

how do i do research

matthias felleisen.

racketeer.

plt. northeastern

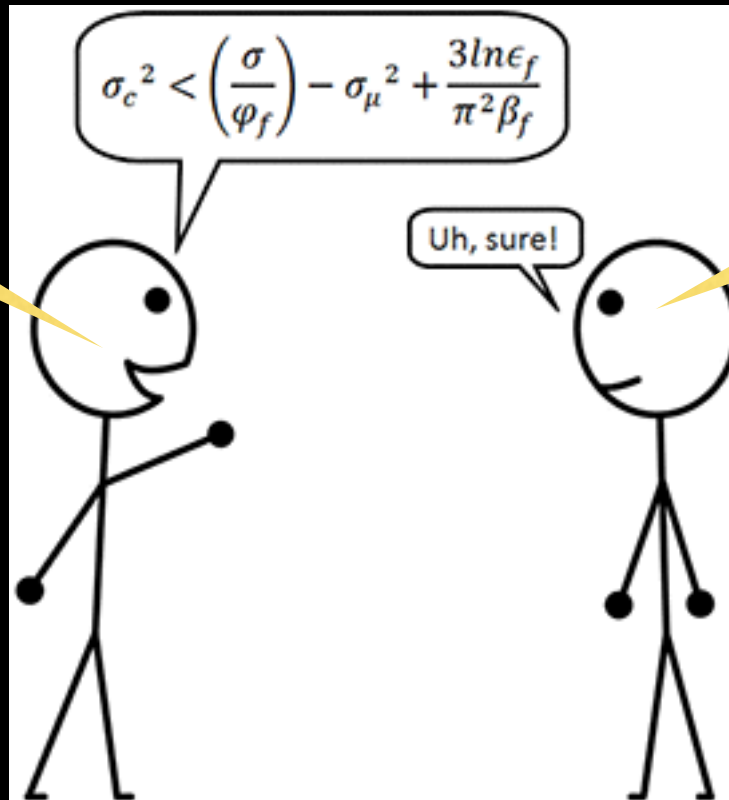
how do I do research?

how ~~do~~ I do research?

would

if I were you

your
advisor



& you

how do *I* work with my
my PhD students?

how would I do
research if I were you?

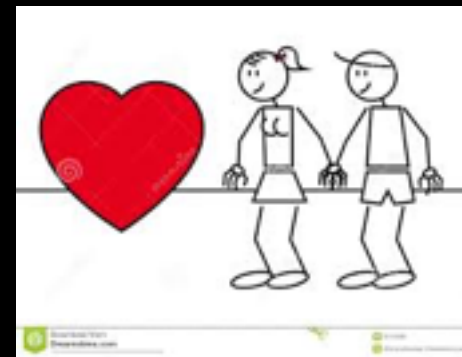
how do *I* do research?

**how do i
relate to my
PhD students**

I have *never, ever*
hired a PhD student.
Period.



Instead my students and I
find a topic we both love.



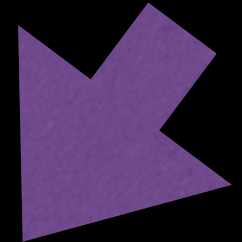
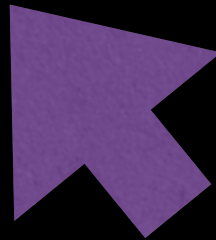
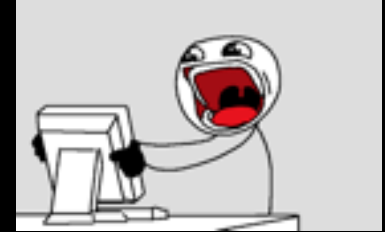
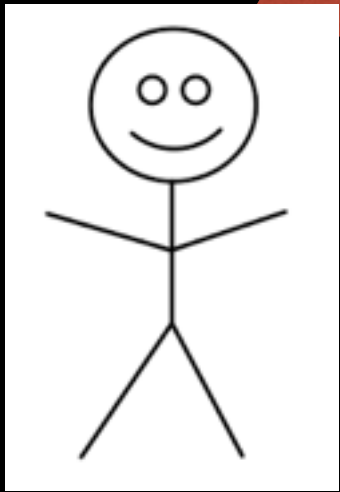
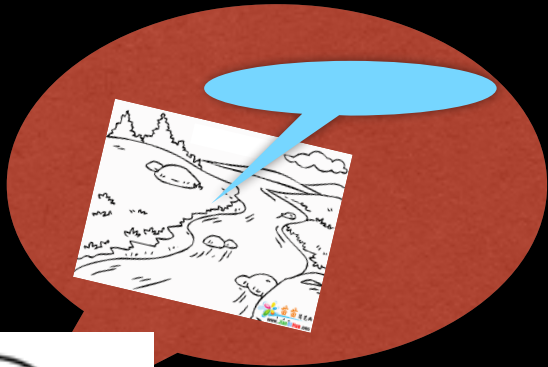
Functional I/O (ICFP '09)

Typed Racket
(ICFP '10)



Compiler
Coaching (OOPSLA '12)

Laziness, what is it (good for)?
(JFP 1996)



And that's what's called 'doing research.'

**how would I
do research if
I were you**

Two Case Studies



Asumu
Takikawa



Tony
Garnock-Jones

Kuhn, *The Structure of Scientific Revolution*

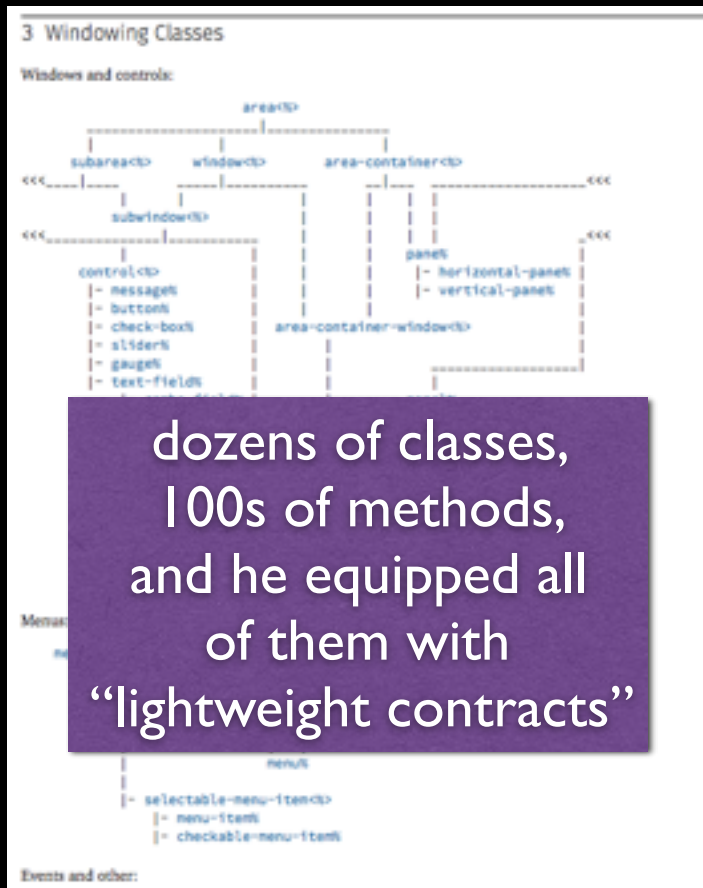
Types for Classes



Typed Racket
(ICFP '10)



Topic: *Gradual* Types for *First-class* Classes



Is Sound Gradual Typing Dead?
Programming Objects with ML-ART
An extension to ML with Abstract and Record Types
Didier Rémy

INFORMATION AND COMPUTATION 93, 1-15 (1991)

Type Inference for Record Concatenation and Multiple Inheritance*
MITCHELL WAND

Complete Type Inference for Simple Objects

Mitchell Wand
College of Computer Science
Northeastern University
300 Huntington Avenue, 051CN
Boston, MA 02115, USA

Abstract
We consider the problem of strong typing for a model of object-oriented programming systems. These systems permit values which are records of other values, and in which fields inside these records are retrieved by name. We propose a type system, which allows us to classify these kinds of values and to classify programs by the type of their result, as is usual in strongly-typed programming languages. Our type system has two important properties: it admits multiple inheritance, and it has a syntactically complete type inference system.

The function *somehow* should be applicable to both cars and submarines. We can think of cars and submarines as inheriting from movable objects. This model also permits multiple inheritance: a submarine is both a movable object and a weapons system, because any function applicable to a weapons system will be applicable to a submarine.

Cardelli [Cardelli 84] has proposed a type system (which we call C84) that accounts for inheritance of this sort. He proved the soundness of a semantics for this system. Unfortunately, C84 sacrifices a useful property of the simply-typed lambda-calculus (as exemplified by the ML system [Gordon et al. 78]): the solvability of the type inference problem. That is, we would like to

1. Introduction

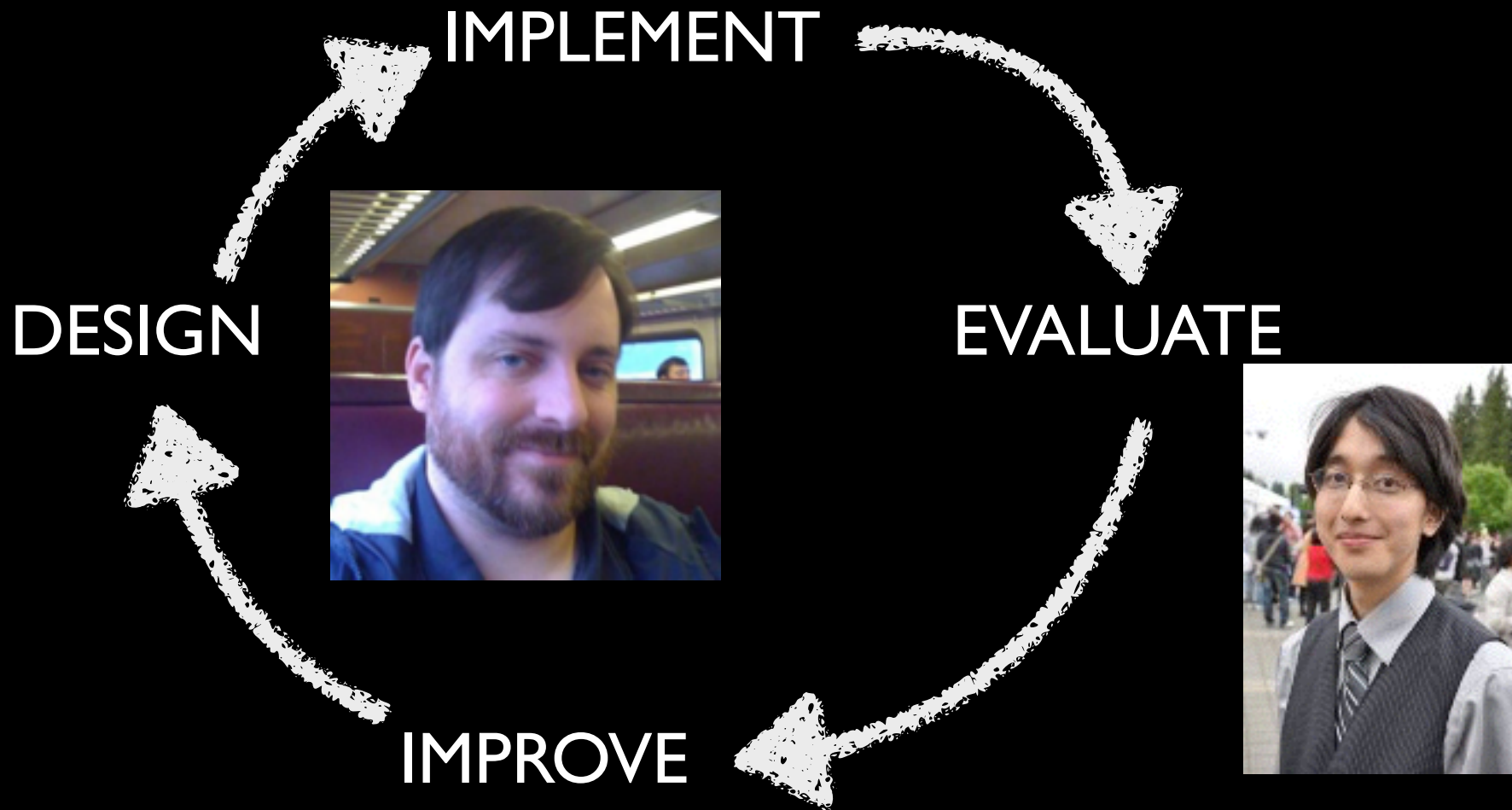
here
more
s. to
ugh,
think
light-

Es-
nd
il-
in
10
s.
10
s.
10

ds,
lus
is,
pe
the
ng
us,
en
NE

tional

Previous Topic: Contracts for Classes & Objects



Takikawa & Greenman '15

Is Sound Gradual Typing Dead?

Dr. Double B. Reviewing, I
In Famous University
turing@award.com

Dr. Double B. Reviewing, II
In Famous University
turing@award.com

Dr. Double B. Reviewing, III
Somewhat Famous University
turing@award.com

Dr. Double B. Reviewing, V
In Famous University
turing@award.com

Dr. Double B. Reviewing, VI
Less Famous University
turing@award.com

EVALUATE

Towards Practical Gradual Typing

Asumu Takikawa¹, Daniel Feltey¹, Earl Dean
Robert Bruce Findler², Sam Tobin-Hochstadt
Felleisen³

- 1 Northeastern University
Boston, Massachusetts
asumu@ccs.neu.edu, dfeltey@ccs.neu.edu, matthias
- 2 Indiana University
Bloomington, Indiana
asath@cs.indiana.edu, edean@cs.indiana.edu
- 3 University of Utah
Salt Lake City, Utah
mflatt@cs.utah.edu
- 4 Northw...
Evanston, Illinois
robby@cs...

Abstract

Programmers have come to embrace dynamically-typed languages for prototyping and delivering large and complex systems. When it comes to maintaining and evolving these systems, the lack of explicit static typing becomes a bottleneck. In response, researchers have explored the idea of gradually-typed programming languages which allow the post-hoc addition of type annotations to software written in one of these untyped languages. Some of these new...

1. Gradual Typing and Performance

Over the past couple of decades dynamically-typed languages have become a staple of the software engineering world. Programmers use these languages to build all kinds of software systems. In many cases, the systems start as innocent prototypes. Soon enough, though, they grow into complex, multi-module programs, at which point the engineers realize that they are facing a maintenance nightmare, mostly due to the lack of reliable type information.

IMPLEMENT

Grad

Asumu T

Abstract

Over the past 20 years, programmers have embraced dynamically-typed programming languages. By now, they have also come to realize that programs in these languages lack reliable type information for software engineering purposes. Gradual typing addresses this problem; it empowers programmers to annotate an existing system with sound type information on a piecemeal basis. This paper presents an implementation of a gradual type system for a full-featured class-based language as well as a novel performance evaluation framework for gradual typing.

1998 ACM Subject Classification D.3 Programming Languages

Gradual typing, object-oriented programming, performance evaluation

10.1145/1144590.1144599

Gradual Typing for Classes

Gradual type systems allow programmers to add type information to software systems in dynamically typed languages on an incremental basis [39, 48]. The ethos of gradual typing is that programmers can gradually add type information to existing code. For example, JavaScript [19] and Perl [31]. Formal models have validated soundness for gradual type systems, allowing seamless interoperation between sister languages [22, 27, 32].

Contracts for First-Class

T. STEPHEN STRICKLAND,
MATTHIAS FELLEISEN, Nor

Abstract

Dynamic type-checkers often go hand-in-hand with first-class classes, Ruby, and JavaScript programming. When scripts evolve into large programs, the discipline reduces maintainability. A programmer may want to migrate parts of such scripts to a static type system. Unfortunately, existing type systems do not support the flexible OO composition mechanisms found in scripting languages nor accommodate sound interoperation with untyped code.

First-class classes enable programs with new forms of object-oriented calls for tools to control the compiler that has seen much use in object cope with first-class classes. On the other hand, classes are contained within class definitions.

This paper presents the design as a two-pronged evaluation. The first part, consisting of benchmarks and case studies, demonstrates the need for the rich contract language and validates that our implementation approach is performant with respect to time.

Categories and Subject Descriptors: D.3.3 Software Engineering—Coding Tools and Techniques—Object-oriented programming; D.3.4 Program Verification—Programming by contract; I.2.3 Languages—Theory—Semantics

General Terms: Design, Languages, Verification

Additional Key Words and Phrases:

ACM Reference Format:
ACM Trans. Program. Lang. Syst. V, N, Article A (January YYYY), 57 pages.
DOI: <http://dx.doi.org/10.1145/9000000.0000000>

1. FIRST-CLASS CLASSES AND CONTRACTS

First-class classes enable the programmer to dynamically pick context-appropriate base classes, to load new classes at run-time to implement a plug-in architecture, or

DESIGN

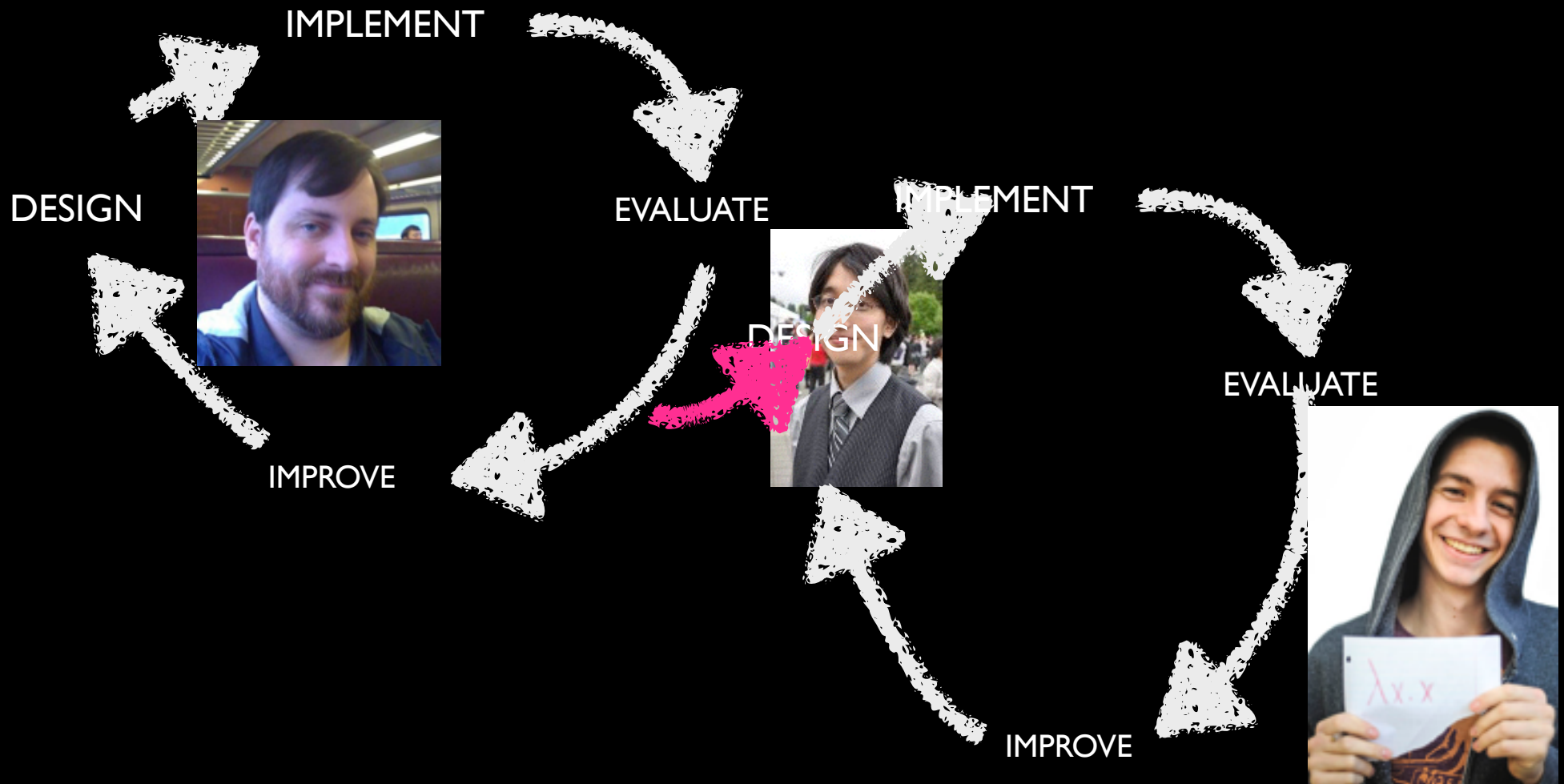
EVALUATE

Takikawa '15

Takikawa & Strickland '13

Strickland & Takikawa '12

A Positive (Self-perpetuating) Feedback Loop



Functional I/O (ICFP '09)

RabbitMQ



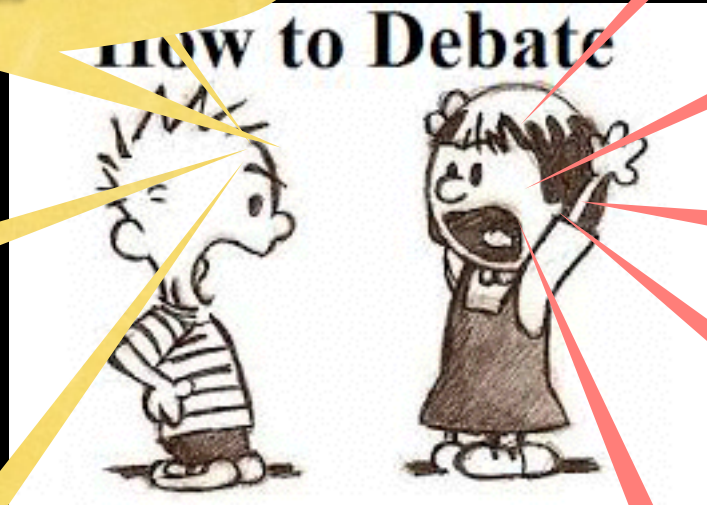
苗苗 简笔画
www.JianBiHua.org

truly functional GUIs

messages as events

communicating
worlds

from
freshman programs
to systems



actors

networks

publish
subscribe

failures!

message
brokers

Functional I/O &
Communicating Worlds

networking systems



DNS
Proxy

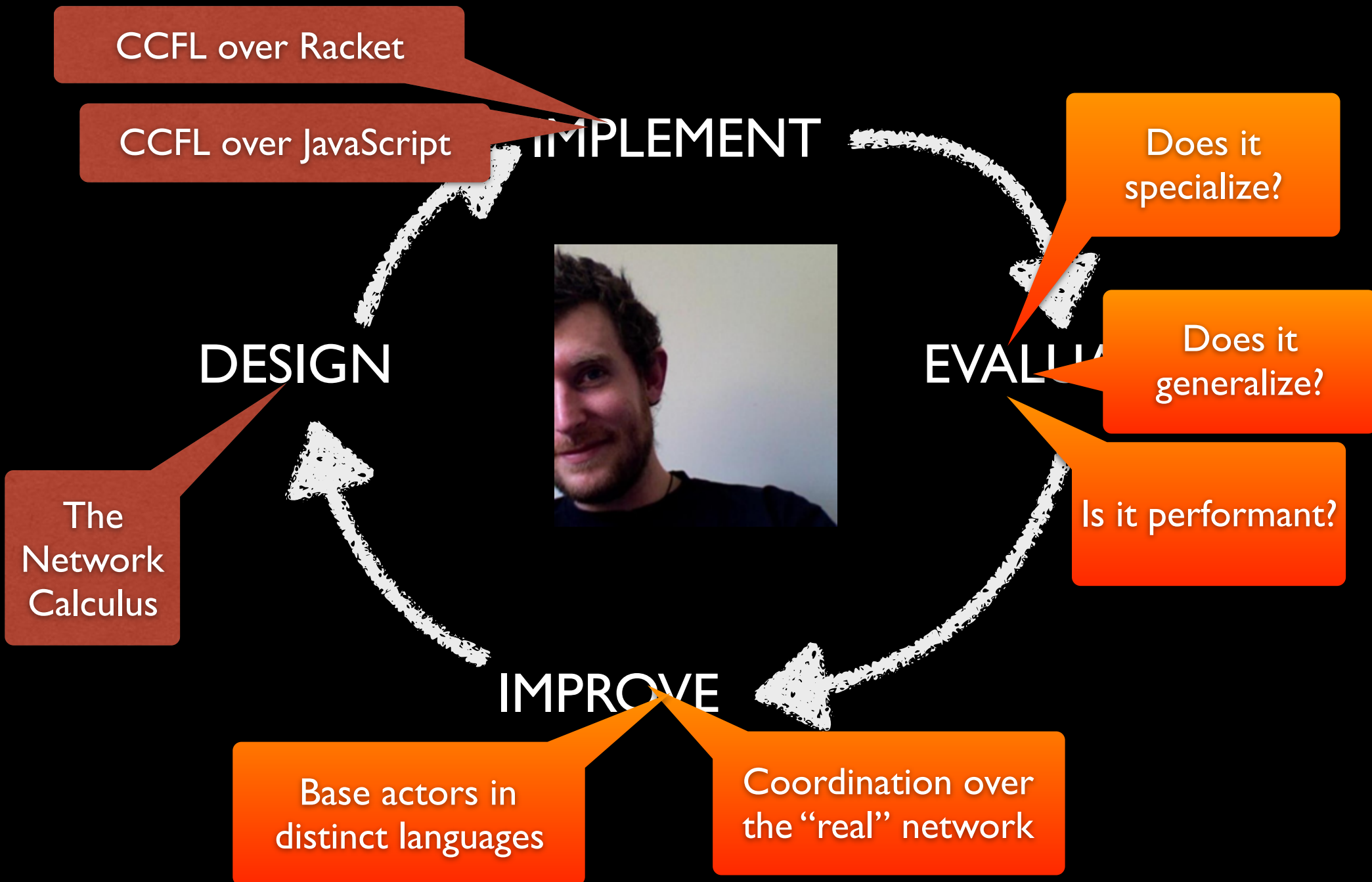
SSH
Server

Chat
Room

TCP
Stack

DSL for
comm.
actors

Topic: *Coordinated Concurrent Functional Language*



What is the cost of breaking open a new field?



5 years



6.5 years

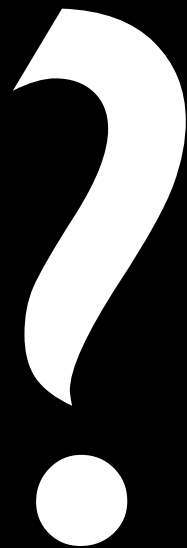
**how did I do
research as a
PhD student**

My Story

Dan Friedman


$$(f (g (call/cc k))) \\ = \\ (k (\lambda (x) (f (g x))))$$

Go, implement it. See what happens.



What does it mean to *implement* equations

Theoretical Computer Science 1 (1975) 125-159. © North-Holland Publishing Company

CALL-BY-NAME, CALL-BY-VALUE AND THE λ -CALCULUS

G. D. PLOTKIN

*Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh,
Edinburgh, United Kingdom*

Communicated by R. Milner

Received 1 August 1974

Abstract. This paper examines the old question of the relationship between ISWIM and the λ -calculus, using the distinction between call-by-value and call-by-name. It is held that the relationship should be mediated by a standardization theorem. A new λ -calculus is introduced.

I had read that paper.

... in two hours.

I read it again.

NOT 4 hours

NOT 4 days

I spent 4 MONTHS studying this paper.

What does it mean to *implement* equations

Theoretical Computer Science 1 (1975) 125–159. © North-Holland Publishing Company

CALL-BY-NAME, CALL-BY-VALUE AND THE λ -CALCULUS

G. D. PLOTKIN

*Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh,
Edinburgh, United Kingdom*

Communicated by R. Milner

Received 1 August 1974

Abstract. This paper examines the old question of the relationship between ISWIM and the λ -calculus, using the distinction between call-by-value and call-by-name. It is held that the relationship should be mediated by a standardisation theorem. Since this leads to difficulties, a new λ -calculus is introduced whose standardisation theorem gives a good correspondence

+

\neq

STUDIES IN LOGIC
AND
THE FOUNDATIONS OF MATHEMATICS
VOLUME III
J. BARWISE / D. KAPLAN / H.J. KEISLER / P. SUPPES / A.S. TROELSTRA
EDITORS

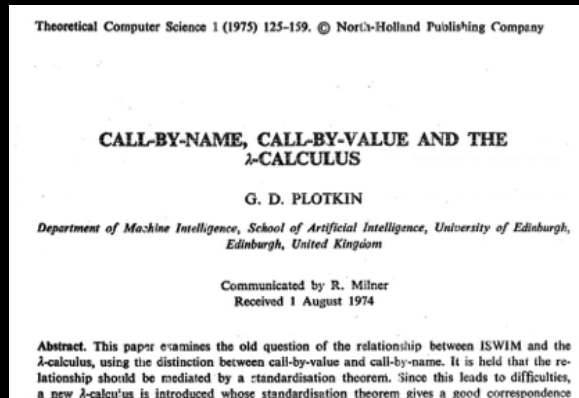
The Lambda Calculus Its Syntax and Semantics

REVISED EDITION

H.P. BARENDRECHT

NORTH-HOLLAND
AMSTERDAM • NEW YORK • OXFORD

What did four months of reading yield



How do calculi correspond to eval?

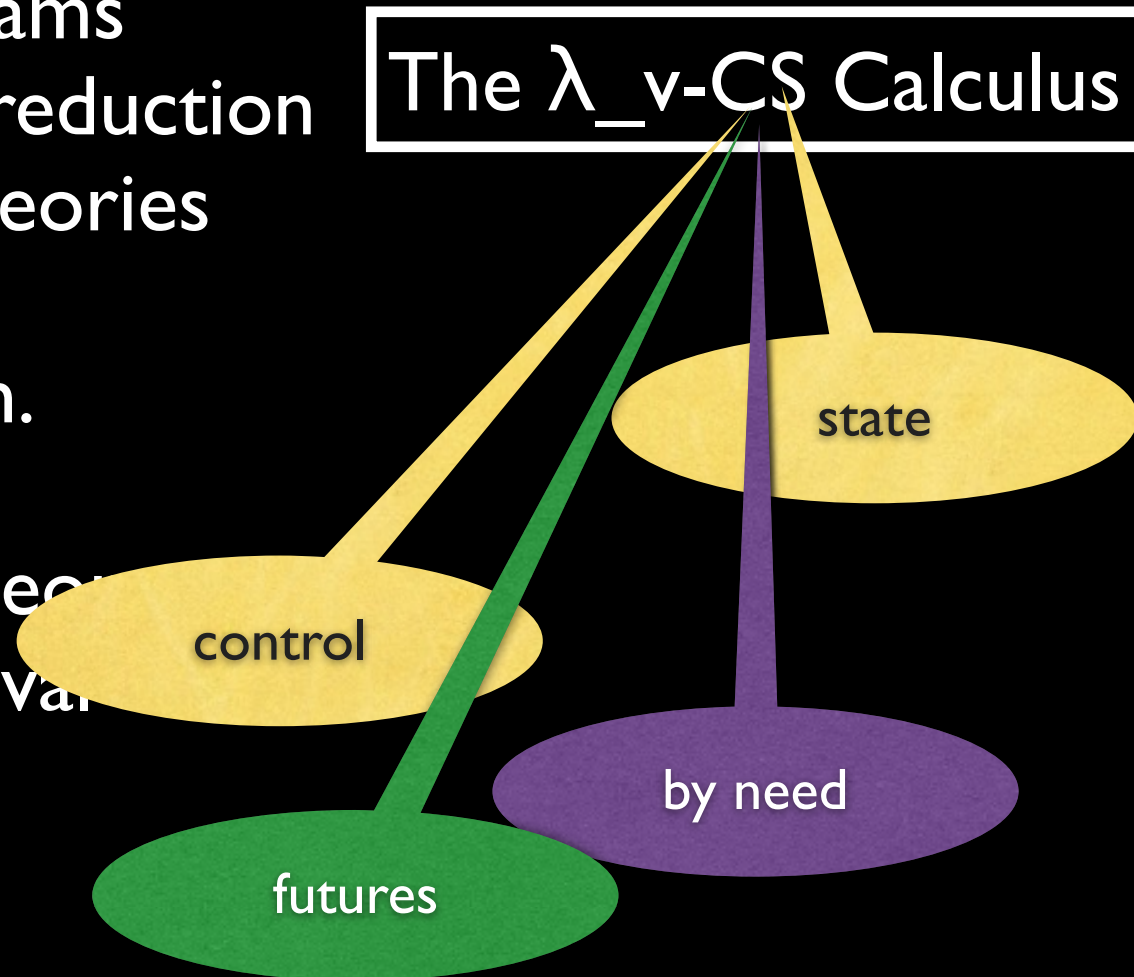
- start from an abstract syntax
- identify values & programs
- define basic notion of reduction
- inductively generate theories
- $\text{eval} \rightarrow$ and $\text{eval} =$
- Church & Rosser Thm.
- Thm. $\text{eval} \rightarrow = \text{eval} =$
- Standard Reduction Theorem
- Thm: $\text{eval-standard} = \text{eval} \rightarrow$

My dissertation: “This” works for imperative features, too.

How do calculi correspond to eval?

- start from an abstract syntax
- identify values & programs
- define basic notion of reduction
- inductively generate theories
- $\text{eval} \rightarrow$ and $\text{eval} =$
- Church & Rosser Thm.
- Thm. $\text{eval} \rightarrow = \text{eval} =$
- Standard Reduction Theorem
- Thm: $\text{eval} \text{-standard} = \text{eval} =$

The λ_v -CS Calculus



Lessons

Know to distinguish the *good* from the *bad*
in your advisor's suggestions.

Good paper require 'deep study'
not just a 'reading.'

Really good paper are 'research programs'
not just results.

**how do I do
research now**

problem I can solve

problem I can solve

problem I can solve

problem I can solve

problem I can solve

problem I can solve

problem I can solve

problem I can solve

problem I can solve

problem I can solve

problem I can solve

problem I can solve

problem I can solve

problem I can solve

paper I can write

paper I can write

paper I can write

paper I can write

paper I can write

paper I can write

pa

paper I can write

paper I can write

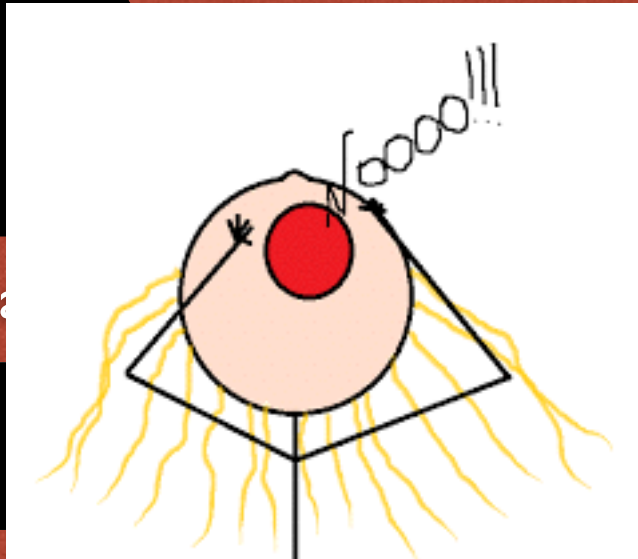
paper I can write

paper I can write

paper I can write

paper I can write

paper I can write



NOOOOOOO

More papers does not mean better researcher.



Think *big*, think long-term.

Lesson

Good researchers say “no” to many problems.
They focus on those that they care about.

My Long-term Projects

How can programmers design programs *systematically*?
(1985)

How do you teach 12, 14, 16 year olds programming and what benefit does this have?
(1995, last day of POPL)

How do types fit into untyped languages?
(1988)

What is linguistic power and why is a DSL better than an algorithm?
(1985)

What do such long-term
projects look like?

How do you launch
long-term projects?

What do such long-term
projects look like?

How do types fit into
untyped languages?
(1988)

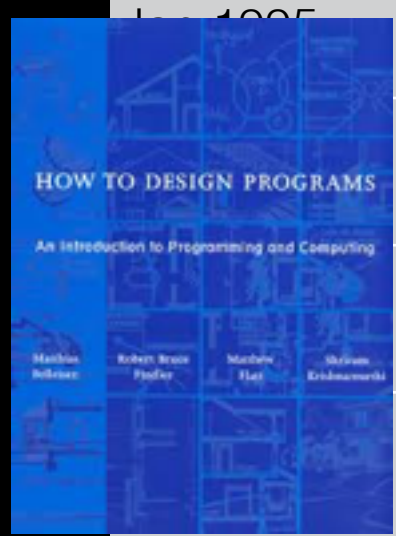
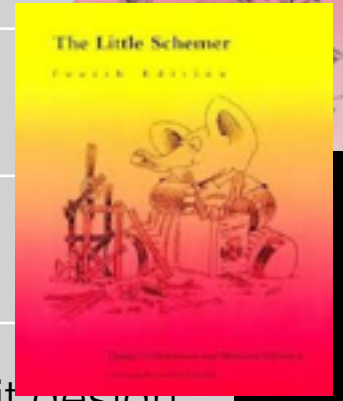
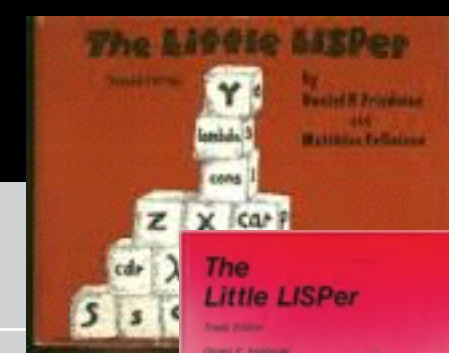
The “Gradual Typing” Dissertations

1990	Mike Fagan	Soft Typing (***)
1994	Andrew Wright	Practical Soft Typing
1998	Cormac Flanagan	Componential SBA
2002	Robby Findler	Higher-order Contracts
2005	Philippe Meunier	Modular SBA from Contracts
2006	Sam Tobin-H. (2010)	From Scripts to Programs
2012	Stevie Strickland	Contracts for First-class Classes
2015	Asumu Takikawa	Types for First-class Classes

How can programmers design programs systematically?

How to Design

How do you teach I2, <http://www.codeskulptor.org/>



1985

The Little LISper, 2nd ed.

F'1992

Teaching my first introductory programming course

1993/94

@ CMU, "Bob" "doing it all wrong"

1995

Launch *TeachScheme!* — FP and algebra in high schools

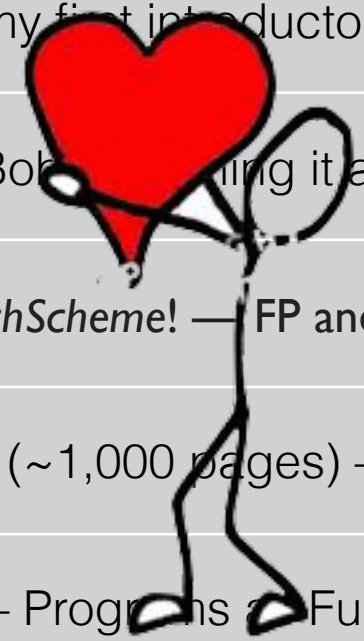
The Dump (~1,000 pages) — re-focusing on explicit design

HtDP/1e — Programs as Functions

Designing, implementing, evaluating

2007-2015

HtDP, 2nd ed. — Programs are not



“Macros”

What is linguistic power and why is a DSL better than an algorithm?
(1985)

1985	with Kohlbecker et al	Hygienic Macros
1986	with Bruce Duba	Macros in Phases
88/89	John Greiner, Steve Weeks	Programming Abstract Syntax
1991	Todd Yonkers	Extensible Syntax
1994	Matthew Flatt	Connecting DSLs into Applications
95/97	PLT	Teaching languages
95/99	Shriram Krishnamurthi	Parameterizing over Language
2002	Matthew Flatt	You want it when?
03/08	Ryan Culpepper	Protecting Macros
08/09	Ryan Culpepper	Debugging Macros

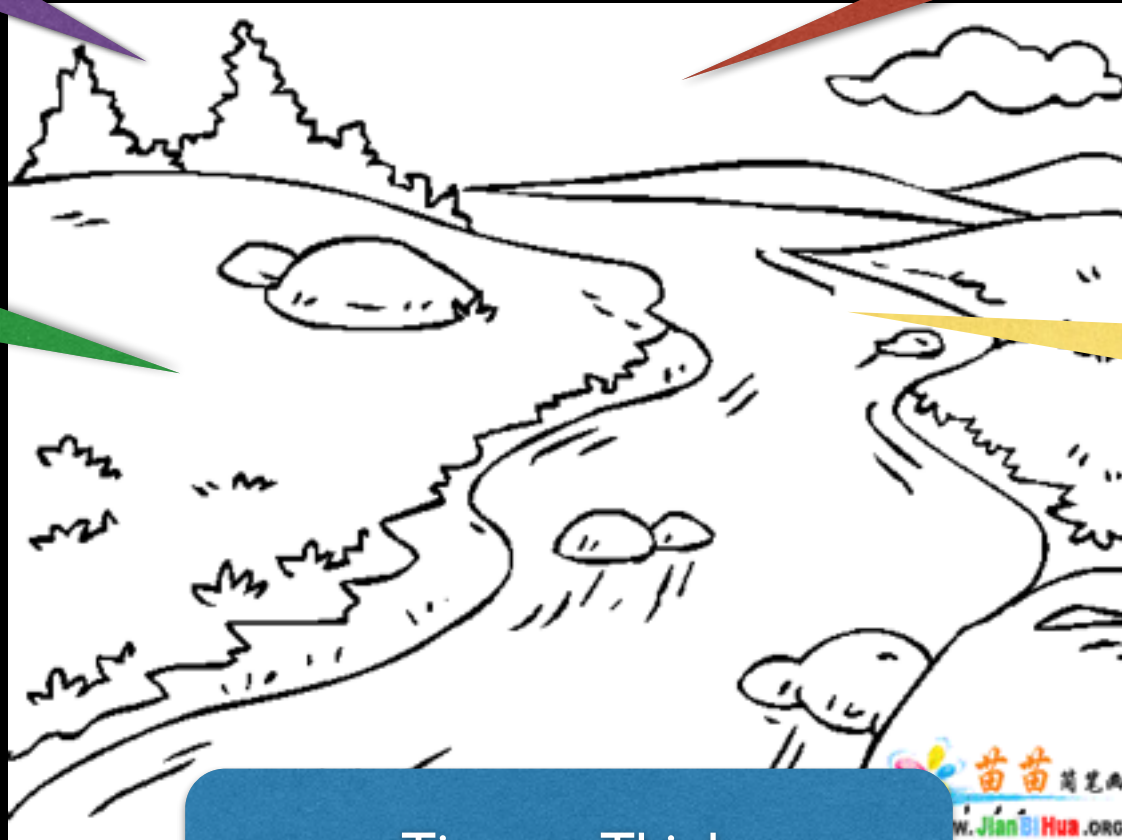
How do you launch
long-term projects?

People

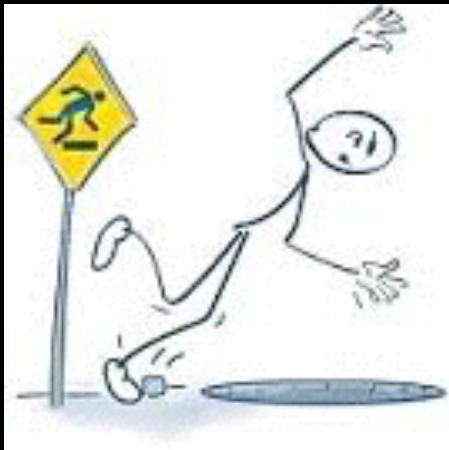
Readings

Teaching

“Reality”



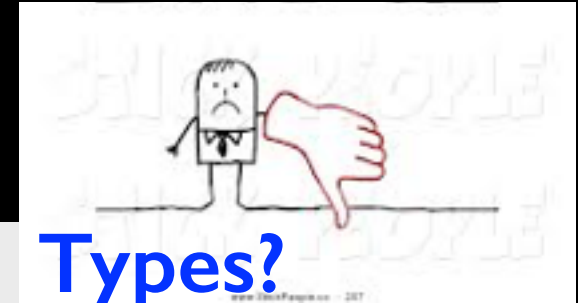
Time to Think



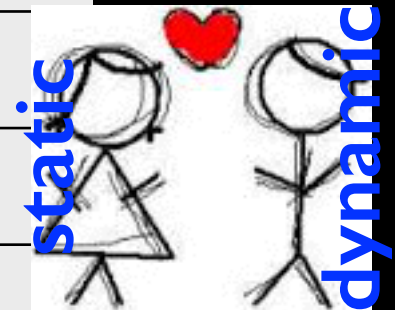
Sometimes you stumble into a topic.

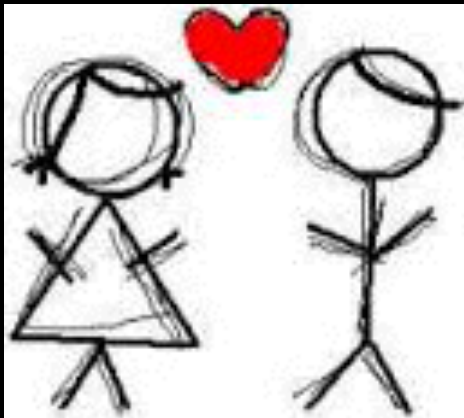
How do types fit into
untyped languages?
(1988)

The “Gradual Typing” Dissertations



1990	Mike Fagan	Soft Typing (***)
1994	Andrew Wright	Practical Soft Typing
1998	Cormac Flanagan	Componential SBA
2001	Robby Findler	Higher-order Contracts
2005	Philippe Meunier	Regular SBA from Contracts
2006	Sam Tobin-H. (2)	Contracts to Programs
2012	Stevie Strickland	First-class Classes
2015	Asumu Takikawa	Classes





Sometimes it is love at first sight.

How can programmers design programs systematically?

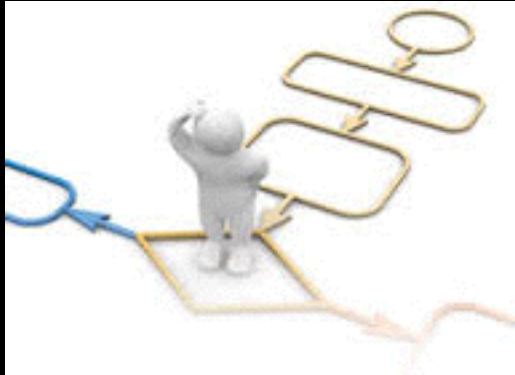
How do you teach 12, 14, 16 year olds programming
What benefit does it have?
(1995, las

An “entertaining” thought

1985	The Litte Lisper, 2nd ed.
F'1992	Teaching my first introductory programming course
1993/94	@ CMU, “Bob’s teaching it all wrong”
Jan 1995	Launch <i>TeachScheme!</i> — FP and algebra in high schools
Spring 1995	The Dump (~1,000 pages) — re-focusing on explicit design
1996-2001	HtDP/1e — Programs
2002-2005	Designing, implement... /O”
2007-2015	HtDP, 2nd ed. — Progra

Cormac asked the one critical question

We knew what we had to do: software, curriculum, teaching



Sometimes it develops as a necessity.

“Macros”

What is linguistic power and why is a DSL better than an algorithm?
(1985)

1985	with Kohlbecker et al	Hygienic Macros
1986	with Bruce Duba	Macros in Phases
88/89	John Greiner, Steve Weeks	Programming Abstract Syntax
1991	Todd Yonkers	Extensible Syntax
1994	Matthew Flatt	Connecting DSLs into Applications
95/97	PLT	Teaching languages
95/99	Shriram Krishnamurthi	Parameterizing over Language
2002	Matthew Flatt	You want it when?
03/08	Ryan Culpepper	Protecting Macros
08/09	Ryan Culpepper	Debugging Macros

**what to
remember?**

As a student, you need to

- develop a sense of the landscape
- follow your heart
- plan out design, implementation, evaluation.

No matter what, keep in mind that the number of your papers is *unrelated* to the quality of your work.

As a researcher, I

- look for long-term projects
- follow my heart
- use teaching (for the 99%) for inspiration
- develop dissertation-size goals
- plan for hand-over
- and have my eyes open for new ideas.

The End