Static and Dynamic Type Checking: A Synopsis

Ronald Garcia
University of British Columbia

Trends in Functional Programming, Volume 8

Selected papers from the Eighth Symposium on Trends in Functional Programming (TFP07),
New York City, USA, April 2-4, 2007.
Marco T. Morazán (editor)
Intellect, Bristol, UK.
ISBN 978-1-84150-196-3

Table of Contents

Trends in Functional Programming, Volume 8

Selected papers from the Eighth Symposium on Trends in Functional Programming (TFP07),
New York City, USA, April 2-4, 2007.
Marco T. Morazán (editor)
Intellect, Bristol, UK.
ISBN 978-1-84150-196-3

Table of Contents

Trends in Functional Programming, Volume 8

Selected papers from the Eighth Symposium on Trends in Functional Programming (TFP07),
New York City, USA, April 2-4, 2007.
Marco T. Morazán (editor)
Intellect, Bristol, UK.
ISBN 978-1-84150-196-3

Table of Contents

Gradual Typing

Static Checking:
(Robustness)

Dynamic Checking:
(Agility)

Complementary
Disciplines

Typing, Gradually

```
struct Point { x }

fun move(p, dx) {
    Point(p.x + dx)
}

let
    a = 1
    p = Point(7)
in
    move(p,a)
```

Typing, Gradually

```
struct Point { x : int }

fun move(p, dx) {
    Point(p.x + dx)
}

let
    a = 1
    p = Point(7)
in
    move(p,a)
```

Typing, Gradually

```
struct Point { x : int }

fun move(p : Point, dx) {
    Point(p.x + dx)
}

let
    a = 1
    p = Point(7)
in
    move(p,a)
```

Typing, Gradually

```
struct Point { x : int }

fun move(p : Point, dx : int) {
    Point(p.x + dx)
}

let
    a : int = 1
    p : Point = Point(7)
in
    move(p,a)
```

More Annotations
=
More Static Checks!

Typing, Gradually

```
if (SomethingTrue()) {
   x = 7.0
} else {
   x = “Good Grief!”
}
y = sqrt(x)
```

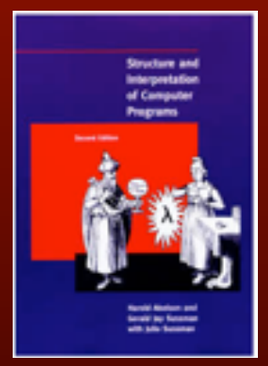
Flexibility
On
Demand!

Recurring Trend

Static Languages

Haskell

Dynamic Languages



Scheme

Recurring Trend

Static Languages

Dynamic Languages

DLR

TypeScript

Java

DuctileJ

Haskell

Deferred Type Errors

Racket

Scheme

Facebook wanted its coders to keep moving fast in the comfort of their native tongue, but it didn't want them to have to break things as they did it.

James Somers, "Toolkits for the Mind"
Technology Review, April 2, 2015

Why would I want to use it?

When you're prototyping, you may not want the compiler bothering you about broken code that you don't care about. Or you may want to use *duck typing* while you're fleshing out the design of your interfaces

DuctileJ homepage

Lots of Theories

Hybrid Type Checking

Dynamic typing:

Quasi-static Typing

Soft Typing

Gradual Typing

Lots of Theories

Hybrid Type Checking

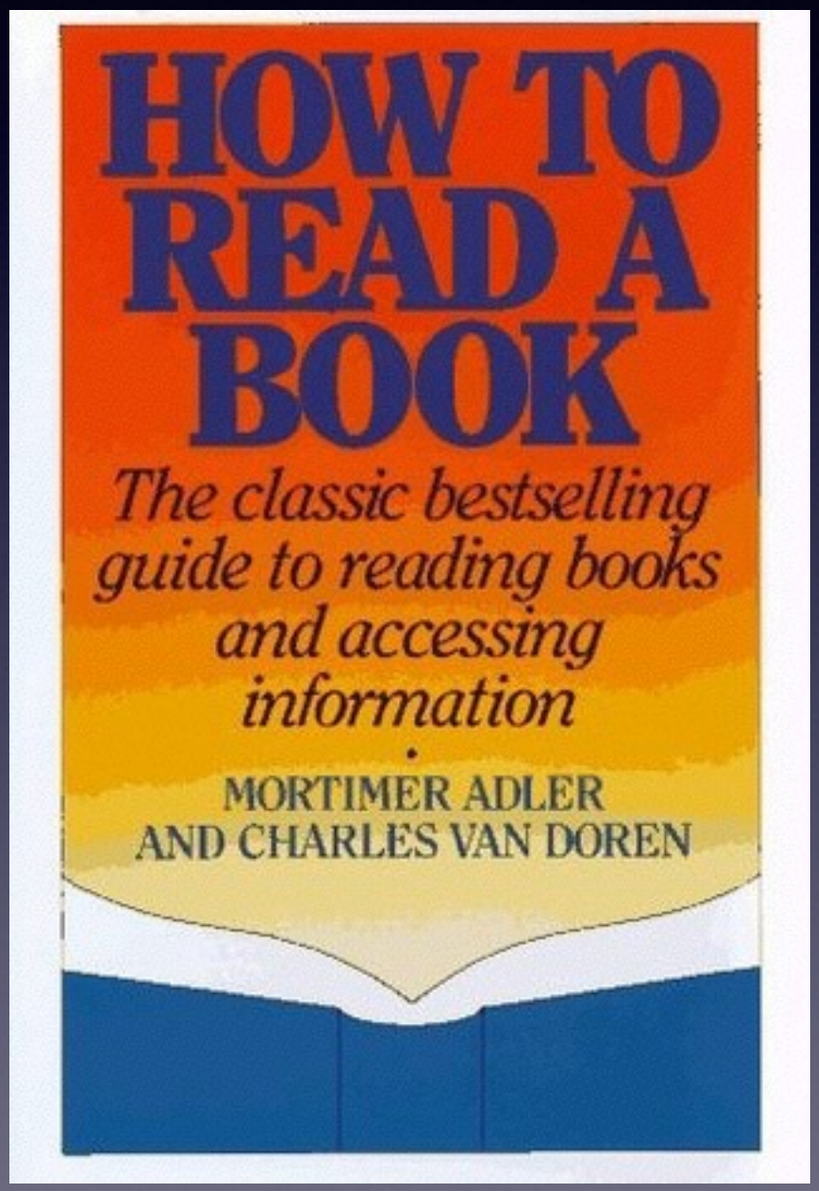
Dynamic typing:

Quasi-static Typing

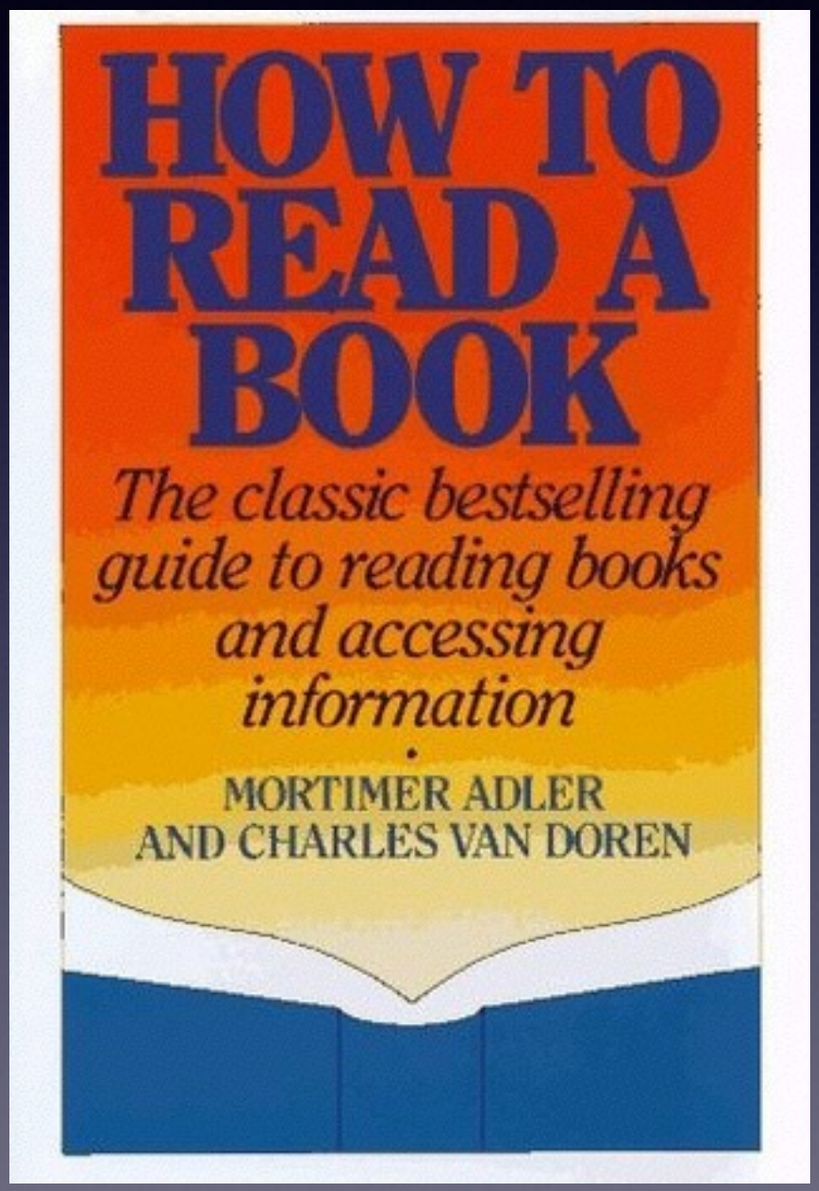
Soft Typing

Gradual Typing

Synopsis

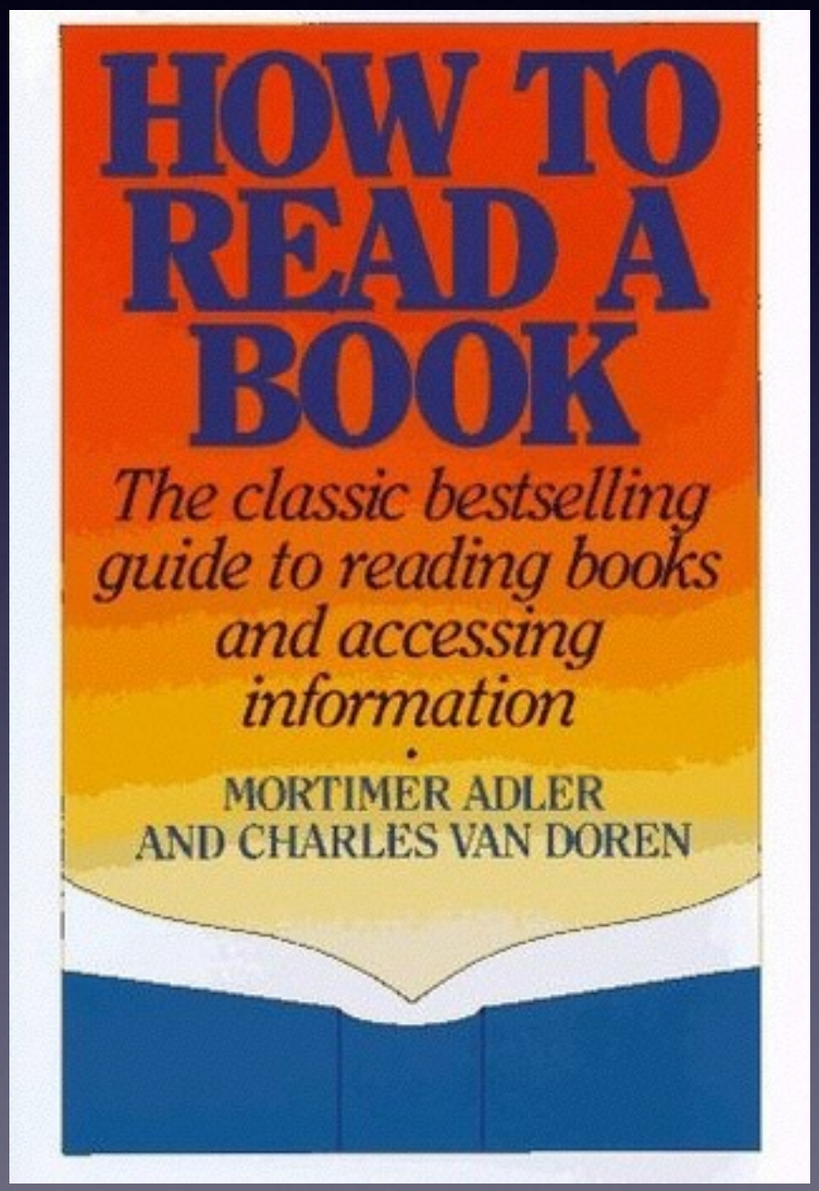

HOW TO READ A BOOK
The classic bestselling guide to reading books and accessing information
MORTIMER ADLER AND CHARLES VAN DOREN

Synopsis



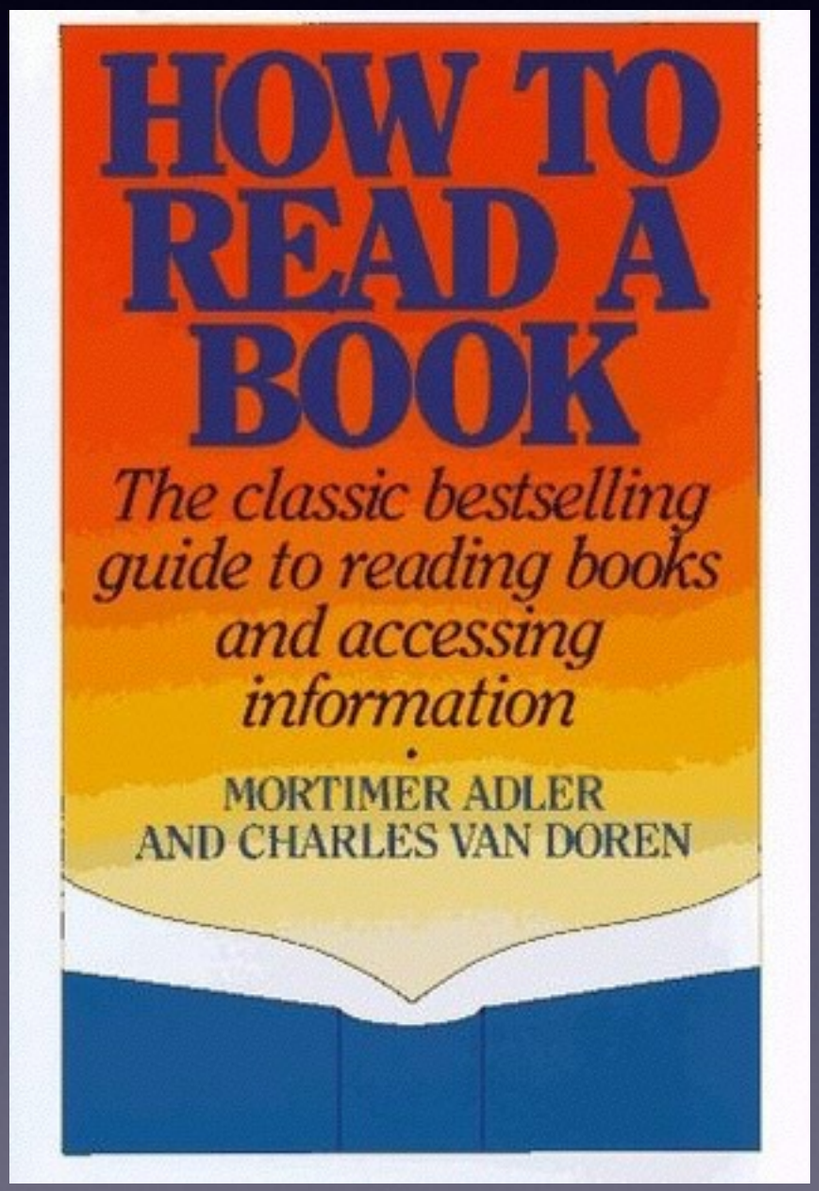
- Levels of Reading:
  - Elementary
  - Inspectional
  - Analytical
  - Synoptical

Synopsis



“The difficulty begins as soon as we examine the phrase ‘two or more books on the same subject.’”

Synopsis



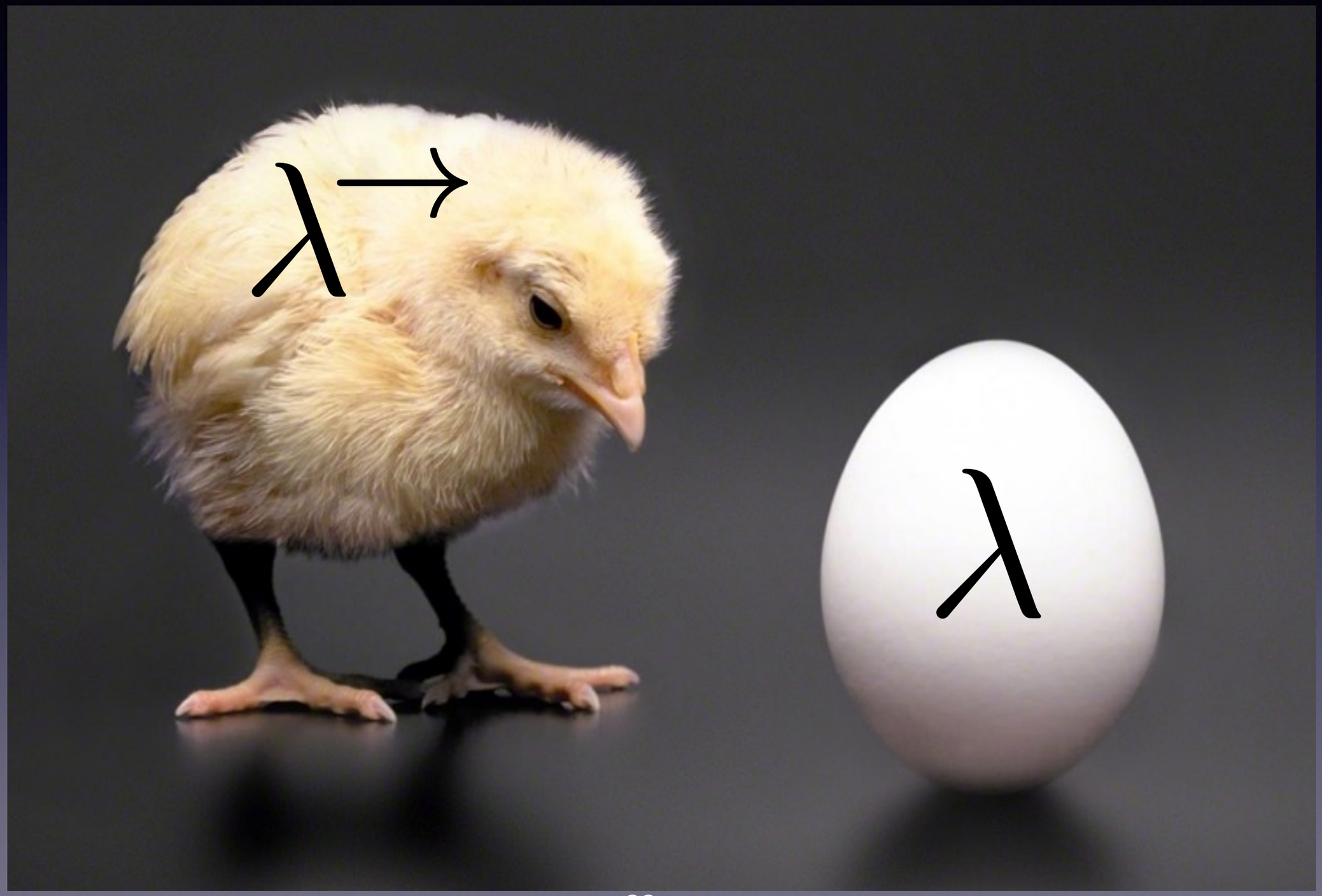
"The difficulty begins as soon as we examine the phrase 'two or more books on the same subject.'"

"What do we mean by 'same subject'?"

Structure of the Talk

- Synopsis: Dynamic and Static Type Checking
- Focus: Gradual Information-Flow Typing

1: Which Comes First?



A Theory of Type Polymorphism in Programming

ROBIN MILNER

Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received October 10, 1977; revised April 19, 1978

A Theory of Type Polymorphism

$$e ::= x \mid (ee') \mid \textit{if } e \textit{ then } e' \textit{ else } e'' \mid \lambda x \cdot e \mid \textit{fix } x \cdot e \mid \textit{let } x = e \textit{ in } e'.$$

$\mathscr{E}[\![x]\!]\eta = \eta[\![x]\!]$

$\mathscr{E}[\![(e_1 e_2)]\!]\eta = v_1 \mathsf{E}\, F \to (v_2 \mathsf{E}\, W \to \text{wrong}, (v_1 \mid F)v_2),$ wrong
where v_i is $\mathscr{E}[\![e_i]\!]\eta \qquad (i = 1, 2).$

$\mathscr{E}[\![\textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3]\!]\eta = v_1 \mathsf{E}\, B_0 \to (v_1 \mid B_0 \to v_2, v_3),$ wrong
where v_i is $\mathscr{E}[\![e_i]\!]\eta \qquad (i = 1, 2, 3)$

$\mathscr{E}[\![\lambda x \cdot e]\!]\eta = (\lambda v \cdot \mathscr{E}[\![e]\!]\, \eta\{v/x\})$ in V

$\mathscr{E}[\![\textit{fix } x \cdot e]\!]\eta = Y(\lambda v \cdot \mathscr{E}[\![e]\!]\, \eta\{v/x\})$

$\mathscr{E}[\![\textit{let } x = e_1 \textit{ in } e_2]\!]\eta = v_1 \mathsf{E}\, W \to \text{wrong}, \mathscr{E}[\![e_2]\!]\, \eta\{v_1/x\}$
where $v_1 = \mathscr{E}[\![e_1]\!]\eta.$

A Theory of Type Polymorphism

$$e ::= x \mid (ee') \mid \textit{if } e \textit{ then } e' \textit{ else } e'' \mid \lambda x \cdot e \mid \textit{fix } x \cdot e \mid \textit{let } x = e \textit{ in } e'.$$

$$\mathscr{E}[\![x]\!]\eta = \eta[\![x]\!]$$

$$\mathscr{E}[\![(e_1 e_2)]\!]\eta = v_1 \,\mathsf{E}\, F \to (v_2 \,\mathsf{E}\, W \to \text{wrong}, (v_1 \mid F)v_2), \text{wrong}$$
where v_i is $\mathscr{E}[\![e_i]\!]\eta \quad (i = 1, 2)$.

$$\mathscr{E}[\![\textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3]\!]\eta = v_1 \,\mathsf{E}\, B_0 \to (v_1 \mid B_0 \to v_2, v_3), \text{wrong}$$
where v_i is $\mathscr{E}[\![e_i]\!]\eta \quad (i = 1, 2, 3)$

$$\mathscr{E}[\![\lambda x \cdot e]\!]\eta = (\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\}) \text{ in } V$$

$$\mathscr{E}[\![\textit{fix } x \cdot e]\!]\eta = Y(\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\})$$

$$\mathscr{E}[\![\textit{let } x = e_1 \textit{ in } e_2]\!]\eta = v_1 \,\mathsf{E}\, W \to \text{wrong}, \mathscr{E}[\![e_2]\!]\,\eta\{v_1/x\}$$
where $v_1 = \mathscr{E}[\![e_1]\!]\eta$.

Starts with a Dynamic Language!

A Theory of Type Polymorphism

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 $\models \rho : \Gamma$ and $\Gamma \vdash t : T$ then $\models \hat{\rho}(t) : T$.

A Theory of Type Polymorphism

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 $\models \rho : \Gamma$ and $\Gamma \vdash t : T$ then $\models \hat{\rho}(t) : T$.

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

A Theory of Type Polymorphism

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 $\models \rho : \Gamma$ and $\Gamma \vdash t : T$ then $\models \hat{\rho}(t) : T$.

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

"Proof Theory"

BNF on steroids

A Theory of Type Polymorphism

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 $\models \rho : \Gamma$ and $\Gamma \vdash t : T$ then $\models \hat{\rho}(t) : T$.

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

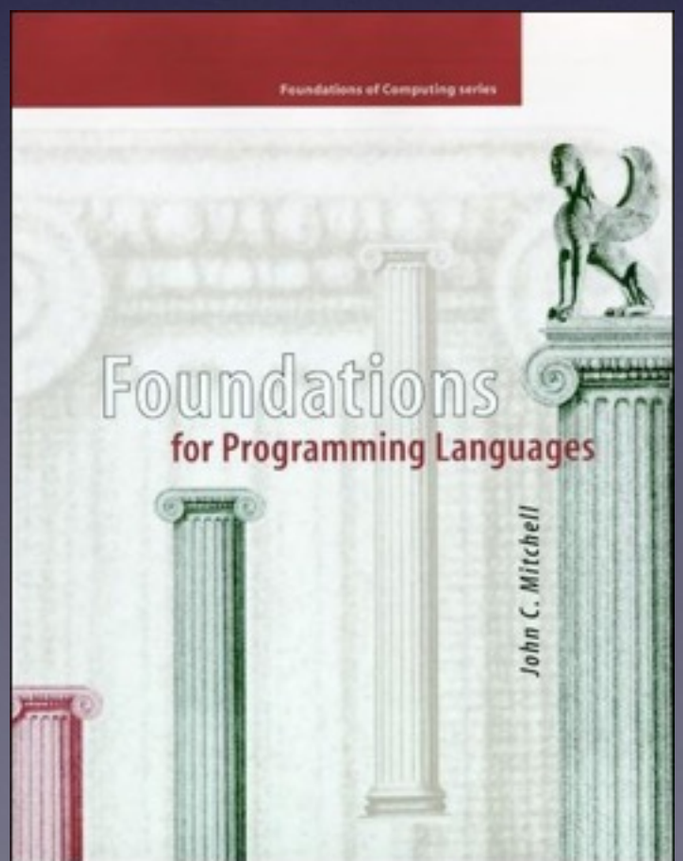
Formal claim about Execution

"Model Theory"

An important aspect of adopting a typed theory of programming languages is that untyped languages arise naturally as a special case. The main idea behind studying untyped languages in a typed framework is that we could use a type *untyped* for all the untyped expressions of a language. A specific example can be given using the syntax of the programming language ML (Mil85a), MTH90, MT91, Ull94]. If we have an untyped language with integers, booleans pairs and functions, we can represent the associated "untyped value space" using the following ML datatype:

```
datatype untyped = N of int | B of bool | P of untyped*untyped
                 | F of untyped → untyped
```

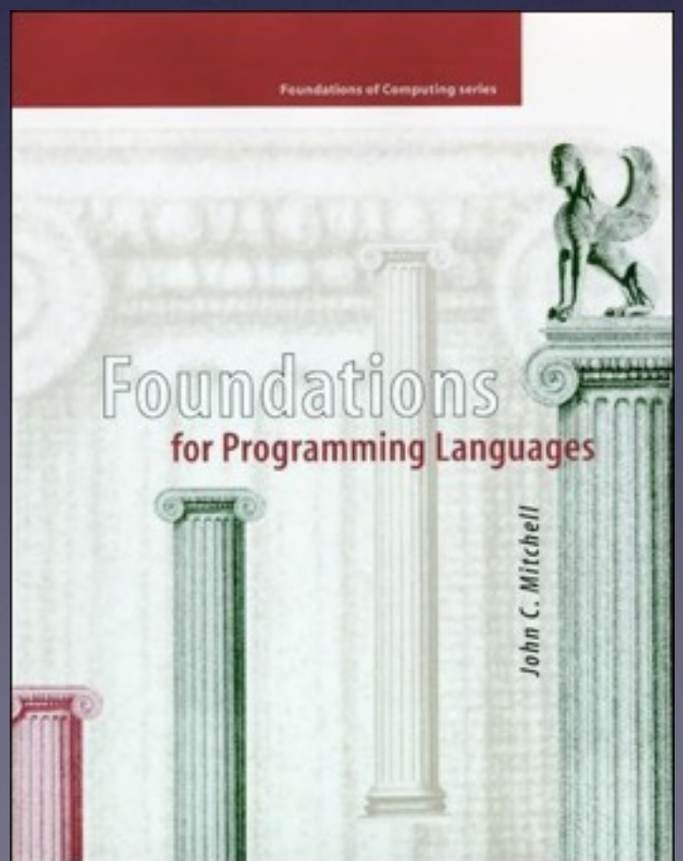
1996



John C. Mitchell

An important aspect of adopting a typed theory of programming languages is that untyped languages arise naturally as a special case. The main idea behind studying untyped languages in a typed framework is that we could use a type *untyped* for all the untyped expressions of a language. A specific example can be given using the syntax of the programming language ML (Mil85a), MTH90, MT91, Ull94]. If we have an untyped language with integers, booleans pairs and functions, we can represent the associated "untyped value space" using the following ML datatype:

```
datatype untyped = N of int | B of bool | P of untyped*untyped
                 | F of untyped → untyped
```



1996

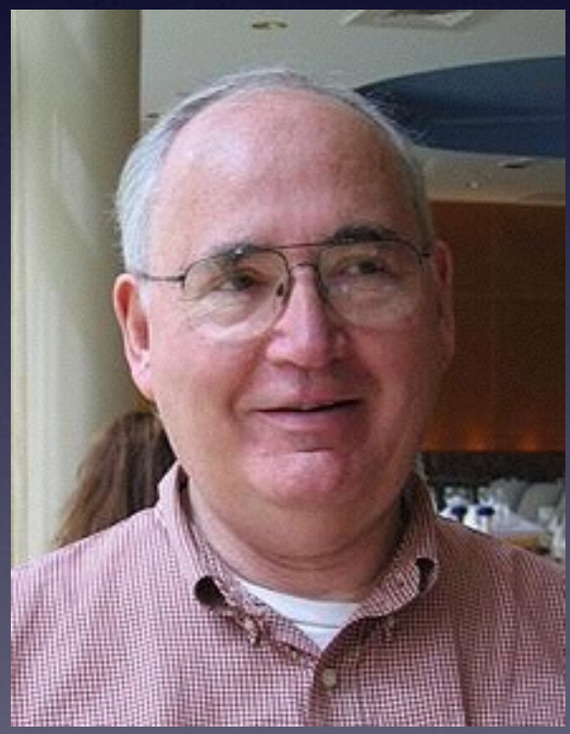
John C. Mitchell

A type-theoretical alternative to ISWIM, CUCH, OWHY

Dana S. Scott

Carnegie-Mellon University, Pittsburgh, PA, USA, and RISC-Linz, Austria

1993 (née 1969)



The paper (first written in 1969 and circulated privately) concerns the definition, axiomatization, and applications of the hereditarily monotone and continuous functionals generated from the integers and the Booleans (plus "undefined" elements). The system is formulated as a typed system of combinators (or as a typed λ-calculus) with a recursion operator (the least fixed-point operator), and its proof rules are contrasted to a certain extent with those of the untyped λ-calculus. For publication (1993), a new preface has been added, and many bibliographical references and comments in footnotes have been appended.

What do types mean? — From intrinsic to extrinsic semantics

John C. Reynolds

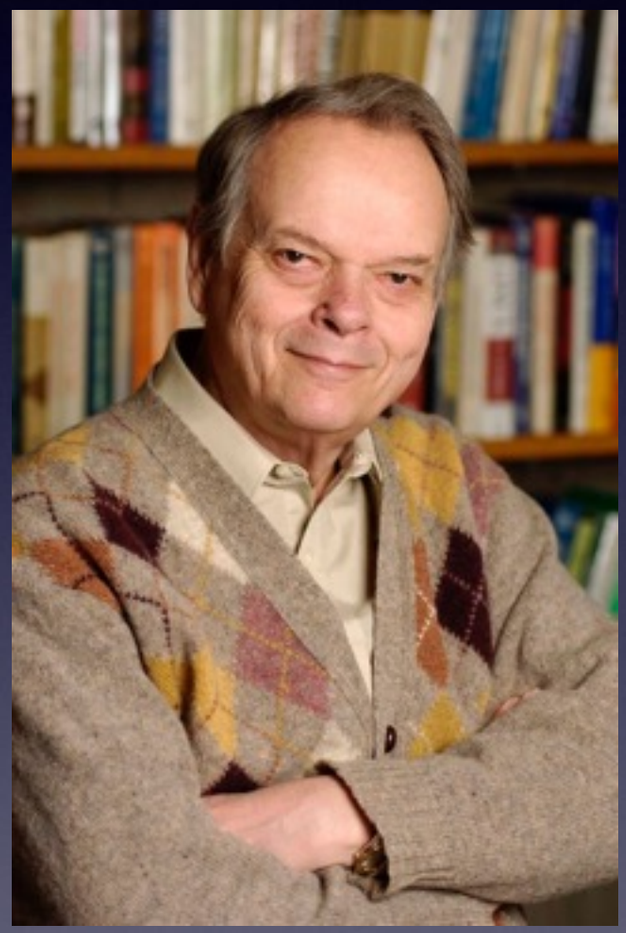
2003

$[\![\mathcal{D} :: \Gamma \vdash t : T]\!]$

Intrinsic Semantics

$[\![t]\!]$

Extrinsic Semantics



2: Mixing Static and Dynamic Checking

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

classes of program expressions that commonly occur in dynamically typed programs that do not type-check in the ML type system.

Extend Milner 1978 with:

$$T = V \mid \mathbf{c}(\ldots) \mid T + T \mid \mathtt{fix}\ x.T$$

constructor types

bare union types

equi-recursive types

$$A \vdash \tau_1 \subseteq \tau_1$$

union-induced subtyping

Soft Typing

Program

Type-Directed
Coercion Insertion

Dynamic Program

1991:
Extrinsic,
Denotational

$$\Downarrow_T^S (v) = \begin{cases} v & v \in T \\ \texttt{fault} & v \in S - T \\ \texttt{wrong} & \text{otherwise} \end{cases}$$

1997:
Intrinsic
Operational

Quasi-static Typing

(Preliminary Report)

Satish R. Thatte*
Department of Mathematics and Computer Science
Clarkson University, Potsdam, NY 13676.

POPL 1990

Given the undecidability of "semantically complete" typechecking for most languages, a system that permits ill-typing proofs has to make a three way division of programs into well-typed, ill-typed, and ambivalent ones.

Quasi-static Typing

"Lack of Information" Type

$$T \leq \Omega$$

All types are a subtypes of the Omega type

$\tau \leq \Omega.$

Quasi-static Typing

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_3 \qquad T_1 <: T_3}{\Gamma \vdash t_1\ t_2 : T_2}$$

$$\frac{\Gamma \vdash t_1 : \Omega \qquad \Gamma \vdash t_2 : T_3}{\Gamma \vdash t_1\ t_2 : \Omega}$$

Automatic Downcasts

Quasi-static Typing

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_3 \qquad T_1 <: T_3}{\Gamma \vdash t_1\ t_2 : T_2}$$

$$\frac{\Gamma \vdash t_1 : \Omega \qquad \Gamma \vdash t_2 : T_3 \qquad [T_3 \to \Omega <: \Omega]}{\Gamma \vdash t_1\ t_2 : \Omega}$$

Automatic Downcasts

Quasi-static Typing

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_3 \qquad T_1 <: T_3}{\Gamma \vdash t_1\ t_2 : T_2}$$

$$\frac{\Gamma \vdash t_1 : \Omega \qquad \Gamma \vdash t_2 : T_3 \qquad [T_3 \to \Omega <: \Omega]}{\Gamma \vdash t_1\ t_2 : \Omega}$$

Automatic Downcasts

(Nearly) Every Program Type Checks

Quasi-Static Typing

Program

Type-Directed
Coercion Insertion

Dynamic Program

Plausibility
Checking

Quasi-Static Program

Type Error!

Hybrid Type Checking

KENNETH KNOWLES CORMAC FLANAGAN
University of California at Santa Cruz

POPL 2006, TOPLAS 2009

Ill-typed programs			**Well-typed programs**
Clearly ill-typed	Subtle programs		Clearly well-typed
Rejected by type checker	Accepted with casts		Accepted without casts
	Casts may fail	Casts never fail	

Hybrid Type Checking

$$S, T ::=$$
$$x : S \to T$$
$$\{x : B \mid t\}$$

First-order Subset Types

First-Order Type

Boolean Term

Hybrid Type Checking

$$
\begin{aligned}
S, T ::= \\
& x : S \to T \\
& \{x : B \mid t\}
\end{aligned}
$$

Dependent Function Types

Hybrid Type Checking

Subtyping $\boxed{E \vdash S <: T}$

$$\frac{E \vdash T_1 <: S_1 \qquad E, x : T_1 \vdash S_2 <: T_2}{E \vdash (x{:}S_1 \to S_2) <: (x{:}T_1 \to T_2)} \quad \text{[S-Arrow]}$$

$$\frac{E, x : B \vdash s \Rightarrow t}{E \vdash \{x{:}B \mid s\} <: \{x{:}B \mid t\}} \quad \text{[S-Base]}$$

Boolean Term Entailment

Undecidable Subtyping Judgment

Hybrid Type Checking

3-valued Subtyping Procedure

$$E \vdash^{a}_{alg} S <: T$$

$$a \in \{\surd, \times, ?\}$$

Yes

No

Maybe

Hybrid Type Checking

Program

Type-Directed
Coercion Insertion

Hybrid Program

$$\frac{E \vdash s \hookrightarrow t : S \qquad E \vdash^{?}_{alg} S <: T}{E \vdash s \hookrightarrow \langle T \triangleleft S \rangle\ t \downarrow T}$$

Gradual Typing for Functional Languages

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

Walid Taha
Rice University
taha@rice.edu

Gradual Typing for Functional Languages

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

Walid Taha
Rice University
taha@rice.edu

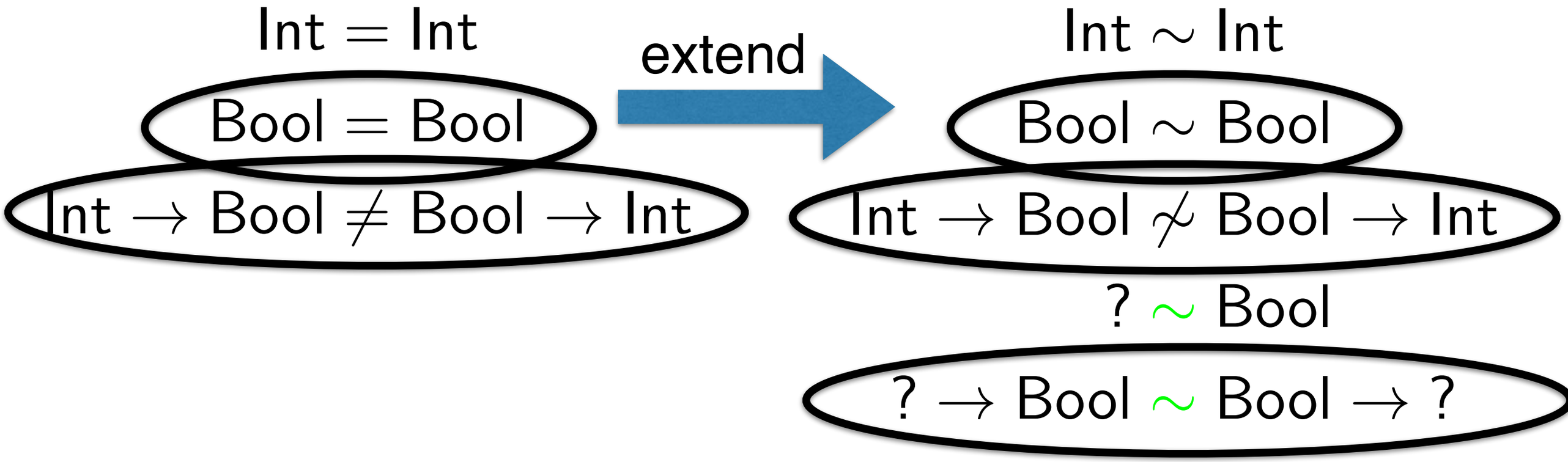
Static Types $T ::= B \mid T \to T$

Gradual Types $\widetilde{T} ::= B \mid ? \mid \widetilde{T} \to \widetilde{T}$

The “Unknown” Type

Gradual Typing

static type equality **gradual type *consistency***



Consistency conservatively extends equality

Gradual Typing

replace equality with consistency

$$\frac{\Gamma \vdash t_1 : \mathsf{Int} \quad \Gamma \vdash t_2 : \mathsf{Int}}{\Gamma \vdash t_1 + t_2 : \mathsf{Int}}$$

$$\frac{\Gamma \vdash t_1 : \widetilde{T_1} \quad \widetilde{T_1} \sim \mathsf{Int} \quad \Gamma \vdash t_2 : \widetilde{T_2} \quad \widetilde{T_2} \sim \mathsf{Int}}{\Gamma \vdash t_1 + t_2 : \mathsf{Int}}$$

Dynamic Checking

Static Semantics

Gradual Language

Type-Directed Translation

Dynamic Semantics

Cast Language

Many Many More!

Synthesis

- Intrinsic vs. Extrinsic Typing
- Accept All Programs vs. Reject Ill-Typed
- Type-directed Translation is Ubiquitous
- Subtyping vs. Consistency
- Dynamic, Simple, Refinement Typing
  - It’s all relative!

3: Gradual Information-Flow Typing

Operating Systems

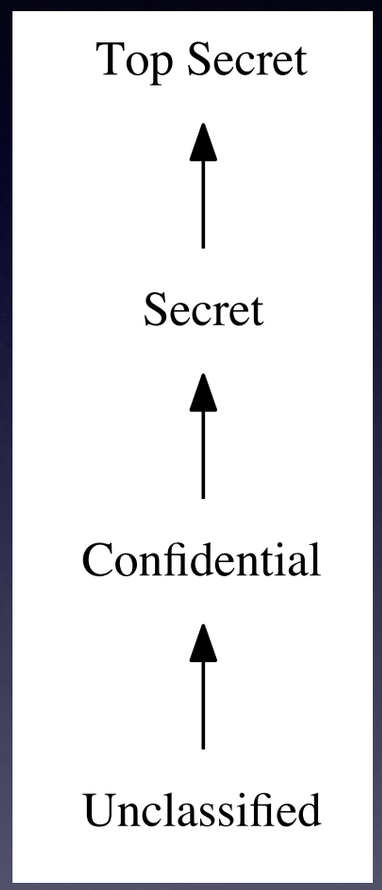
R.S. Gaines Editor

A Lattice Model of Secure Information Flow

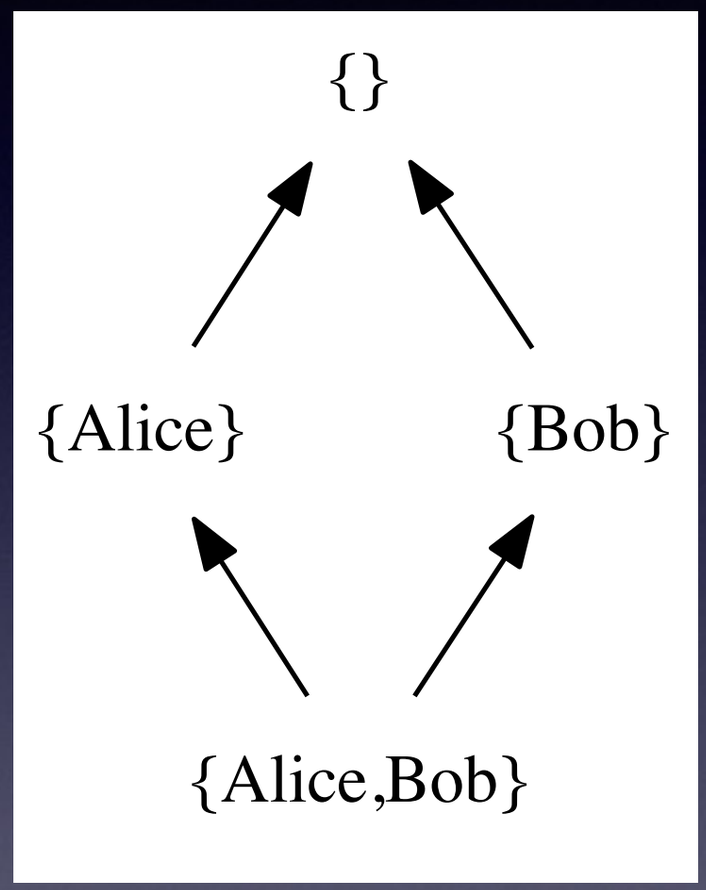
Dorothy E. Denning
Purdue University

[CACM 1976]

Security as a Lattice



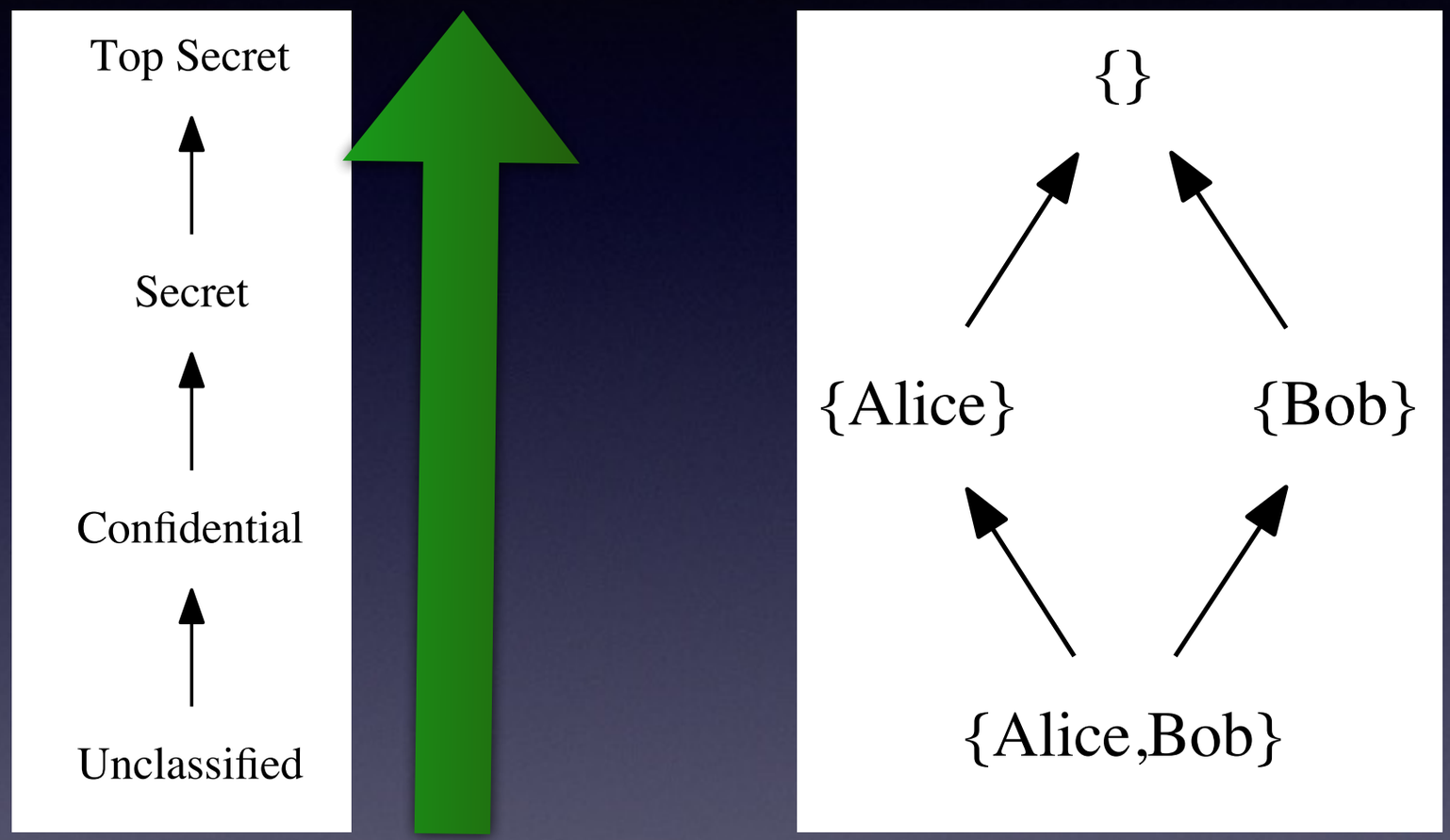
Classification



Permitted Readers

Zdancewic. “Programming Languages for Information Security”

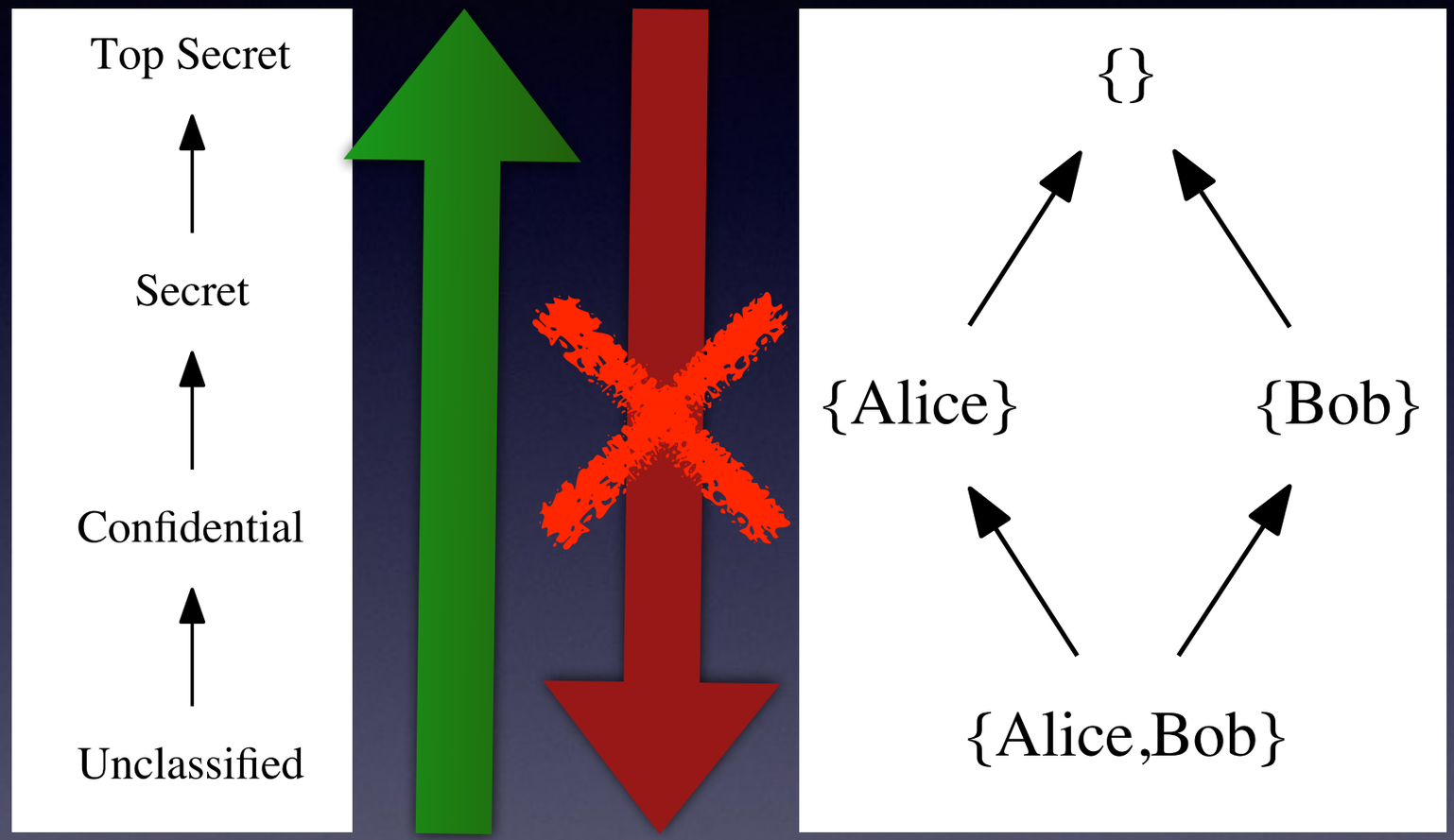
Security as a Lattice



Low-security information may flow
to high-security contexts

Zdancewic. “Programming Languages for Information Security”

Security as a Lattice



High-security information may not flow to low-security contexts

Zdancewic. “Programming Languages for Information Security”

Example

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

Disney and Flanagan. “Gradual Information Flow Typing”

Example

Low Security Data

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

Disney and Flanagan. “Gradual Information Flow Typing”

Example

Low Security Data

High Security Data

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

Disney and Flanagan. “Gradual Information Flow Typing”

Example

Low Security Data

High Security Data

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

Low Security Channel

Disney and Flanagan. “Gradual Information Flow Typing”

Example

Low Security Data

High Security Data

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

Low Security Channel

Security Leak!!

Unchecked Semantic Error

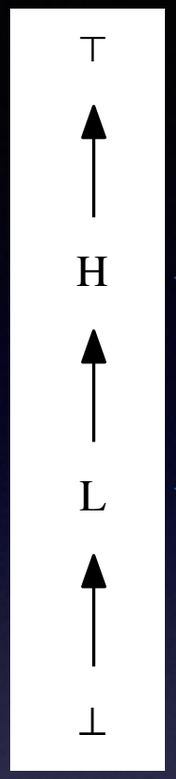
Disney and Flanagan. “Gradual Information Flow Typing”

Security Typing

$$\mathrm{Int} \longrightarrow \mathrm{Int}$$

Simple Type Structure

Security Typing



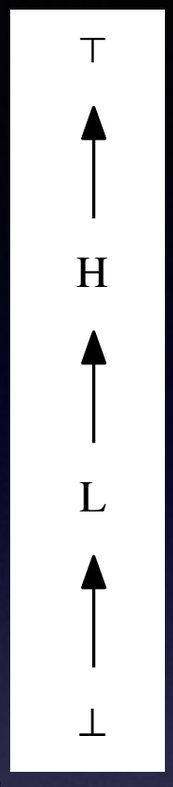
Higher Security

Lower Security

$\text{Int}_H \longrightarrow_L \text{Int}_H$

Security-Indexed Type Constructors

Security Typing



Higher Security

Lower Security

$$\mathsf{Int}_L <: \mathsf{Int}_H$$

$$\mathsf{Int}_H \longrightarrow_L \mathsf{Int}_L \;<:\; \mathsf{Int}_L \longrightarrow_H \mathsf{Int}_H$$

Induced Subtyping Structure

Example

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

Security Leak!!

Unchecked Semantic Error

Disney and Flanagan. “Gradual Information Flow Typing”

Example

```
let age : IntL = 31
let salary : IntH = 58000
let intToString : IntL →L StringL = ...
let print : StringL →L UnitL = ...
print(intToString(salary))
```

Type Error

Checked Semantic Error

Disney and Flanagan. “Gradual Information Flow Typing”

Example

```
let age : IntL = 31
let salary : IntH = 58000
let intToString : IntL →L StringL = ...
let print : StringL →L UnitL = ...
print(intToString(age))
```

Security Typed

Disney and Flanagan. “Gradual Information Flow Typing”

Implicit Security Flows

- High-Security information can affect the **control flow** of a program, and thereby leak information.
- Upgrade the security of the results of high-value computations.

Implicit Security Flows

```
fun b : BoolH =>
  let tt : BoolL = true
  let ff : BoolL = false
  if b then tt else ff
```

High-Security **data** can affect **control** flow of a program

Effects Can Leak Information

- High-Security values can flow from a computation via mutable reference cells

Security Typing

$$\mathsf{Int}_H \longrightarrow_L \mathsf{Int}_H$$

$$\Gamma; \Sigma \vdash t : S$$

Security Typing

$$\mathsf{Int}_H \xrightarrow{L}_L \mathsf{Int}_H$$

$$\Gamma; \Sigma; \ell \vdash t : S$$

Latent Security Effects

Security Typing Gap

Simple Typing

Security Typing

Security Typing Gap

Simple Typing

+

Dynamic Information-Flow

Security Typing

Gradual Typing!

Simple Typing

Gradual Information Flow Typing

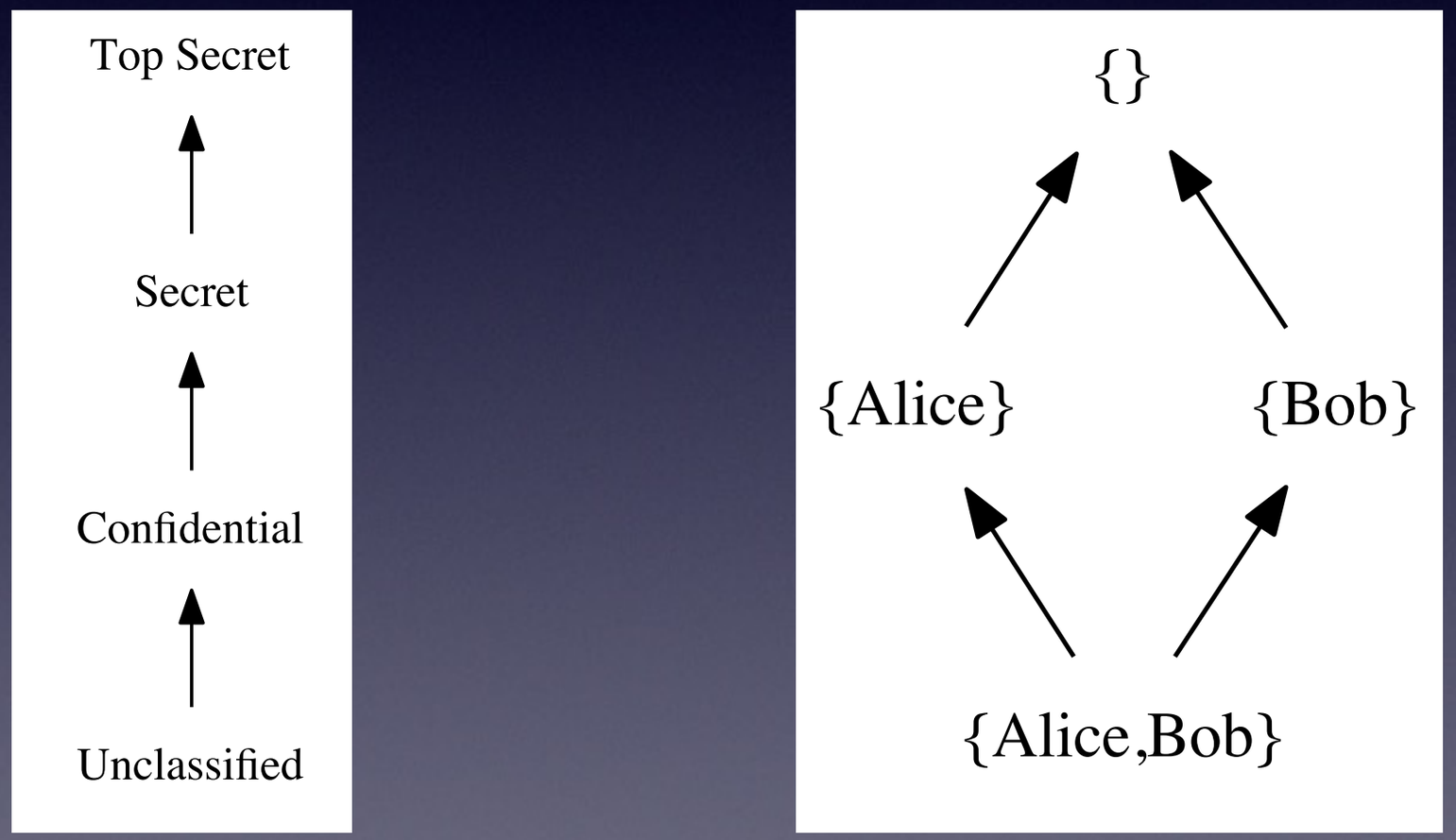
Tim Disney and Cormac Flanagan

University of California Santa Cruz

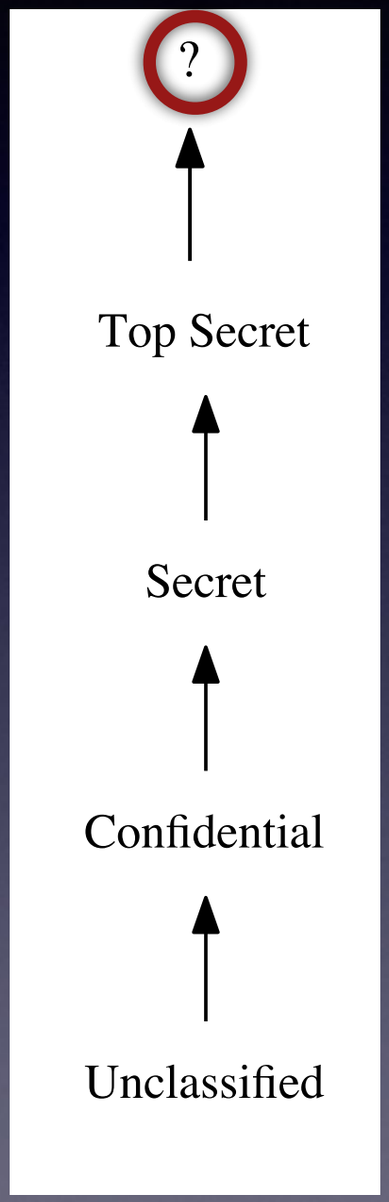
Security Typing

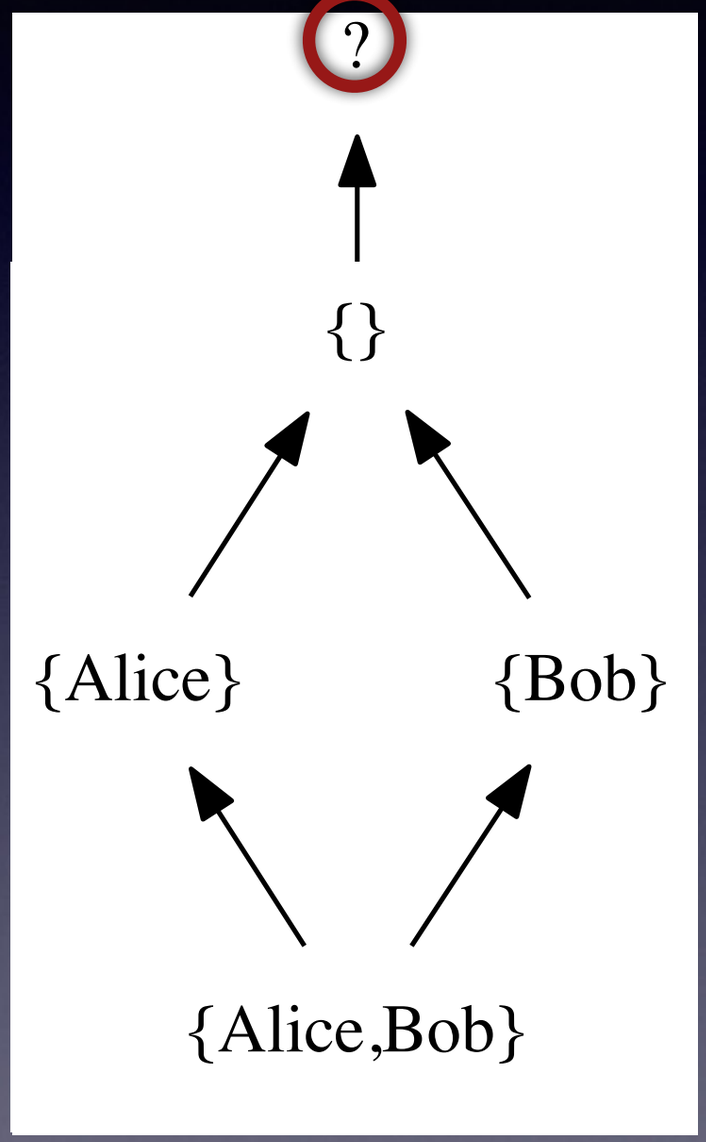
[STOP 2011]

Security Lattice



Security Lattice





Dynamic Security as “Top”

Example

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

Disney and Flanagan. “Gradual Information Flow Typing”

“Dynamic” Program

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

Disney and Flanagan. “Gradual Information Flow Typing”

“Gradual” Program

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

Disney and Flanagan. “Gradual Information Flow Typing”

“Gradual” Program

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

Static Security Labels

Disney and Flanagan. “Gradual Information Flow Typing”

“Gradual” Program

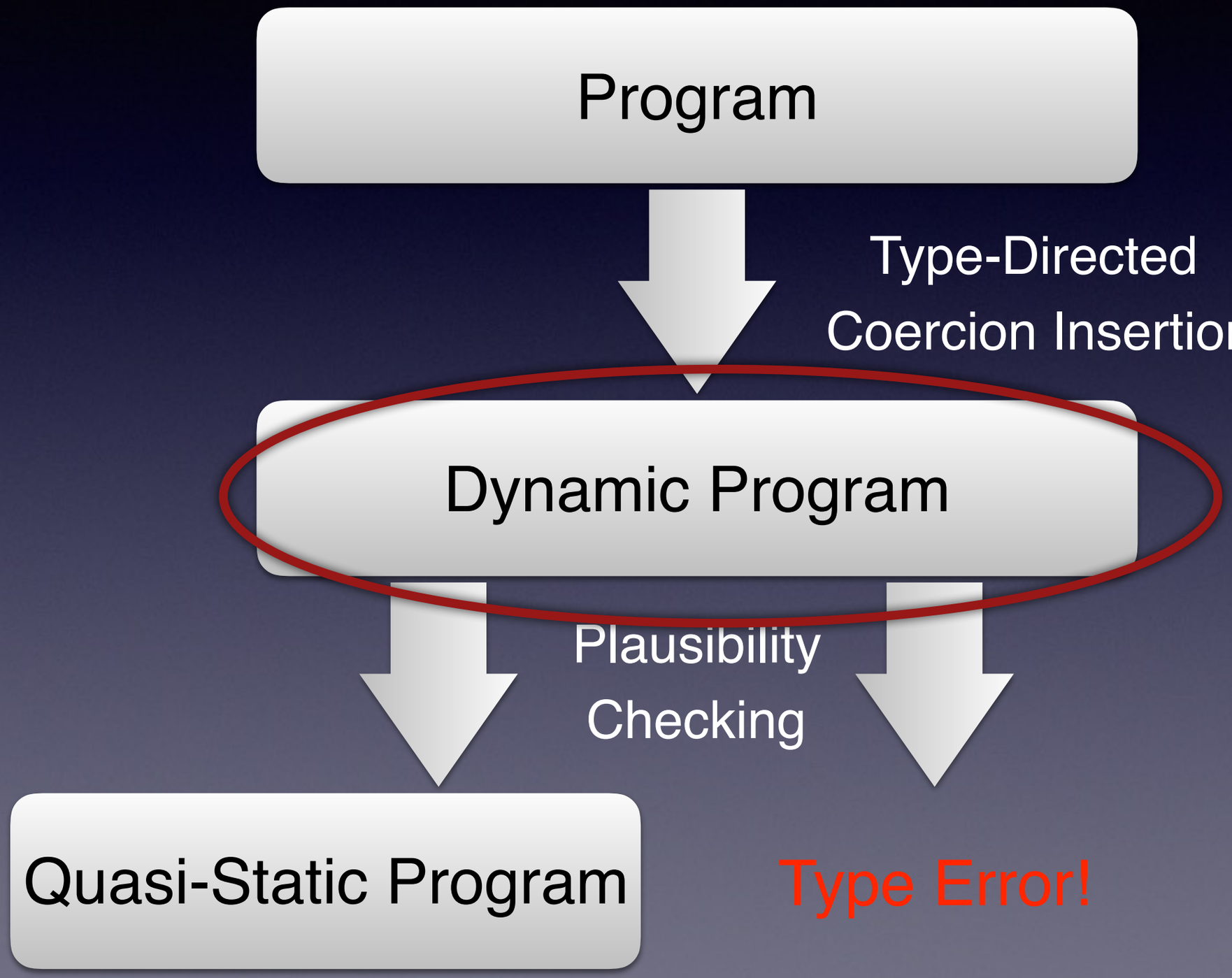
```
let age : Int? = 31
let salary : IntH = 58000
let intToString : Int? →? String? = ...
let print : StringL →? Unit? = ...
print((StringL)intToString(salary))
```

Explicit Runtime Downcasts

closer to “Quasi-static” than “Gradual”

Disney and Flanagan. “Gradual Information Flow Typing”

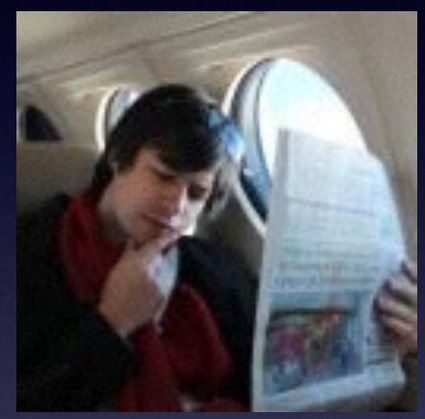
Quasi-Static Style



Gradual Information Flow

Me

Éric Tanter



Matías Toro

(A Whirlwind Tour)

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

Gradual Security

$$\ell \in \textsc{Label}$$

$$\widetilde{\ell} \in \textsc{GLabel} ::= \ell \mid ?$$

Unknown Label

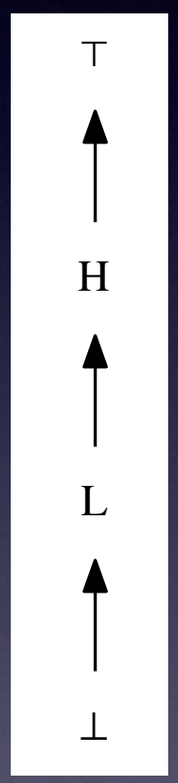
Interpret Gradual Labels

$$\gamma : \textsc{GLabel} \to \mathcal{P}(\textsc{Label})$$

$$\gamma(L) = \{L\}$$

$$\gamma(H) = \{H\}$$

$$\gamma(?) = \textsc{Label}$$



"Concretization"

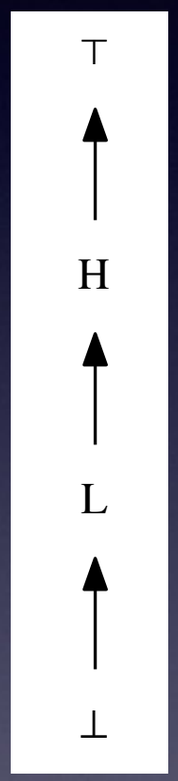
Interpret Gradual Labels

$$\gamma : \textsc{GLabel} \to \mathcal{P}(\textsc{Label})$$

$$\gamma(L) = L$$

$$\gamma(H) = H$$

$$\gamma(?) = \textsc{Label}$$



$$\alpha : \mathcal{P}(\textsc{Label}) \rightharpoonup \textsc{GLabel}$$

$$\alpha(\{L\}) = L$$

$$\alpha(\{L, H\}) = ?$$

"Abstraction"

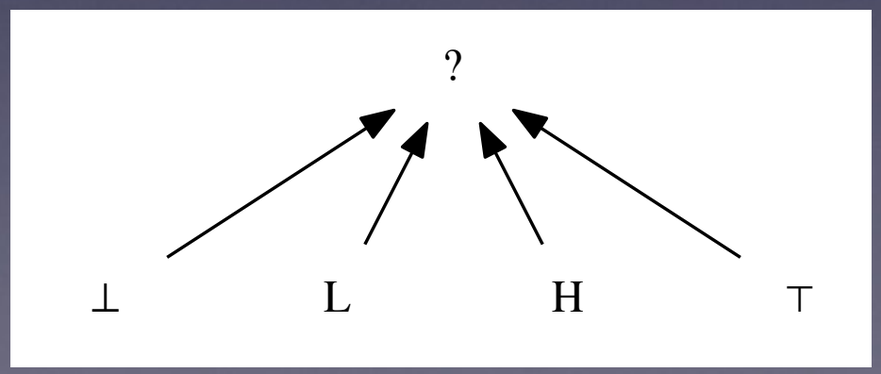
Gradual Label Precision

$$\widetilde{l} \sqsubseteq \widetilde{l}$$

Gradual Label Precision

$$\widetilde{l} \sqsubseteq \widetilde{l}$$

$$\gamma(\widetilde{l_1}) \subseteq \gamma(\widetilde{l_2})$$



Consistent Ordering

$$\tilde{l} \gtrsim \tilde{l}$$

$$l_1 \preccurlyeq l_2 \text{ for some } l_1 \in \gamma(\widetilde{l_1}), l_1 \in \gamma(\widetilde{l_1})$$

Consistent “Ordering”

$$L \precsim H$$
$$L \not\precsim H$$
$$L \precsim L$$
$$? \precsim L$$
$$L \precsim ?$$

Not really an order

Lifted Join

$$\widetilde{l} \ \widetilde{\gamma} \ \widetilde{l}$$

$$\alpha(\gamma(\widetilde{l_1}) \ \gamma^* \ \gamma(\widetilde{l_2}))$$

Lifted Join

$$L \mathbin{\widetilde{\gamma}} H = H$$

$$H \mathbin{\widetilde{\gamma}} H = H$$

$$? \mathbin{\widetilde{\gamma}} H = ?$$

Interpret Gradual Types

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

$$\gamma(\mathsf{Int}_?) = \{\mathsf{Int}_\bot, \mathsf{Int}_L, \mathsf{Int}_H, \mathsf{Int}_\top\}$$

"Concretization"

Interpret Gradual Types

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

$$\gamma(\mathsf{Int}_?) = \{\mathsf{Int}_\bot, \mathsf{Int}_L, \mathsf{Int}_H, \mathsf{Int}_\top\}$$

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

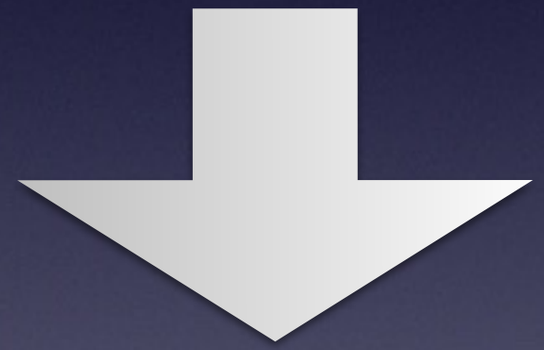
$$\alpha(\{\mathsf{Int}_L\}) = \mathsf{Int}_L$$

$$\alpha(\{\mathsf{Int}_L, \mathsf{Int}_H\}) = \mathsf{Int}_?$$

"Abstraction"

Lifting Judgments

Subtyping $T <: T$



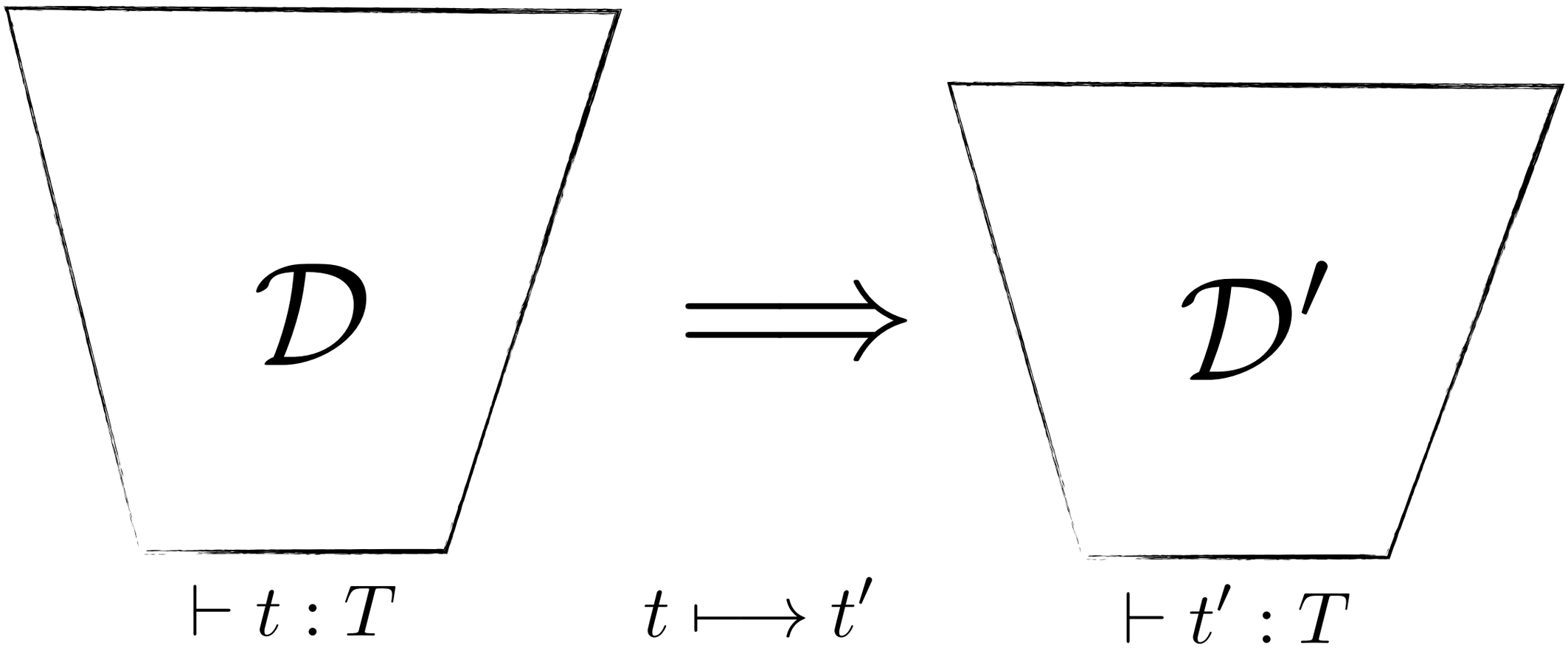
Consistent Subtyping $\tilde{T} \lesssim \tilde{T}$

Lifting Typing Rules

$$\frac{\Gamma;\Sigma;\ell_c \vdash t_1 : T_{11} \xrightarrow{\ell'}_{\ell_x} T_{12} \qquad \Gamma;\Sigma;\ell_c \vdash t_2 : T_2 \qquad S_2 <: S_{11} \qquad \ell_c \curlyvee \ell_x \preccurlyeq \ell'}{\Gamma;\Sigma;\ell_c \vdash t_1\ t_2 : T_{12} \curlyvee \ell_x}$$

$$\frac{\Gamma;\Sigma;\widetilde{\ell}_c \vdash \widetilde{t}_1 : \widetilde{T}_{11} \xrightarrow{\widetilde{\ell}'}_{\ell_x} T_{12} \qquad \Gamma;\underbrace{\Sigma;\widetilde{\ell}_c} \vdash \widetilde{t}_2 : \widetilde{T}_2 \qquad \widetilde{T}_2 \lesssim \widetilde{T}_{11} \qquad \widetilde{\ell}_c \curlyvee \widetilde{\ell}_x \sqsubseteq \widetilde{\ell}'}{\Gamma;\Sigma;\widetilde{\ell}_c \vdash \widetilde{t}_1\ \widetilde{t}_2 : \widetilde{T}_{12} \mathbin{\widetilde{\curlyvee}} \widetilde{\ell}_x}$$

Lift Type Safety Argument



Preservation Theorem

Gradual Program

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

No Casts Necessary!

Disney and Flanagan. “Gradual Information Flow Typing”

Conclusion

Hybrid Type Checking

Dynamic typing:

Quasi-static Typing

Soft Typing

Gradual Typing

Conclusion

Hybrid Type Checking

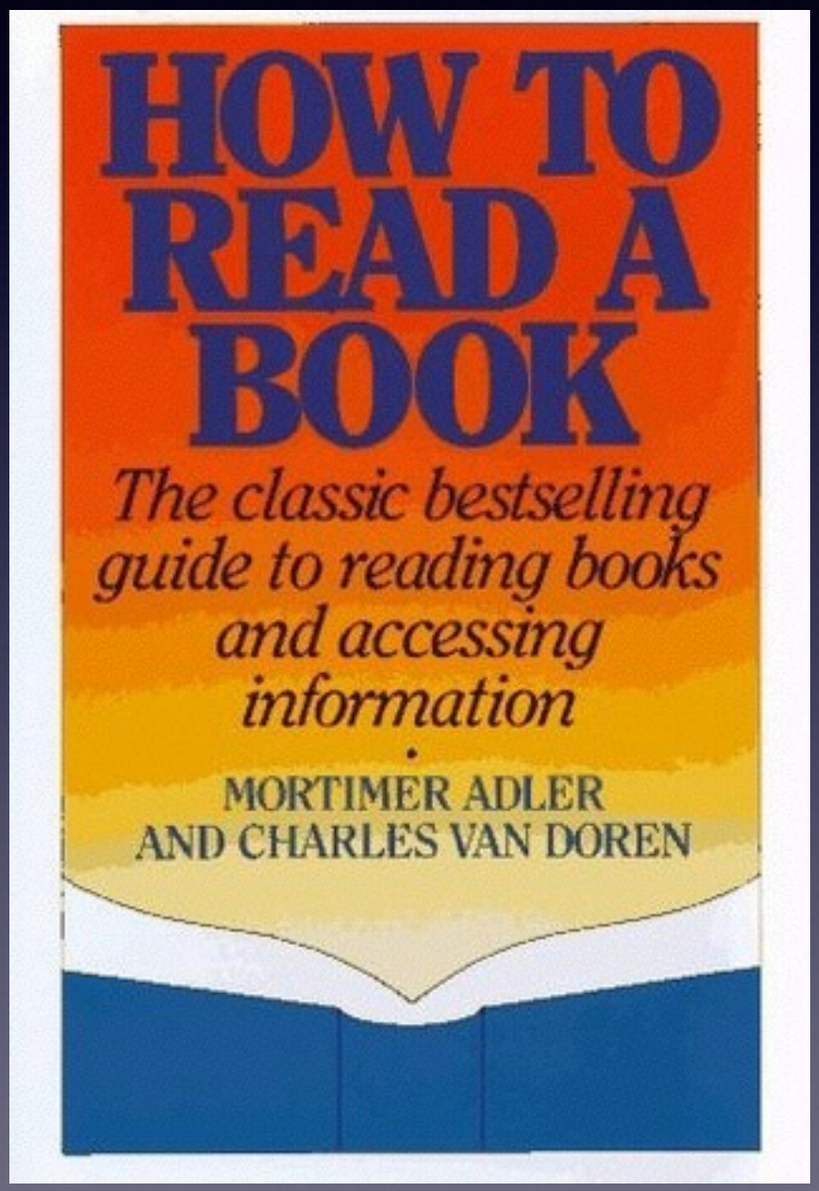
Dynamic typing:

Quasi-static Typing

Soft Typing

Gradual Typing

Synopsis



"…it is in a sense true that the identification of the subject matter must follow the reading, not precede it."