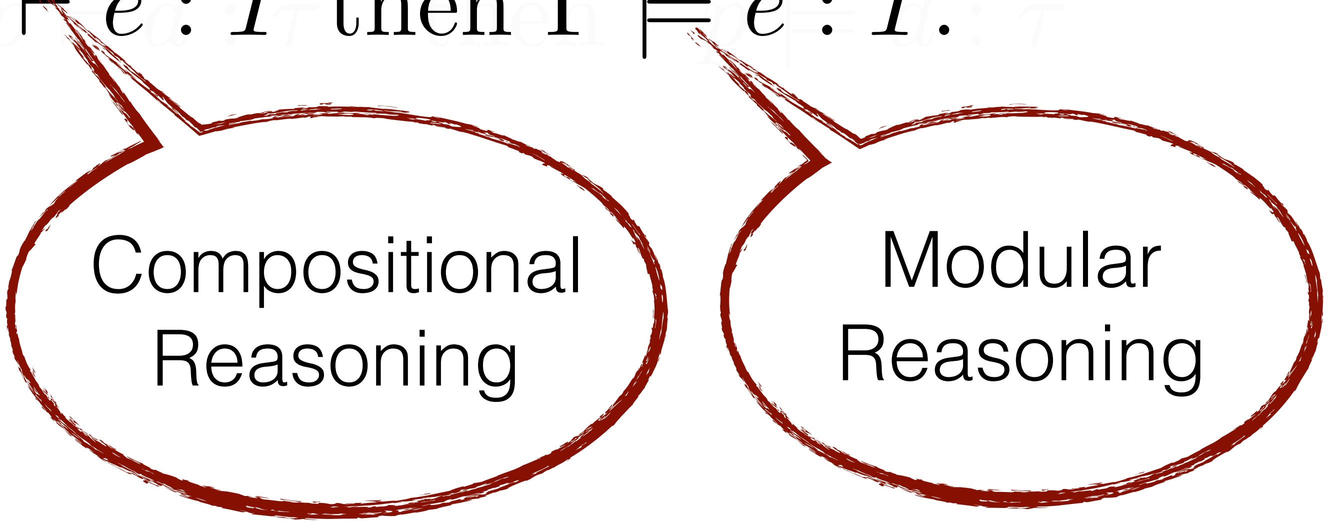
[Milner 1978]

Semantic Soundness

A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot "go wrong"

THEOREM 1 (Semantic Soundness). *If* η *respects* $\bar{p}$ *and* $\bar{p} \mid \bar{d}_\tau$ *is well typed then* $\mathscr{E}[\![d]\!]\eta : \tau$.

i.e., if $\Gamma \vdash e : T$ then $\Gamma \models e : T$.



Compositional Reasoning

Modular Reasoning

[Milner 1978]

Semantic Soundness

A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot "go wrong"

THEOREM 1 (Semantic Soundness). *If* η *respects* $\bar{p}$ *and* $\bar{p} \mid \bar{d}_\tau$ *is well typed then* $\mathcal{E}[\![d]\!]\eta : \tau$.

i.e., if $\Gamma \vdash e : T$ then $\Gamma \models e : T$.

Syntax

Semantics

As a corollary, under the conditions of the theorem we have

$$\mathcal{E}[\![d]\!]\eta \neq \text{wrong},$$

Whole-Program Payoff!

since wrong has no type.

[Milner 1978]

Semantic Soundness

A Semantic Soundness Theorem (based on a formal semantics for the language) implies ~~states~~ that well-type programs cannot "go wrong"

THEOREM 1 (Semantic Soundness). *If η respects $\bar{p}$ and $\bar{p} \mid \bar{d}_\tau$ is well typed then* $\mathscr{E}[\![d]\!]\eta : \tau$.

i.e., if $\Gamma \vdash e : T$ then $\Gamma \models e : T$.

Syntax

Semantics

As a corollary, under the conditions of the theorem we have

$$\mathscr{E}[\![d]\!]\eta \neq \text{wrong},$$

since wrong has no type.

Whole-Program Payoff!

[Milner 1978]

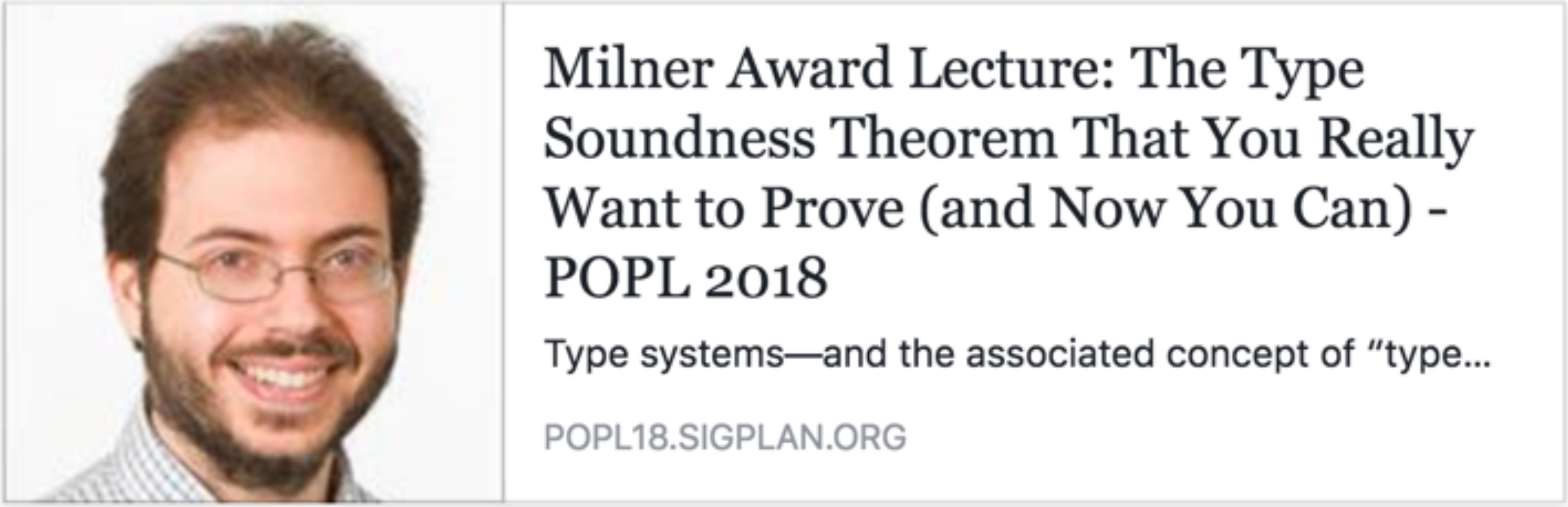
Semantic Soundness

implies

A Semantic Soundness Theorem (based on a formal semantics for the language) ~~states~~ that well-type programs cannot "go wrong"

THE … *ed then*

$\mathscr{E}[\![d]\!]\eta$ …



Milner Award Lecture: The Type Soundness Theorem That You Really Want to Prove (and Now You Can) - POPL 2018

Type systems—and the associated concept of "type...

POPL18.SIGPLAN.ORG

As a corollary, under the conditions of the theorem we have

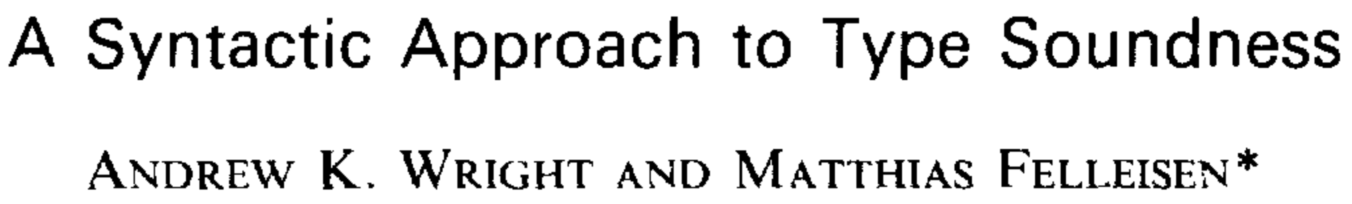
$$\mathscr{E}[\![d]\!]\eta \neq \text{wrong},$$

since wrong has no type.

Whole-Program Payoff!

[Milner 1978]

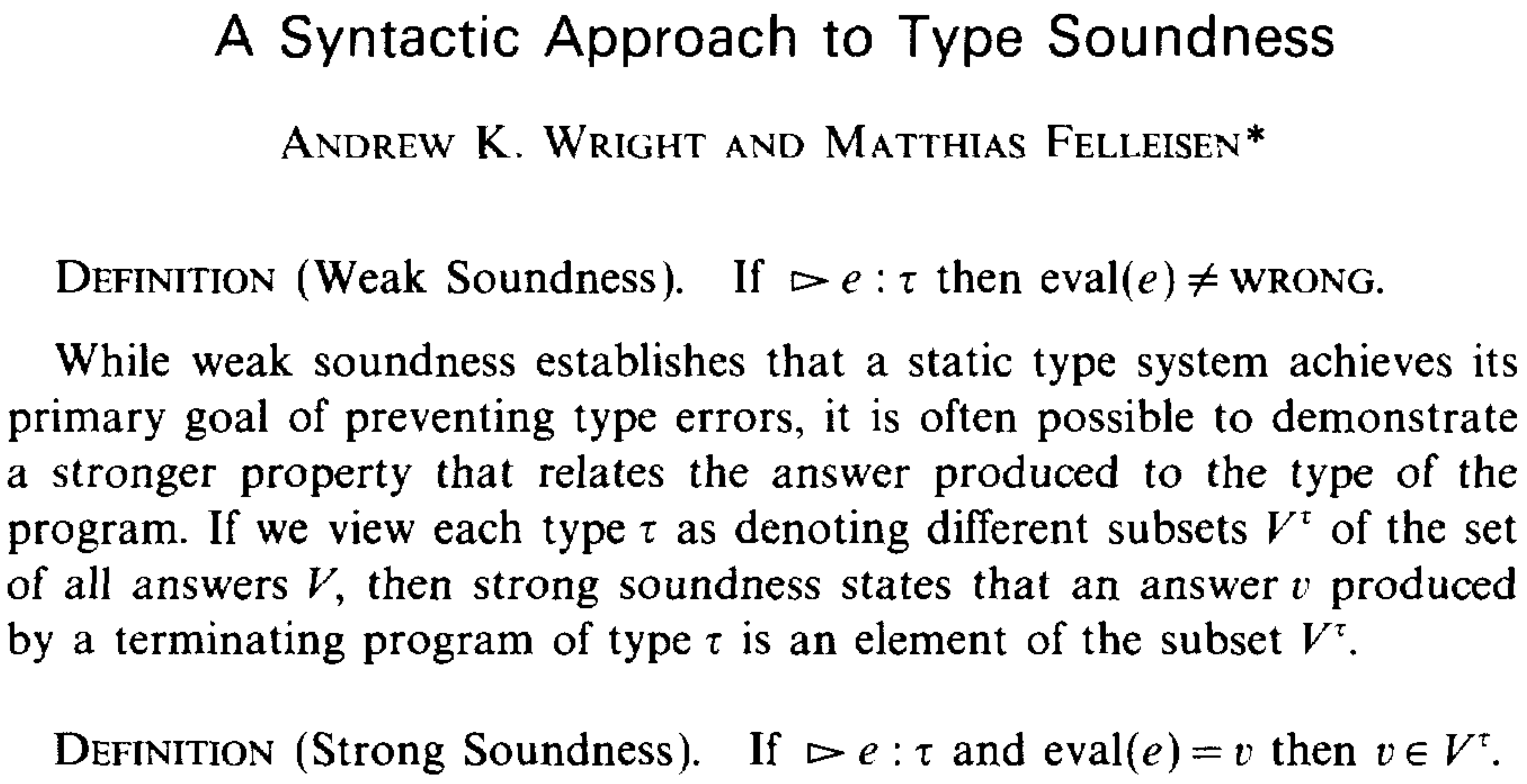
Discourse On The Method



A Syntactic Approach to Type Soundness

ANDREW K. WRIGHT AND MATTHIAS FELLEISEN*

Discourse On The Method



A Syntactic Approach to Type Soundness

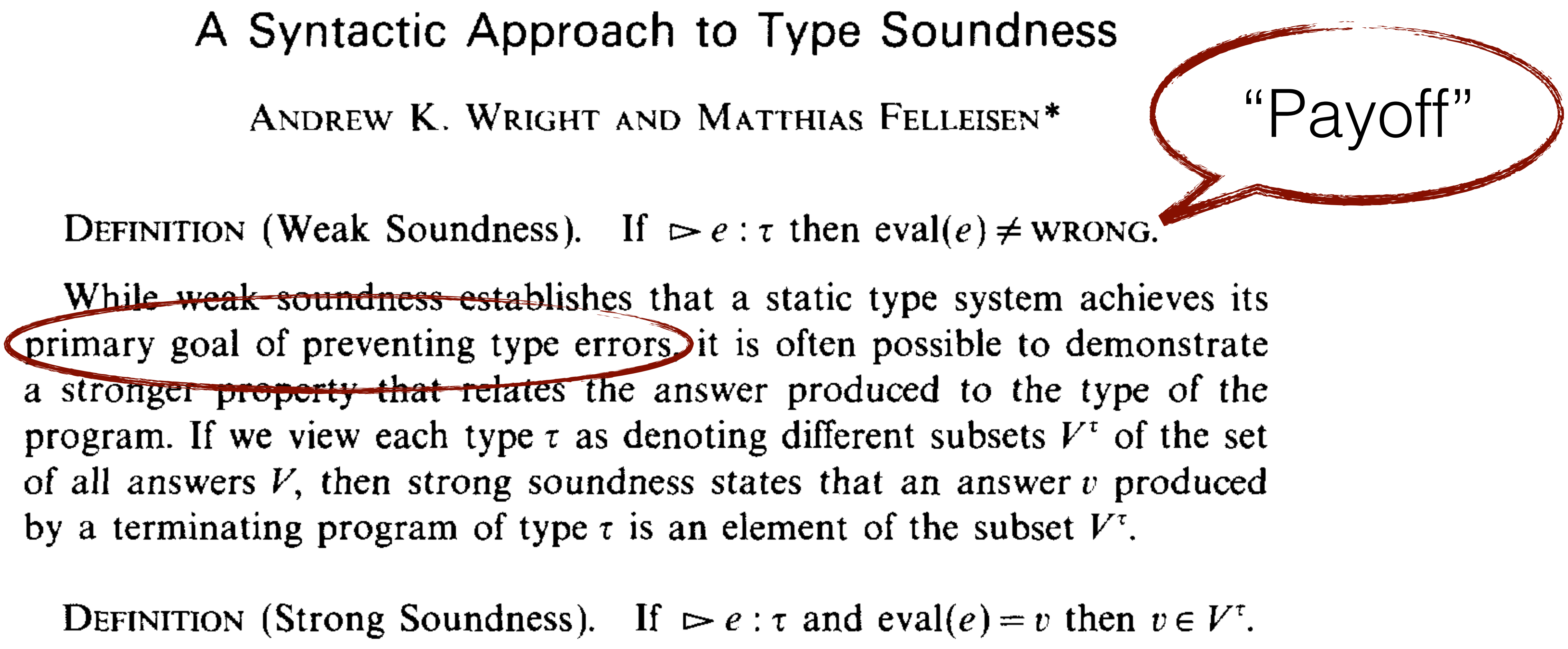
ANDREW K. WRIGHT AND MATTHIAS FELLEISEN*

DEFINITION (Weak Soundness). If $\rhd e : \tau$ then $\text{eval}(e) \neq \text{WRONG}$.

While weak soundness establishes that a static type system achieves its primary goal of preventing type errors, it is often possible to demonstrate a stronger property that relates the answer produced to the type of the program. If we view each type τ as denoting different subsets V^{τ} of the set of all answers V, then strong soundness states that an answer v produced by a terminating program of type τ is an element of the subset V^{τ}.

DEFINITION (Strong Soundness). If $\rhd e : \tau$ and $\text{eval}(e) = v$ then $v \in V^{\tau}$.

Discourse On The Method



A Syntactic Approach to Type Soundness

ANDREW K. WRIGHT AND MATTHIAS FELLEISEN*

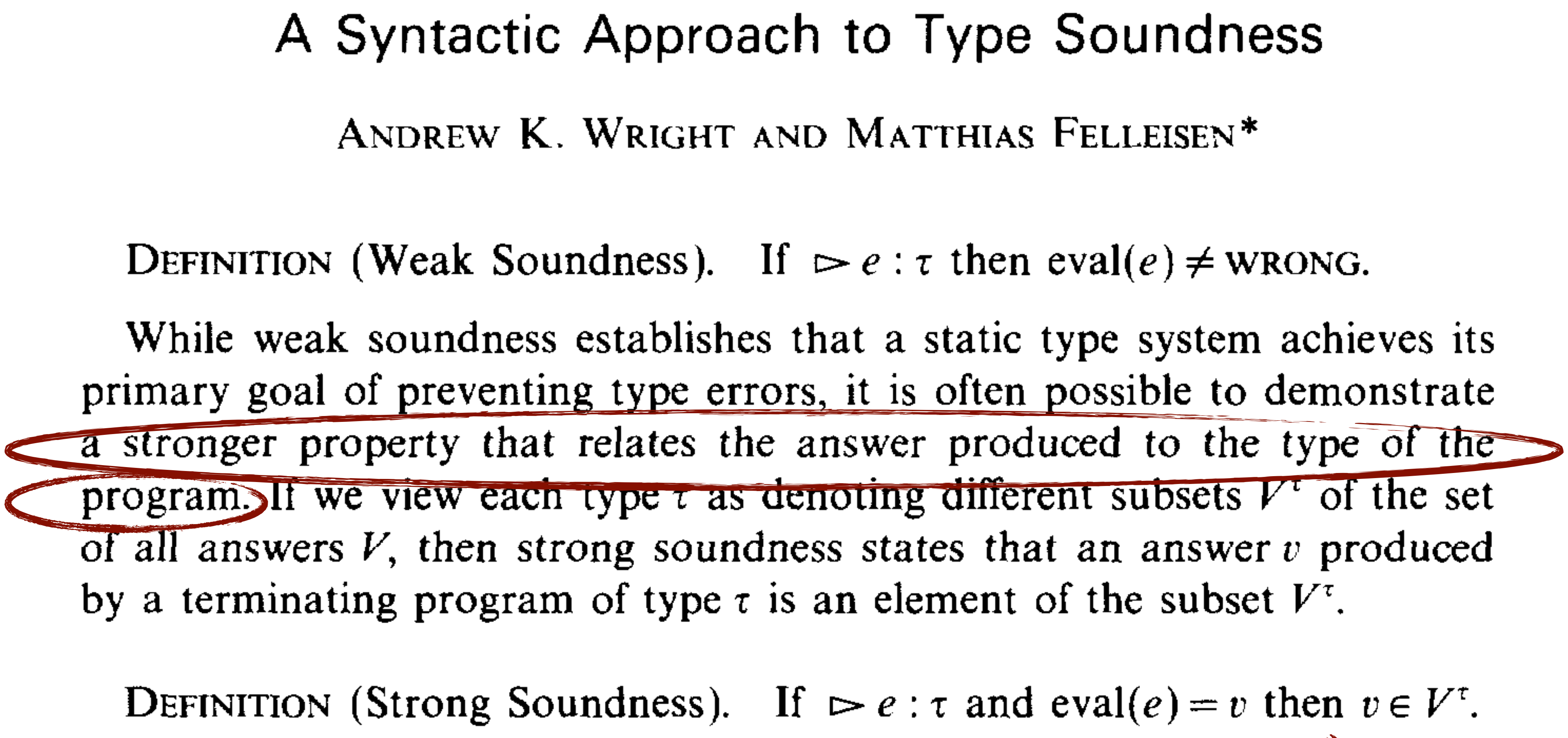
“Payoff”

DEFINITION (Weak Soundness). If $\rhd e : \tau$ then $\text{eval}(e) \neq \text{WRONG}$.

While weak soundness establishes that a static type system achieves its primary goal of preventing type errors, it is often possible to demonstrate a stronger property that relates the answer produced to the type of the program. If we view each type τ as denoting different subsets V^τ of the set of all answers V, then strong soundness states that an answer v produced by a terminating program of type τ is an element of the subset V^τ.

DEFINITION (Strong Soundness). If $\rhd e : \tau$ and $\text{eval}(e) = v$ then $v \in V^\tau$.

Discourse On The Method



A Syntactic Approach to Type Soundness

ANDREW K. WRIGHT AND MATTHIAS FELLEISEN*

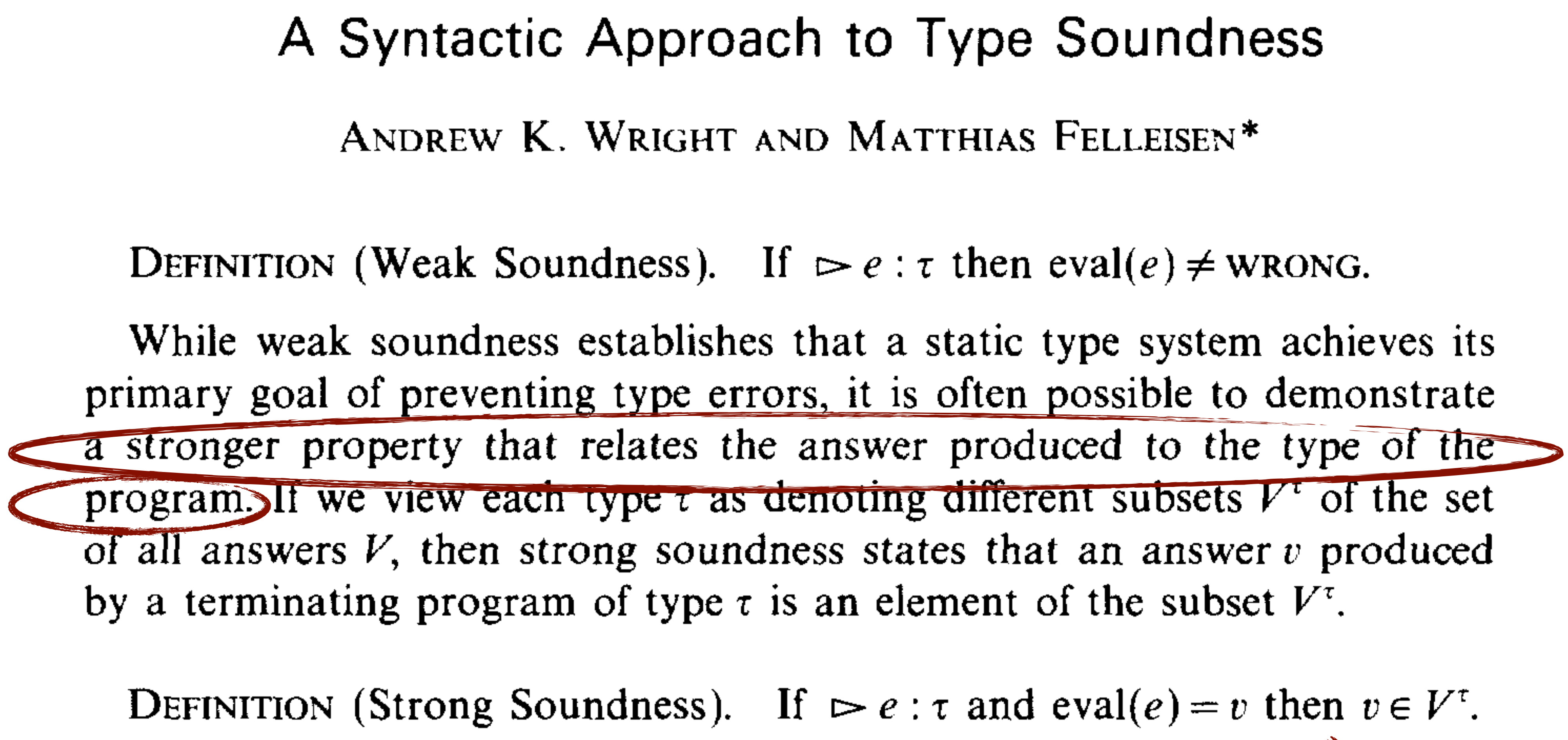
DEFINITION (Weak Soundness). If $\rhd e : \tau$ then $\mathrm{eval}(e) \neq$ WRONG.

While weak soundness establishes that a static type system achieves its primary goal of preventing type errors, it is often possible to demonstrate a stronger property that relates the answer produced to the type of the program. If we view each type τ as denoting different subsets V^{τ} of the set of all answers V, then strong soundness states that an answer v produced by a terminating program of type τ is an element of the subset V^{τ}.

DEFINITION (Strong Soundness). If $\rhd e : \tau$ and $\mathrm{eval}(e) = v$ then $v \in V^{\tau}$.

Behavioural
Invariant

Discourse On The Method



A Syntactic Approach to Type Soundness

ANDREW K. WRIGHT AND MATTHIAS FELLEISEN*

DEFINITION (Weak Soundness). If $\rhd e : \tau$ then $\mathrm{eval}(e) \neq \textsc{wrong}$.

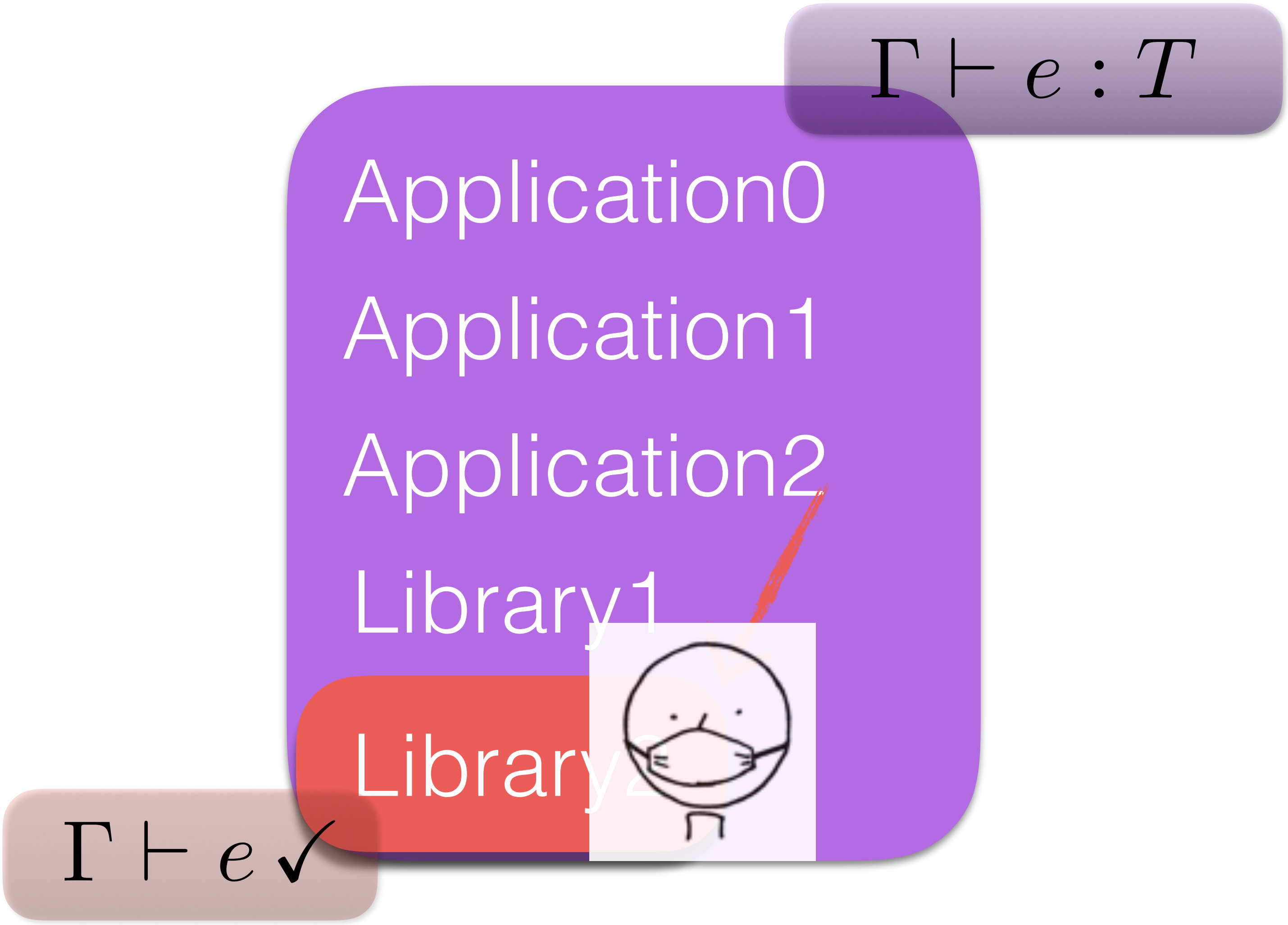
While weak soundness establishes that a static type system achieves its primary goal of preventing type errors, it is often possible to demonstrate a stronger property that relates the answer produced to the type of the program. If we view each type τ as denoting different subsets V^{τ} of the set of all answers V, then strong soundness states that an answer v produced by a terminating program of type τ is an element of the subset V^{τ}.

DEFINITION (Strong Soundness). If $\rhd e : \tau$ and $\mathrm{eval}(e) = v$ then $v \in V^{\tau}$.

Fragment Soundness
is often(*) a Corollary

Behavioural
Invariant

Syntactic Thinking



$\Gamma \vdash e : T$

Application0

Application1

Application2

Library1

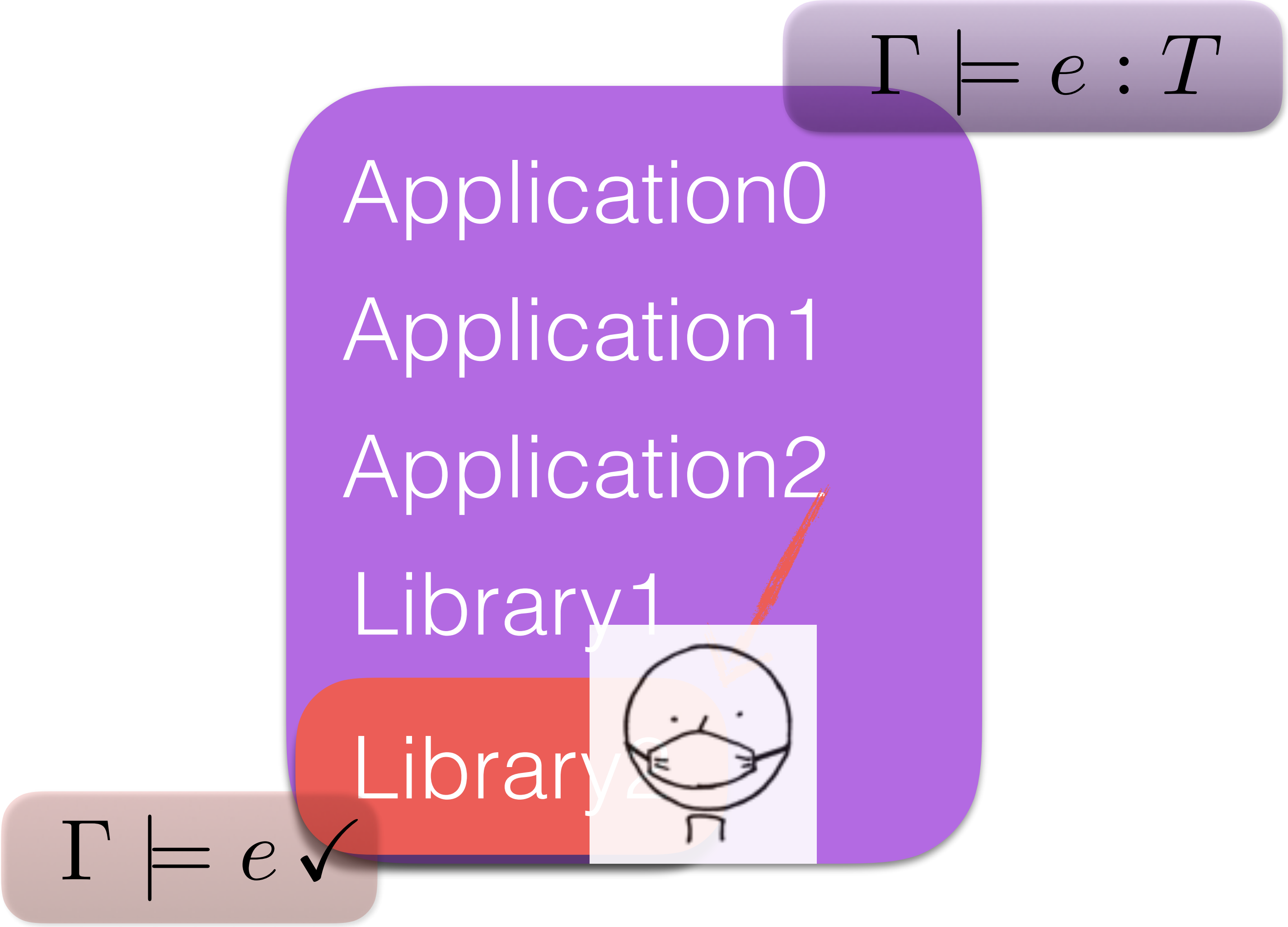
Library2

$\Gamma \vdash e \checkmark$

Two Typing Judgments = Two “Behavioural Contracts”

Conflicts Signal Runtime Errors (go wrong!)

Semantic Thinking



$\Gamma \models e : T$

Application0

Application1

Application2

Library1

Library2

$\Gamma \models e \checkmark$

(Semantically) Sound Gradual Typing

Semantic Judgments Denote “Behavioural Contracts”

“Appropriate Linking” Enforces Contracts

Outline

- Motivating Example (In Two Acts)
- Gradual Typing For All!
- Typin Small Pieces
- Meat
- Strands and Related Works

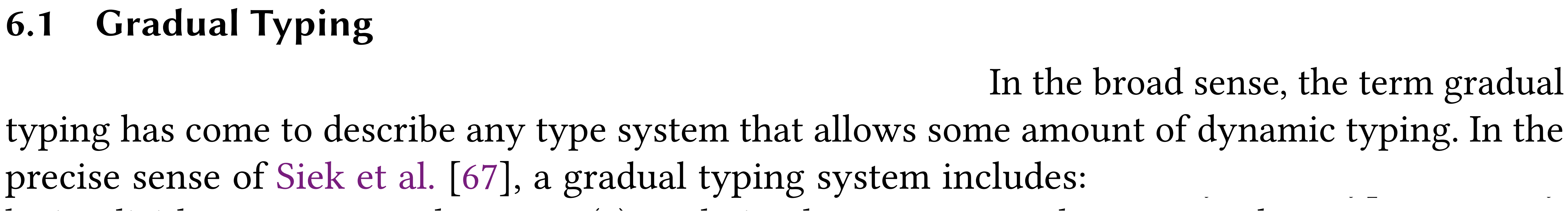
Outline

- Motivating Example (In Two Acts)
- Gradual Typing For All!
- Typin Small Pieces

- Meat →
- Strands and Related Works

Gradual Typing

“Gradual” in which sense?

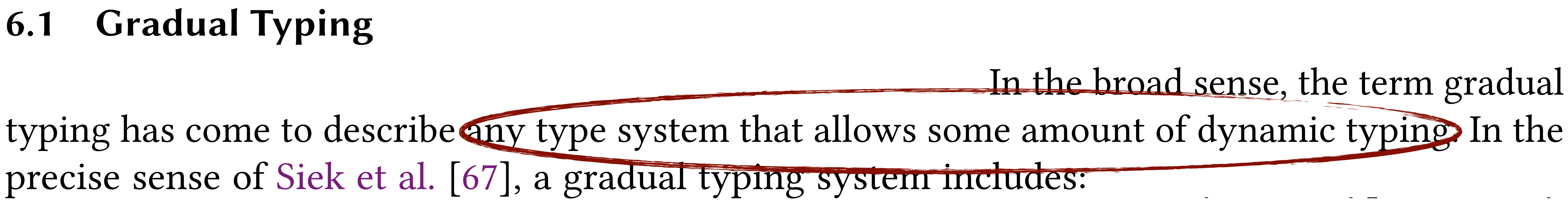


6.1 Gradual Typing

In the broad sense, the term gradual typing has come to describe any type system that allows some amount of dynamic typing. In the precise sense of Siek et al. [67], a gradual typing system includes:

[Greenman & Felleisen ICFP18]

“Gradual” in which sense?

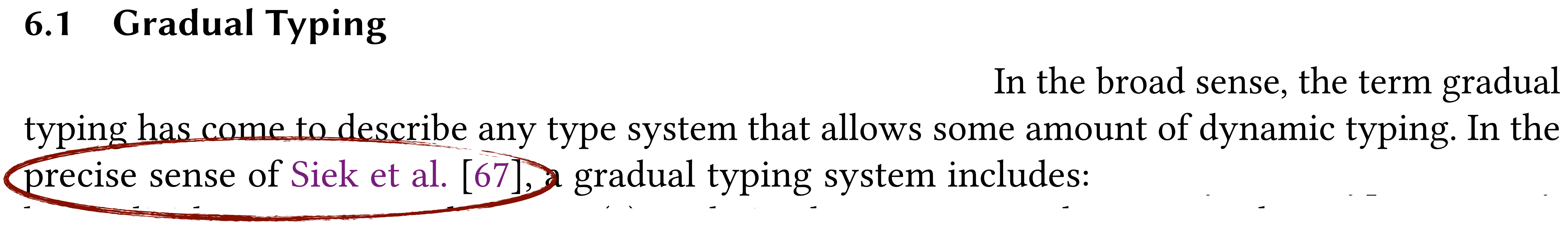


6.1 Gradual Typing

In the broad sense, the term gradual typing has come to describe any type system that allows some amount of dynamic typing. In the precise sense of Siek et al. [67], a gradual typing system includes:

[Greenman & Felleisen ICFP18]

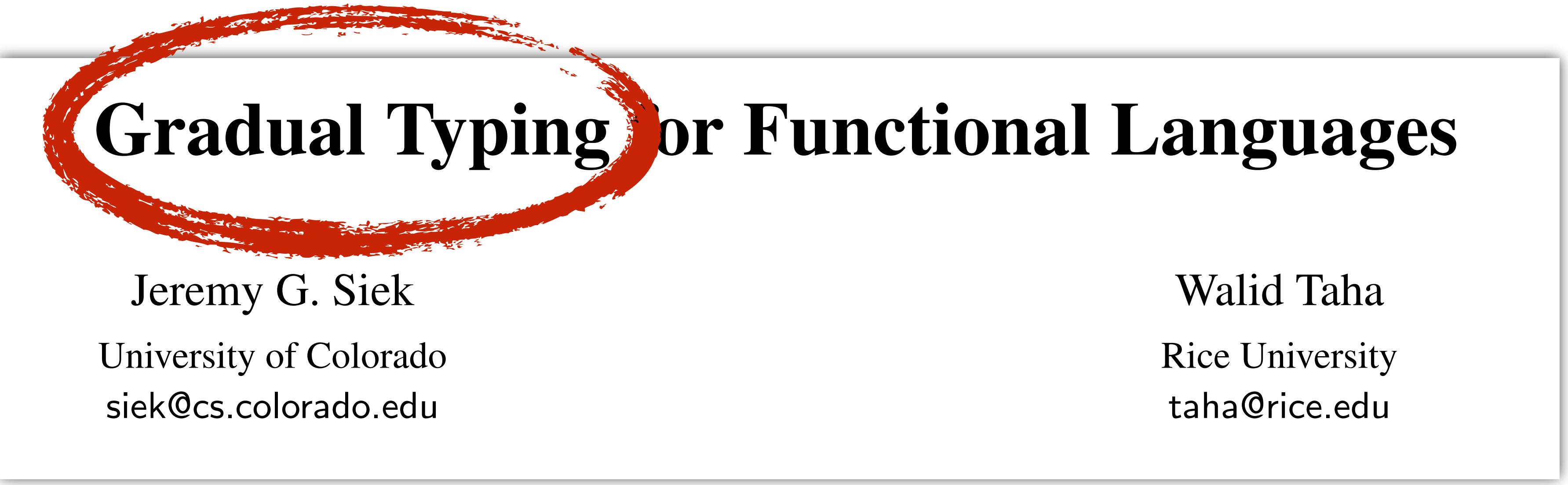
“Gradual” in which sense?



6.1 Gradual Typing

In the broad sense, the term gradual typing has come to describe any type system that allows some amount of dynamic typing. In the precise sense of Siek et al. [67], a gradual typing system includes:

[Greenman & Felleisen ICFP18]



Gradual Typing for Functional Languages

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

Walid Taha
Rice University
taha@rice.edu

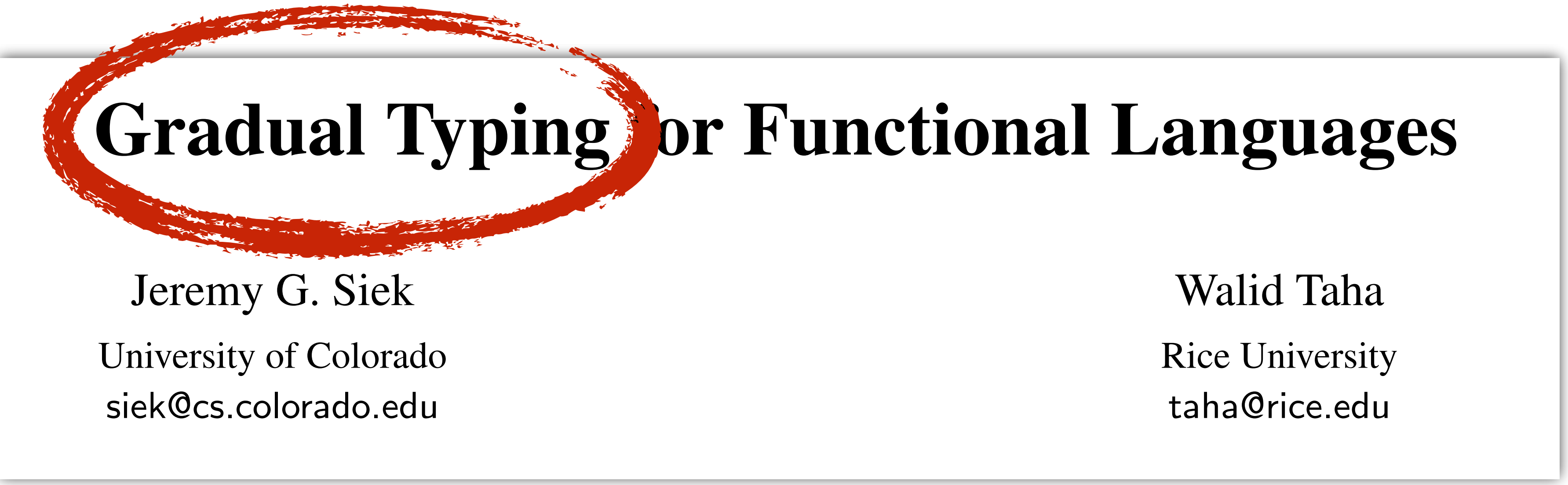
Scheme 2006

Gradual Typing for Functional Languages

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

Walid Taha
Rice University
taha@rice.edu

Rejected from ICFP 2006



Gradual Typing for Functional Languages

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

Walid Taha
Rice University
taha@rice.edu

Rejected from ICFP 2006

>300 citations

Typing Gradually

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

Mixed Checking

Typing Gradually

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

Types might be inferred”

[Siek and Vachharajani DLS08]
[Garcia and Cimini POPL15]

Mixed Checking

Gradual Enforcement

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

✗

static

error

Gradual Enforcement

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

✗

static error

$\Gamma \vdash e : T$

Gradual Enforcement

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

runtime error

Gradual Enforcement

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

`h(f)` → ✗

runtime error

$\Gamma \vdash e : T$

$\Gamma \vdash e \checkmark$

Refined Criteria for Gradual Typing*

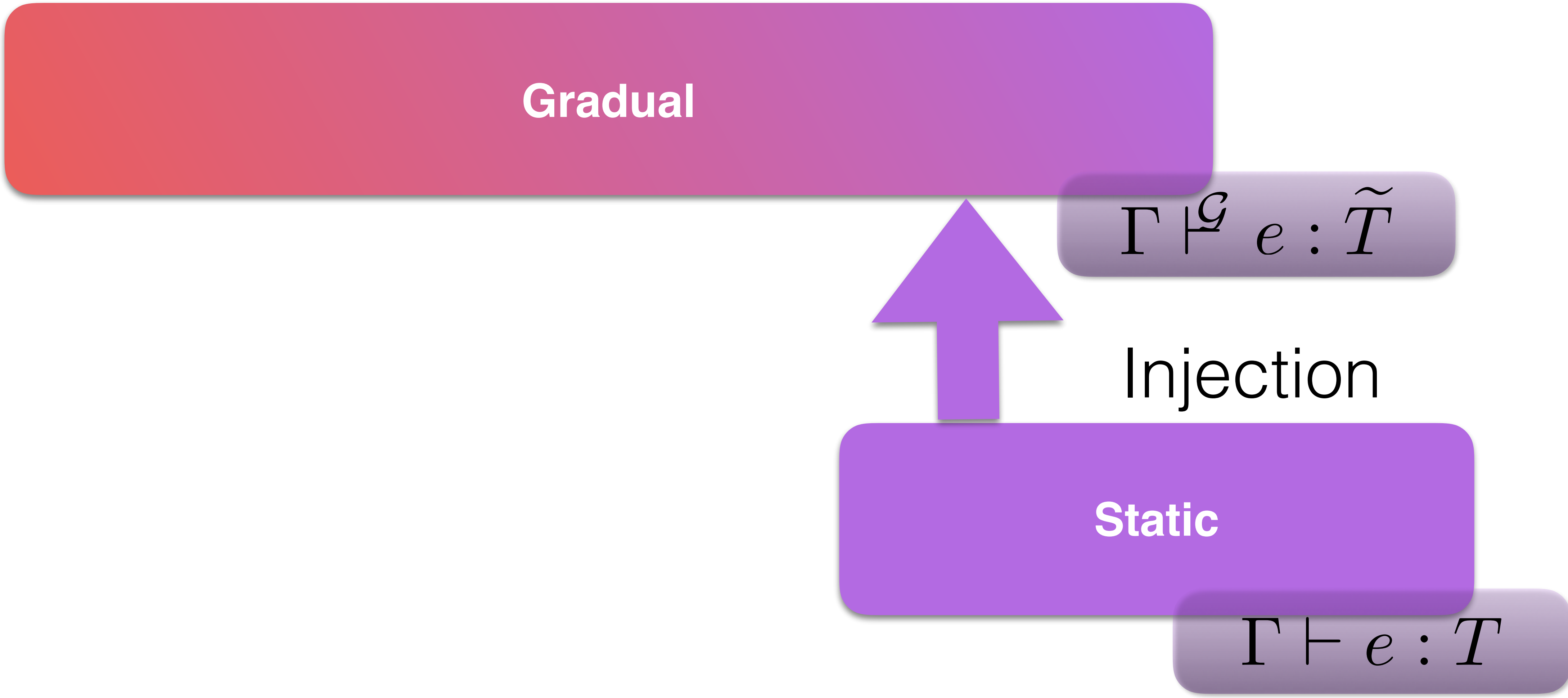
Jeremy G. Siek[1], Michael M. Vitousek[2], Matteo Cimini[3], and John Tang Boyland[4]

Gradual

$$\Gamma \vdash^{\mathcal{G}} e : \widetilde{T}$$

Refined Criteria for Gradual Typing*

Jeremy G. Siek[1], Michael M. Vitousek[2], Matteo Cimini[3], and John Tang Boyland[4]



Gradual

$\Gamma \vdash^{\mathcal{G}} e : \widetilde{T}$

Injection

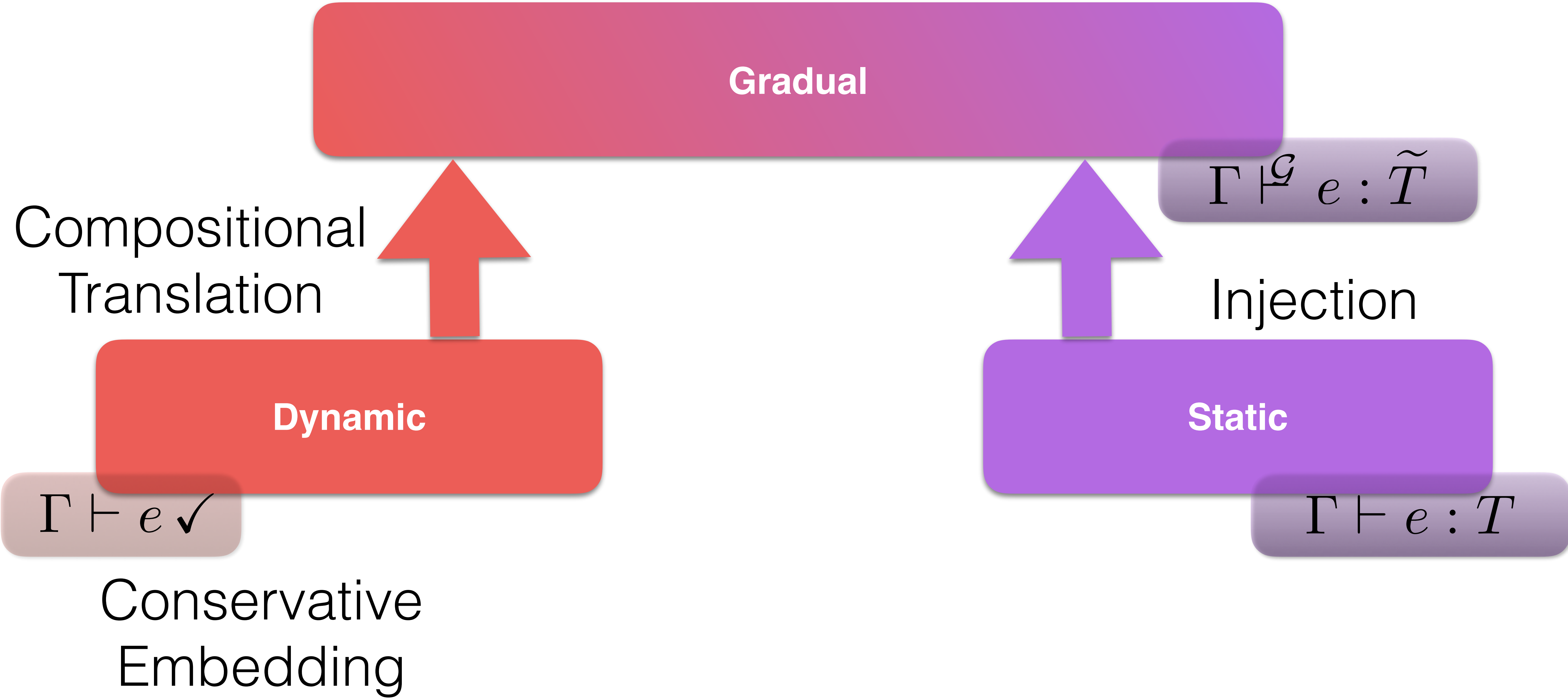
Static

$\Gamma \vdash e : T$

Conservative Embedding

Refined Criteria for Gradual Typing[*]

Jeremy G. Siek[1], Michael M. Vitousek[2], Matteo Cimini[3], and John Tang Boyland[4]



Gradual

$\Gamma \vdash^{\mathcal{G}} e : \widetilde{T}$

Compositional Translation

Injection

Dynamic

Static

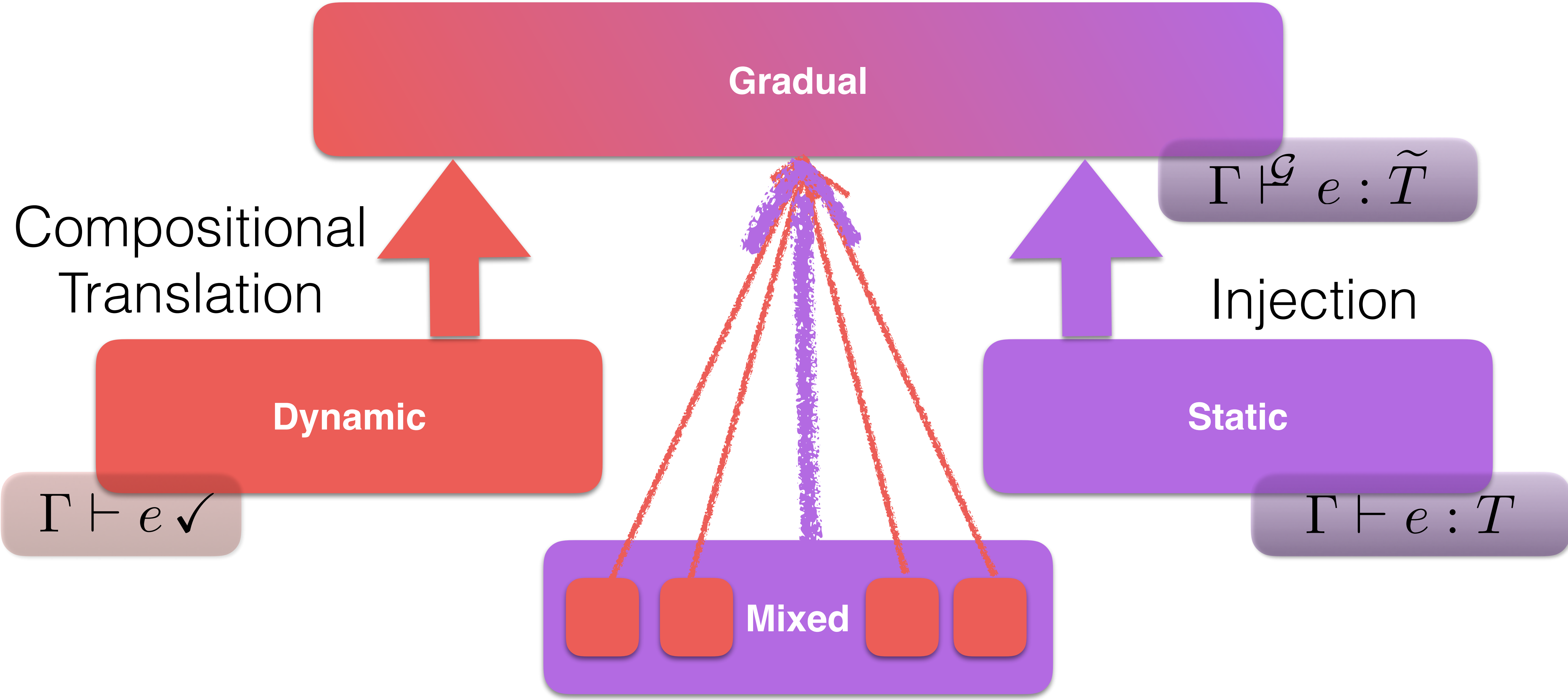
$\Gamma \vdash e \checkmark$

$\Gamma \vdash e : T$

Conservative Embedding

Refined Criteria for Gradual Typing*

Jeremy G. Siek[1], Michael M. Vitousek[2], Matteo Cimini[3], and John Tang Boyland[4]



Gradual

$\Gamma \vdash^{\mathcal{G}} e : \widetilde{T}$

Compositional Translation

Injection

Dynamic

$\Gamma \vdash e \checkmark$

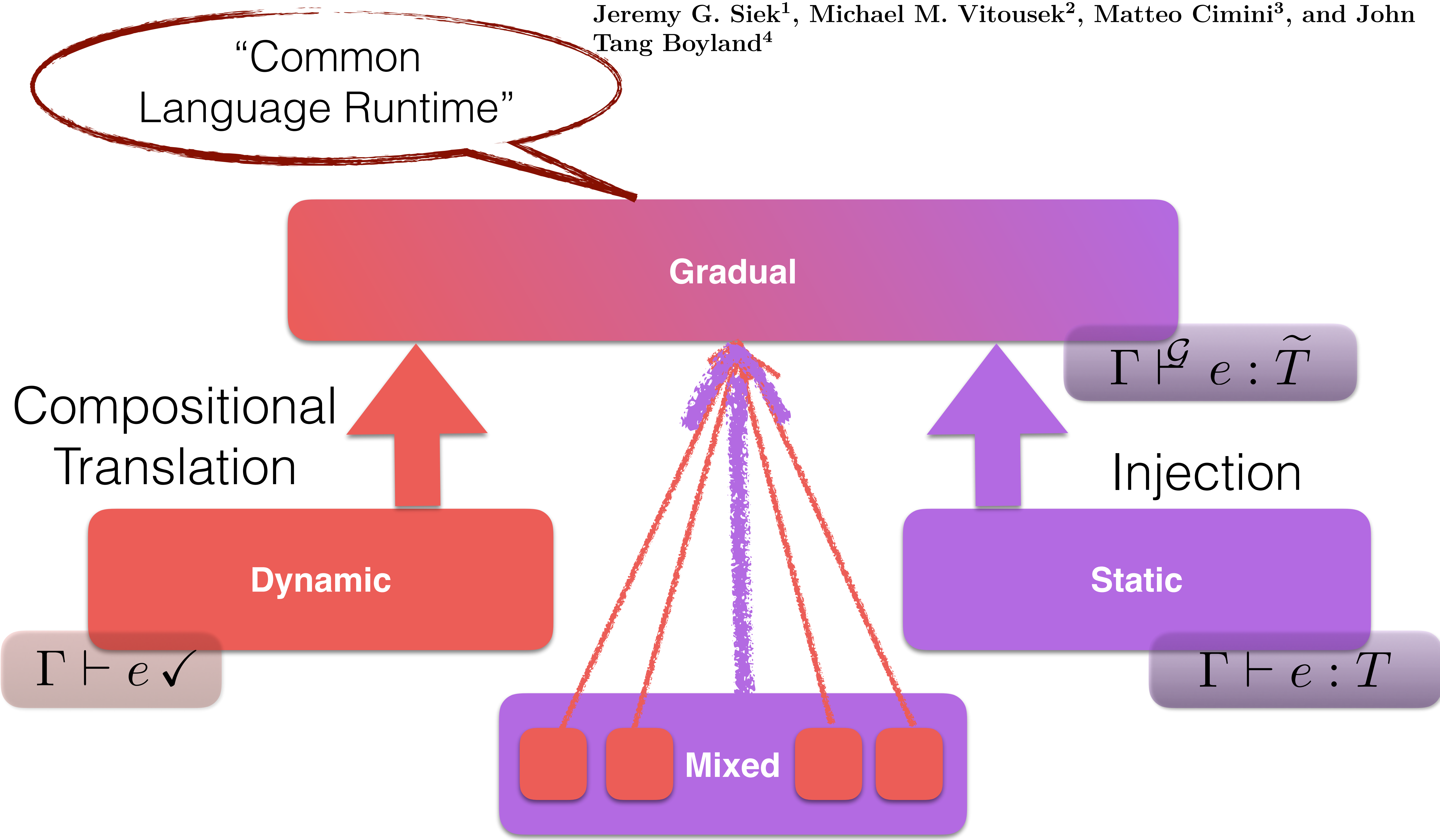
Static

$\Gamma \vdash e : T$

Mixed

Refined Criteria for Gradual Typing*

Jeremy G. Siek[1], Michael M. Vitousek[2], Matteo Cimini[3], and John Tang Boyland[4]



“Common Language Runtime”

Gradual

$\Gamma \vdash^{\mathcal{G}} e : \widetilde{T}$

Compositional Translation

Injection

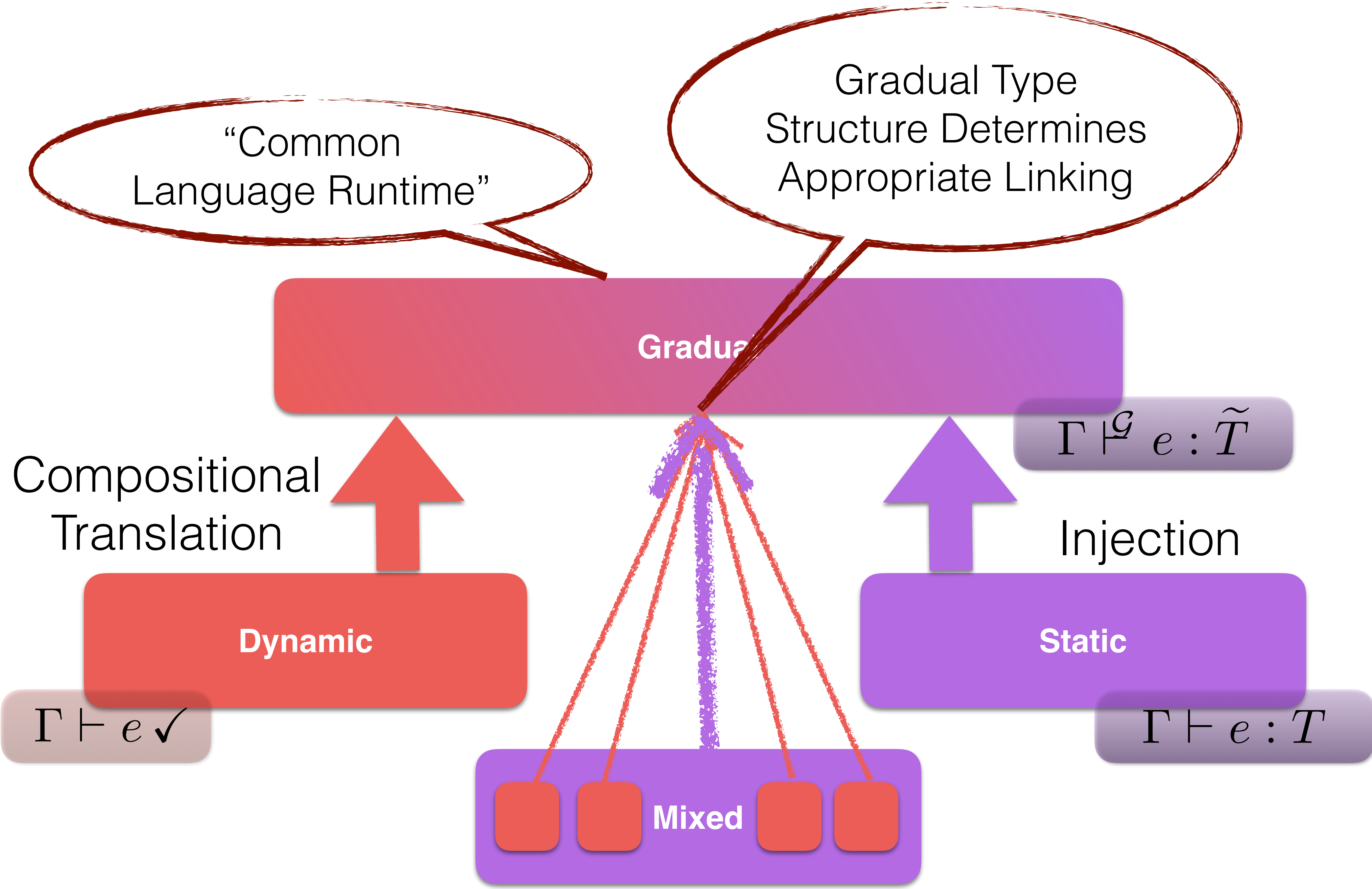
Dynamic

$\Gamma \vdash e \checkmark$

Static

$\Gamma \vdash e : T$

Mixed



“Common Language Runtime”

Gradual Type Structure Determines Appropriate Linking

Gradua

$\Gamma \vdash^{\mathcal{G}} e : \widetilde{T}$

Compositional Translation

Injection

Dynamic

$\Gamma \vdash e \checkmark$

Static

$\Gamma \vdash e : T$

Mixed

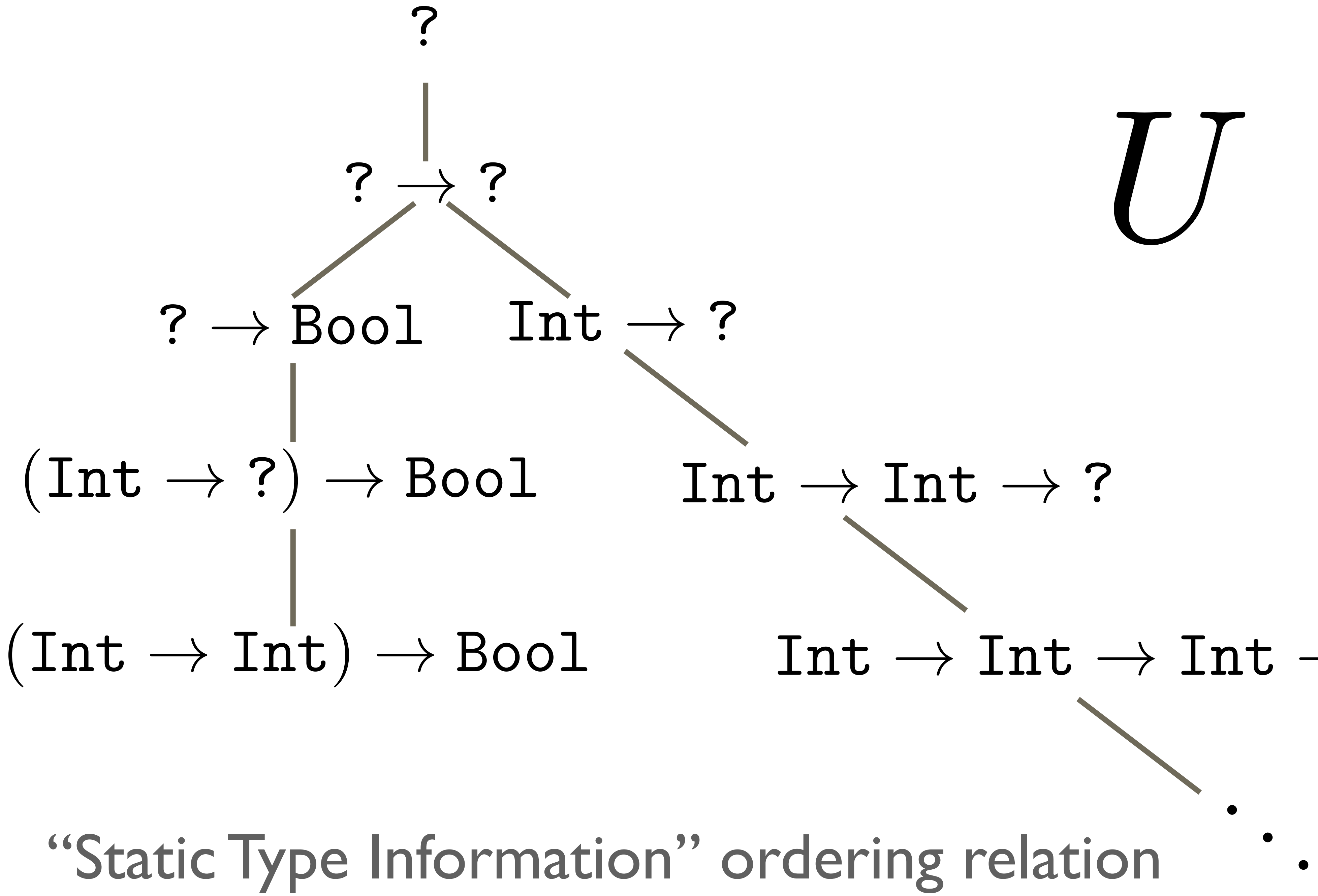
Simple Gradual Types

Static Types (Type) $T ::= B \mid T \rightarrow T$

Gradual Types (GType) $U ::= \texttt{?} \mid B \mid U \rightarrow U$

$$\textsc{Type} \subseteq \textsc{GType}$$

Gradual Type Precision



$\texttt{?}$

$\texttt{?} \to \texttt{?}$

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

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

$(\texttt{Int} \to \texttt{?}) \to \texttt{Bool}$

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

$(\texttt{Int} \to \texttt{Int}) \to \texttt{Bool}$

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

$\ddots$

$$U \sqsubseteq U$$

"Static Type Information" ordering relation

Consistent Lifting(*)

$$U_1 \sim U_2$$

Gradual Type
Consistency

(*) Reformulation of original definition

Consistent Lifting(*)

$$\begin{array}{ccc} U_1 & \sim & U_2 \\ \rotatebox{90}{$\sqsubseteq$} & & \rotatebox{90}{$\sqsubseteq$} \\ T_1 & = & T_2 \end{array}$$

Gradual Type Consistency

if and only if

Static Type Equality

For some T1 and T2

(*) Reformulation of original definition

Static Checking

static type equality

$$\mathsf{Int} = \mathsf{Int}$$

$$\mathsf{Bool} = \mathsf{Bool}$$

$$\mathsf{Int} \rightarrow \mathsf{Bool} \neq \mathsf{Bool} \rightarrow \mathsf{Int}$$

extend

gradual type consistency

$$\mathsf{Int} \sim \mathsf{Int}$$

$$\mathsf{Bool} \sim \mathsf{Bool}$$

$$\mathsf{Int} \rightarrow \mathsf{Bool} \not\sim \mathsf{Bool} \rightarrow \mathsf{Int}$$

$$? \sim \mathsf{Bool}$$

$$? \rightarrow \mathsf{Bool} \sim \mathsf{Bool} \rightarrow ?$$

Consistency conservatively extends equality

Static Checking

static type equality

$$\text{Int} = \text{Int}$$

$$\boxed{\text{Bool} = \text{Bool}}$$

$$\text{Int} \to \text{Bool} \neq \text{Bool} \to \text{Int}$$

extend

gradual type consistency

$$\text{Int} \sim \text{Int}$$

$$\boxed{\text{Bool} \sim \text{Bool}}$$

$$\text{Int} \to \text{Bool} \not\sim \text{Bool} \to \text{Int}$$

$$? \sim \text{Bool}$$

$$? \to \text{Bool} \sim \text{Bool} \to ?$$

Consistency conservatively extends equality

Static Checking

static type equality

$$\mathsf{Int} = \mathsf{Int}$$

$$\mathsf{Bool} = \mathsf{Bool}$$

$$\mathsf{Int} \to \mathsf{Bool} \neq \mathsf{Bool} \to \mathsf{Int}$$

extend

gradual type consistency

$$\mathsf{Int} \sim \mathsf{Int}$$

$$\mathsf{Bool} \sim \mathsf{Bool}$$

$$\mathsf{Int} \to \mathsf{Bool} \not\sim \mathsf{Bool} \to \mathsf{Int}$$

$$? \sim \mathsf{Bool}$$

$$? \to \mathsf{Bool} \sim \mathsf{Bool} \to ?$$

Consistency conservatively extends equality

Static Checking

static type equality

$$\text{Int} = \text{Int}$$

$$\text{Bool} = \text{Bool}$$

$$\text{Int} \to \text{Bool} \neq \text{Bool} \to \text{Int}$$

extend

gradual type consistency

$$\text{Int} \sim \text{Int}$$

$$\text{Bool} \sim \text{Bool}$$

$$\text{Int} \to \text{Bool} \not\sim \text{Bool} \to \text{Int}$$

$$? \sim \text{Bool}$$

$$? \to \text{Bool} \sim \text{Bool} \to ?$$

Consistency conservatively extends equality

Consistent Lifting(*)

$$U_1 \lesssim U_2$$

Consistent Subtyping

if and only if

$$\begin{array}{ccc} U_1 & \lesssim & U_2 \\ \rotatebox{90}{$\sqsubseteq$} & & \rotatebox{90}{$\sqsubseteq$} \\ T_1 & <: & T_2 \end{array}$$

Static Subtyping

For some T1 and T2

(*) Reformulation of original definition

Consistent Lifting

$$\texttt{Int} \lesssim \texttt{Int}$$

$$\texttt{Int} \not\lesssim \texttt{Bool}$$

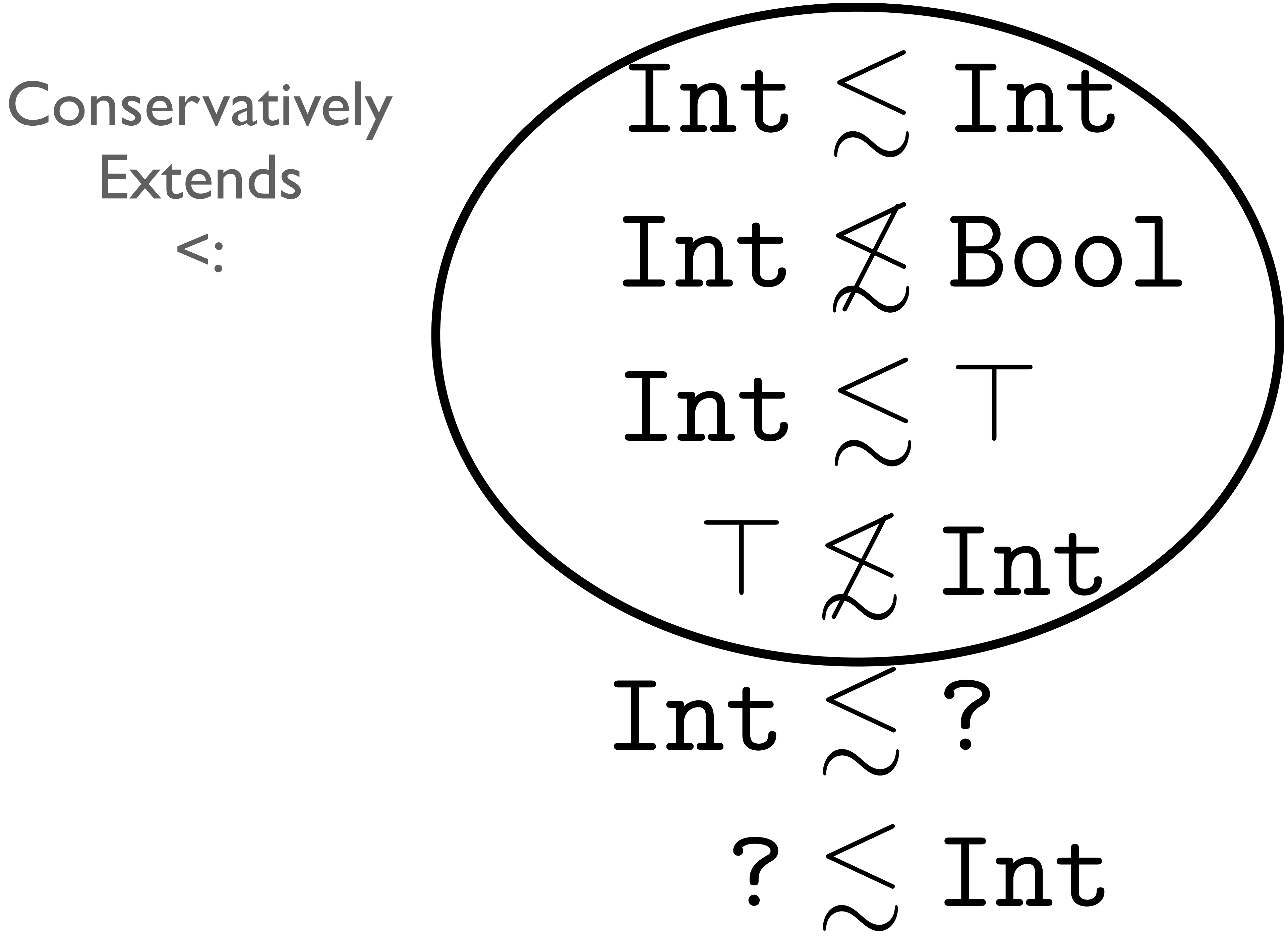
$$\texttt{Int} \lesssim \top$$

$$\top \not\lesssim \texttt{Int}$$

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

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

Consistent Lifting



Conservatively
Extends
<:

$$\texttt{Int} \lesssim \texttt{Int}$$

$$\texttt{Int} \not\lesssim \texttt{Bool}$$

$$\texttt{Int} \lesssim \top$$

$$\top \not\lesssim \texttt{Int}$$

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

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

Consistent Lifting

$$\mathtt{Int} \lesssim \mathtt{Int}$$

$$\mathtt{Int} \not\lesssim \mathtt{Bool}$$

$$\mathtt{Int} \lesssim \top$$

$$\top \not\lesssim \mathtt{Int}$$

$$\mathtt{Int} \lesssim \mathtt{?}$$

$$\mathtt{?} \lesssim \mathtt{Int}$$

"unknown"
is not the
"top" type

Lift Typing RuLes

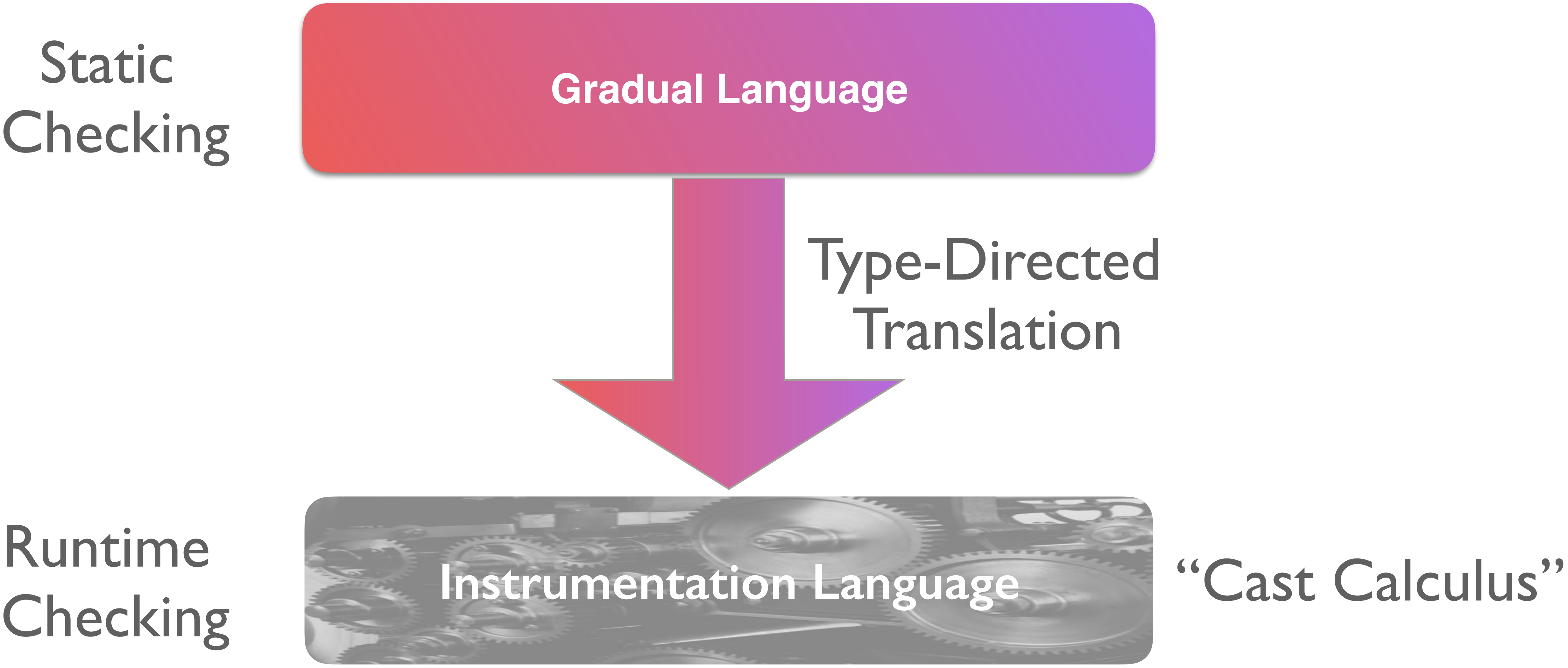
Static Type System

$$\frac{\Gamma \vdash t_1 : T_1 \qquad T_1 = \texttt{Int} \qquad \Gamma \vdash t_2 : T_2 \qquad T_2 = \texttt{Int}}{\Gamma \vdash t_1 + t_2 : \texttt{Int}}$$

Gradual Type System

$$\frac{\Gamma \vdash t_1 : U_1 \qquad U_1 \sim \texttt{Int} \qquad \Gamma \vdash t_2 : U_2 \qquad U_2 \sim \texttt{Int}}{\Gamma \vdash t_1 + t_2 : \texttt{Int}}$$

Dynamic Semantics



Static Checking

Gradual Language

Type-Directed Translation

Runtime Checking

Instrumentation Language

“Cast Calculus”

“Common Language Runtime”

Gradual

“Common Language Runtime”

Gradual

Static Types (Type) $T ::= B \mid T \to T$

Gradual Types (GType) $U ::= \textcircled{?} \mid B \mid U \to U$

Also Works as a
Surface Language!

Gradual

Static Types (Type) $T ::= B \mid T \to T$

Gradual Types (GType) $U ::= ? \mid B \mid U \to U$

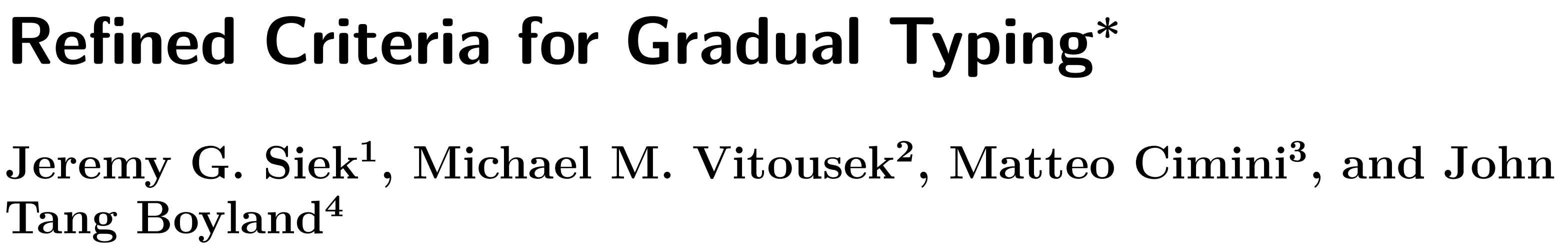
Also Works as a
Surface Language!

Gradual

Static Types (Type) $T ::= B \mid T \to T$

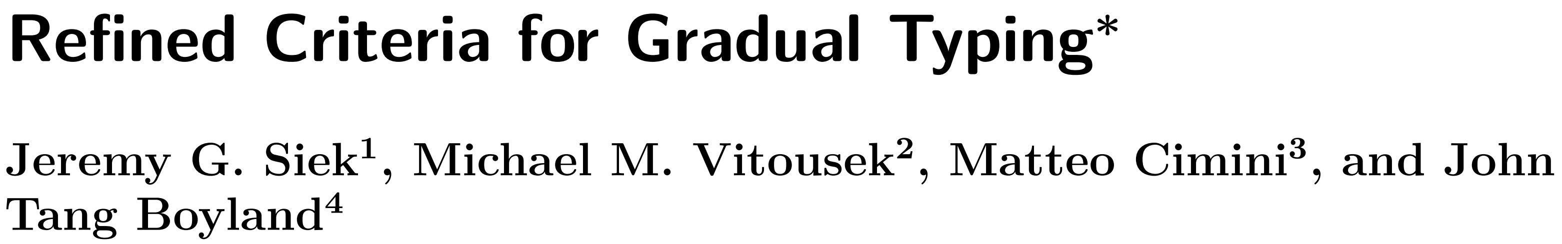
Gradual Types (GType) $U ::= \textcircled{?} \mid B \mid U \to U$

Much of the Literature
is Written This Way



Refined Criteria for Gradual Typing*

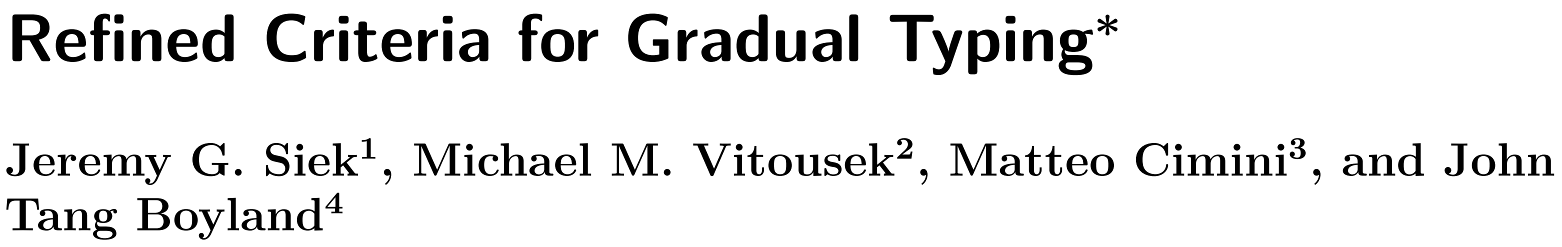
Jeremy G. Siek[1], Michael M. Vitousek[2], Matteo Cimini[3], and John Tang Boyland[4]



Refined Criteria for Gradual Typing*

Jeremy G. Siek[1], Michael M. Vitousek[2], Matteo Cimini[3], and John Tang Boyland[4]

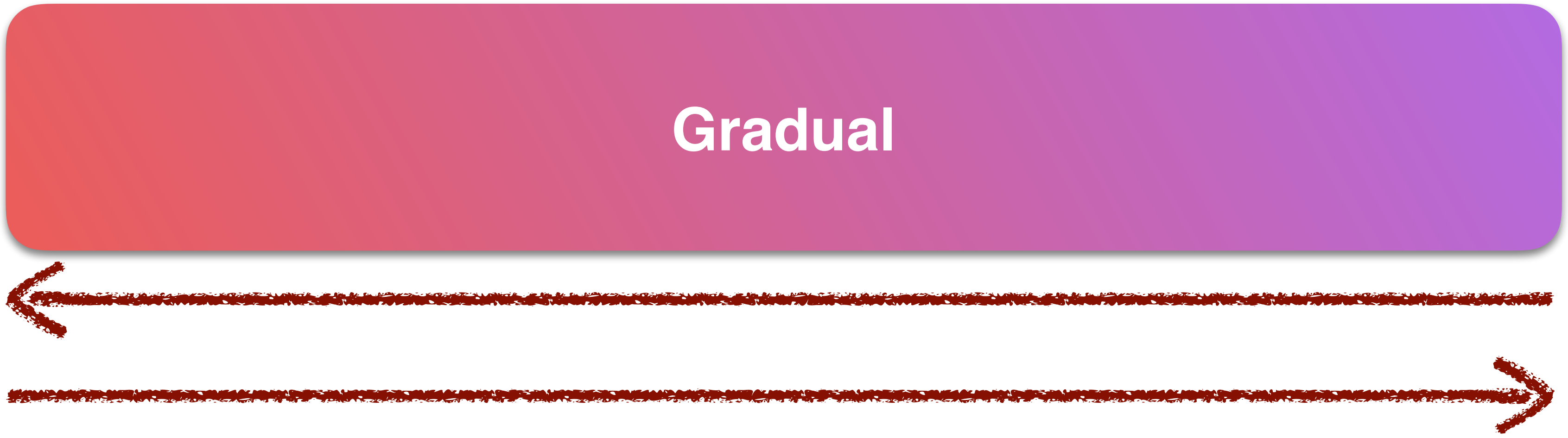
Static and Dynamic Gradual Guarantee!

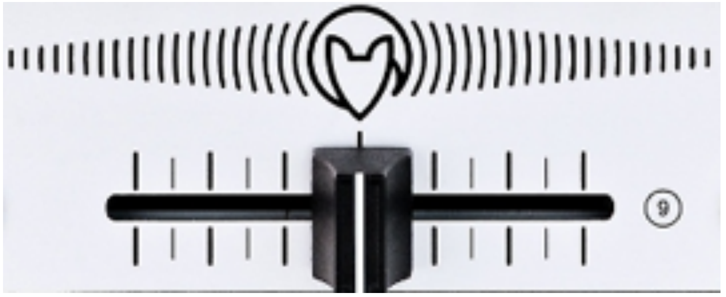


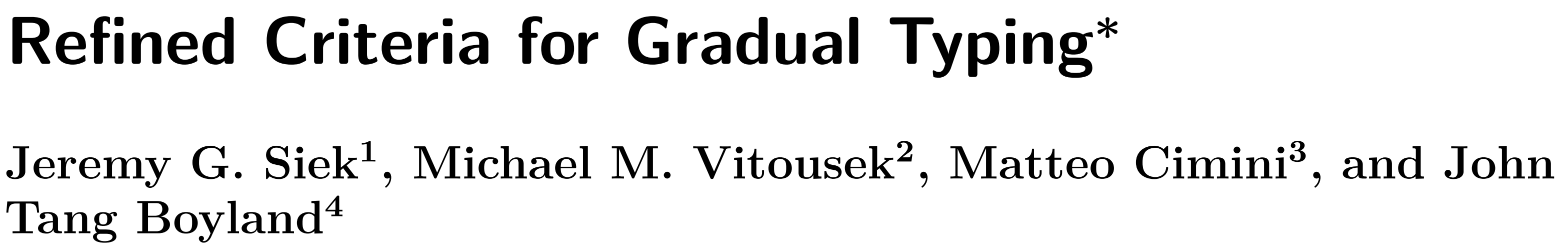
Refined Criteria for Gradual Typing*

Jeremy G. Siek[1], Michael M. Vitousek[2], Matteo Cimini[3], and John Tang Boyland[4]

Static and Dynamic Gradual Guarantee!



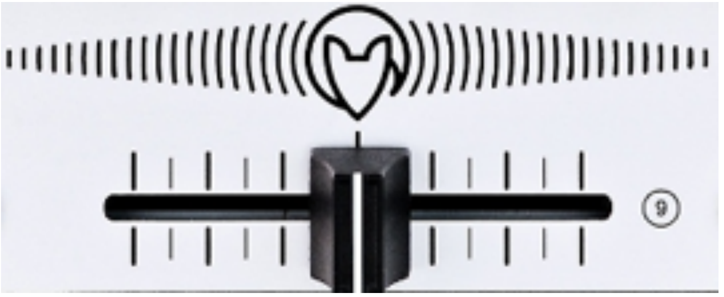




Refined Criteria for Gradual Typing*

Jeremy G. Siek[1], Michael M. Vitousek[2], Matteo Cimini[3], and John Tang Boyland[4]

Static and Dynamic Gradual Guarantee!



Varying The Type Precision of a Program
Monotonically Changes **only**
static and dynamic type errors

Robust Theoretical Framework

“Dynamic” **Gradual** **“Static”**

Robust Theoretical Framework

“**Dynamic**” **Gradual** “**Static**”

Unityped [Siek and Taha 06] Simple

Robust Theoretical Framework

“Dynamic”	**Gradual**	**“Static”**
Unityped	[Siek and Taha 06]	Simple
Unityped	[Siek and Taha 08]	Subtyping

Robust Theoretical Framework

“Dynamic”	**Gradual**	**“Static”**
Unityped	[Siek and Taha 06]	Simple
Unityped	[Siek and Taha 08]	Subtyping
Unityped	[Siek and Vachharajani 08]	Hindley/Milner

Robust Theoretical Framework

“**Dynamic**”	**Gradual**	“**Static**”
Unityped	[Siek and Taha 06]	Simple
Unityped	[Siek and Taha 08]	Subtyping
Unityped	[Siek and Vachharajani 08]	Hindley/Milner
Simple	[Lehmann and Tanter 17]	Refinement

Robust Theoretical Framework

“**Dynamic**”	**Gradual**	“**Static**”
Unityped	[Siek and Taha 06]	Simple
Unityped	[Siek and Taha 08]	Subtyping
Unityped	[Siek and Vachharajani 08]	Hindley/Milner
Simple	[Lehmann and Tanter 17]	Refinement
Simple	[Bañados et al. 14]	Type&Effect

Robust Theoretical Framework

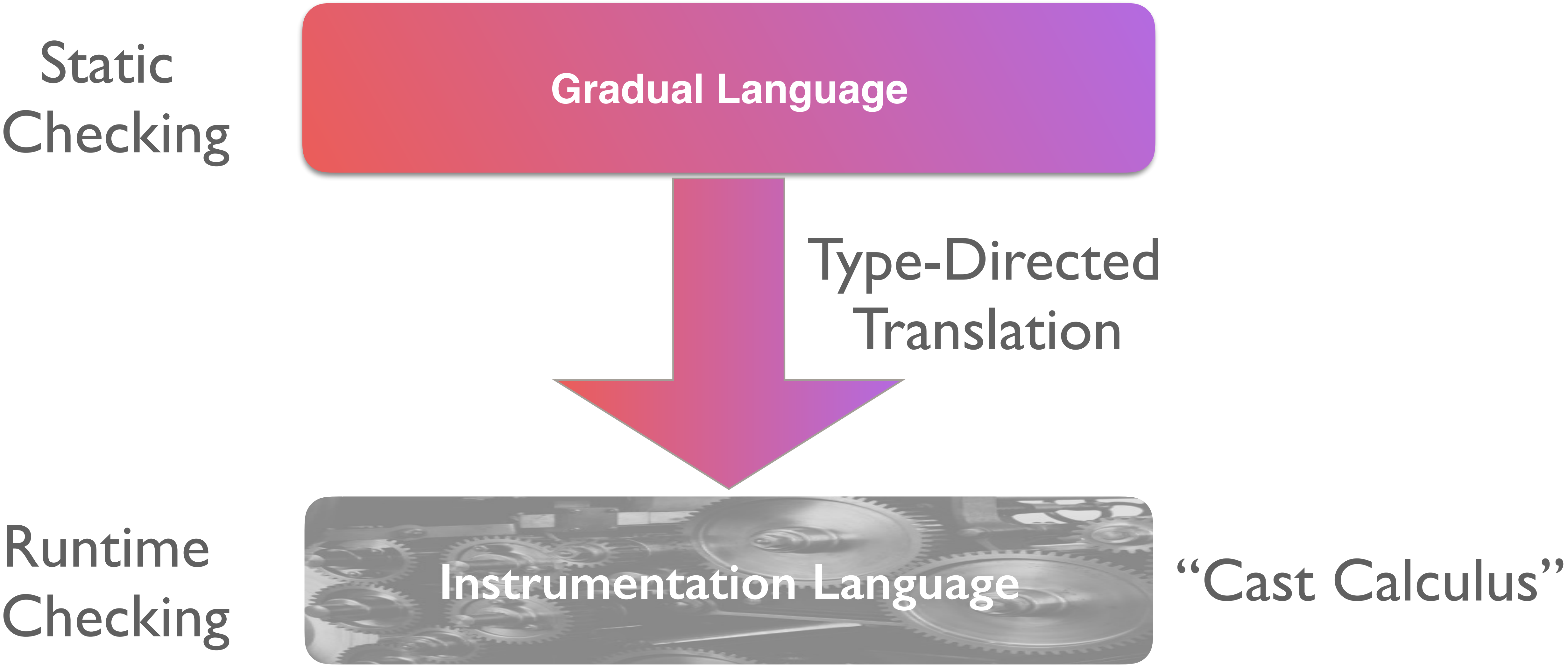
“**Dynamic**”	**Gradual**	“**Static**”
Unityped	[Siek and Taha 06]	Simple
Unityped	[Siek and Taha 08]	Subtyping
Unityped	[Siek and Vachharajani 08]	Hindley/Milner
Simple	[Lehmann and Tanter 17]	Refinement
Simple	[Bañados et al. 14]	Type&Effect
Simple	[Toro et al. to appear]	Security

Robust Theoretical Framework

“**Dynamic**”	**Gradual**	“**Static**”
Unityped	[Siek and Taha 06]	Simple
Unityped	[Siek and Taha 08]	Subtyping
Unityped	[Siek and Vachharajani 08]	Hindley/Milner
Simple	[Lehmann and Tanter 17]	Refinement
Simple	[Bañados et al. 14]	Type&Effect
Simple	[Toro et al. to appear]	Security

And More!

Challenge: Dynamics



Static Checking

Gradual Language

Type-Directed Translation

Runtime Checking

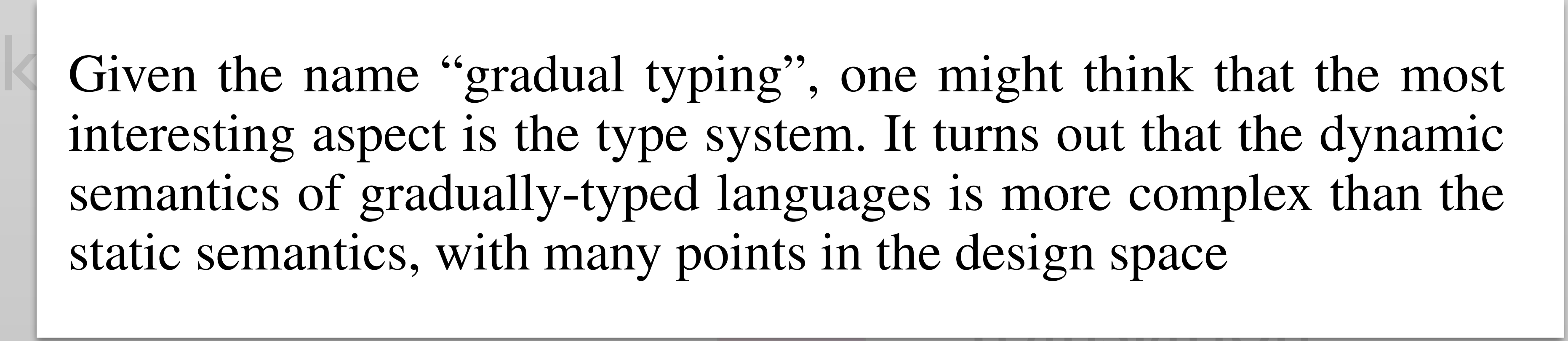
Instrumentation Language

“Cast Calculus”

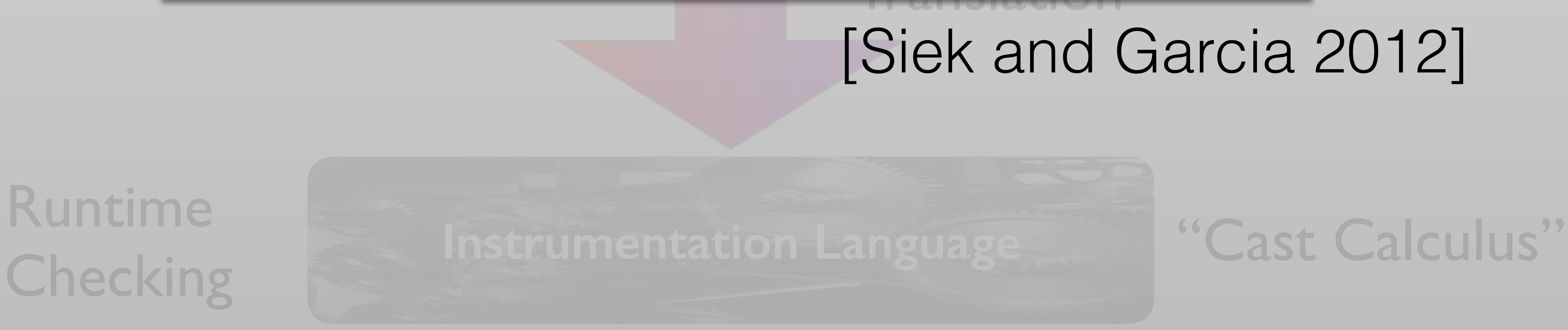
Challenge: Dynamics

Static
Check

Gradual Language



Given the name “gradual typing”, one might think that the most interesting aspect is the type system. It turns out that the dynamic semantics of gradually-typed languages is more complex than the static semantics, with many points in the design space



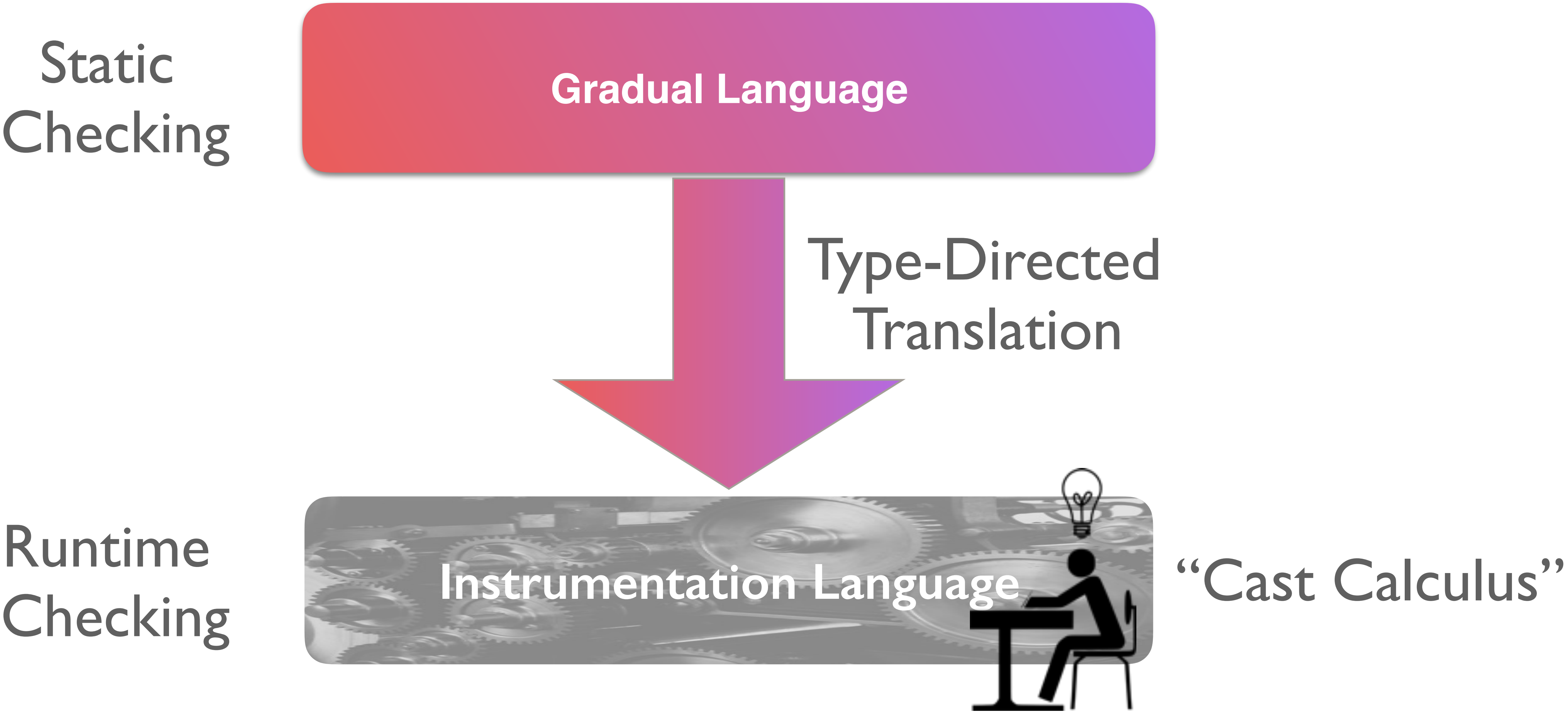
[Siek and Garcia 2012]

Runtime
Checking

Instrumentation Language

“Cast Calculus”

Challenge: Dynamics



Static Checking

Gradual Language

Type-Directed Translation

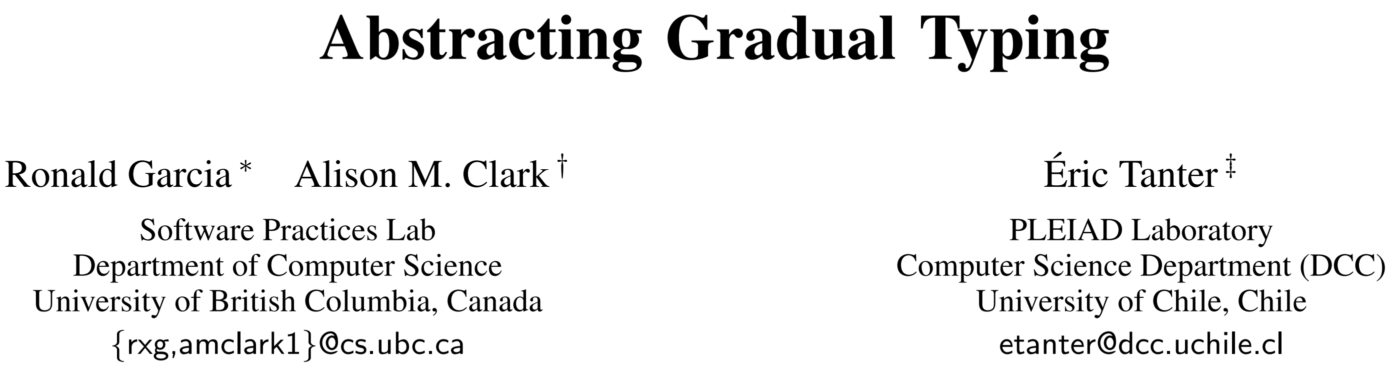
Runtime Checking

Instrumentation Language

“Cast Calculus”

static type system &
type safety proof

interpretation of
gradual types



Abstracting Gradual Typing

Ronald Garcia [*] Alison M. Clark [†]
Software Practices Lab
Department of Computer Science
University of British Columbia, Canada
{rxg,amclark1}@cs.ubc.ca

Éric Tanter [‡]
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile, Chile
etanter@dcc.uchile.cl

POPL 2016

gradual language

TYPE SYSTEM

DYNAMIC SEMANTICS

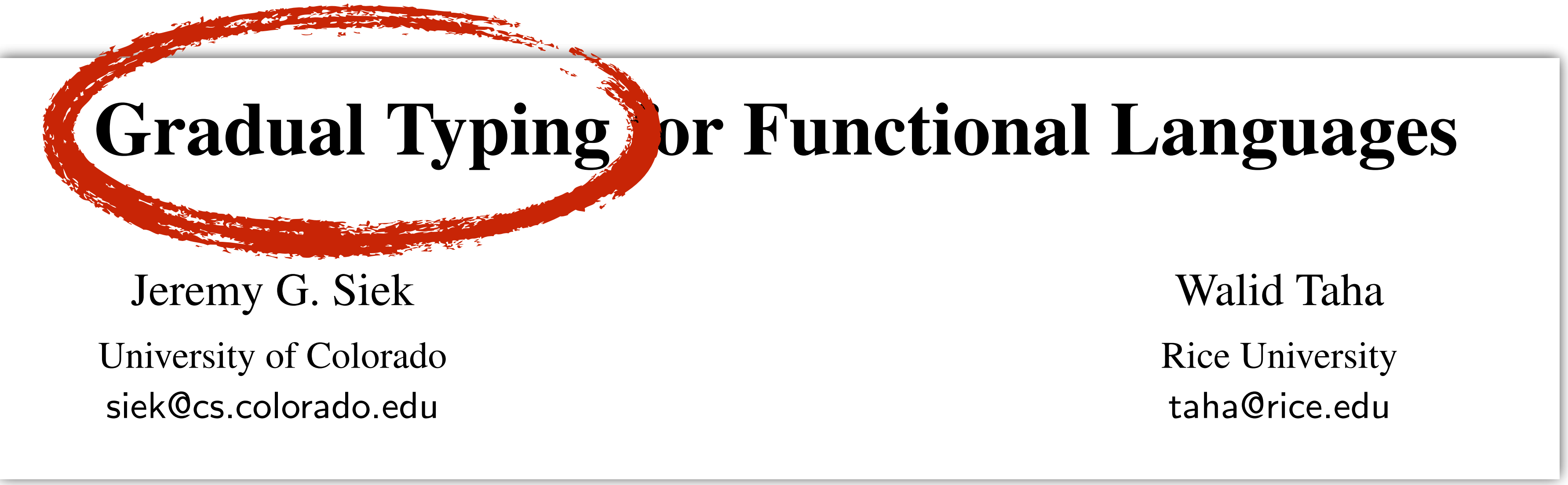
Breadth of AGT

- Applications of AGT so far

records with subtyping	POPL’16
gradual rows (à la row polymorphism)	
security typing	TOPLAS‘18
effect typing	ICFP’14 (statics)
refinement types	POPL‘17
set-theoretic types	ICFP’17 (statics)
parametric polymorphism	ongoing work

Outline

- Motivating Example (In Two Acts)
- Gradual Typing For All!
- Typing in Small Pieces
- Meat
- Strands and Related Works



Gradual Typing for Functional Languages

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

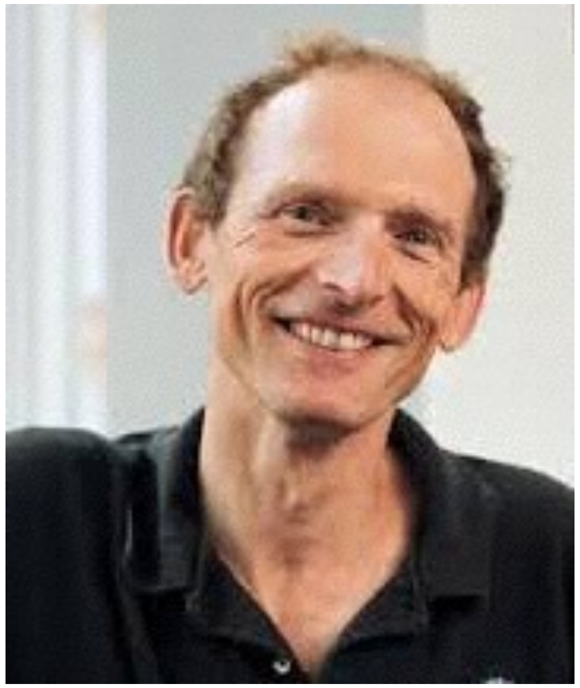
Walid Taha
Rice University
taha@rice.edu

Scheme 2006

Interlanguage Migration: From Scripts to Programs

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

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

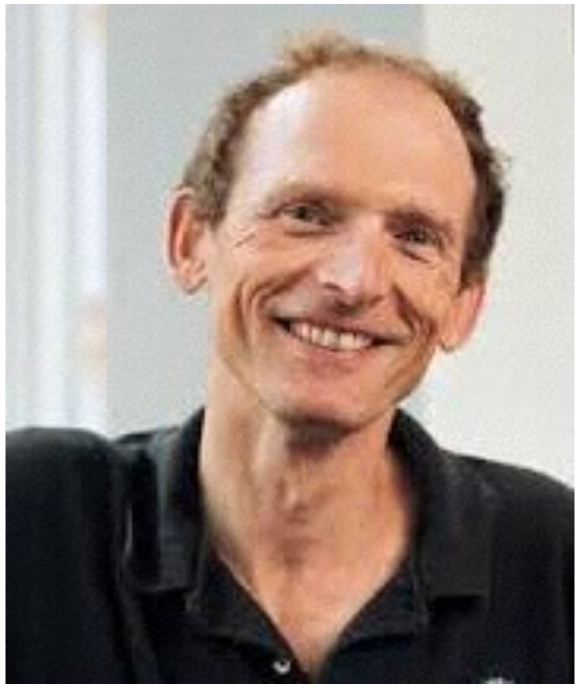


DLS 2006

Interlanguage Migration: From Scripts to Programs

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

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

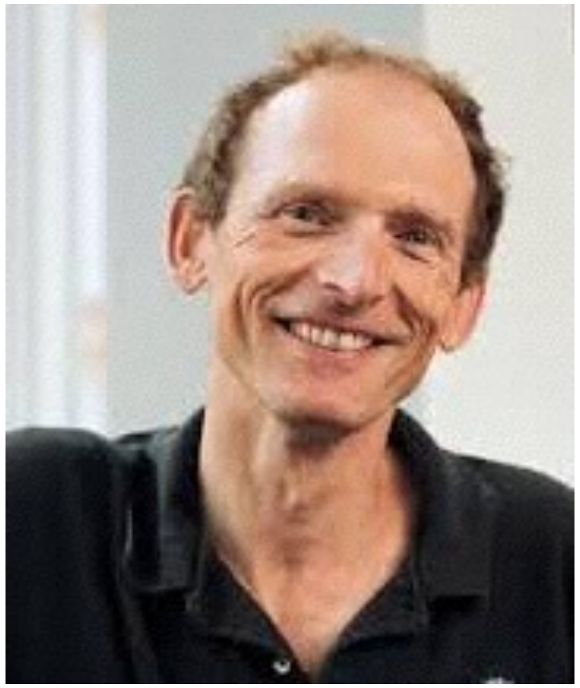


DLS 2006

Interlanguage Migration: From Scripts to Programs

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

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

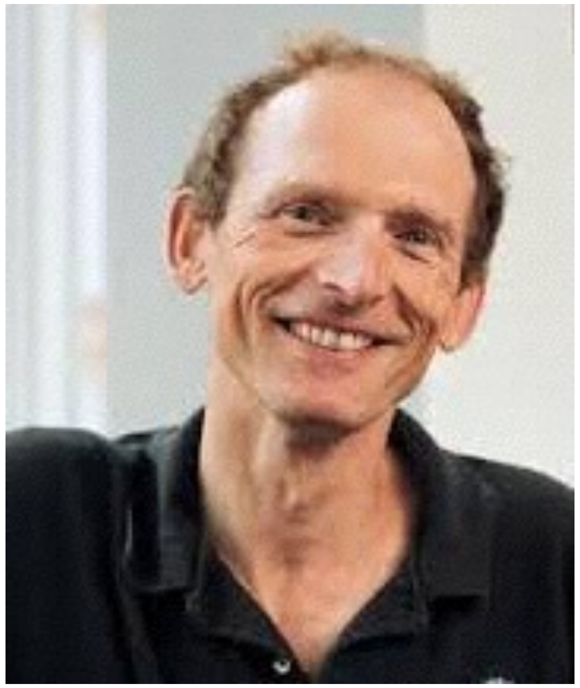


DLS 2006

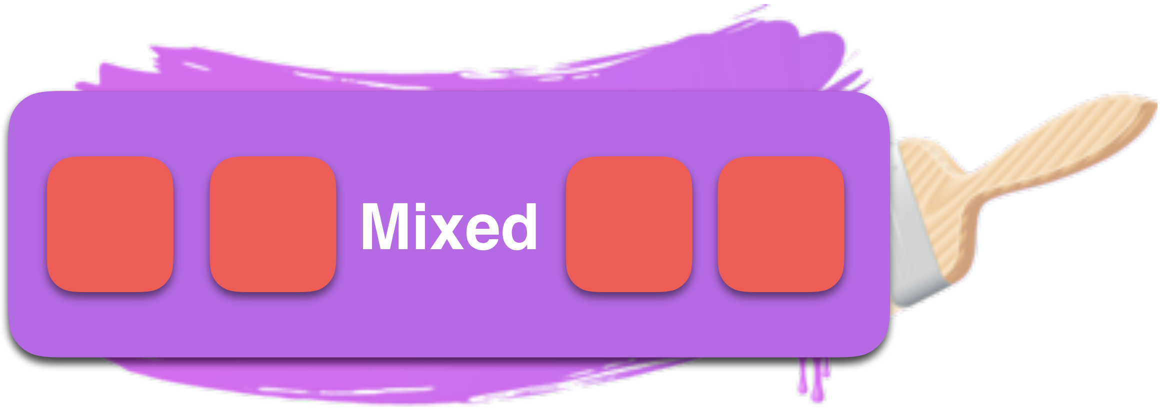
Interlanguage Migration: From Scripts to Programs

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

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



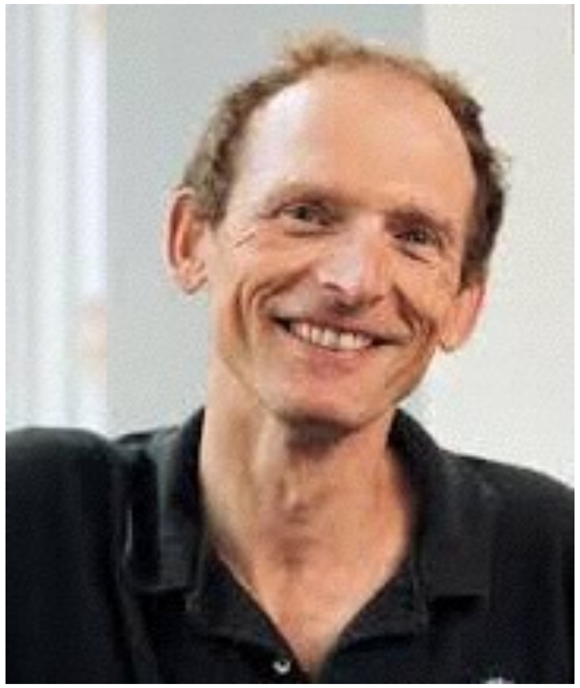
DLS 2006



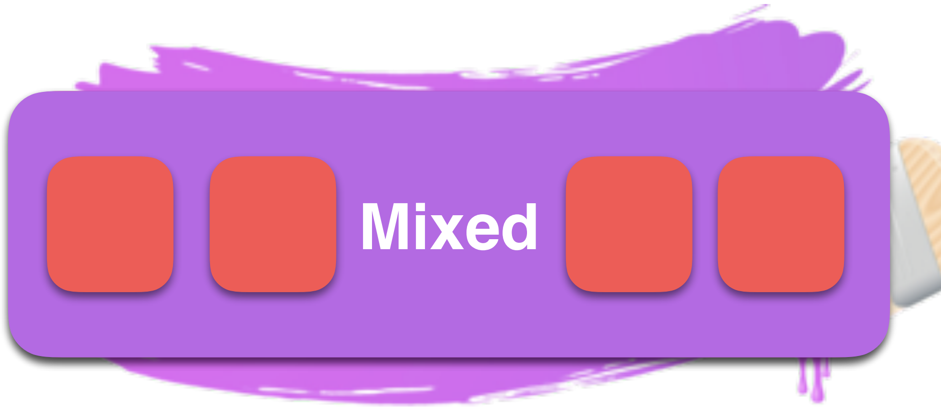
Interlanguage Migration: From Scripts to Programs

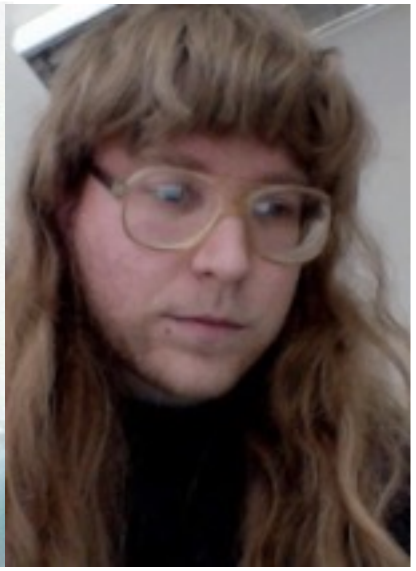
Sam Tobin-Hochstadt
Northeastern University
Boston, MA
samth@ccs.neu.edu

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



DLS 2006





Retrospective

Migratory Typing: Ten Years Later*

Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, Asumu Takikawa[1]

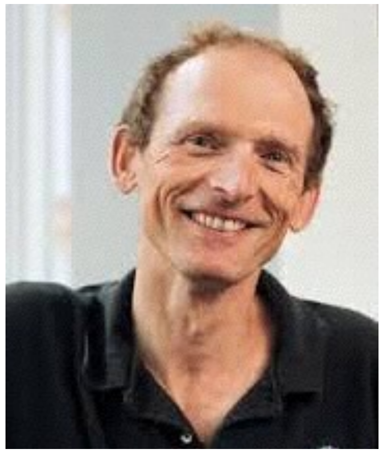
[1] **PLT** `*@racket-lang.org`

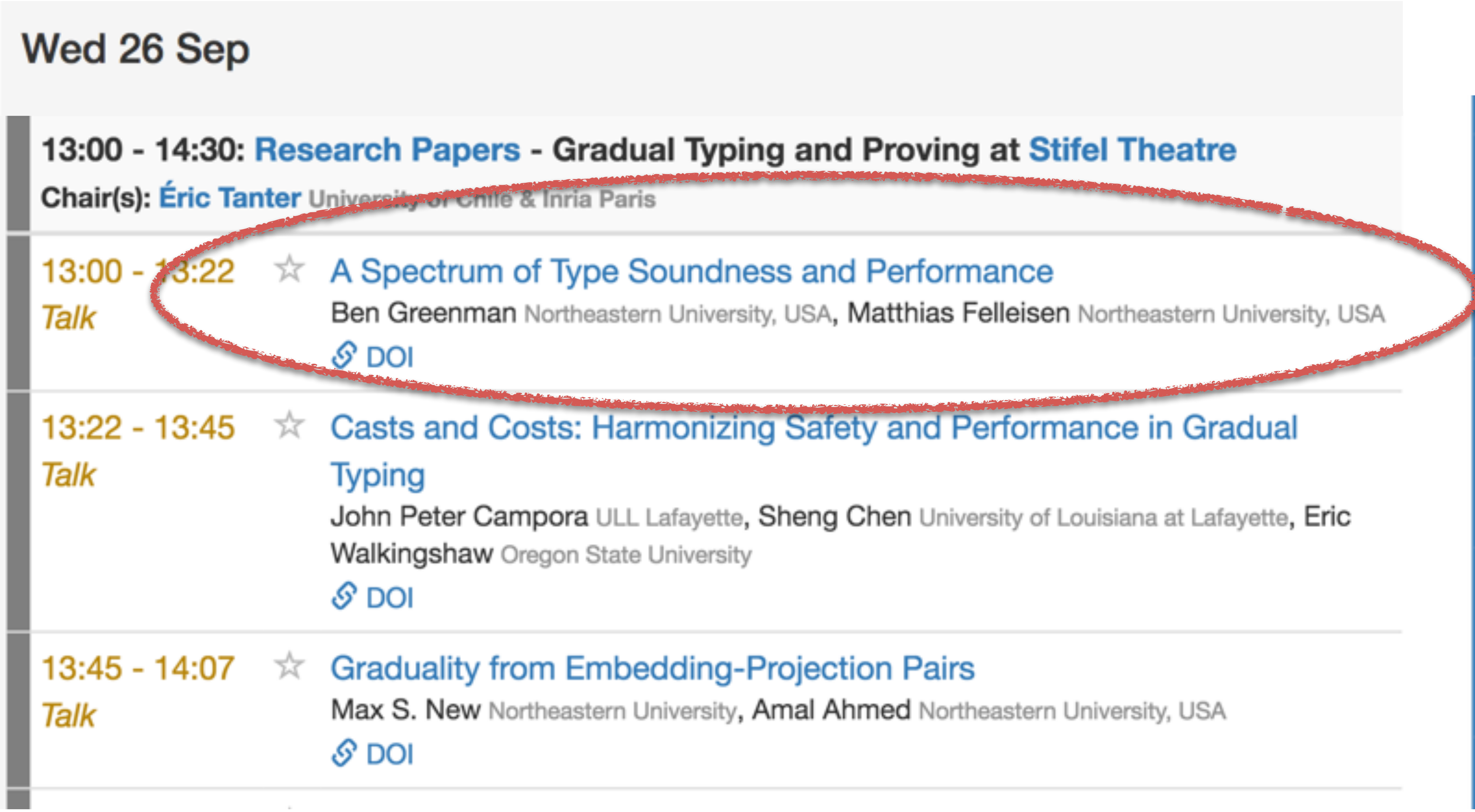
Abstract

In this day and age, many developers work on large, untyped code repositories. Even if they are the creators of the code, they notice that they have to figure out the equivalent of method signatures every time they work on old code. This step is time consuming and error prone.

Ten years ago, the two lead authors outlined a linguistic solution to this problem. Specifically they proposed the creation of typed twins for untyped programming languages so that developers could migrate scripts from the untyped world to a typed one in an incremental manner. Their programmatic paper also spelled out three guiding design principles concerning the acceptance of grown idioms, the soundness of mixed-typed programs, and the units of migration.

This paper revisits this idea of a migratory type system as implemented for Racket. It explains how the design principles have been used to produce the Typed Racket twin and presents an assessment of the project's status, highlighting successes and failures.





Wed 26 Sep

13:00 - 14:30: Research Papers - Gradual Typing and Proving at Stifel Theatre

Chair(s): **Éric Tanter** University of Chile & Inria Paris

13:00 - 13:22 *Talk* ☆ A Spectrum of Type Soundness and Performance

Ben Greenman Northeastern University, USA, Matthias Felleisen Northeastern University, USA

DOI

13:22 - 13:45 *Talk* ☆ Casts and Costs: Harmonizing Safety and Performance in Gradual Typing

John Peter Campora ULL Lafayette, Sheng Chen University of Louisiana at Lafayette, Eric Walkingshaw Oregon State University

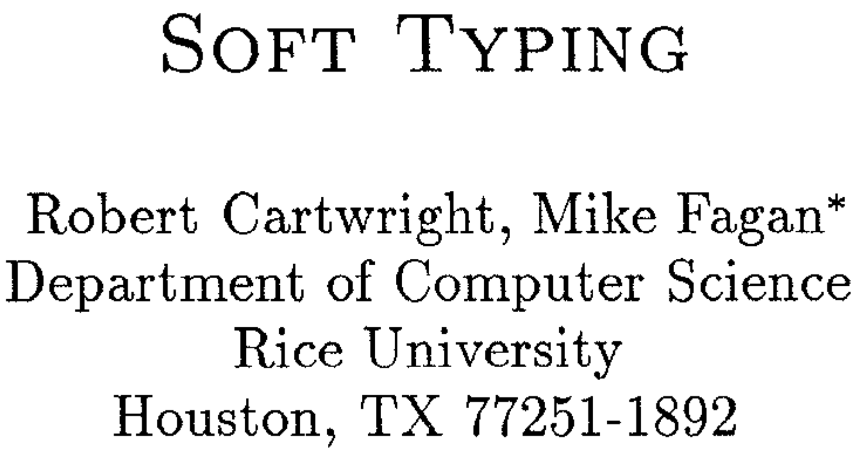
DOI

13:45 - 14:07 *Talk* ☆ Graduality from Embedding-Projection Pairs

Max S. New Northeastern University, Amal Ahmed Northeastern University, USA

DOI

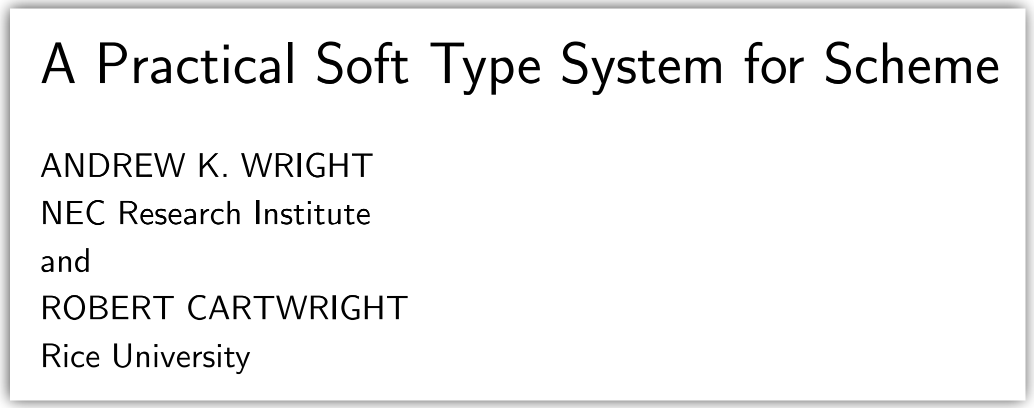
Soft Typing



SOFT TYPING

Robert Cartwright, Mike Fagan*
Department of Computer Science
Rice University
Houston, TX 77251-1892

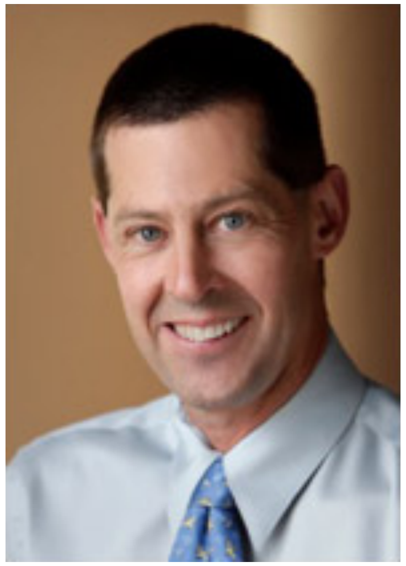
PLDI 1991



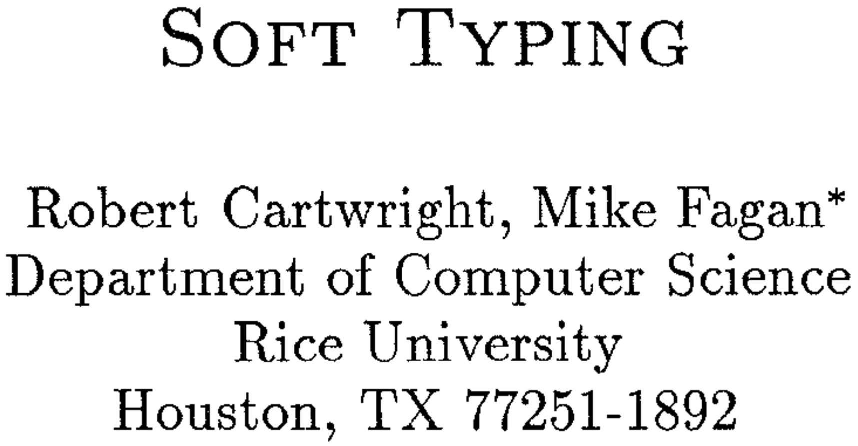
A Practical Soft Type System for Scheme

ANDREW K. WRIGHT
NEC Research Institute
and
ROBERT CARTWRIGHT
Rice University

TOPLAS 1997



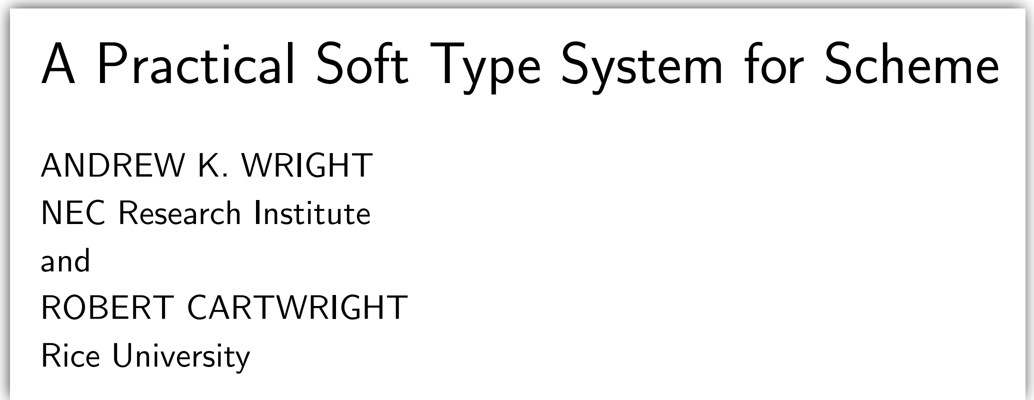
Soft Typing



SOFT TYPING

Robert Cartwright, Mike Fagan*
Department of Computer Science
Rice University
Houston, TX 77251-1892

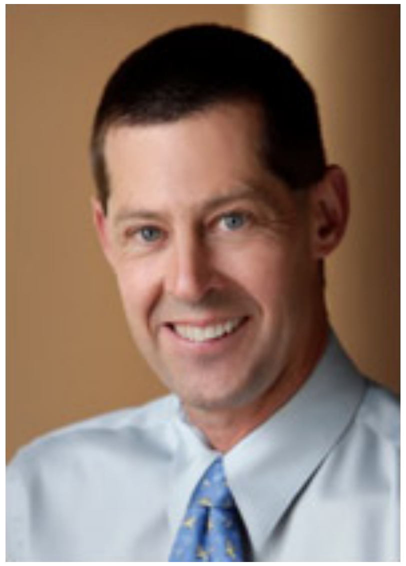
PLDI 1991



A Practical Soft Type System for Scheme

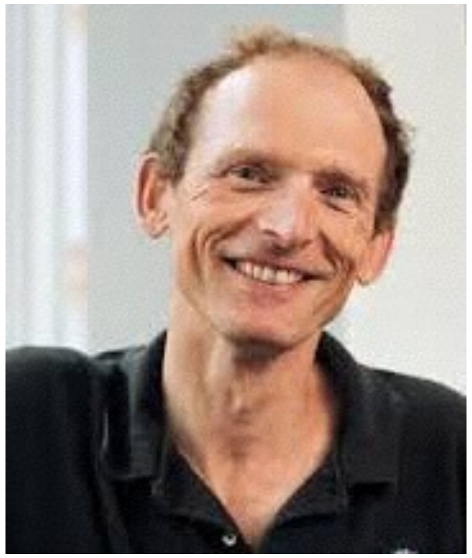
ANDREW K. WRIGHT
NEC Research Institute
and
ROBERT CARTWRIGHT
Rice University

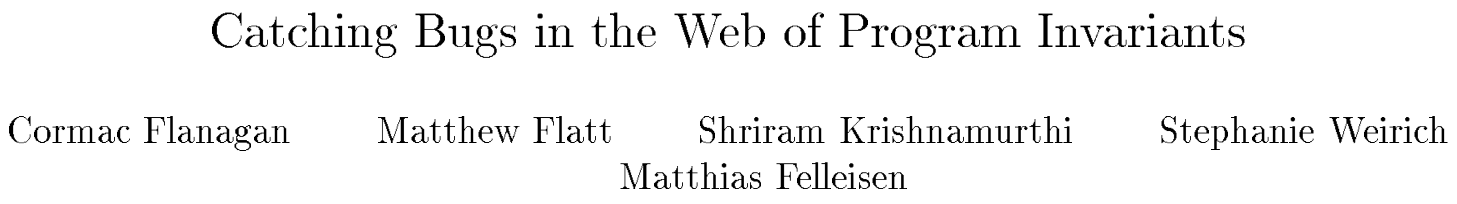
TOPLAS 1997



Idea: Use H/M Type Inference to Migrate Dynamic Programs

Set-Based Analysis

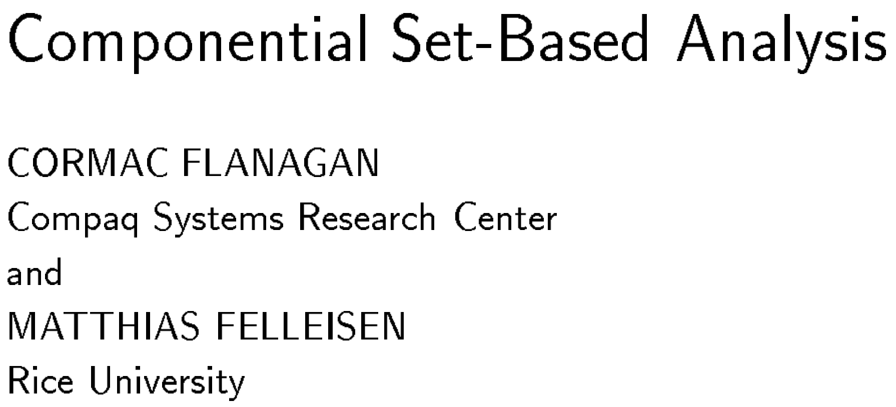




Catching Bugs in the Web of Program Invariants

Cormac Flanagan Matthew Flatt Shriram Krishnamurthi Stephanie Weirich
Matthias Felleisen

PLDI96



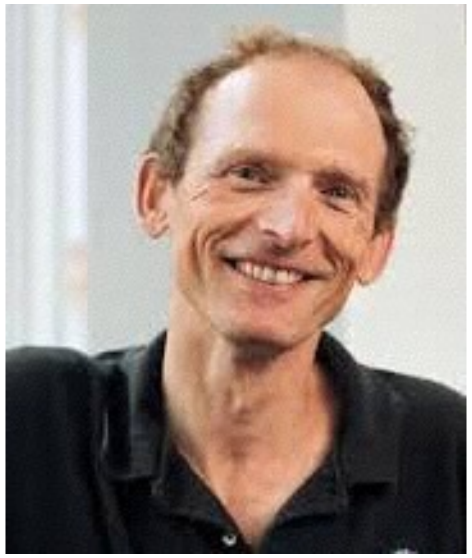
Componential Set-Based Analysis

CORMAC FLANAGAN
Compaq Systems Research Center
and
MATTHIAS FELLEISEN
Rice University

TOPLAS99

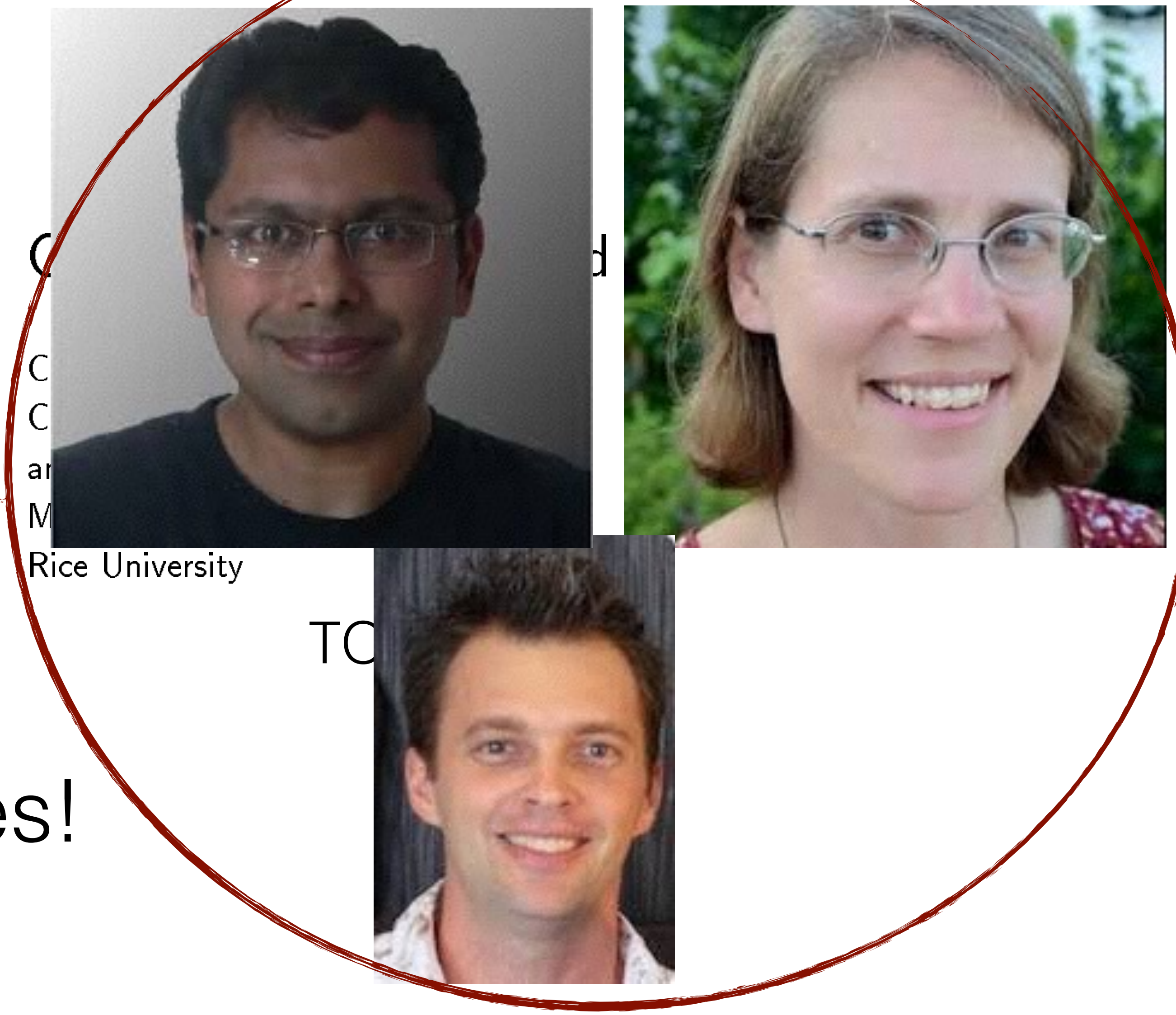
Not Types!

Set-Based Analysis



Catching Bugs in the Web of Program Invariants

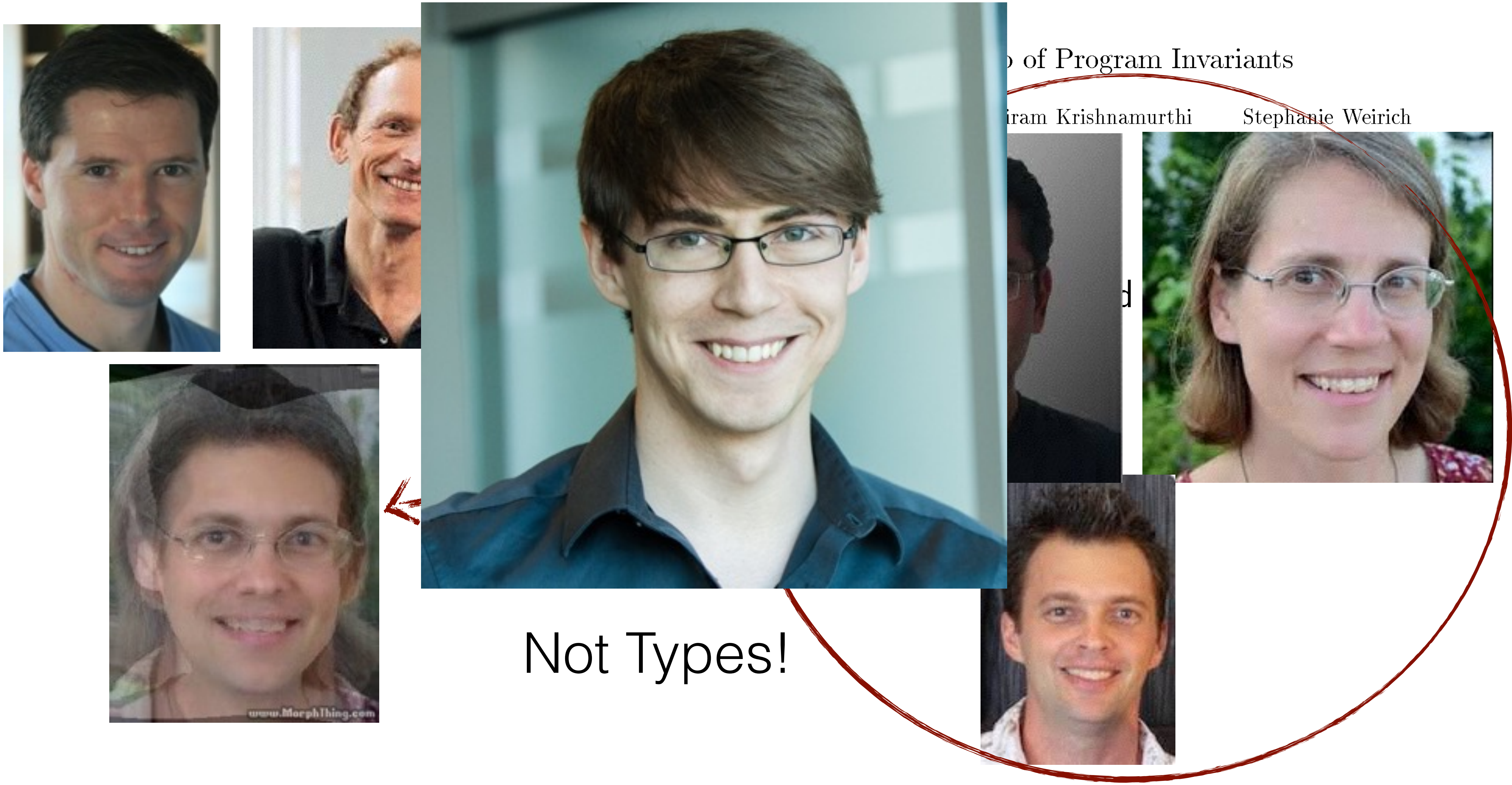
Cormac Flanagan Matthew Flatt Shriram Krishnamurthi Stephanie Weirich



Not Types!

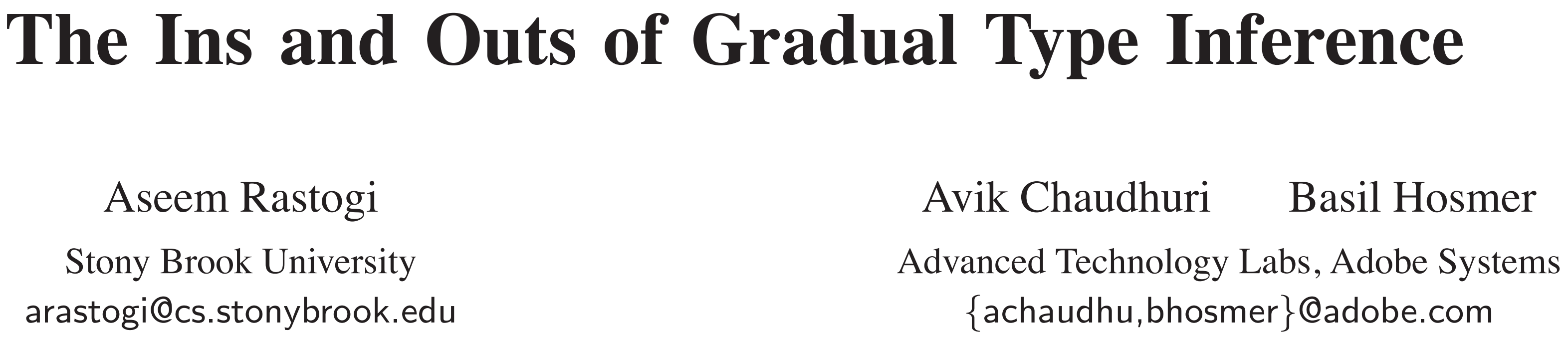
Set-Based Analysis

William Bowman



Not Types!

Migration By Inference

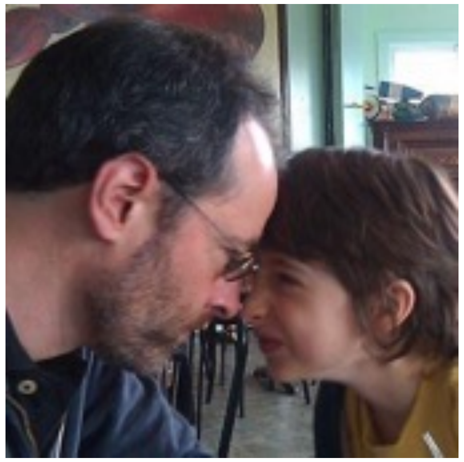


The Ins and Outs of Gradual Type Inference

Aseem Rastogi
Stony Brook University
arastogi@cs.stonybrook.edu

Avik Chaudhuri Basil Hosmer
Advanced Technology Labs, Adobe Systems
{achaudhu,bhosmer}@adobe.com

POPL 2012



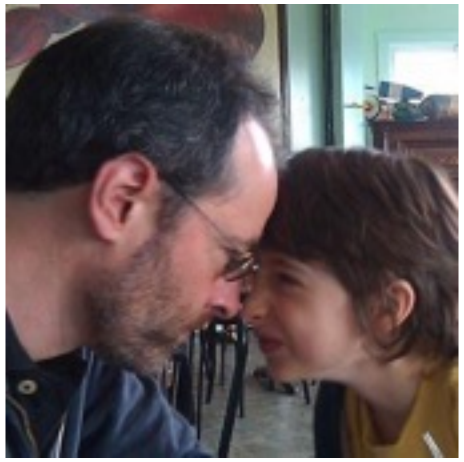
Migration By Inference

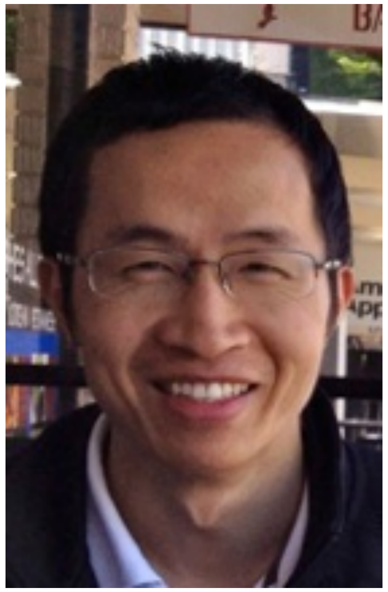
The Ins and Outs of Gradual Type Inference

Aseem Rastogi
Stony Brook University
arastogi@cs.stonybrook.edu

Avik Chaudhuri Basil Hosmer
Advanced Technology Labs, Adobe Systems
{achaudhu,bhosmer}@adobe.com

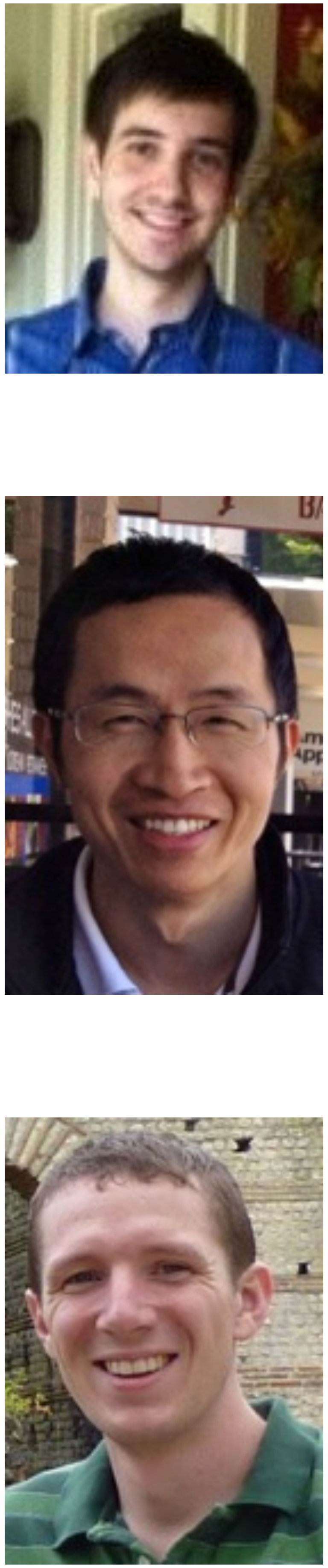
POPL 2012

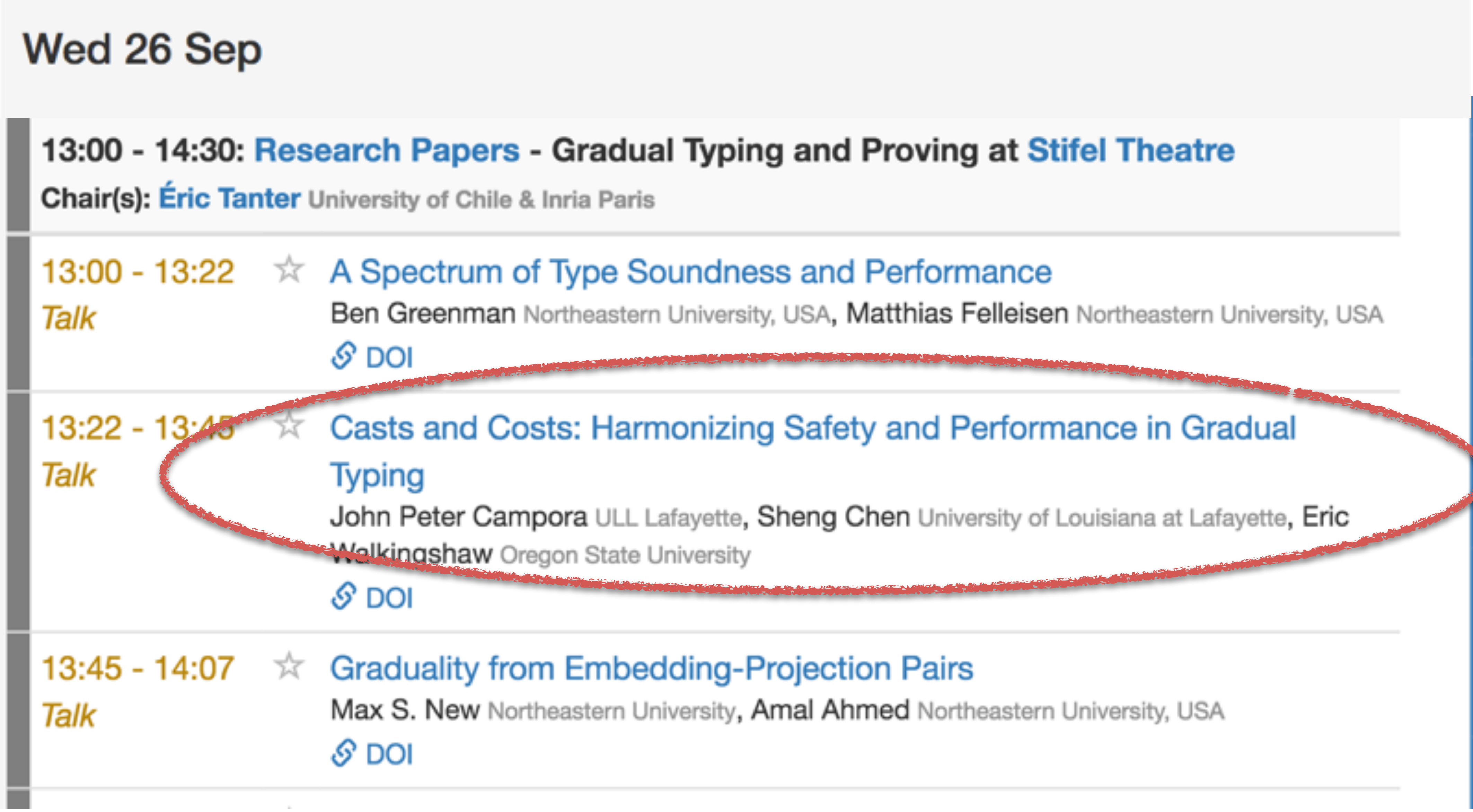




Migrating Gradual Types

JOHN PETER CAMPORA III, University of Louisiana at Lafayette
SHENG CHEN, University of Louisiana at Lafayette
MARTIN ERWIG, Oregon State University
ERIC WALKINGSHAW, Oregon State University





Wed 26 Sep

13:00 - 14:30: Research Papers - Gradual Typing and Proving at Stifel Theatre

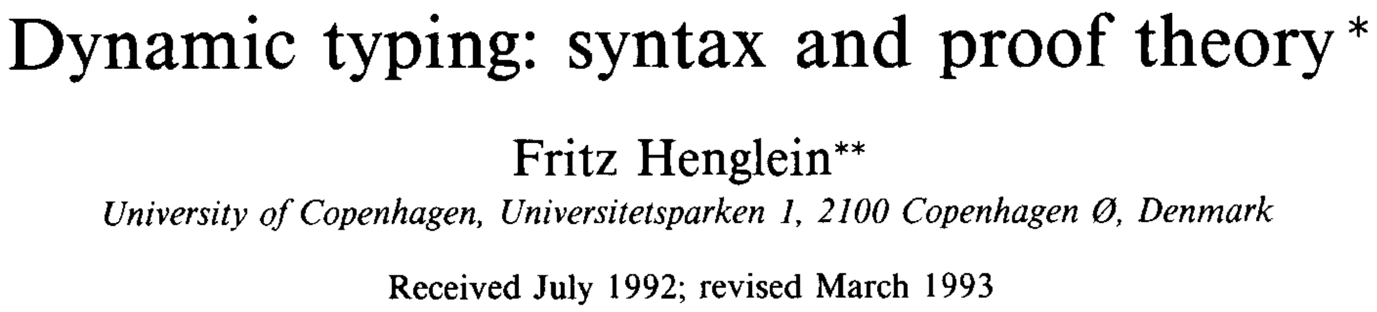
Chair(s): Éric Tanter University of Chile & Inria Paris

13:00 - 13:22 ☆ A Spectrum of Type Soundness and Performance
Talk
Ben Greenman Northeastern University, USA, Matthias Felleisen Northeastern University, USA
DOI

13:22 - 13:45 ☆ Casts and Costs: Harmonizing Safety and Performance in Gradual Typing
Talk
John Peter Campora ULL Lafayette, Sheng Chen University of Louisiana at Lafayette, Eric Walkingshaw Oregon State University
DOI

13:45 - 14:07 ☆ Graduality from Embedding-Projection Pairs
Talk
Max S. New Northeastern University, Amal Ahmed Northeastern University, USA
DOI

Dynamic Typing



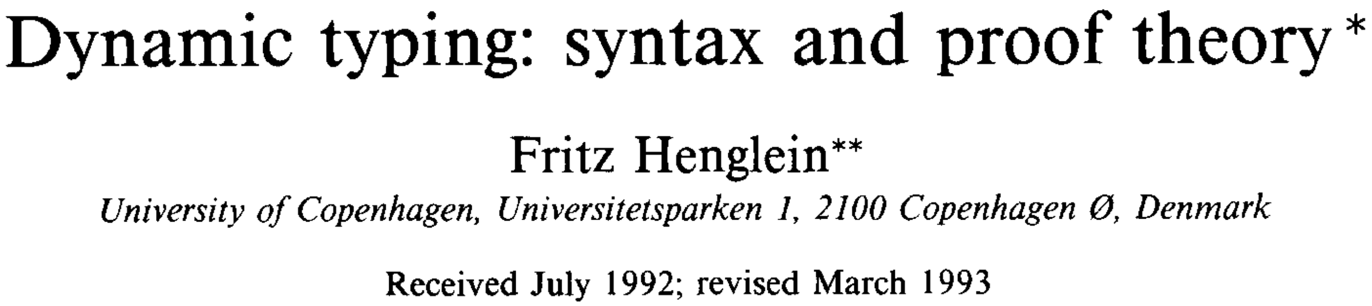
Dynamic typing: syntax and proof theory*

Fritz Henglein**

University of Copenhagen, Universitetsparken 1, 2100 Copenhagen Ø, Denmark

Received July 1992; revised March 1993

Dynamic Typing



Dynamic typing: syntax and proof theory *

Fritz Henglein**

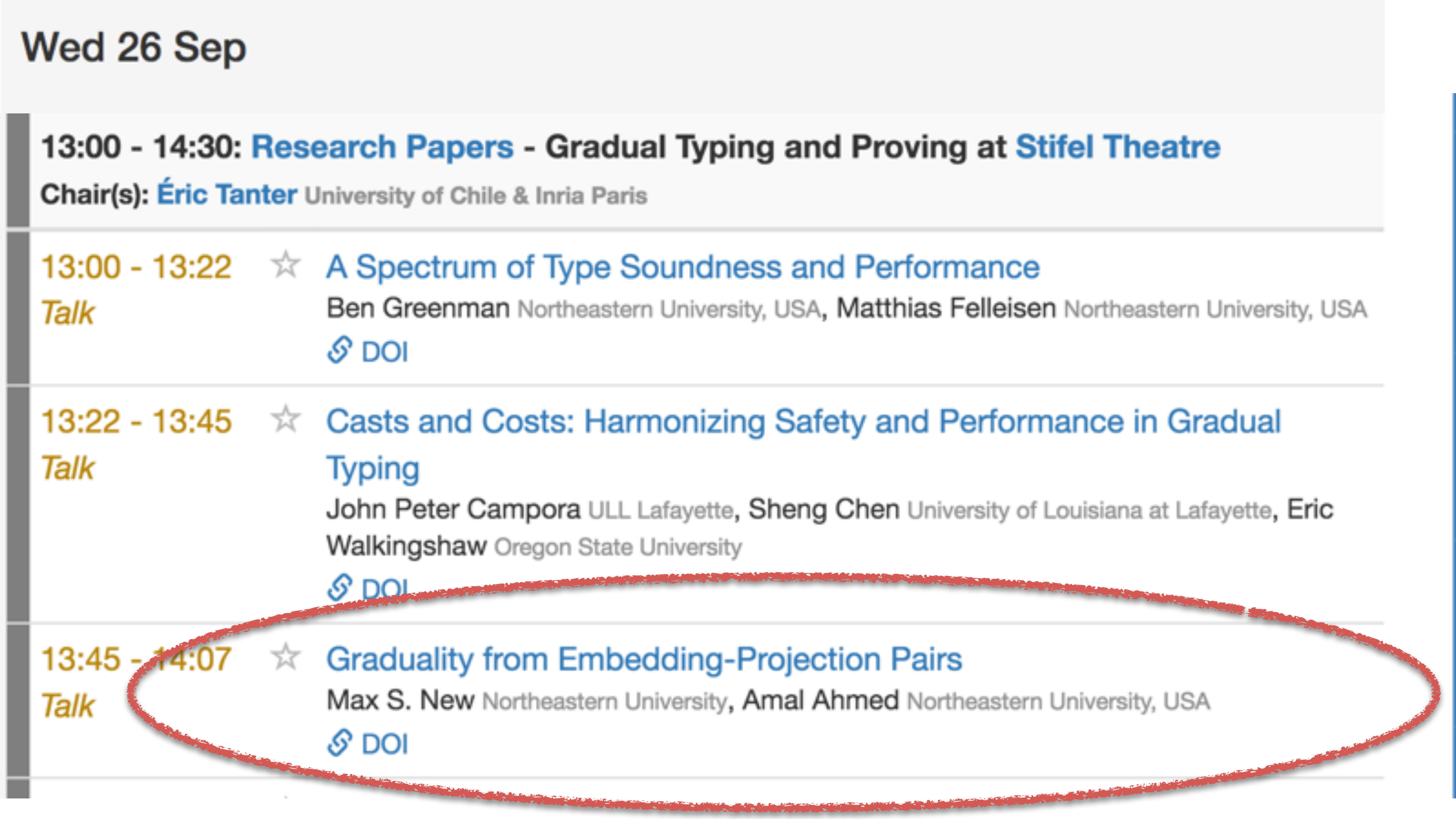
University of Copenhagen, Universitetsparken 1, 2100 Copenhagen Ø, Denmark

Received July 1992; revised March 1993

Influence

- Herman, et al. [TFP 2007]
- Siek, Garcia, Taha [ESOP 2008]
- Siek and Wadler [POPL 2010]
- Garcia [ICFP 2013]
- Siek et al. [PLDI 2015]

Fresh Influence



Wed 26 Sep

13:00 - 14:30: Research Papers - Gradual Typing and Proving at Stifel Theatre
Chair(s): Éric Tanter University of Chile & Inria Paris

13:00 - 13:22 ☆ A Spectrum of Type Soundness and Performance
Talk
Ben Greenman Northeastern University, USA, Matthias Felleisen Northeastern University, USA
DOI

13:22 - 13:45 ☆ Casts and Costs: Harmonizing Safety and Performance in Gradual Typing
Talk
John Peter Campora ULL Lafayette, Sheng Chen University of Louisiana at Lafayette, Eric Walkingshaw Oregon State University
DOI

13:45 - 14:07 ☆ Graduality from Embedding-Projection Pairs
Talk
Max S. New Northeastern University, Amal Ahmed Northeastern University, USA
DOI

Outline

- Motivating Example (In Two Acts)
- Gradual Typing For All!
- Typing in Small Pieces
- Meat
- Strands and Related Works

Gratitude


CLASS
OF
1997

It Takes a Village

Andy Lumsdaine

Dan Friedman

Frank Pfenning

Amr Sabry

It Takes a Village

Andy Lumsdaine

Dan Friedman

Frank Pfenning

Amr Sabry

It Takes a Village

Andy Lumsdaine

Dan Friedman

Frank Pfenning

Amr Sabry

It Takes a Village

Andy Lumsdaine

Dan Friedman

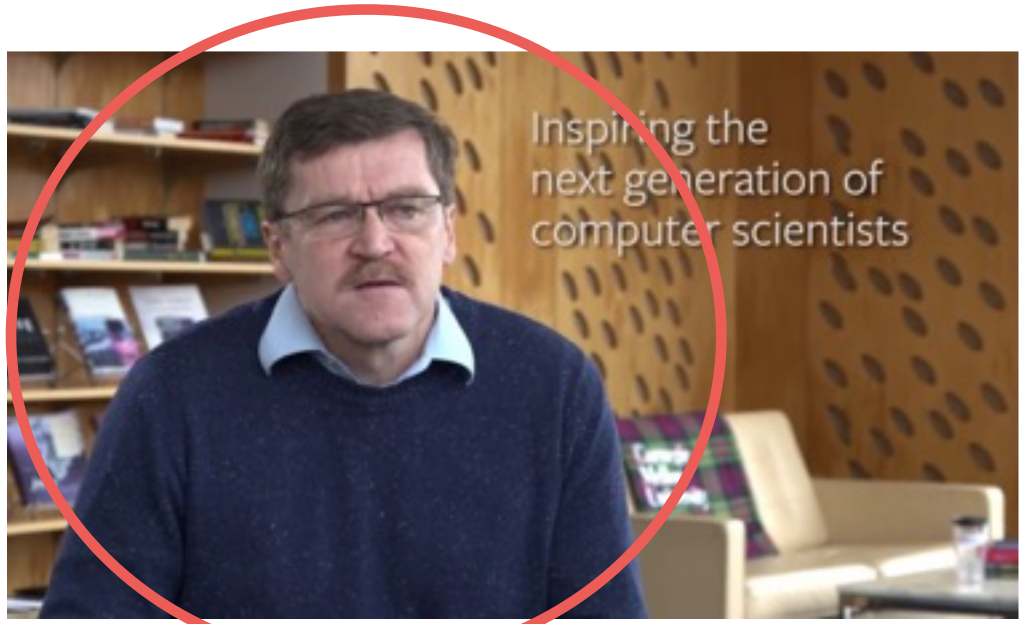
Frank Pfenning

Amr Sabry

It Takes a Village

Andy Lumsdaine

Dan Friedman



Frank Pfenning

Amr Sabry

It Takes a Village

Andy Lumsdaine

Dan Friedman

Me

Frank Pfenning

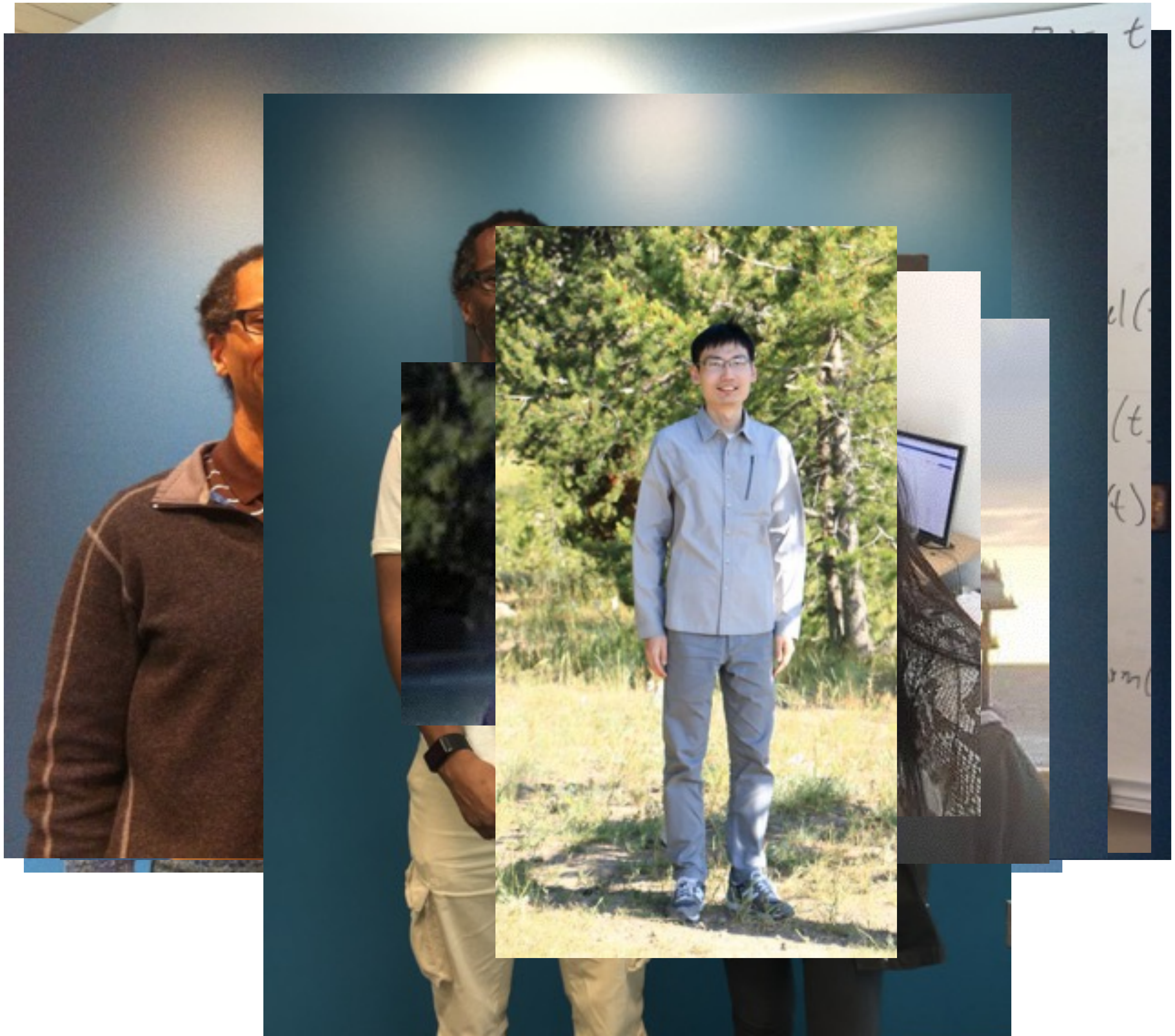
Amr Sabry

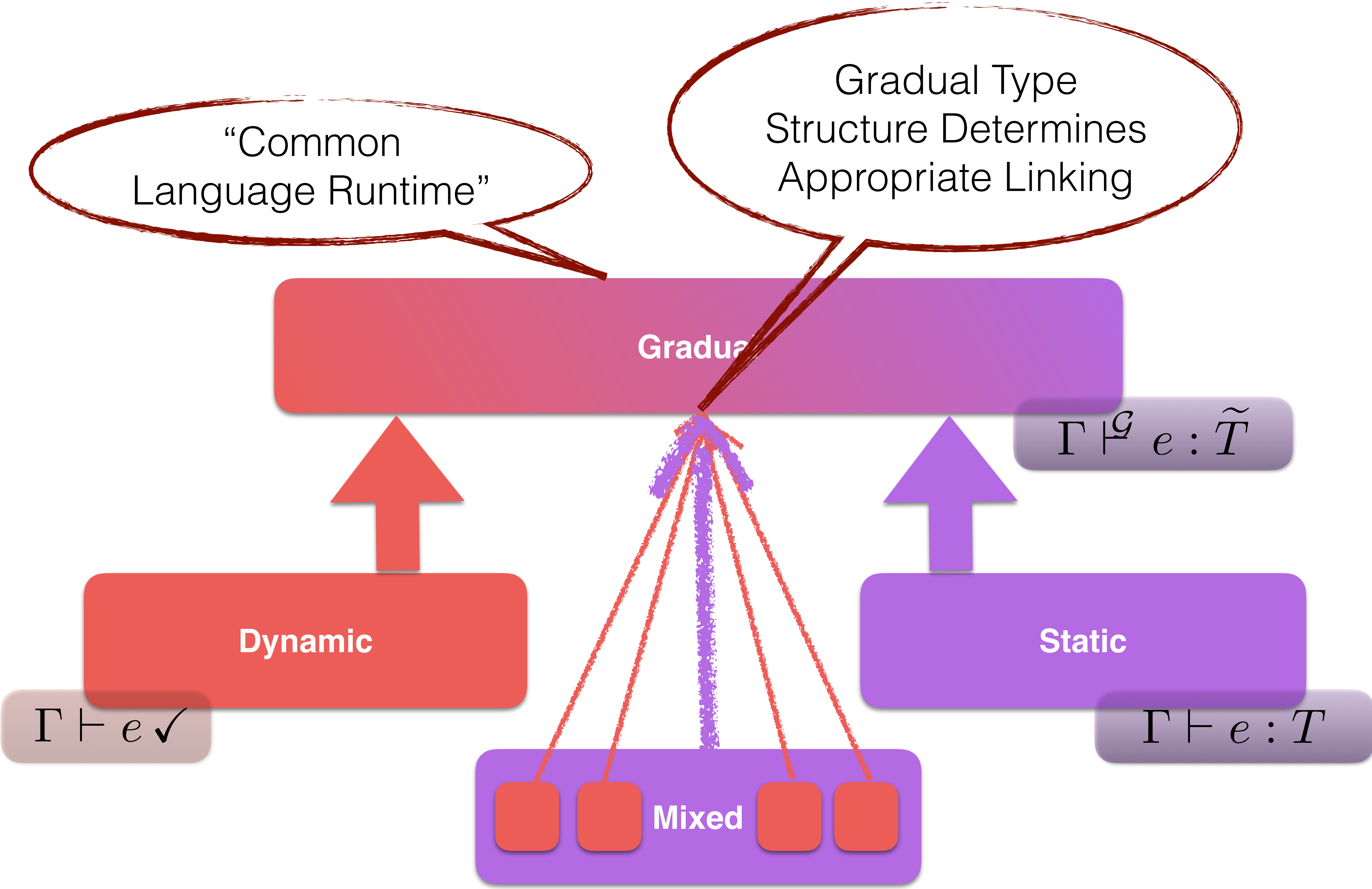

©2004

©2004

Students

Students





“Common Language Runtime”

Gradual Type Structure Determines Appropriate Linking

$\Gamma \vdash^{\mathcal{G}} e : \widetilde{T}$

$\Gamma \vdash e \checkmark$

$\Gamma \vdash e : T$

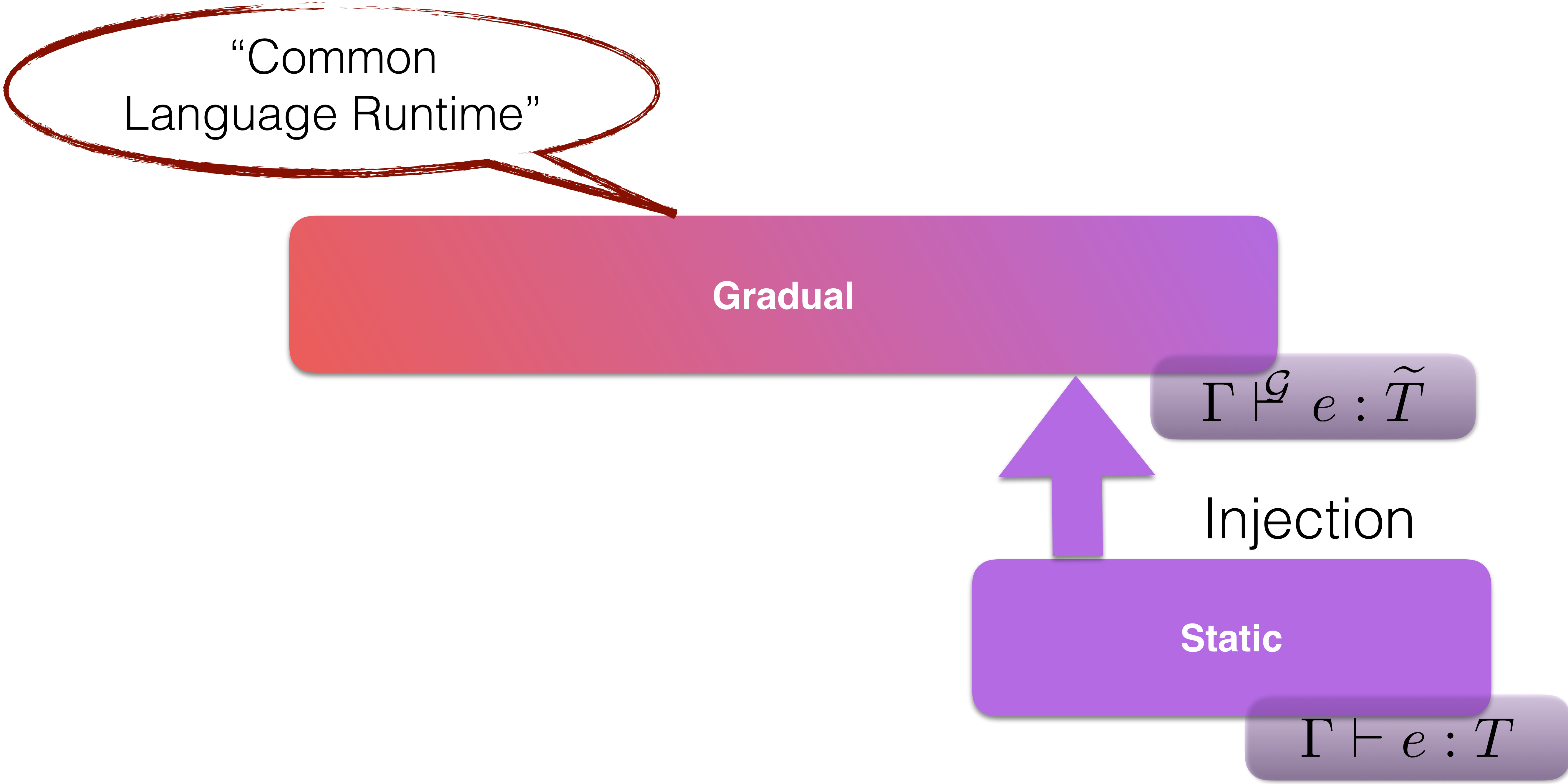
> “Dear Today Ron,
> You went one slide too far.
> Go back one slide.”
>
> –Yesterday Ron

Bonus Tracks

“Common Language Runtime”

Gradual

$\Gamma \vdash^{\mathcal{G}} e : \widetilde{T}$



"Common Language Runtime"

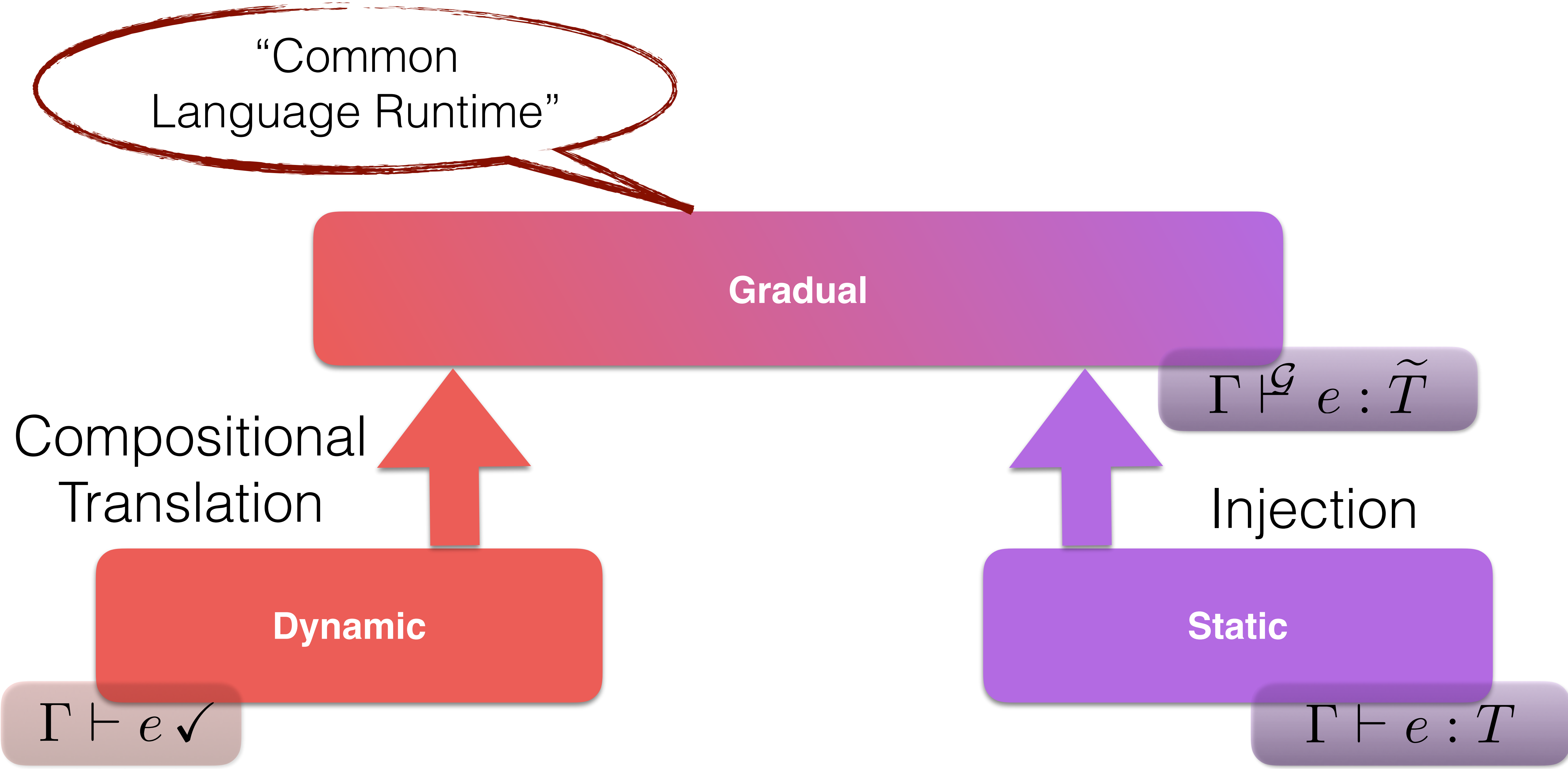
Gradual

$\Gamma \vdash^{\mathcal{G}} e : \widetilde{T}$

Injection

Static

$\Gamma \vdash e : T$



“Common Language Runtime”

Gradual

$\Gamma \vdash^{\mathcal{G}} e : \widetilde{T}$

Compositional Translation

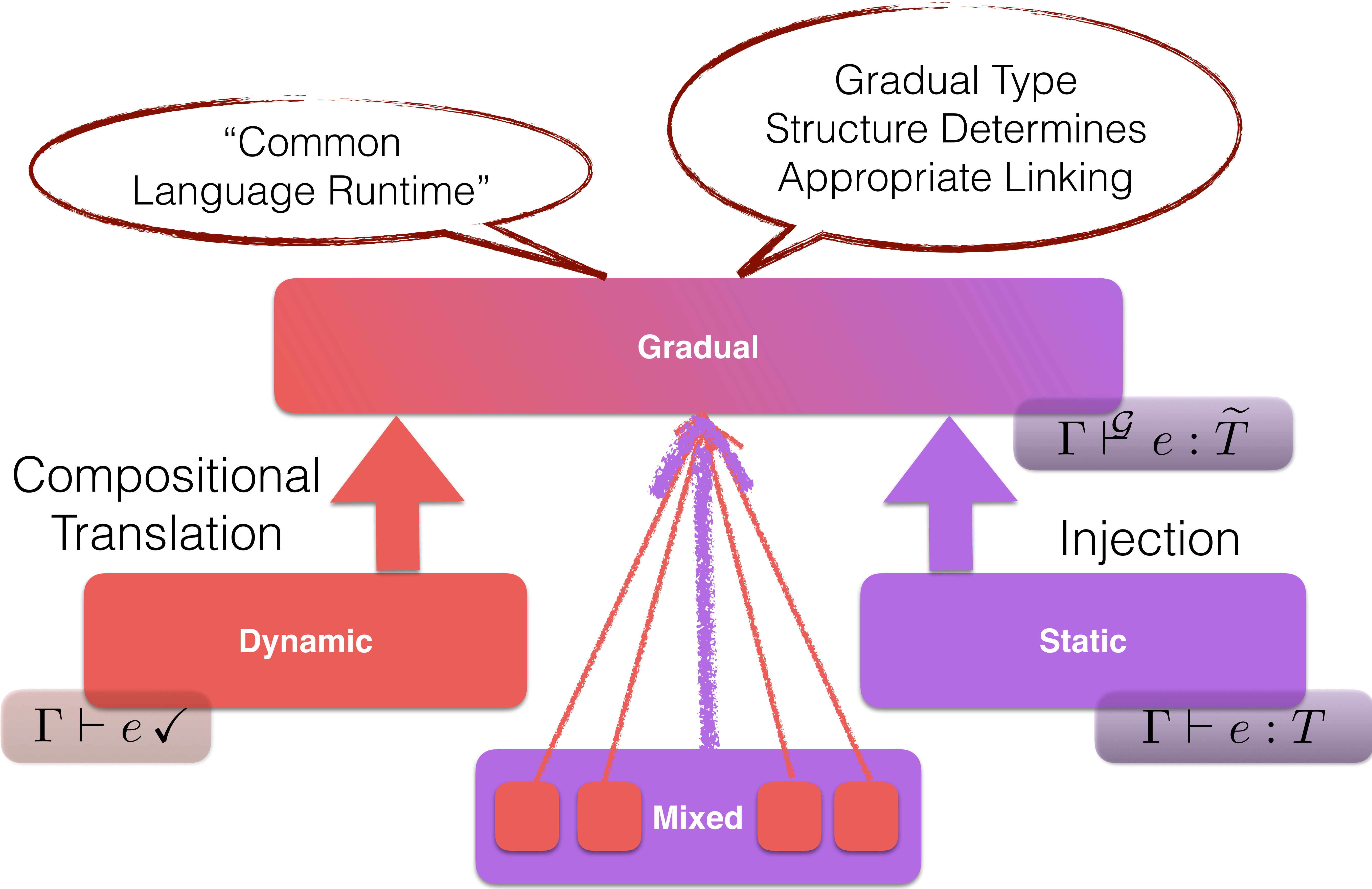
Injection

Dynamic

$\Gamma \vdash e \checkmark$

Static

$\Gamma \vdash e : T$



“Common Language Runtime”

Gradual Type Structure Determines Appropriate Linking

Gradual

$\Gamma \vdash^{\mathcal{G}} e : \widetilde{T}$

Compositional Translation

Injection

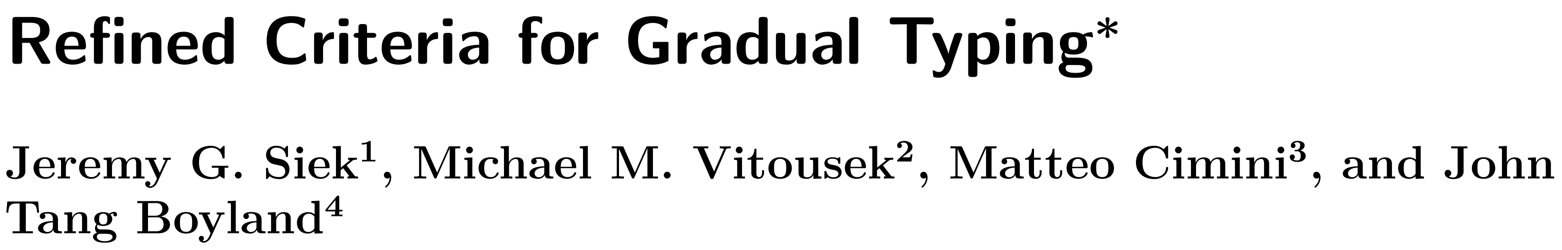
Dynamic

$\Gamma \vdash e \checkmark$

Static

$\Gamma \vdash e : T$

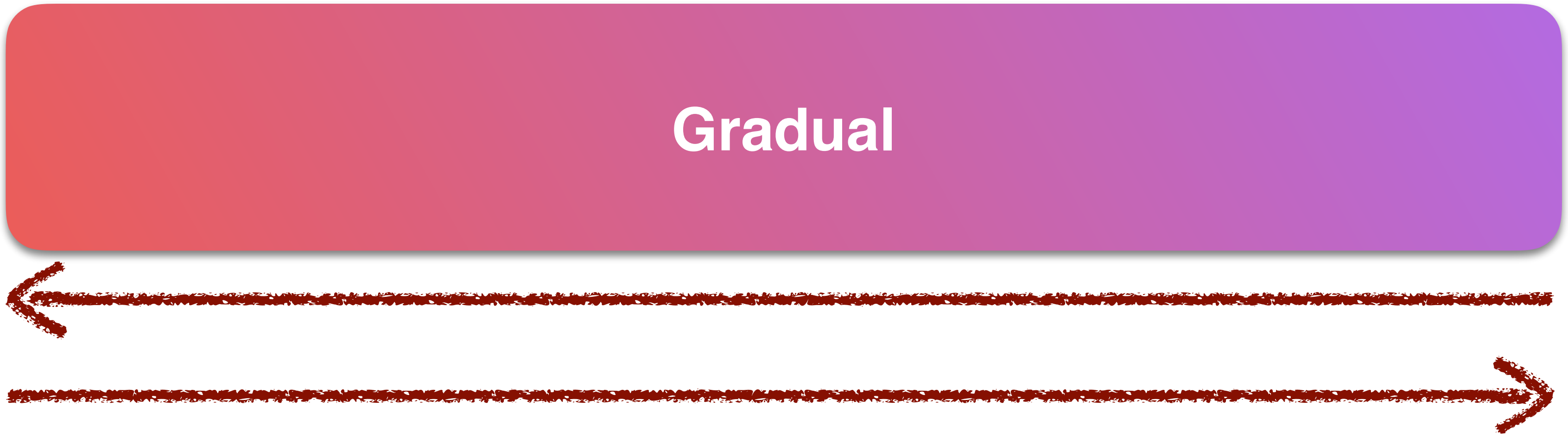
Mixed



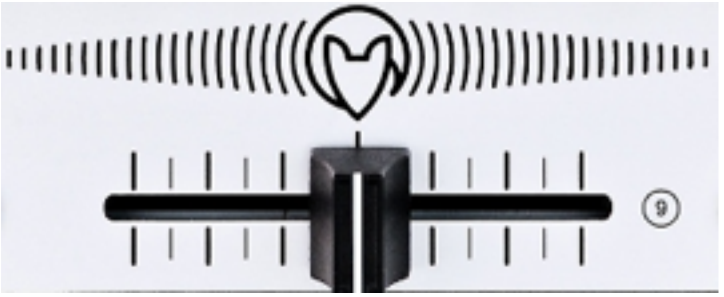
Refined Criteria for Gradual Typing*

Jeremy G. Siek[1], Michael M. Vitousek[2], Matteo Cimini[3], and John Tang Boyland[4]

Static and Dynamic Gradual Guarantee!



Gradual



Varying The Type Precision of a Program
Monotonically Changes **only**
static and dynamic type errors

Blame

Theorems about Blame

- Tobin-Hochstadt and Felleisen 2006
- Wadler and Findler 2008
- Dimoulas et al.
- Dimoulas …
- Takikawa …

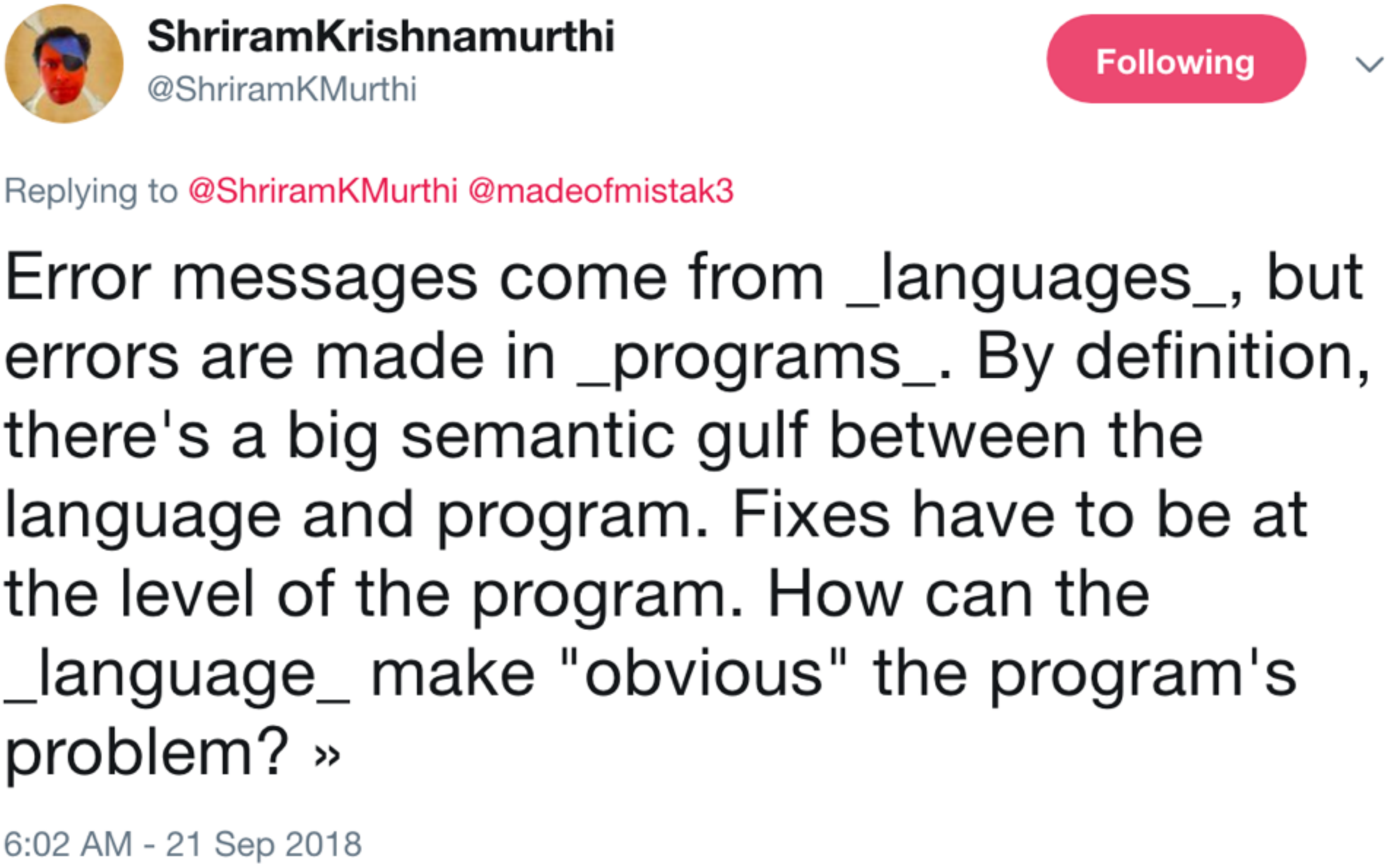
Racket Contract Blame


Robby Findler

```
point-in?: contract violation
  expected: real?
  given: #f
  in: the 2nd argument of
      (-> pict? real? real? boolean?)
  contract from: point-in-module
  blaming: top-level
   (assuming the contract is correct)
```

Findler PLMW@ICFP15

Wherein Shriram Unwittingly Writes My Blame Schpiel For Me

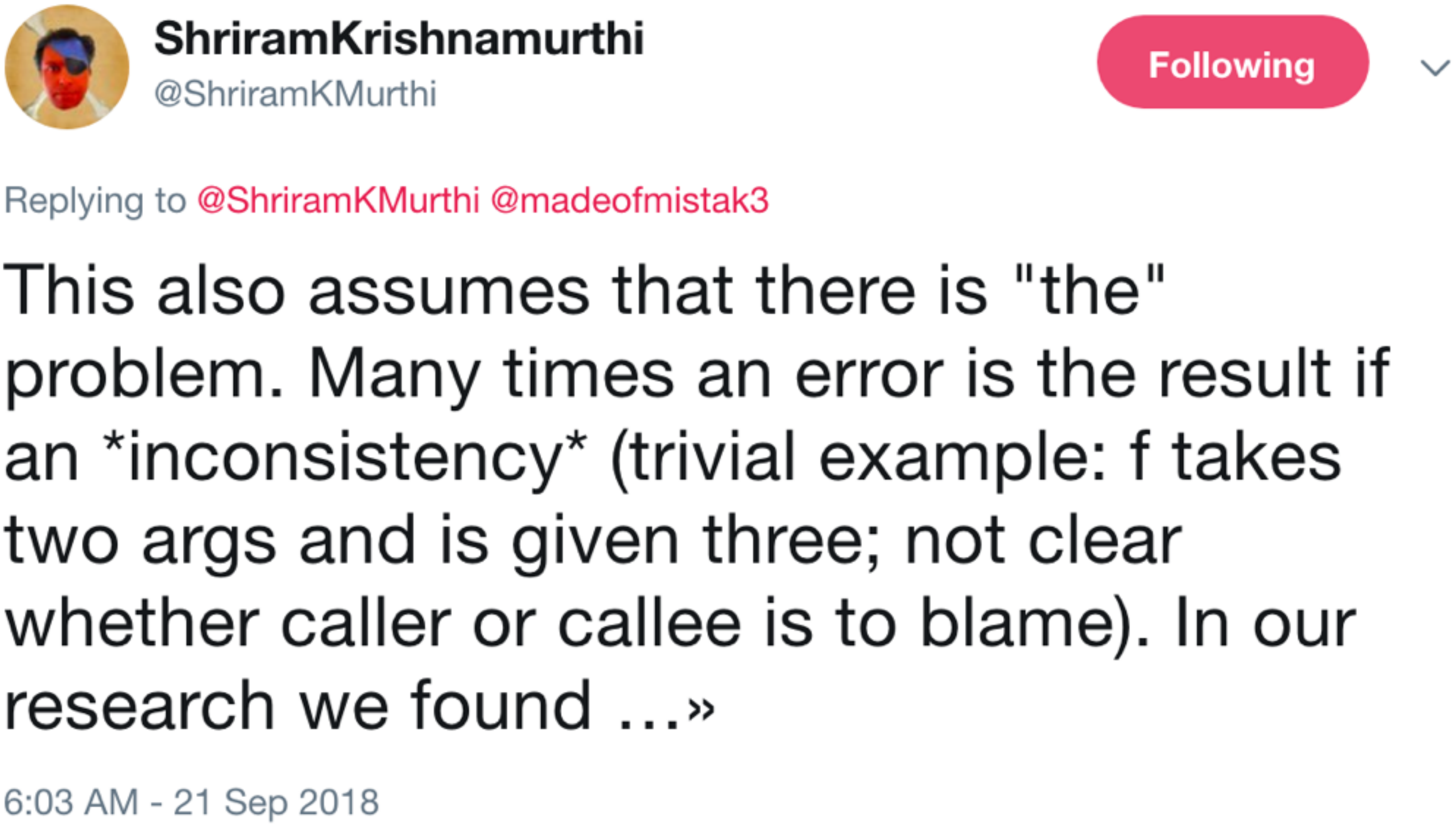


ShriramKrishnamurthi
@ShriramKMurthi

Replying to @ShriramKMurthi @madeofmistak3

Error messages come from _languages_, but errors are made in _programs_. By definition, there's a big semantic gulf between the language and program. Fixes have to be at the level of the program. How can the _language_ make "obvious" the program's problem? »

6:02 AM - 21 Sep 2018

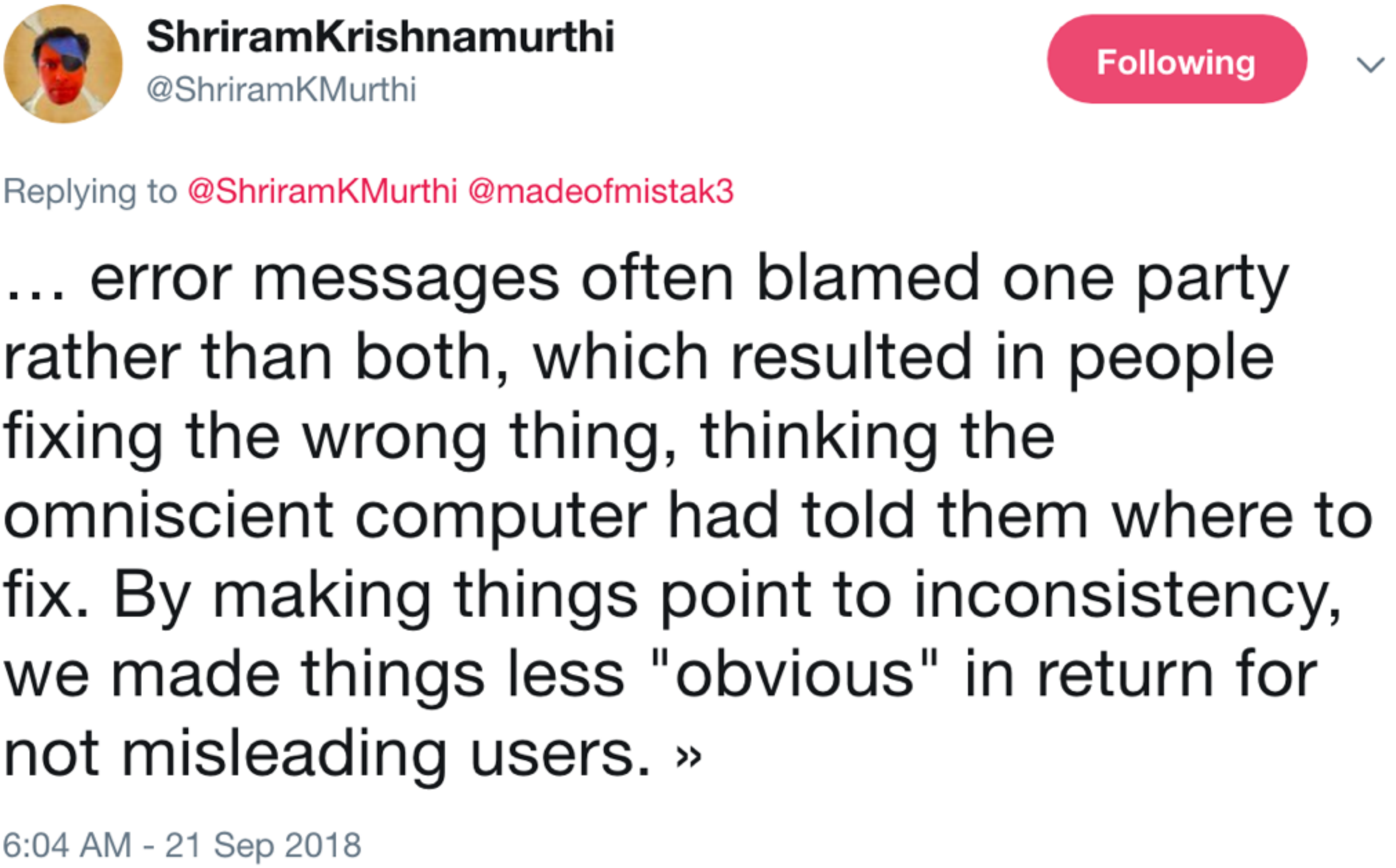


ShriramKrishnamurthi
@ShriramKMurthi

Replying to @ShriramKMurthi @madeofmistak3

This also assumes that there is "the" problem. Many times an error is the result if an *inconsistency* (trivial example: f takes two args and is given three; not clear whether caller or callee is to blame). In our research we found …»

6:03 AM - 21 Sep 2018



ShriramKrishnamurthi
@ShriramKMurthi

Replying to @ShriramKMurthi @madeofmistak3

… error messages often blamed one party rather than both, which resulted in people fixing the wrong thing, thinking the omniscient computer had told them where to fix. By making things point to inconsistency, we made things less "obvious" in return for not misleading users. »

6:04 AM - 21 Sep 2018

Racket Contract Blame

```
point-in?: contract violation
  expected: real?
  given: #f
  in: the 2nd argument of
      (-> pict? real? real? boolean?)
  contract from: point-in-module
  blaming: top-level
   (assuming the contract is correct)
```

Robby Findler

Findler PLMW@ICFP15