A Featherweight Approach to FOOL

Atsushi Igarashi
Kyoto Univ.

What I Have Been Working On

Type theory and its *applications*:

- Static program analysis based on type inference
- Behavioral types for concurrent programs
- Multi-stage programming
  - Type systems based on modal logic
- Object-oriented programming

What is FOOL?

"Foundations of Object-Oriented Languages"

- Semantics
- Type Theory
- Verification techniques

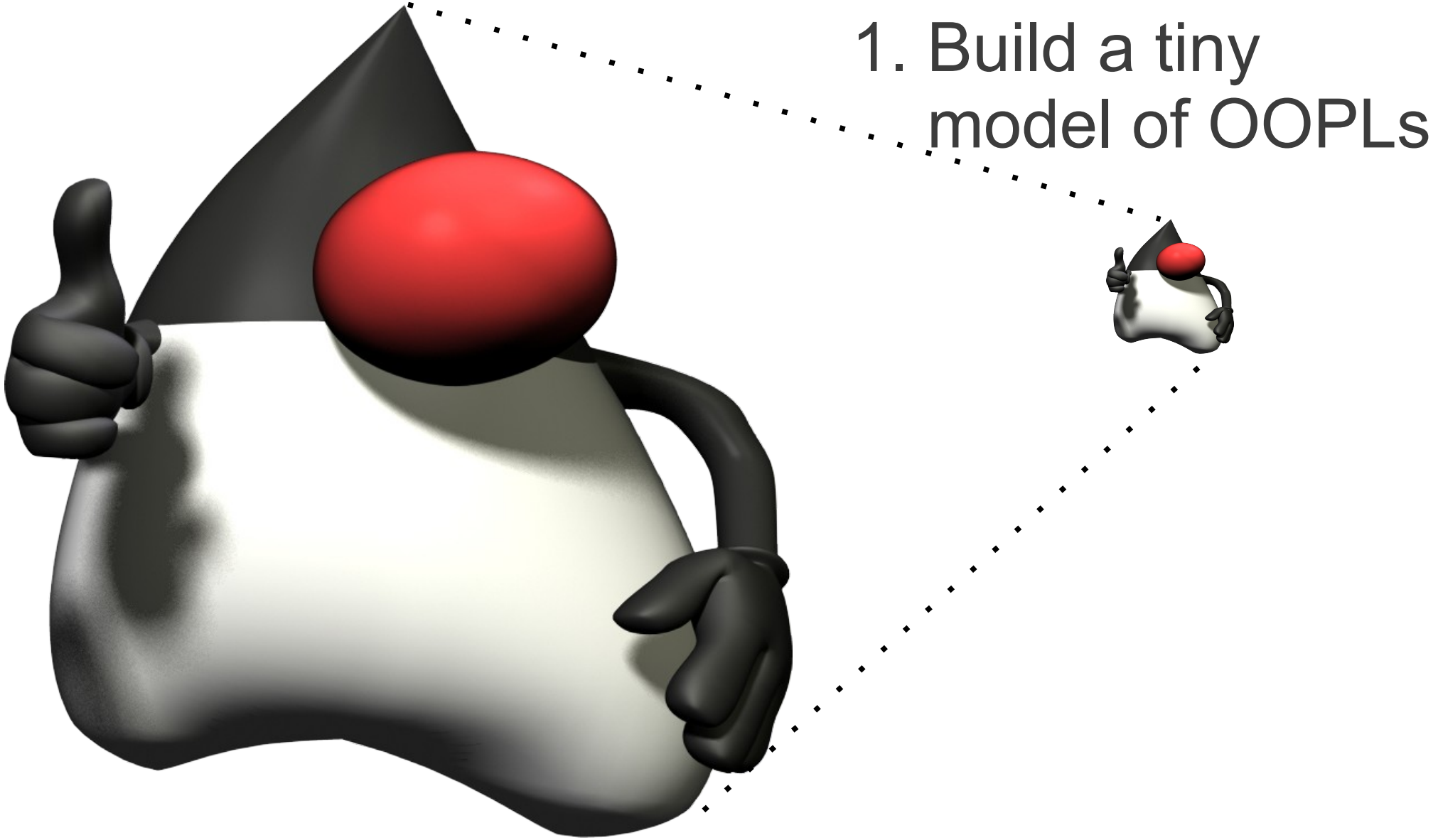
For the development of

- *Correct* systems
- *Correct* compilers

What I Mean By a "Featherweight Approach"

- Usual scientific approach to a complex problem:
  - Discarding irrelevant details
  - To concentrate on central issues
- With a stronger emphasis on simplicity
  - Even "lighter" than *lightweight*

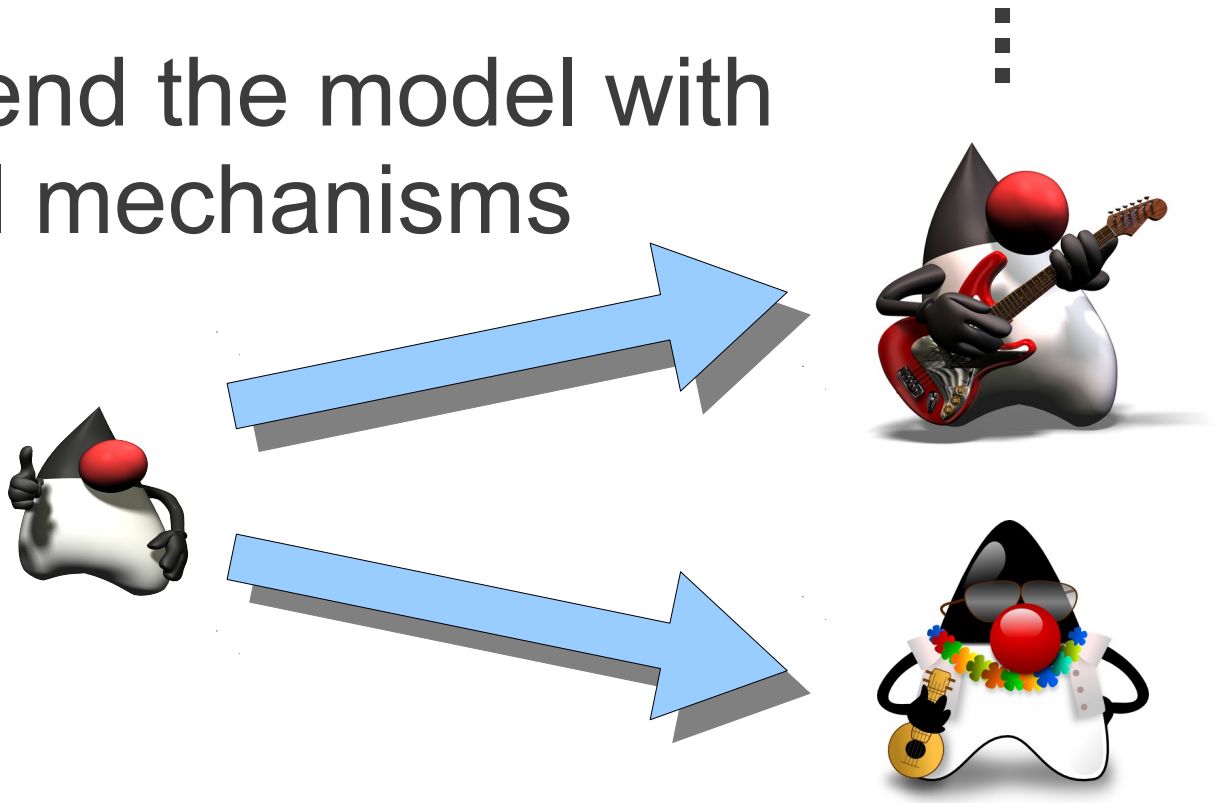
My Featherweight Approach to FOOL



1. Build a tiny model of OOPLs

My Featherweight Approach to FOOL

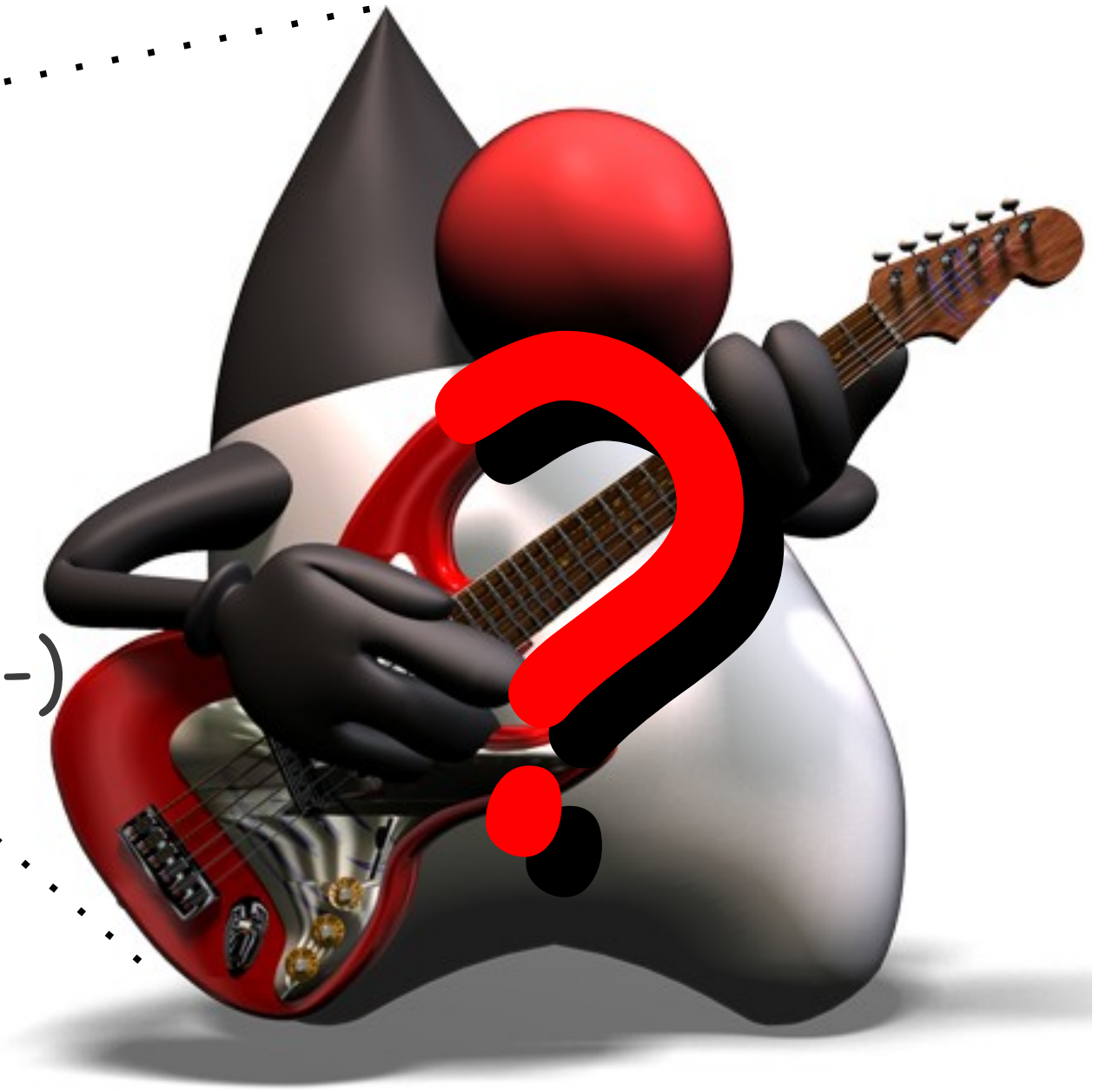
2. Extend the model with cool mechanisms



3. Study their theories to show they are reasonable and ...

My Featherweight Approach to FOOL

... hope someone implements them :-)



Longman Dictionary of Contemporary English says...

lightweight² ***adj*** **1** weighing less than average: *special lightweight fabric*

Longman Dictionary of Contemporary English says...

lightweight[2] ***adj*** **1** weighing less than average: *special lightweight fabric* **2** showing a lack of serious thought: *She's written nothing but light-weight novels.*

Aim of This Talk

- Convince that featherweight approaches
  - have been useful for FOOL
    - esp. language extensions
  - (are not lightweight in the second sense)
- Share some lessons I learned over the years

Table of Contents

A Brief History of FOOL Study in '80s & '90s

Caveats

- (Un)intentionally oversimplified
- Focusing only on Smalltalk-style languages
  - Class-based
  - Single-dispatch

Early FOOL Study

Main questions:

- What are objects?
- What are inheritance, subtyping, and their relationship?
- What is a static type system for objects?

Approach: “Landin reductionism”

Slogan: “Express everything in the λ-calculus!”

- Object as a recursive record of functions
- Message send as field projection
- Class as a function from “self” to a method suite
  - Parameterization to express late binding
- Inheritance as record extension
  - `{x=3} ++ {y=2}` ⟶ `{x=3; y=2}`

Early Type Theory for Objects

Object type ≒ (recursive) record types

'84: Record and function subtyping [Cardelli]

'85: Bounded quantification [Cardelli&Wegner]

'89: F-bounded quantification [Canning et al.]

'91: Subtyping recursive types [Amadio&Cardelli]

Lots of cool ideas,
but somewhat overwhelming...

FOOL Study in '90s

Simpler approaches to *typed* objects

'92: Existential encoding [Pierce&Turner92]

'93～'96: Calculi of *primitive* objects [Abadi&Cardelli][Fisher,Honsel&Mitchell]

- A slight departure from Landin reductionism
  - Base calculi are still very primitive

Between Theory and Practice

aka. declaration-based

	Theory	Practice
Subtyping	Structural	Nominal (♠)
Classes	First-class (♣)	Second-class

♠ Result from (intentional?) confusion between

- classes and types
- inheritance and subtyping

♣ A consequence of Ladin reductionism

- Classes have to be given types

Summarizing FOOL Study until Mid '90s

- Encoding objects and classes into very primitive calculi
  - Successful especially for untyped objects
- Lots of interesting and substantial type theory
- Still gaps from mainstream languages

Table of Contents

Boom of Java!

Triggered two lines of research:

- "Is Java really safe?"
  - A FOOL-ish question, obviously!
- "It's a chance to add my cool idea to this new popular language!"
  - Generics, Multi-methods, Virtual Types, Mixins...

Research on Type Safety of Java

- “Java is not Type Safe” [Saraswat97]
- “Java is Type Safe – Probably” [Drossopoulou&Eisenbach97]
- “Java$_{light}$ is Type Safe – Definitely” [Nipkow&von Oheimb98]

and many other interesting papers...

Research on Type Safety of Java

- "Java is not type safe" [Saraswat97]
  - Pointing out a class loader bug
- "Java is Type Safe – Probably" [Drossopoulou&Eisenbach97]
  - Formal model of a significant subset of Java
  - Type safety proofs
- "Java$_{light}$ is Type Safe – Definitely" [Nipkow&von Oheimb98]
  - Model and proofs mechanized on Isabelle/HOL

Language Extensions Reseach

- Few papers really discuss foundational issues
- Some notable exceptions:
  - “Classes and Mixins” [Flatt, Krishnamurthi & Felleisen'98]
    - ClassicJava: A subset of imprerative Java
    - MixedJava: ClassicJava with mixins
  - “Ownership Types for Flexible Alias Protection” [Clarke, Potter & Noble'98]

“Featherweight Java: A Minimal Core Calculus for Java and GJ”

by I., B.C. Pierce, & P. Wadler [OOPSLA'99, TOPLAS'01]

- A sublanguage of Java with a formal type system and (operational) semantics
- Minimal set of features
  - (Second-class) classes with (single) inheritance
  - Recursion through **`this`**
  - Dynamic typecast
  - No assignments
- The choice of features depended upon the main motivation, namely...

Main Motivation

- Study of foundational issues of generics for Java (in particular, GJ [Bracha et al. 98])
  - Type safety
  - Correctness of “erasure” compilation to JVML
- Not to prove type safety of as large a subset of Java as possible

Main Technical Results

- Def. of Featherweight Java (FJ)
- Type safety theorem of FJ
- Def. of Featherweight GJ (FGJ)
  - An extension of FJ with generics
  - “Direct” operational semantics
- Type safety of Featherweight GJ
- Def. of compilation from FGJ to FJ
- Theorem of compilation correctness

FJ: Some Points of Interest

- Classes are second-class citizens
- Nominal subtyping
  - Reminiscent of “Amber rule”
- Dynamic casts
  - Needed to model erasure compilation
- Minimal set of language features
  - Lack of assignments

Inherited from earlier work

Classes are Second-Class

- Classes are not part of expressions to "run"

P (programs) ::= $(L_1, \ldots, L_n, e)$ $\quad$ L (classes) ::= ...

e (expr.) ::= $x \mid e.m(e_1,\ldots,e_n) \mid$ **new** $C(e_1,\ldots,e_n) \mid \ldots$

  - Reduction: $e \rightarrow e'$ (under a *fixed* set of classes)
    - c.f. Term rewriting systems
- Classes do not really compose
  - No need for fancy operations on records
- No type *expressions* for classes
  - Expression typing: $x_1{:}C_1,\ldots,x_n{:}C_n \vdash e : C$
  - Class typing: L ok

Nominal Subtyping

- Subtyping relation $C <: D$ is extracted from **`extends`** clauses of the given classes
- Subtyping is “confirmed” to be safe only *after* typechecking
  - esp. after checking correct method overriding
  - c.f. Subtyping for recursive types (Amber rule)

Declaration-based Subtyping

- Subtyping relation $C <: D$ is extracted from **`extends`** clauses of the given classes
- Subtyping is "confirm
  typechecking
  - esp. after checking
  - c.f. Subtyping for recursive types (Amber rule)

$$\frac{X <: Y \vdash S(X) <: T(Y)}{\vdash \mu X.S(X) <: \mu Y.T(Y)}$$

μObj.{clone: ()→Obj, …}

<:

μNum.{clone: ()→Num, …}

Dynamic Casts

- Hard to express in typed lambda-calculus
  - Casts bypass typechecking
  - Casts do run-time checking, which compares *class names* according to **`extends`**
- Required (only) to model erasure compilation
  - Obvious candidates of further simplification

Minimal Set of Features

- This is the *whole* syntax of FJ!

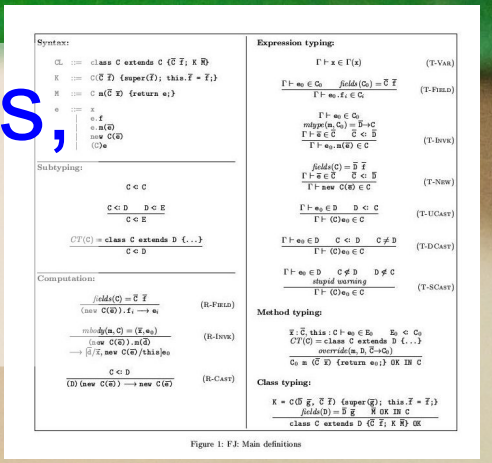
P (programs) ::= (L_1,...,L_n, *e*)
L (classes) ::=
    class *C* **extends** *C* **{** *C f;*... *C f;* *K* *M* ... *M* **}**
K (constructors) ::= ...
M (methods) ::= *C m* **(** *C x* **,** ... **,** *C x* **)** **{** **return** *e;* **}**
e (expressions) ::=
    x | **this** | *e* **.** *f* | *e* **.** *m* **(** ~*e* **)** | **new** *C* **(** ~*e* **)** | **(** *C* **)** *e*

Quantitative Evaluation

Quantitative Evaluation : –)

FJ: 1 page, in 9pt, two columns,

0.0 pounds



Syntax:

CL ::= class C extends C {C̄ f̄; K M̄}

K ::= C(C̄ f̄) {super(f̄); this.f̄ = f̄;}

M ::= C m(C̄ x̄) {return e;}

e ::= x | e.f | e.m(ē) | new C(ē) | (C)e

Subtyping:

$$\frac{}{C <: C} \qquad \frac{C <: D \quad D <: E}{C <: E}$$

$$\frac{CT(C) = \text{class } C \text{ extends } D \ \{\ldots\}}{C <: D}$$

Computation:

$$\frac{fields(C) = \overline{C}\ \overline{f}}{(\text{new } C(\overline{e})).f_i \longrightarrow e_i} \ (\text{R-Field})$$

$$\frac{mbody(m, C) = (\overline{x}, e_0)}{(\text{new } C(\overline{e})).m(\overline{d}) \longrightarrow [\overline{d}/\overline{x}, \text{new } C(\overline{e})/\text{this}]e_0} \ (\text{R-Invk})$$

$$\frac{C <: D}{(D)(\text{new } C(\overline{e})) \longrightarrow \text{new } C(\overline{e})} \ (\text{R-Cast})$$

Expression typing:

$$\Gamma \vdash x \in \Gamma(x) \ (\text{T-Var})$$

$$\frac{\Gamma \vdash e_0 \in C_0 \qquad fields(C_0) = \overline{C}\ \overline{f}}{\Gamma \vdash e_0.f_i \in C_i} \ (\text{T-Field})$$

$$\frac{\Gamma \vdash e_0 \in C_0 \quad mtype(m, C_0) = \overline{D} \to C \quad \Gamma \vdash \overline{e} \in \overline{C} \quad \overline{C} <: \overline{D}}{\Gamma \vdash e_0.m(\overline{e}) \in C} \ (\text{T-Invk})$$

$$\frac{fields(C) = \overline{D}\ \overline{f} \quad \Gamma \vdash \overline{e} \in \overline{C} \quad \overline{C} <: \overline{D}}{\Gamma \vdash \text{new } C(\overline{e}) \in C} \ (\text{T-New})$$

$$\frac{\Gamma \vdash e_0 \in D \quad D <: C}{\Gamma \vdash (C)e_0 \in C} \ (\text{T-UCast})$$

$$\frac{\Gamma \vdash e_0 \in D \quad C <: D \quad C \neq D}{\Gamma \vdash (C)e_0 \in C} \ (\text{T-DCast})$$

$$\frac{\Gamma \vdash e_0 \in D \quad C \neq D \quad D \not<: C}{stupid\ warning \quad \Gamma \vdash (C)e_0 \in C} \ (\text{T-SCast})$$

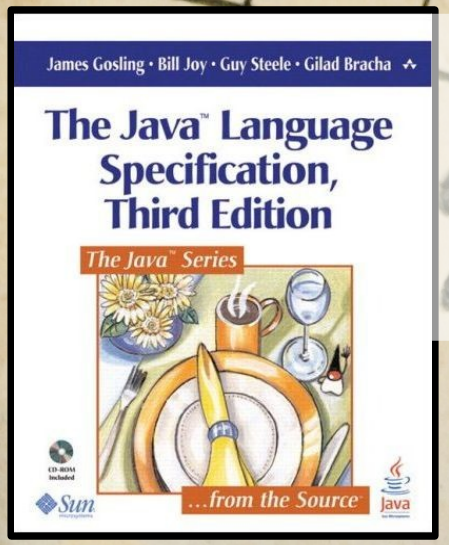
Method typing:

$$\frac{\overline{x} : \overline{C}, \text{this} : C \vdash e_0 \in E_0 \quad E_0 <: C_0 \quad CT(C) = \text{class } C \text{ extends } D \ \{\ldots\} \quad override(m, D, \overline{C} \to C_0)}{C_0\ m\ (\overline{C}\ \overline{x})\ \{\text{return } e_0;\}\ \text{OK IN } C}$$

Class typing:

$$\frac{K = C(\overline{D}\ \overline{g}, \overline{C}\ \overline{f})\ \{\text{super}(\overline{g}); \text{this}.\overline{f} = \overline{f};\} \quad fields(D) = \overline{D}\ \overline{g} \quad \overline{M}\ \text{OK IN } C}{\text{class } C \text{ extends } D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}\ \text{OK}}$$

Figure 1: FJ: Main definitions



JLS 3rd ed.: 650 pages,

2.2 pounds

Re: Lack of Assignments

We felt formalizing assignments wouldn't give us deeper insights (matching the price to pay)

- Some reasonable responses we got:
  - "State change is the essence of OO!"
  - "Do you know ML type inference?"
    - In fact, GJ type inference turned out to be flawed later ;-([Jefferey]
- One (and, perhaps, only) justification (excuse?):
  - Interesting results even without them

Pleasant Surprise!

- FJ has become a popular tool to study (type systems of) language extensions
  - Especially, class-based language abstractions
- Some reasons for wide adoption:
  - The name was catchy, perhaps (thanks, Phil!)
    - Initially called “the J-calculus,” IIRC
  - It doesn't have your favorite mechanism
    - You cannot help adding something!

Applications of FJ

Existing Advanced Class Mechanisms

- Generics
- Inner Classes

New Advanced Class Mechanisms

- Use-site Variance
- Self Type Constructors [OOPSLA'09]
- Lightweight Family Polymorphism [APLAS'05; JFP'08]
- Gradua Typing [OOPSLA'11]
- Variant Path Types [OOPSLA'07]
- Union types [SAC'06; JOT'07]

Applications of FJ

Existing Advanced Class Mechanisms

- Generics
- Inner Classes

New Advanced Class Mechanisms

- Use-site Variance
- Self Type Constructors [OOPSLA'09]
- Lightweight Family Polymorphism [APLAS'05; JFP'08]
- Gradua Typing [OOPSLA'11]
- Variant Path Types [OOPSLA'07]
- Union types [SAC'06; JOT'07]

Formal Semantics for Inner Classes

I. & B.C. Pierce. “On Inner Classes” [ECOOP'00]

Applying FJ to inner classes of Java 1.2 to answer

- How do inheritance and nesting interact when
  - An inner class can access members of an enclosing class
  - A top-level subclass can extend an inner class nested in an unrelated class

  ?

Interesting ~~Nightmarish~~ Aspect of This Work

- Inner Classes Specification didn't help figuring out corner cases
- It was “software physics”:
  - Observe the behavior of software (in this case, **`javac`**)
    - without reading the source code
  - Formalize it!

Indeed, this work led us to (re)discovering (known) compiler bugs

Applications of FJ

Existing Advanced Class Mechanisms

- Generics
- Inner Classes

New Advanced Class Mechanisms

- Use-site Variance
- Self Type Constructors [OOPSLA'09]
- Lightweight Family Polymorphism [APLAS'05; JFP'08]
- Gradua Typing [OOPSLA'11]
- Variant Path Types [OOPSLA'07]
- Union types [SAC'06; JOT'07]

Type System for Use-Site Variance

I. & M. Viroli. “On Variance-Based Subtyping for Parametric Types” [ECOOP'02]

- Slogan: “*More Subtyping for Generics*”
  - Generalization of structural virtual types
- Formalization on top of Featherweight GJ
- First type safety proof of a variance system
  - Existential types as a background theory
- Basis of Java Wildcards

Two Subtyping Schemes for Generics

- Inheritance-based subtyping:
  - Q: When `C<T> <: D<T>` for given type `T`?
  - A: `class C<X> extends D<X> {...}`
- Variance-based subtyping:
  - Q: When `C<S> <: C<T>`?
    - Subtyping between two types from the *same* generic class

`List<Integer> <: List<Number>`?

A few different Answers:

- Unsafe subtyping (Eiffel around '90 [Cook90]):
  "Yes, as long as your program doesn't add a **`Number`** to **`List<Number>`**."
  - "Otherwise, your program may crash :-p "
  - Java array types inherit this
- Definition-site variance (POOL-I [America90])
- Use-site variance

Definition-Site Variance

"Yes, provided that `List` doesn't have public methods to put elements"

- Type parameter declaration with a variance property
- Trade-off between methods and subtyping

```
class List<+X> {
 // List<Int>
 //    <: List<Num>
 // no meth. to put
}
```

```
class List<-X> {
 // List<Num>
 //    <: List<Int>
 // no meth. to get
}
```

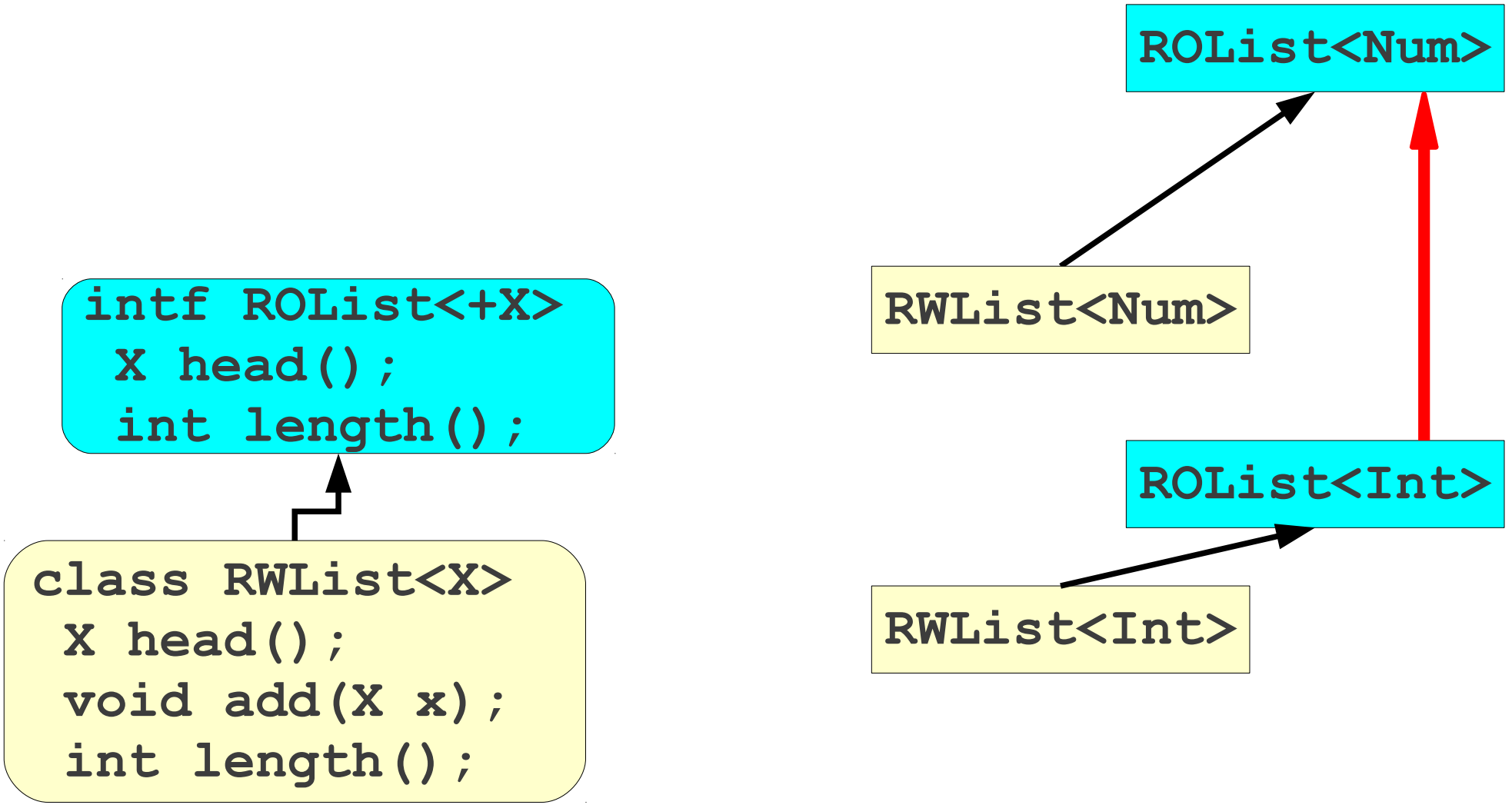
When you need RWLists ...

```
RWList<Num>
```

```
class RWList<X>
 X head();
 void add(X x);
 int length();
```

```
RWList<Int>
```

When you need RWLists ...



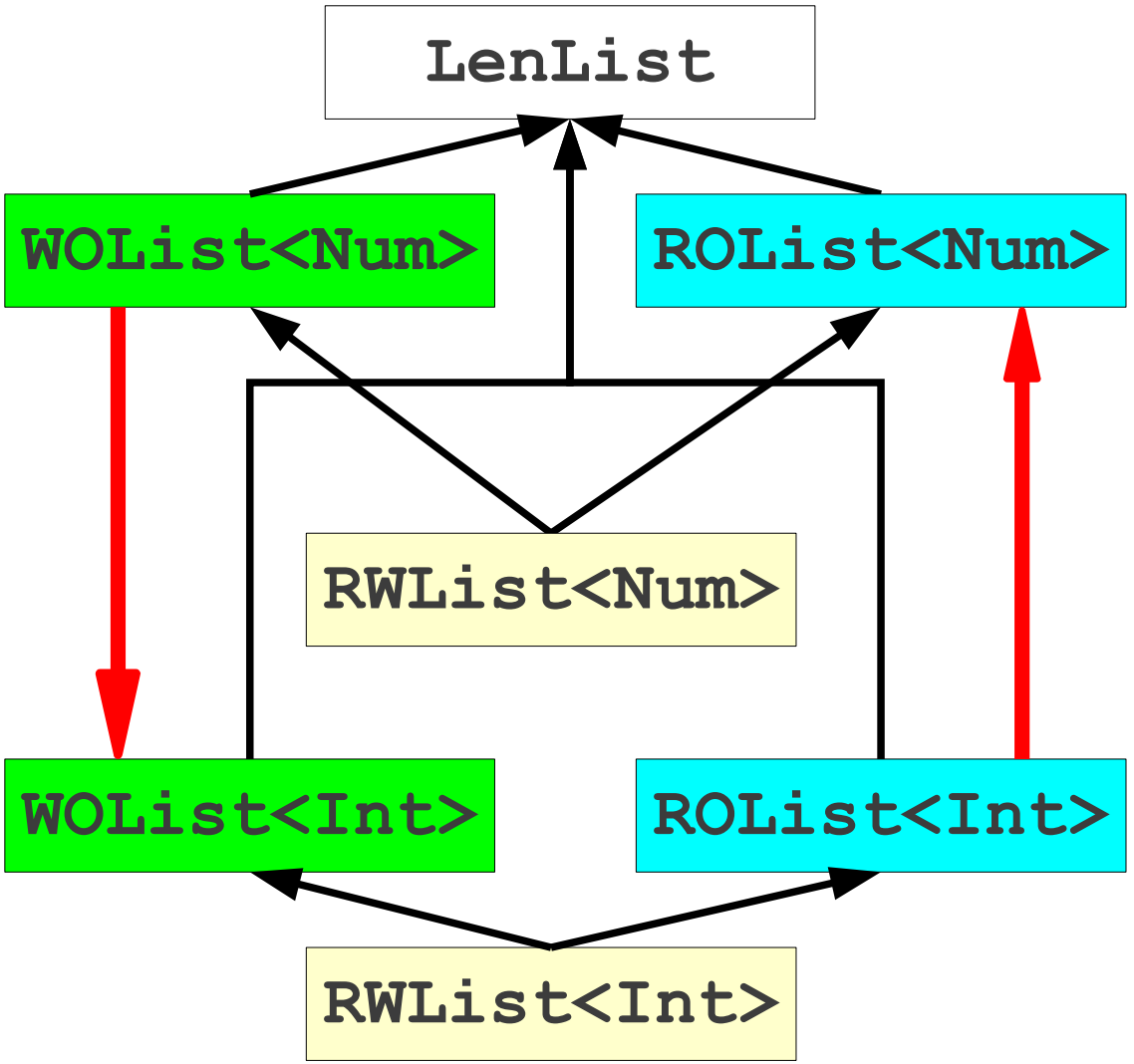
When you need RWLists ...

```
intf LenList
   int length();
```

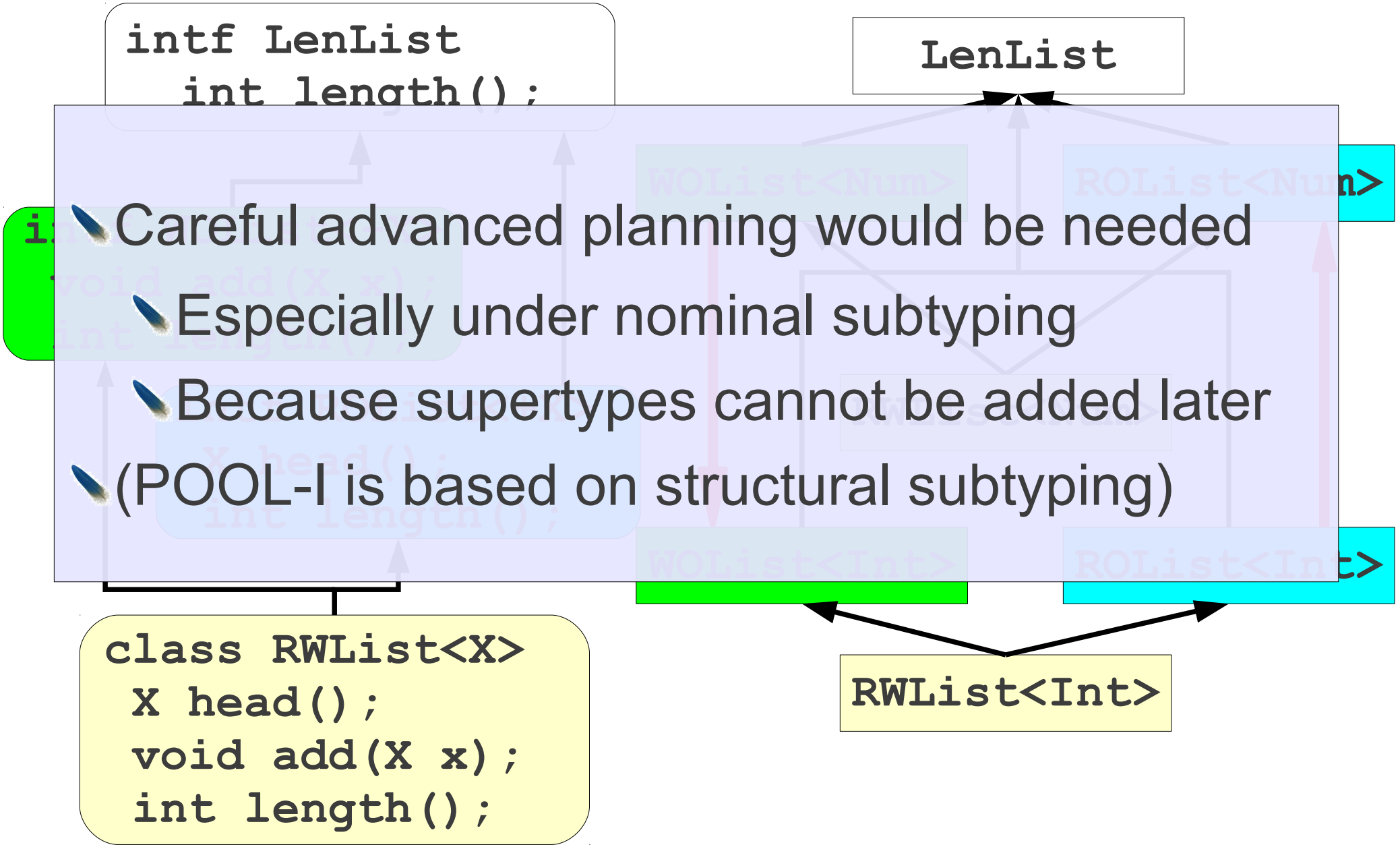
```
intf WOList<-X>
 void add(X x);
 int length();
```

```
intf ROList<+X>
 X head();
 int length();
```

```
class RWList<X>
 X head();
 void add(X x);
 int length();
```



When you need RWLists ...

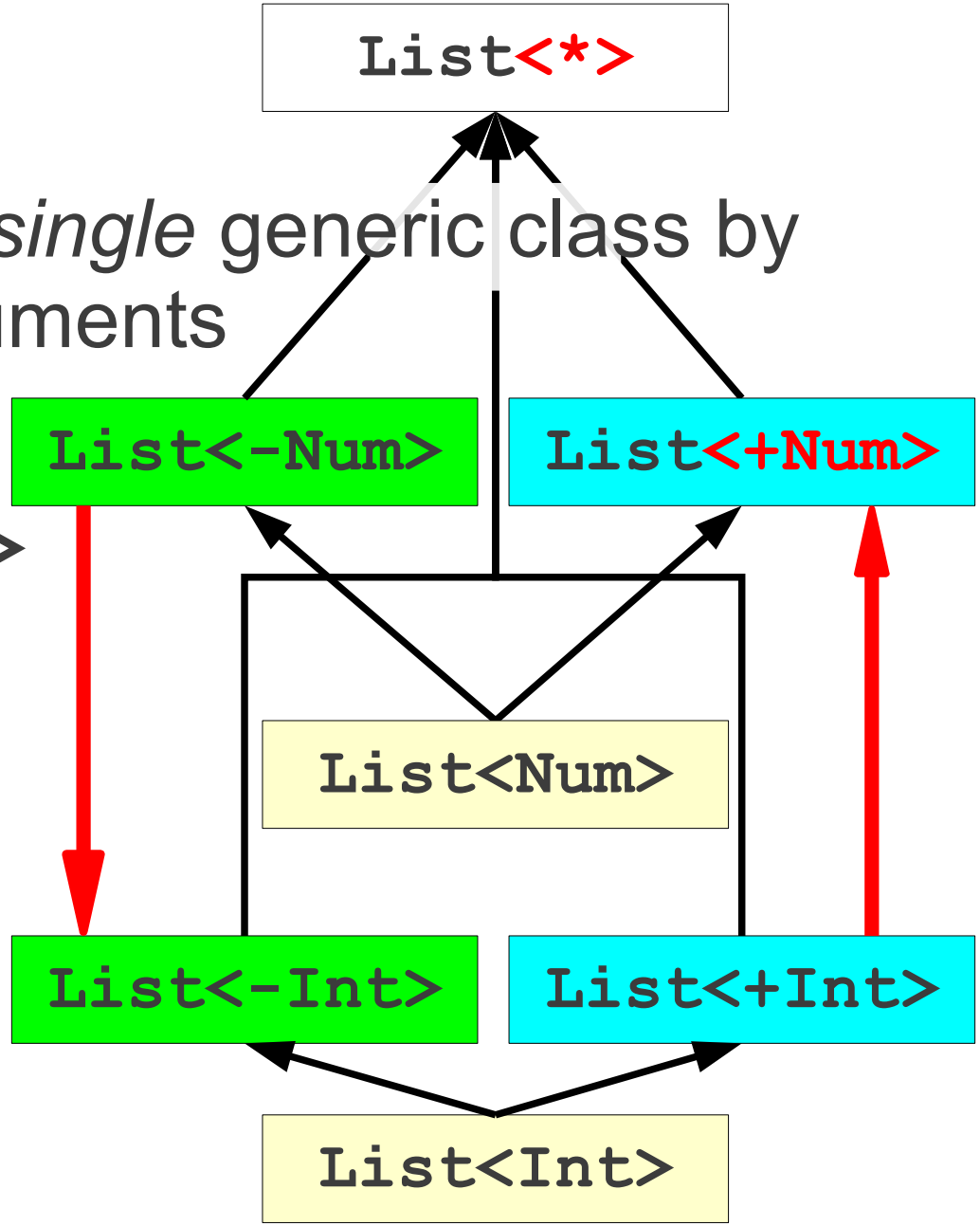


Our Answer: Use-Site Variance

Different interfaces from a *single* generic class by annotating actual type arguments

- `add()` missing in `List<+T>` and `List<*>`
- `head()` missing in `List<-T>` and `List<*>`

```
class List<X>
 X head();
 void add(X x);
 int length();
```



Use-site Variance as Existential Types

- A variable of `List<+Num>` can store
  - `List<Int>`, `List<Float>`, and so on
  - namely, `List` of *some* kind of numbers
- In type theory, such a type is expressed as an existential type $\exists$`X<:Num.List<X>`
  - Similarly, `List<-Num>` = $\exists$`X:>Num.List<X>`

Typing rules for use-site variance follow from this intuitive correspondence!

It's a Natural Idea (to me :-)!

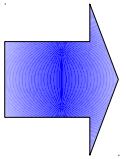
- Virtual types as an alternative to generics [Thorup 97]
- Safe virtual types [Torgersen 98]
- Structural virtual types [Thorup&Torgersen 99]
  - Essentially, use-site variance only with `List<T>` and `List<+T>`
- Modeling virtual types as existentials [I.&Pierce 99]

From Use-Site Variance to Wildcards

"Adding Wildcards to the Java Programming Language"
[Torgersen at al. 04]

- Cosmetic changes ...

```
List<*>          List<?>
List<+Number>    List<? extends Number>
List<-Number>    List<? super Number>
```



  - Emphasizing the existential nature
    - which we tried to hide under the hood :-)
- ... and some other improvements to make them useful
- Adaption of library to take adv. of wildcards

Some Criticisms on Wildcards

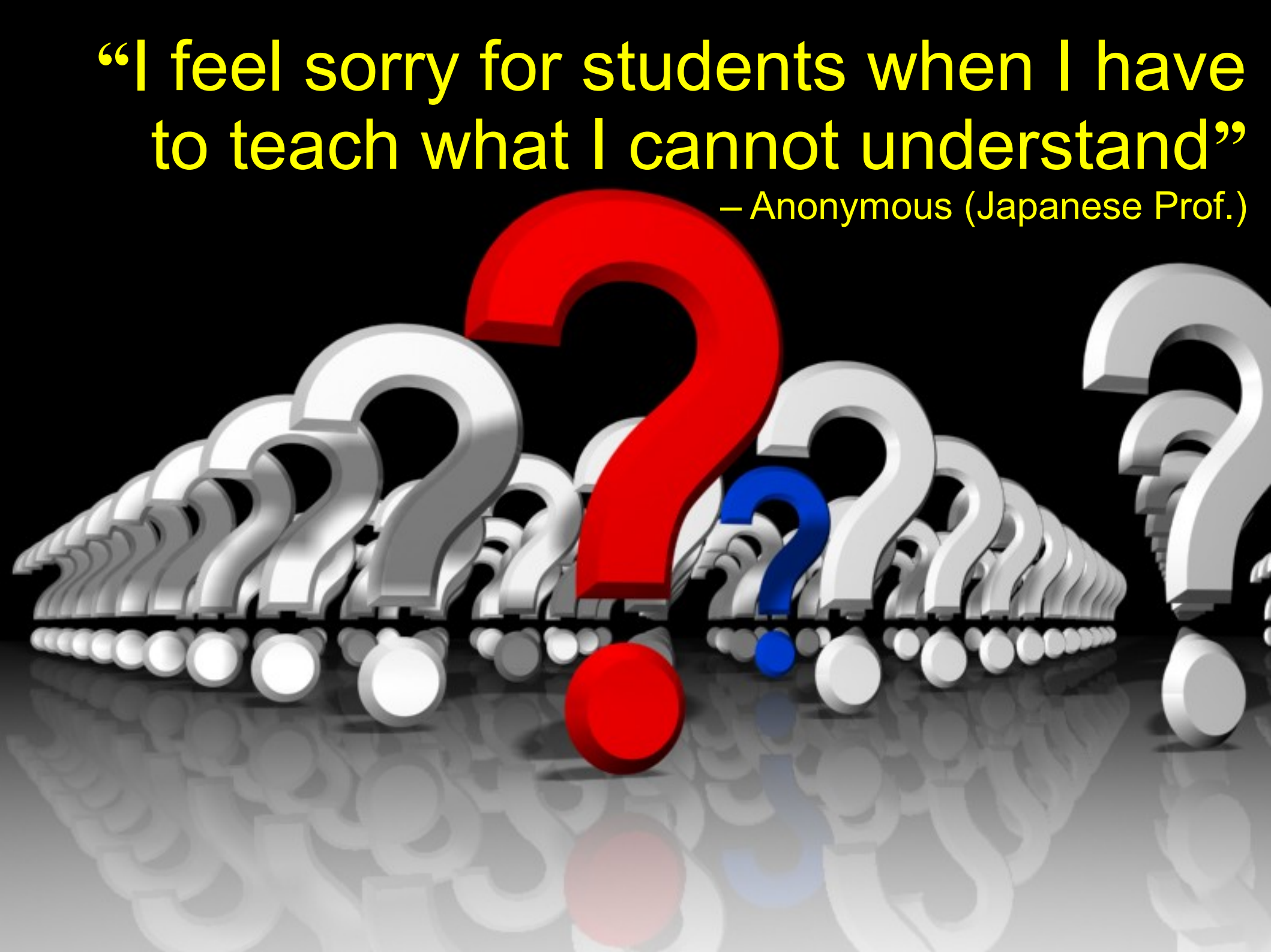
- “Use-site variance places a great burden on the user of generic types” [Emir et al. 06]
  - In fact, OCaml, Scala and C# later adopt definition-site variance
- Decidability of subtyping of use-site variance is still open! [Kennedy&Pierce07]
- And even...

“We simply cannot afford another wildcards”

– Joshua Bloch

“I feel sorry for students when I have to teach what I cannot understand”

– Anonymous (Japanese Prof.)



What's Unusual about Use-Site Variance

- Subtyping with more of structural flavor

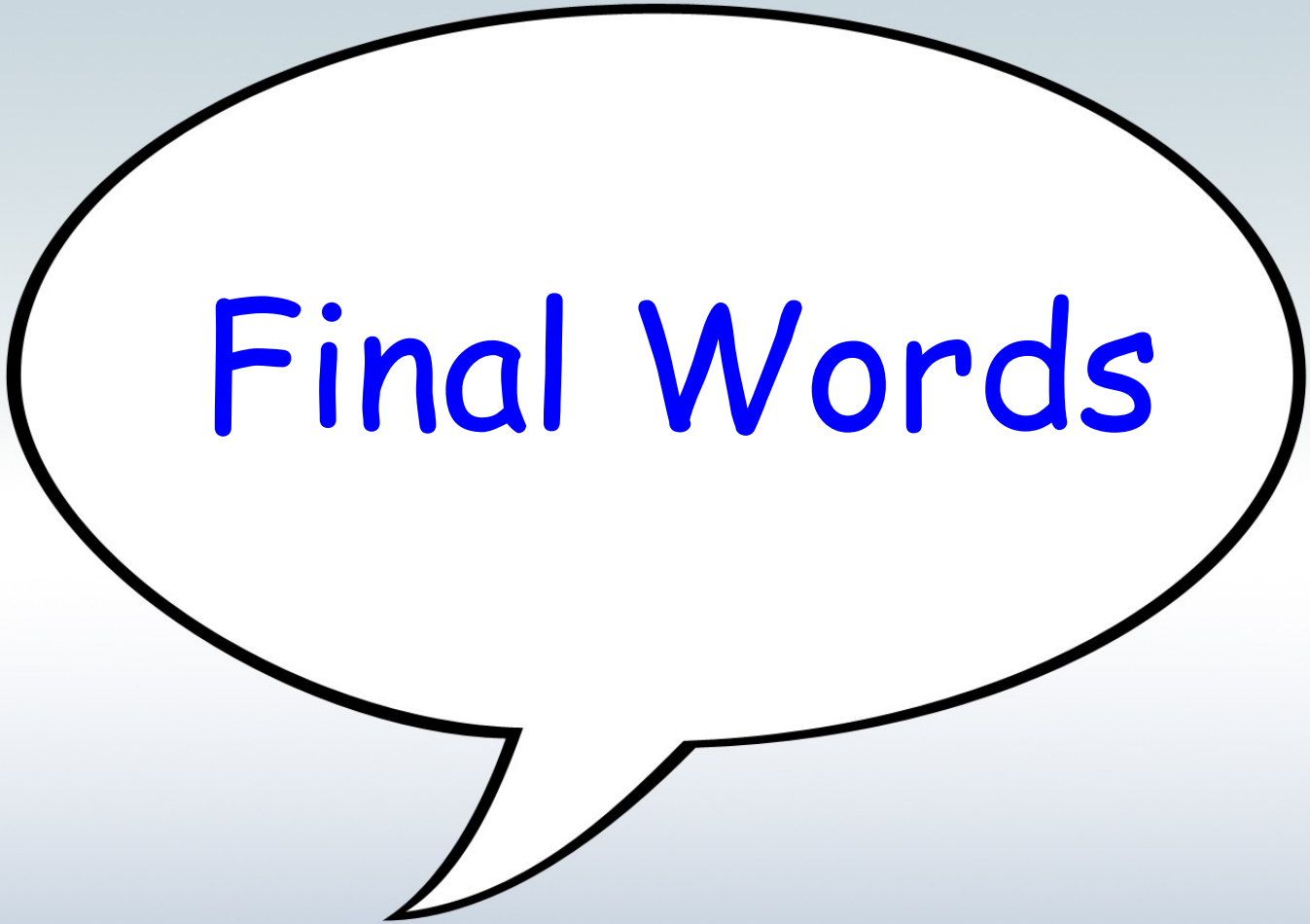
	Types	Type comparison
Java 1.x	Atomic	Atomic
GJ	Structural	Mostly atomic
Wildcards	Structural	Structural

- Separation of static and run-time types
  - There is no instance of `List<+Num>`
- “Post hoc” supertypes

Was It Really a Bad Idea?

- I'm not qualified to judge `:-)`
  - Maybe only history will tell us
- Still, post-hoc supertypes are often very useful
  - `C<S>` and `C<T>` *always* have a common supertype `C<? extends U>` (for `S`, `T <: U`)
    - Otherwise, it might even be `Object`
- Further research is needed, anyway
  - "Taming the Wildcards" [Altidor, Huang, Smaragdakis11]

Table of Contents


Final Words

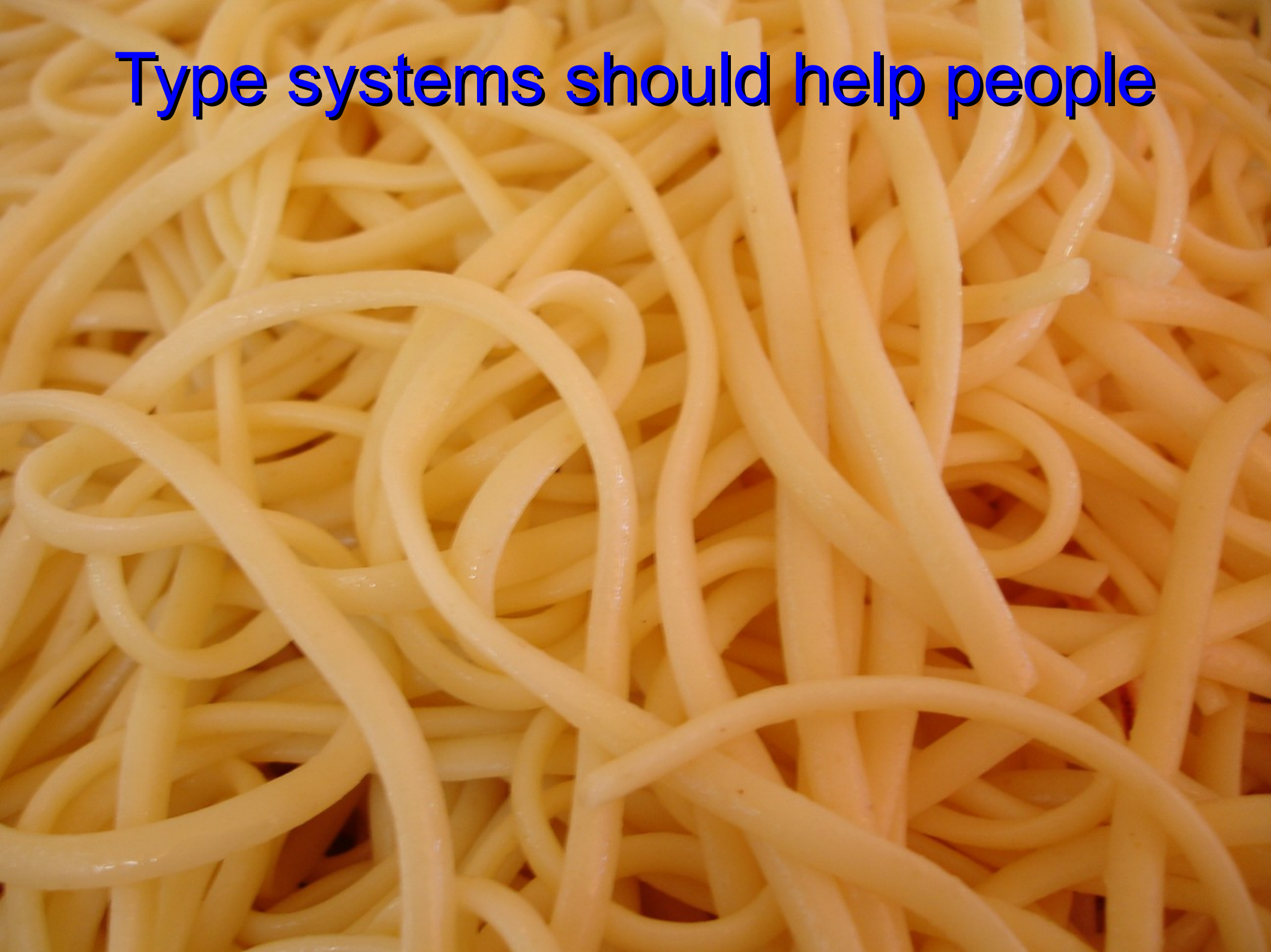
You can throw away most as long as something interesting is left

should

You ~~can~~ throw away most

~~as long as~~ something interesting ~~is left~~

to distill


Type systems should help people

Type systems should help people write programs in *good styles*

Igarashi's Conjecture

If a new, cool programming style emerges, there will be a type system to enforce it.


Working for building FOOL might not look very attractive at first...


… but, the Fun of Your OO Life is Dependent on FOOL!
BALLYS
James' SALT WATER TAFFY
CUT TO FIT THE MOUTH
PIZZA
DINNERS
STEAKS
SALADS
SUBS
James' SALT WATER TAFFY
James' SALT WATER TAFFY
HAUNTED TALES
COLLECTIBLES WORLD
PACIFIC SIDE
CITY SOUVENIRS
ATLANTIC SOUVENIRS
GODFATHERS


… but, the Fun of Your OO Life is Dependent on FOOL!
Really!
BALLYS
James'
SALT WATER TAFFY
CUT TO FIT THE MOUTH
PIZZA
DINNERS
STEAKS
SALADS
SUBS
GODFATHERS

To my collaborators:

João Filipe Belo Shigeru Chiba Michael Greenberg
Robert Hirschfeld Lintaro Ina Masashi Iwaki
Futoshi Iwama Yukiyoshi Kameyama
Naoki Kobayashi Kensuke Kojima Hidehiko Masuhara
Hideshi Nagira Benjamin C. Pierce
Chieri Saito Takafumi Sakurai Masahiko Sato
Naokata Shikuma Manabu Toyama
Takeshi Tsukada Mirko Viroli Philip Wadler
Yosihiro Yuse Salikh Zakirov

… my family and *ALL* users of FJ!