how do i do research

matthias felleisen,
racketeer,
plt, northeastern

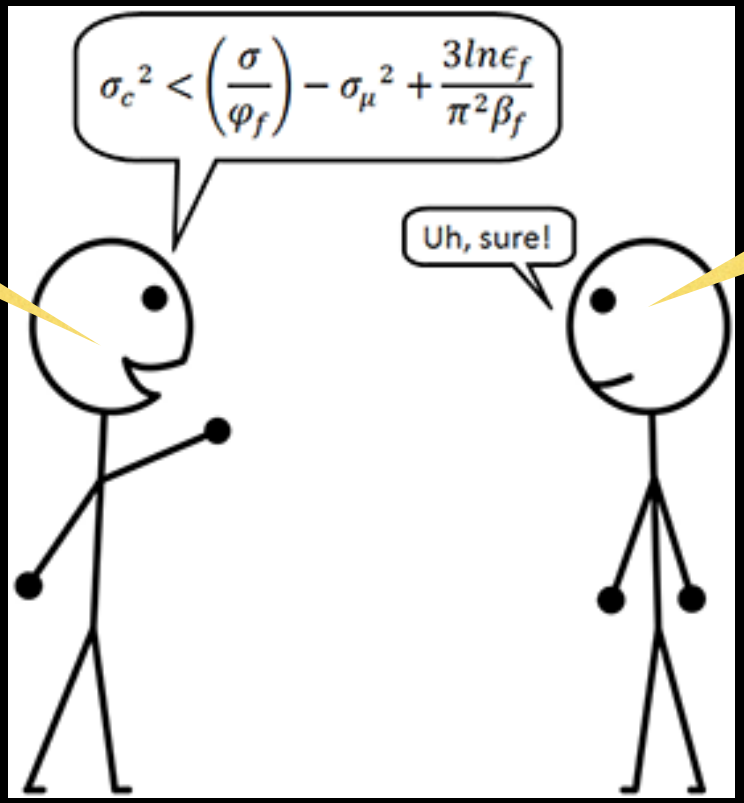
how do *I* do research?

how ~~do~~ I do research?

would

if I were you

your advisor



& you

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

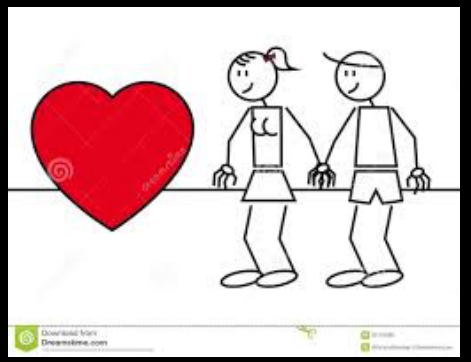
how would I do
research if I were you?

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.

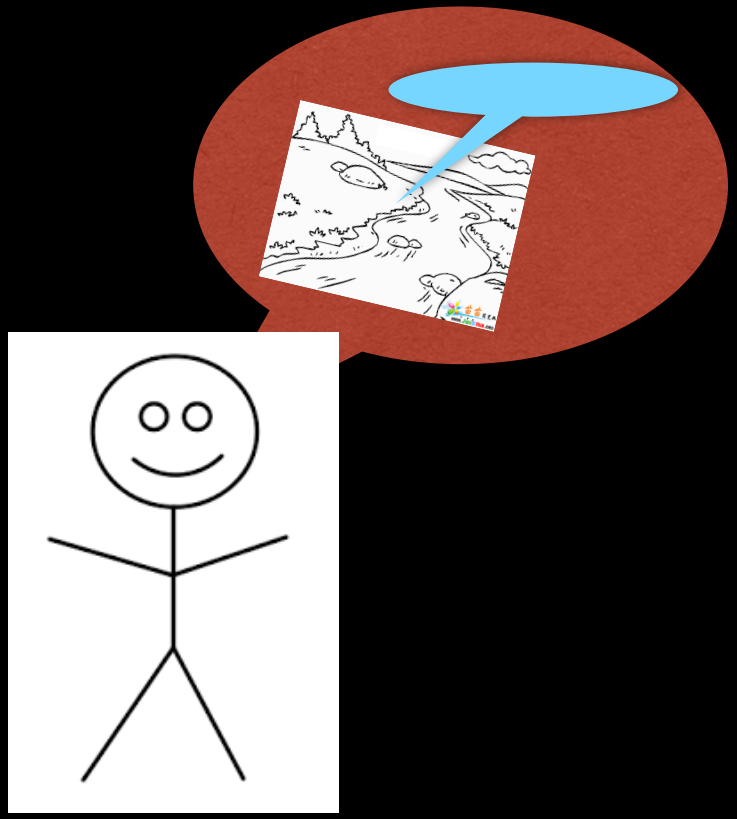


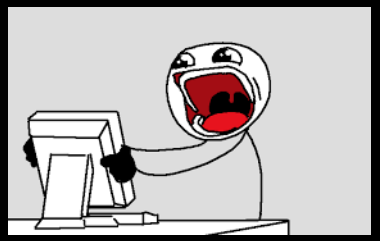
Functional I/O (ICFP '09)

Typed Racket (ICFP '10)

Compiler Coaching (OOPSLA '12)

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



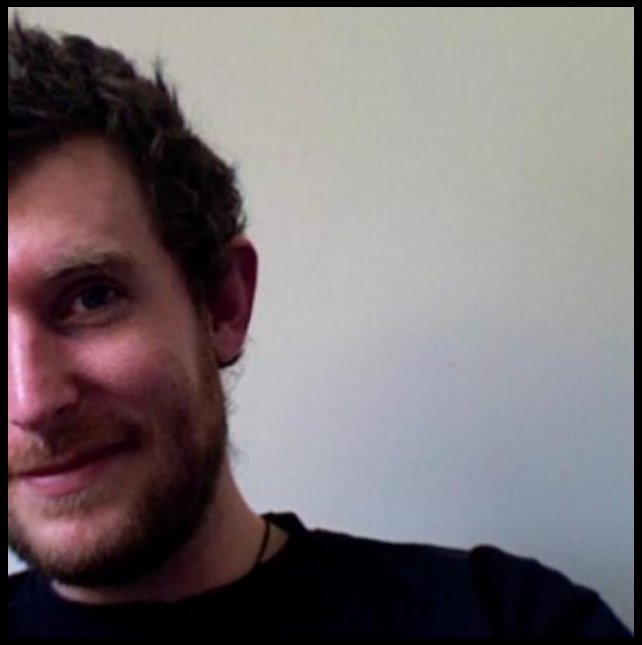


And that's what's called 'doing research.'

how would I do research if I were you

Two Case Studies

Asumu
Takikawa

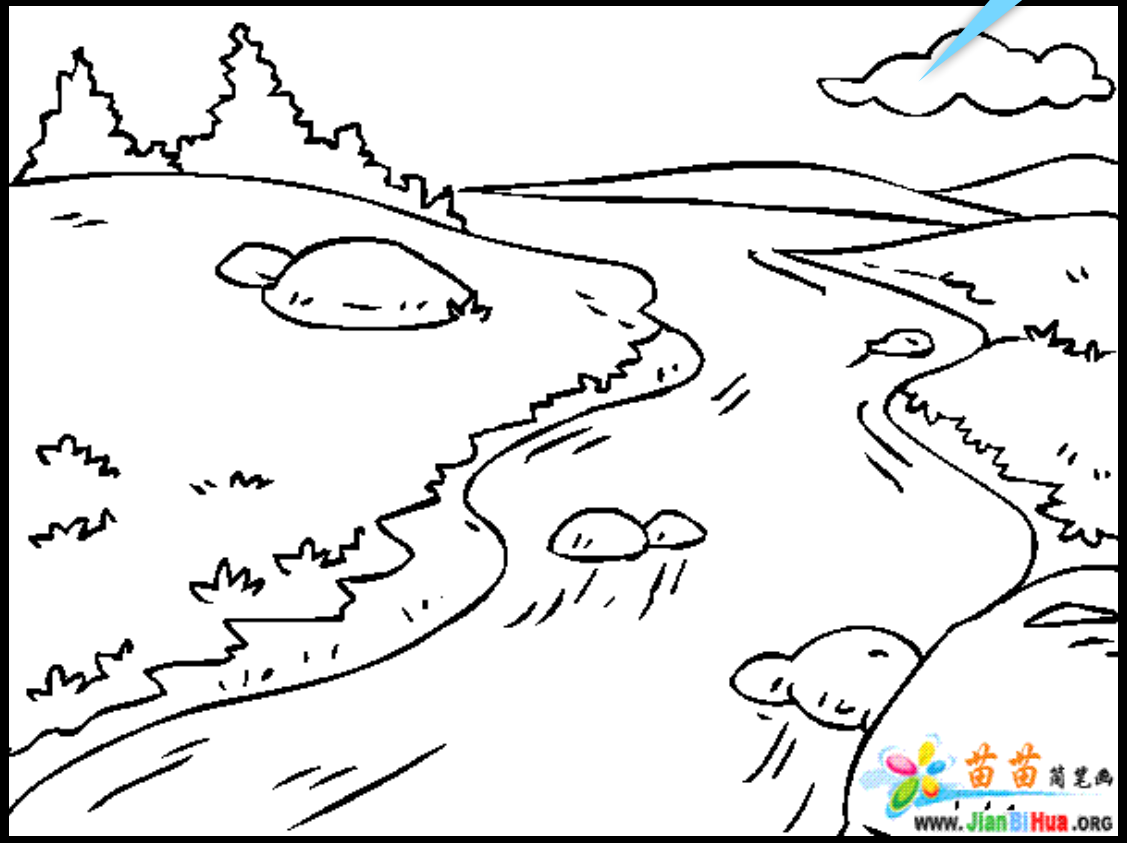


Tony
Garnock-Jones

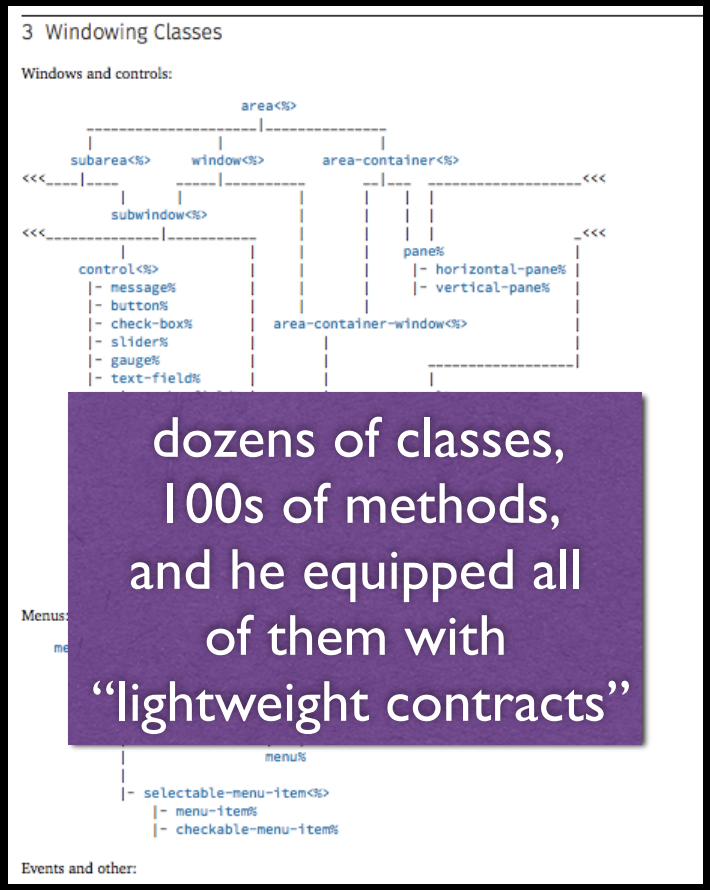
Kuhn, *The Structure of Scientific Revolution*

Types for Classes

Typed Racket
(ICFP '10)



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



3 Windowing Classes

Windows and controls:

area<%>
subarea<%> window<%> area-container<%>
subwindow<%>
pane%
control<%>
|- message%
|- button%
|- check-box%
|- slider%
|- gauge%
|- text-field%
|- horizontal-pane%
|- vertical-pane%
area-container-window<%>

dozens of classes, 100s of methods, and he equipped all of them with "lightweight contracts"

Menus:

menu%
|- selectable-menu-item<%>
|- menu-item%
|- checkable-menu-item%

Events and other:

Dr.

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
360 Huntington Avenue, 161CN
Boston, MA 02115, USA

Abstract

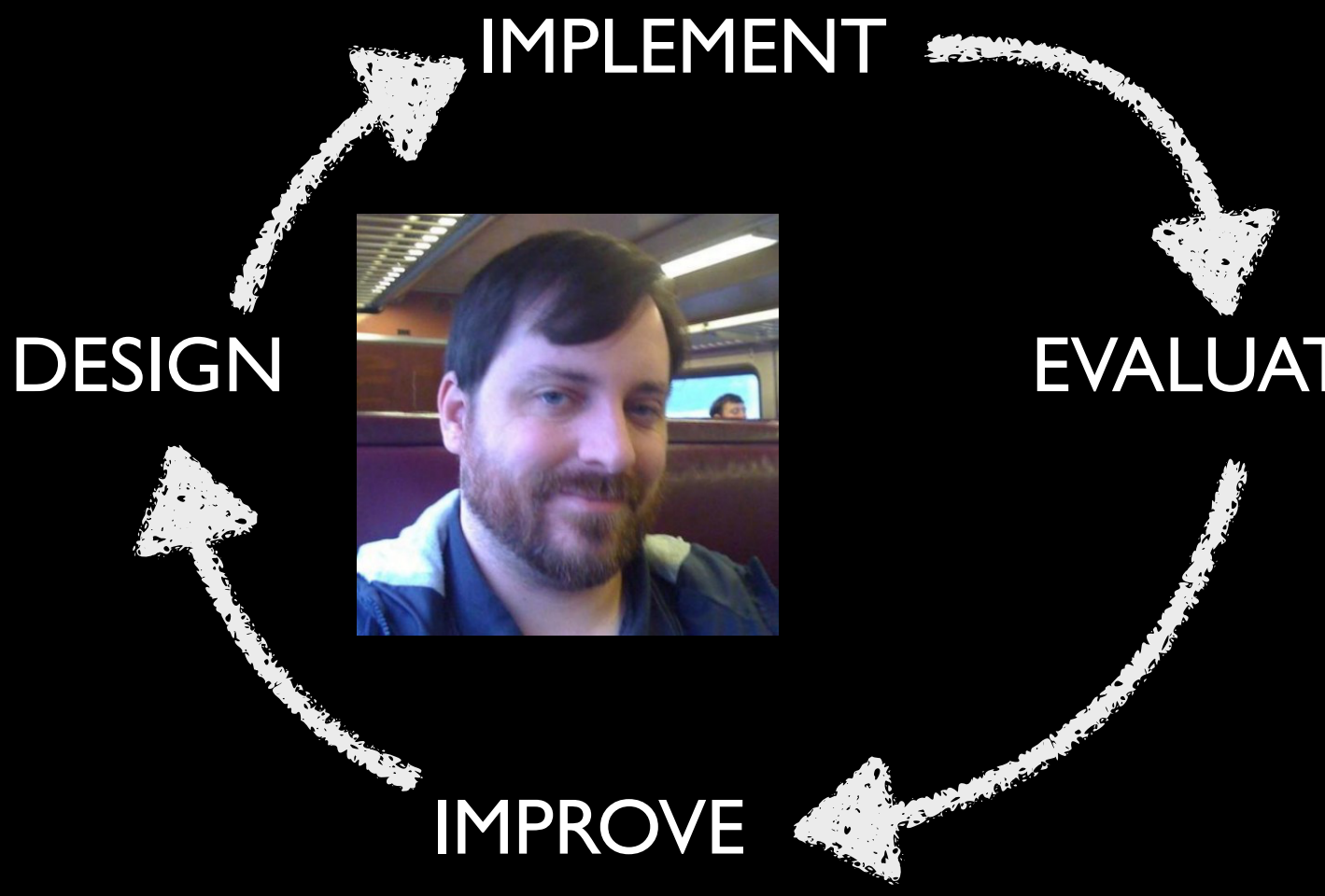
We consider the problem of strong typing for a model of object-oriented programming systems. These systems permit values which are records of other values, and in which fields inside these records are retrieved by name. We propose a type system which allows us to classify these kinds of values and to 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.

1. Introduction

The function momentum 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

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

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

EVALUATE

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.

Towards Practical Gradual Typing

Asumu Takikawa[1], Daniel Feltey[1], Earl Dean
Robert Bruce Findler[4], Sam Tobin-Hochstad
Felleisen[1]

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

IMPLEMENT

Abstrac

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

ual typing, object-oriented programming, performance evaluation

4230/LIPIcs.ECOOP.2015.999

Takikawa ‘15

Contracts for First-Cla

T. STEPHEN STRICKLAND,
MATTHIAS FELLEISEN, Nor

DESIGN

Abstract

Dynamic type-checki
often go hand-in-hand
Ruby, and JavaScript
gramming. When scri
evolve into large programs, the la
pline reduces maintainability. A pr
to migrate parts of such scripts to
static type system. Unfortunately, existing type systems hav
ther support the flexible OO composition mechanisms found
in scripting languages nor accommodate sound interopera-
tion with untyped code.

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
JavaScript [19] and Perl [31]. Formal models have validated soundness for gradual type systems, allowing seamless interoperation between sister languages [22, 27, 32].

Takikawa & Strickland ‘13

First-class classes enable progra
with new forms of object-oriented
calls for tools to control the comp
that has seen much use in object
cope with first-class classes. On t
such as numbers, while classes are
contained within class definitions,

This paper presents the design
as a two-pronged evaluation. The
of our language. The theorem shows that when the contract system assigns blame to a component for a contract violation, the component is indeed responsible for providing the non-conforming value. The second 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.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented p*
contract; I
ogram Verification—*Programming by*
Theory—*Semantics*

General T

Additional

ACM Ref

ACM Trans. Program. Lang. Syst. V, N, Article A (January YYYY), 57 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

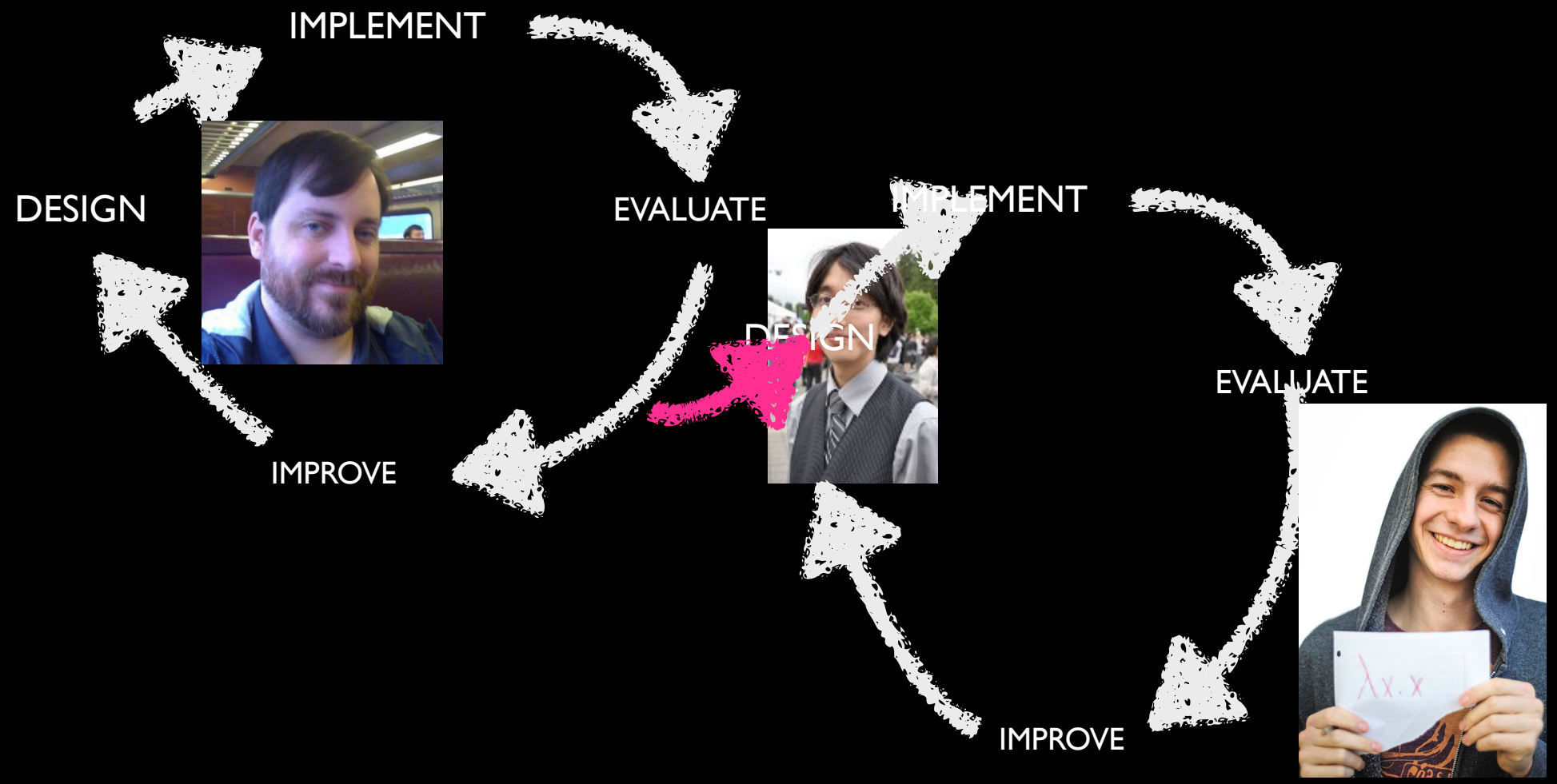
EVALUATE

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

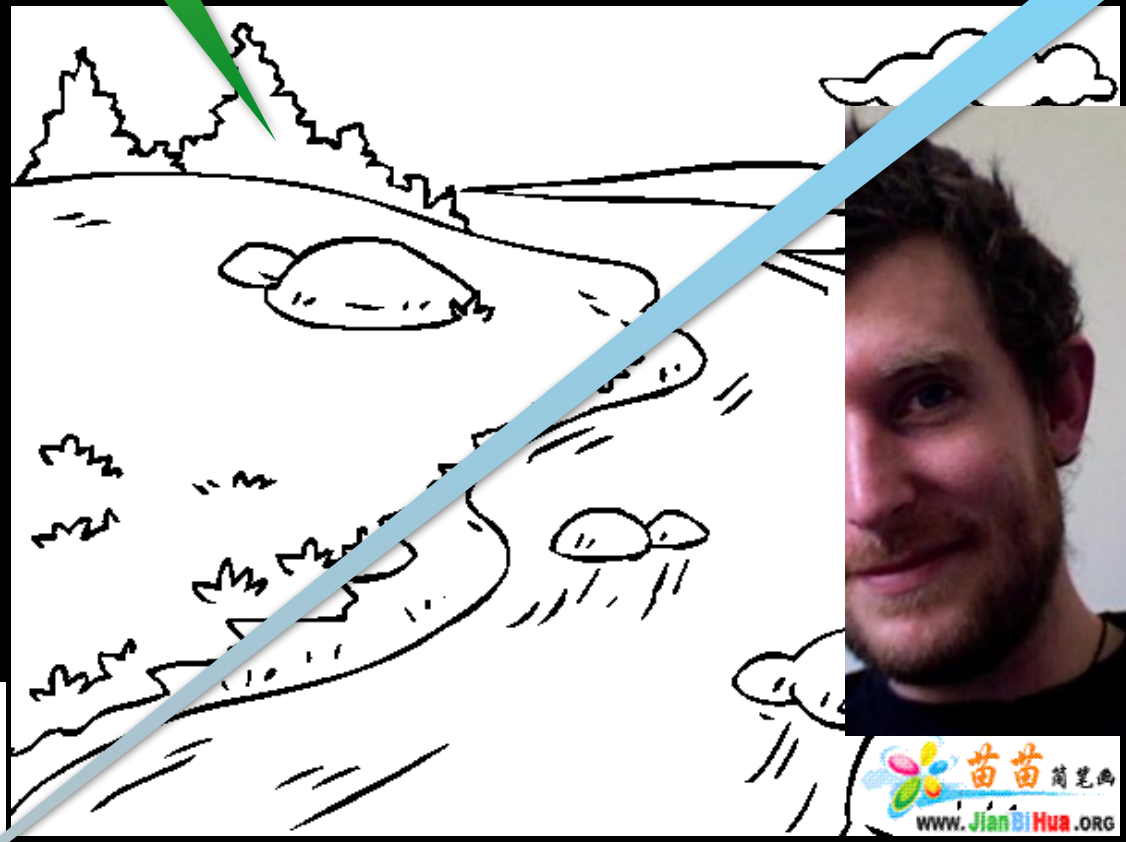
Strickland & Takikawa ‘12

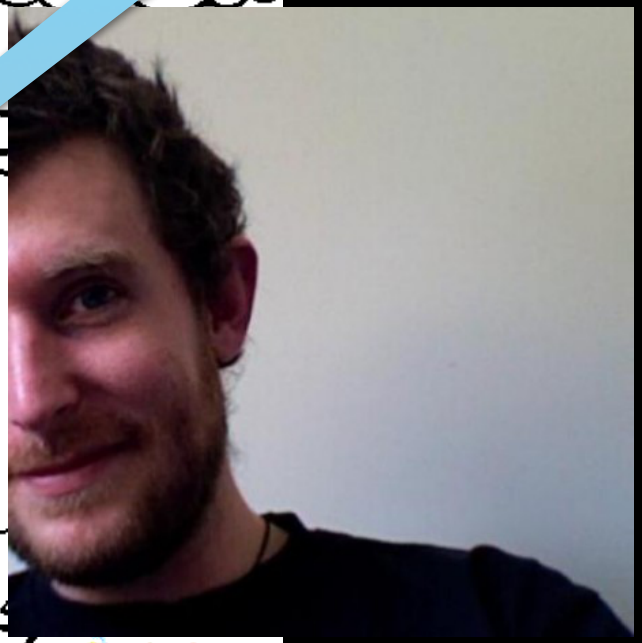
A Positive (Self-perpetuating) Feedback Loop

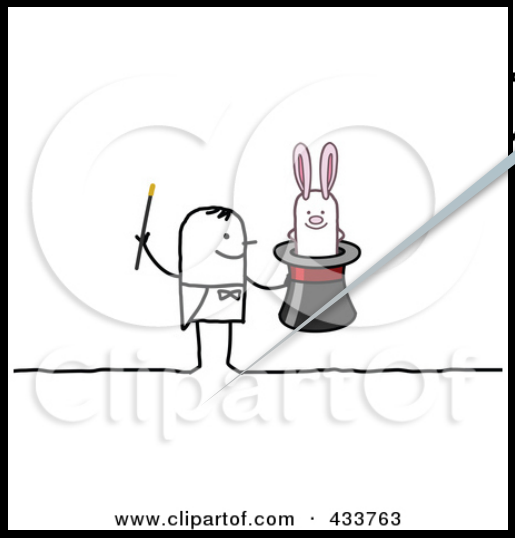


Functional I/O (ICFP '09)

RabbitMQ





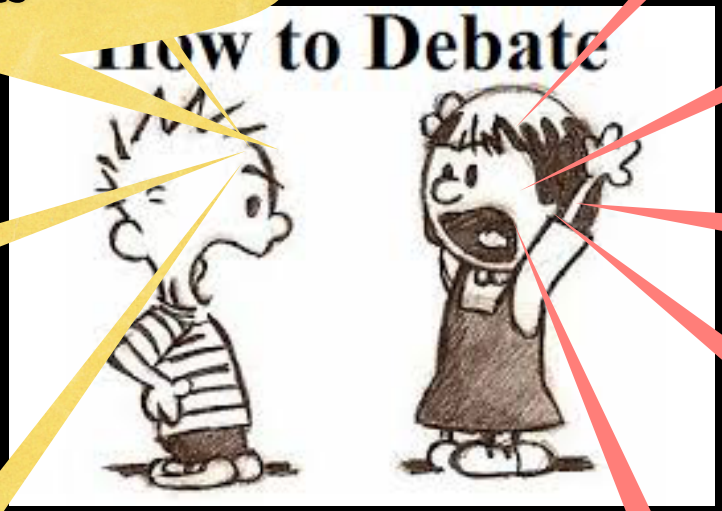


truly functional GUIs

actors

networks

messages as events



publish
subscribe

communicating
worlds

failures!

from
freshman programs
to systems

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*

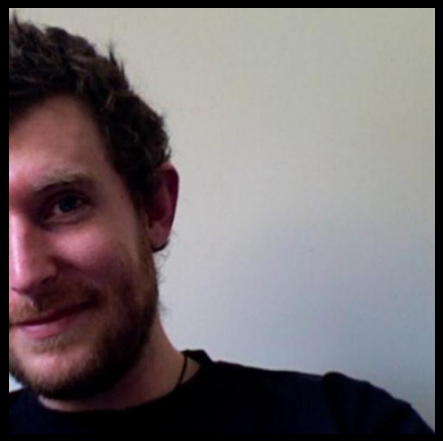
CCFL over Racket

CCFL over JavaScript

IMPLEMENT

Does it specialize?

DESIGN



EVALUATE

Does it generalize?

Is it performant?

The Network Calculus

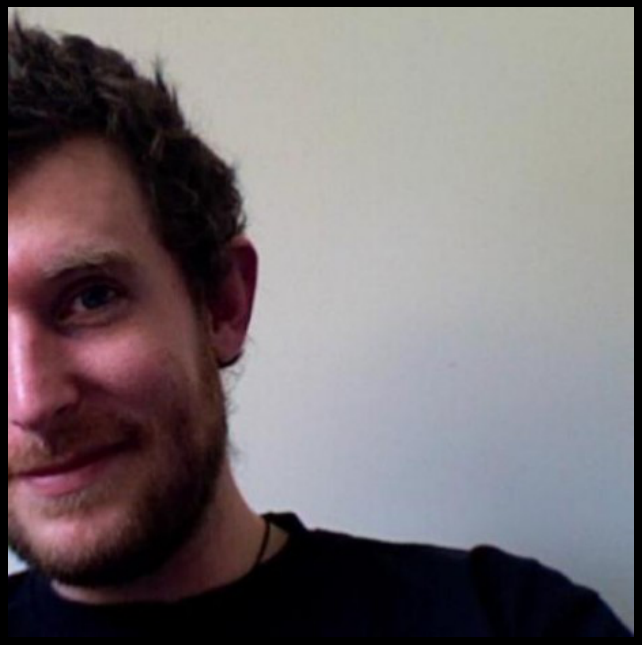
IMPROVE

Base actors in distinct languages

Coordination over the "real" network

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

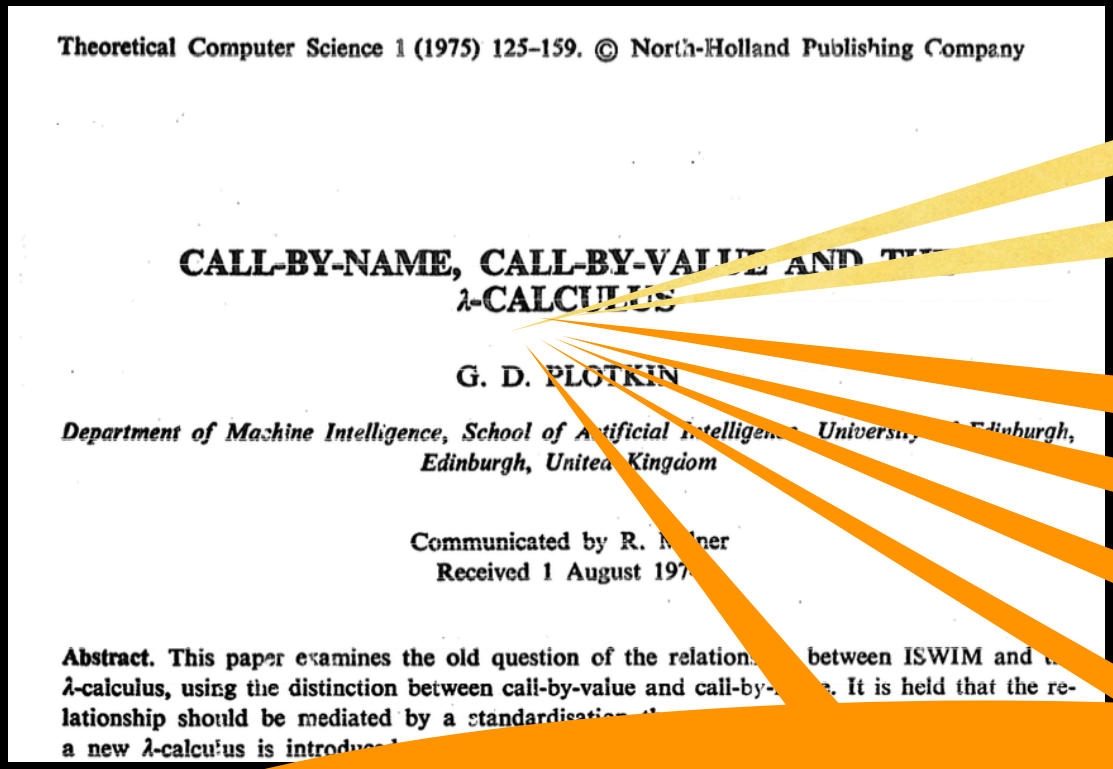
Dan Friedman

```
(f (g (call/cc k))
        =
(k (λ (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 197

Abstract. This paper examines the old question of the relation 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. a new λ-calculus is introduced

I had read that paper.

… in two hours.

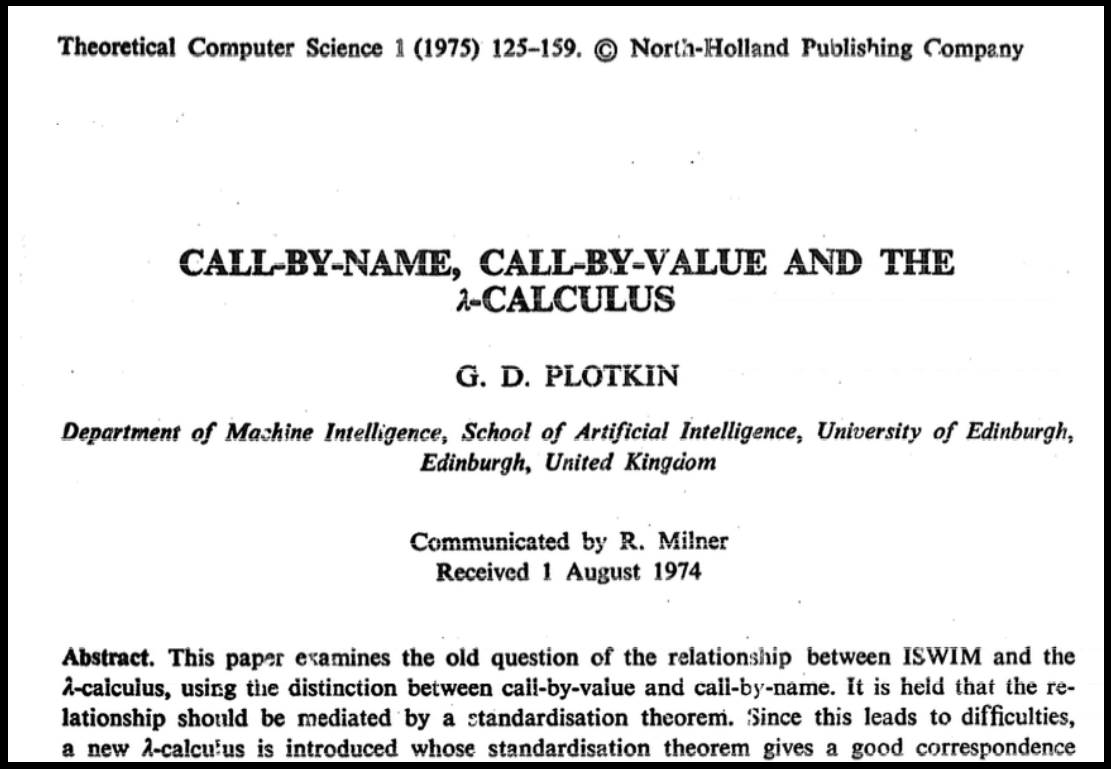
I read it again.

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

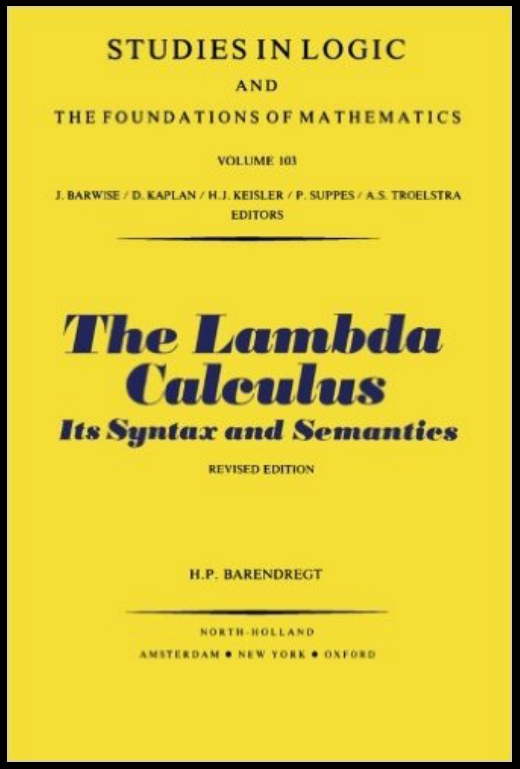
G. D. PLOTKIN

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

+



STUDIES IN LOGIC
AND
THE FOUNDATIONS OF MATHEMATICS
VOLUME 103
J. BARWISE / D. KAPLAN / H.J. KEISLER / P. SUPPES / A.S. TROELSTRA
EDITORS

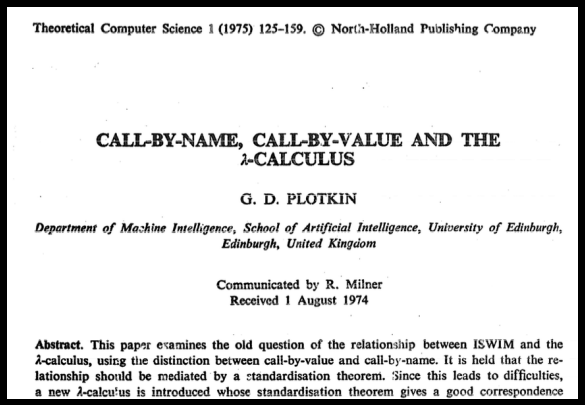
The Lambda Calculus
Its Syntax and Semantics

REVISED EDITION

H.P. BARENDREGT

NORTH-HOLLAND
AMSTERDAM • NEW YORK • OXFORD

What did four months of reading yield



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

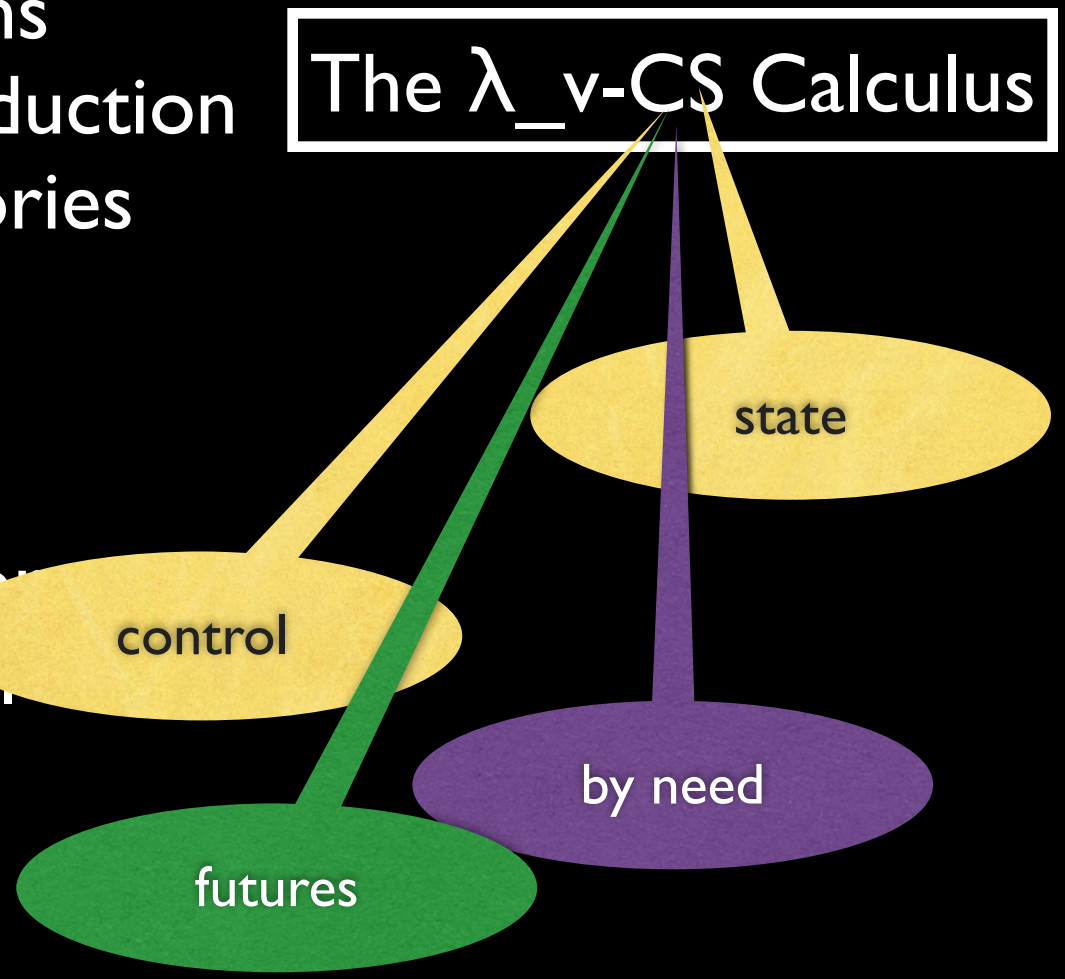
How do calculi correspond to eval?

- start from an abstract syntax
- identify values & programs
- define basic notion of reduction
- inductively generate theories
- eval-> and eval=
- Church & Rossser Thm.
- Thm. eval-> = eval=
- Standard Reduction Theorem
- Thm: eval-standard = eval->

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
- eval-> and eval=
- Church & Rossser Thm.
- Thm. eval-> = eval=
- Standard Reduction Theo
- Thm: eval-standard = eva



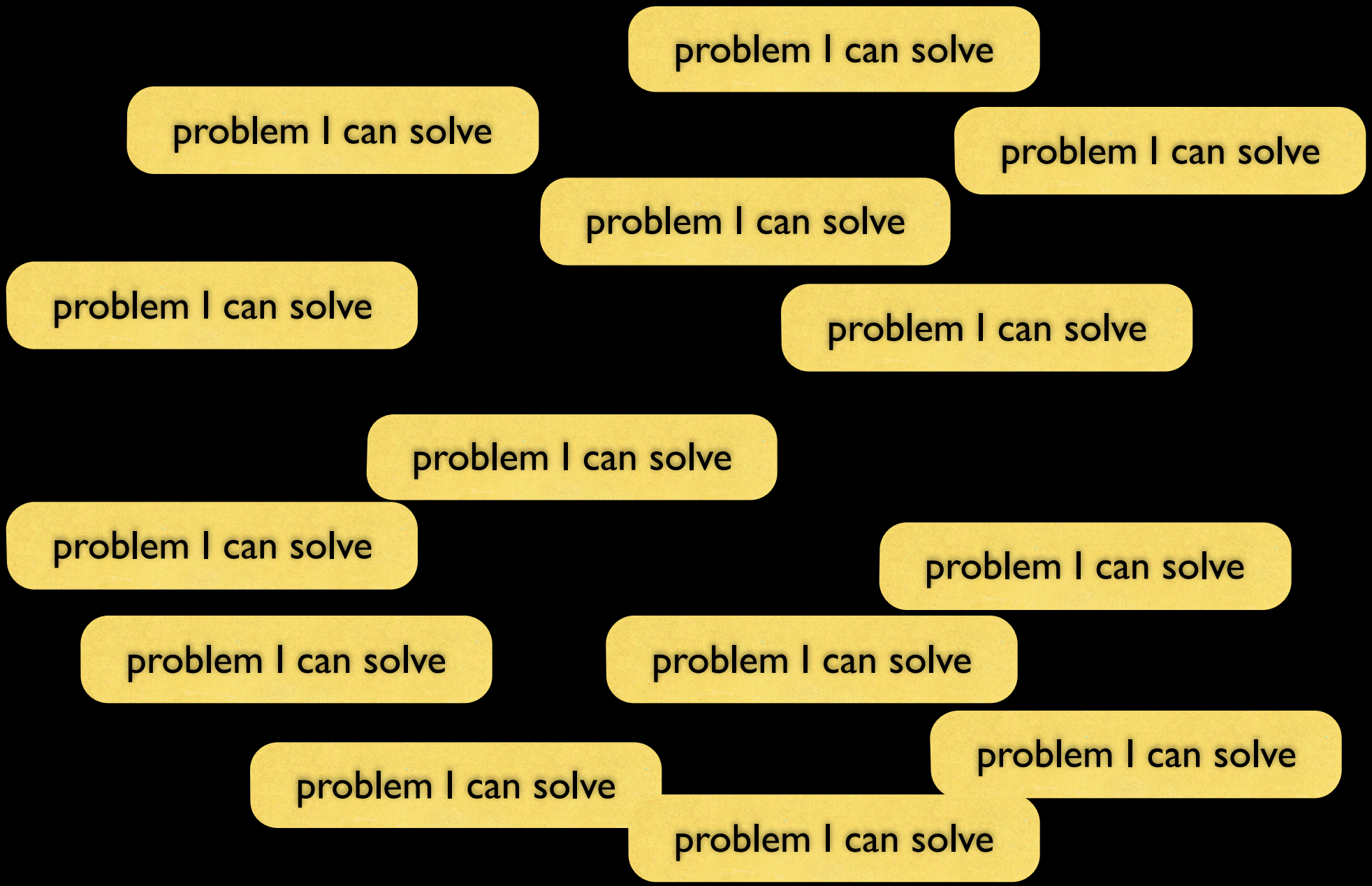
Lessons

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

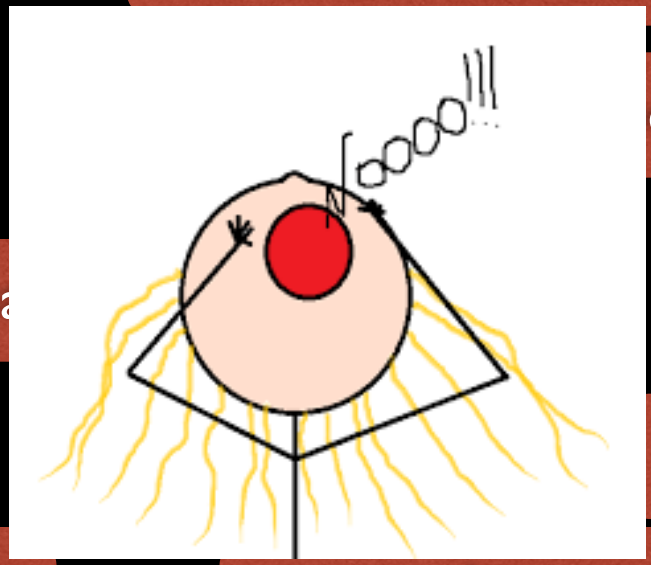
paper I can write

paper I can write

paper I can write

paper I can write

paper I can write



er I can write

paper I can write

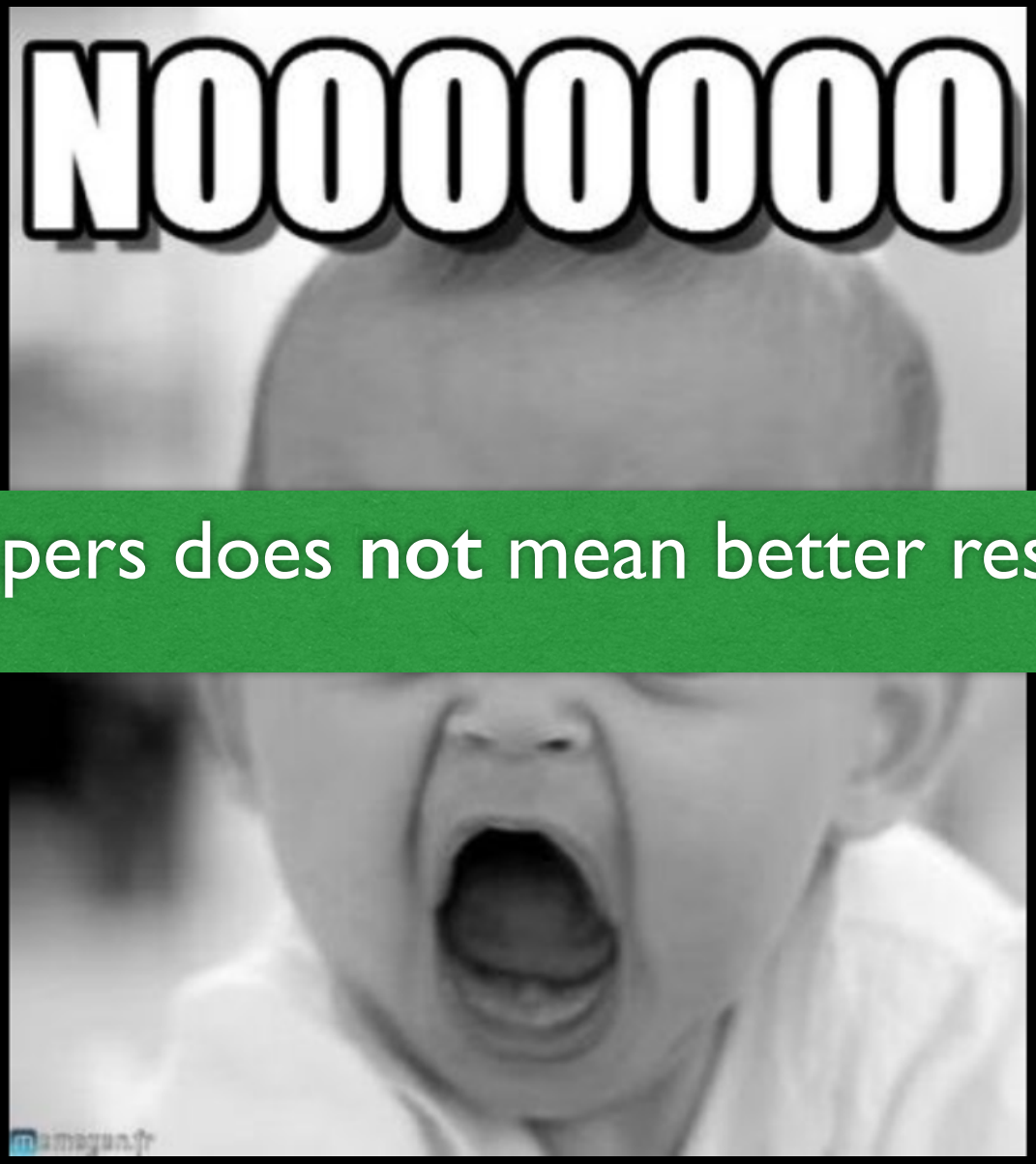
paper I can write

paper I can write

paper I can write

paper I can write

paper I can write



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 do types fit into untyped languages? (1988)

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)

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)

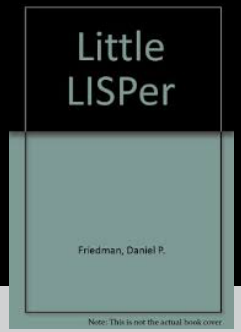
The "Gradual Typing" Dissertations

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

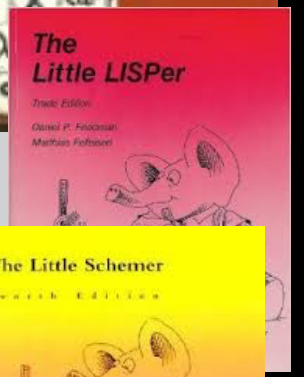
How can programmers design programs *systematically*?

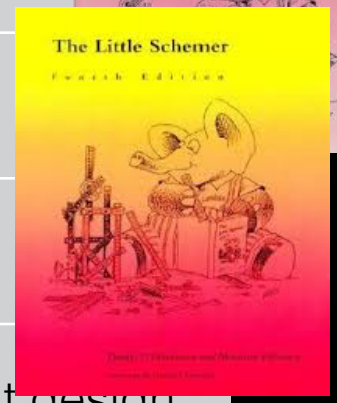
How do you teach 12, http://www.google.co/

ww

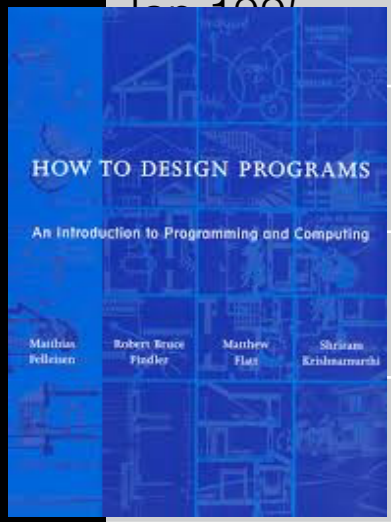


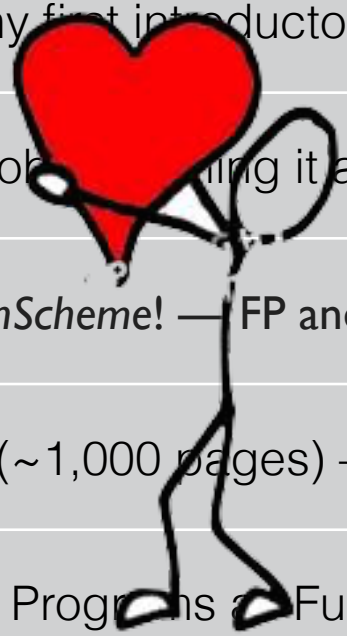
How to Design

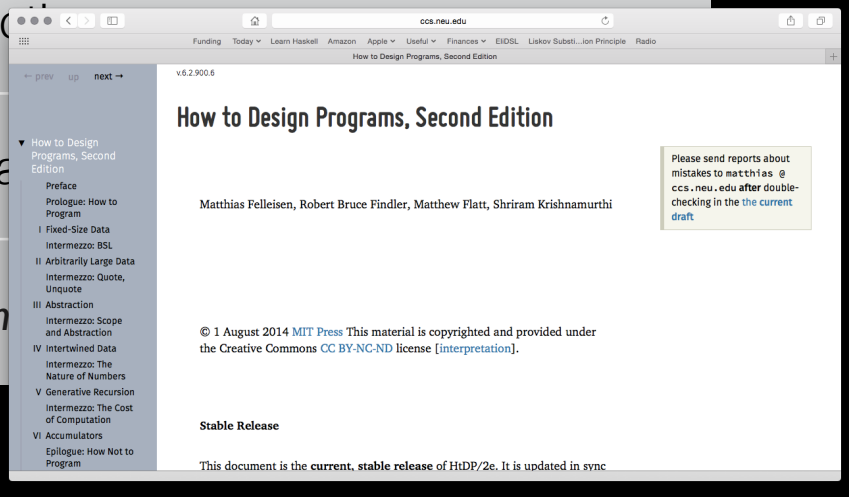




1985	The Litte Lisper, 2nd ed.
F'1992	Teaching my first introductory programming course
1993/94	@ CMU, "Bo ing it all wrong"
Jan 1995	Launch *TeachScheme*! — FP and algebra in high schools
	The Dump (~1,000 pages) — re-focusing on explicit design
	HtDP/1e — Progr ms a Func
	Designing, implementing, eva
2007-2015	HtDP, 2nd ed. — Programs are *n*







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

"Macros"

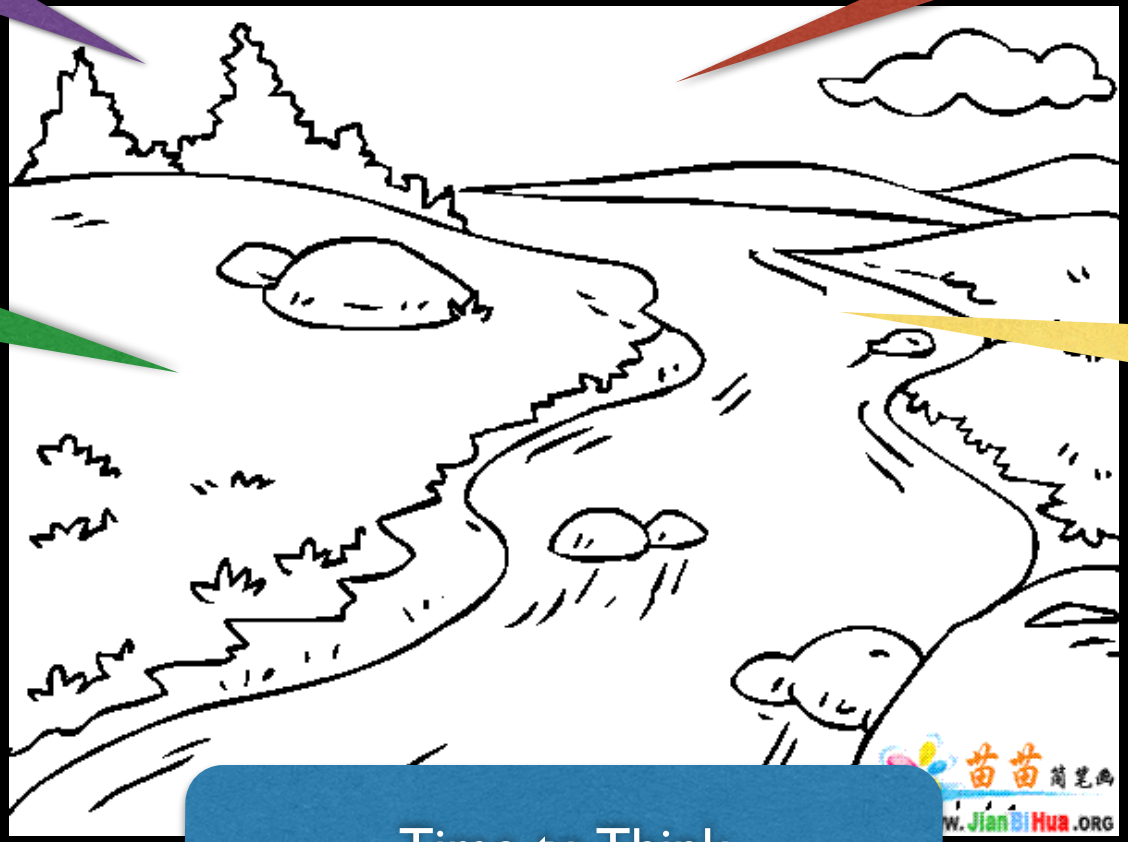
1985	with Kohlbecker et al	Hygienic Maros
1986	with Bruce Duba	Macros in Phases
88/89	John Greiner, Steve Weeks	Programming Abstract Syntax
1991	Todd Yonkers	Extensible Syntax
1994	Matthew Flatt	Connecting DSLs into Applications
95/97	PLT	Teaching languages
95/99	Shriram Krishnamurthi	Parameterizing 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



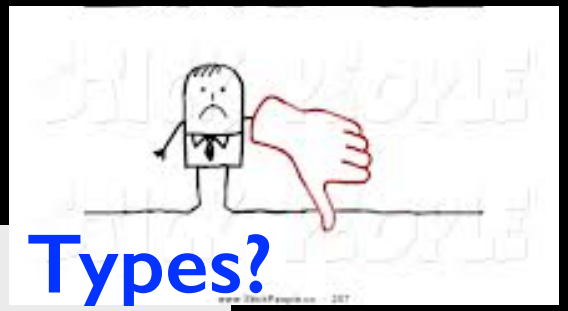
"Reality"

Time to Think

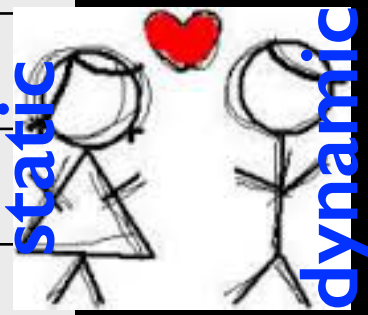
Sometimes you stumble into a topic.

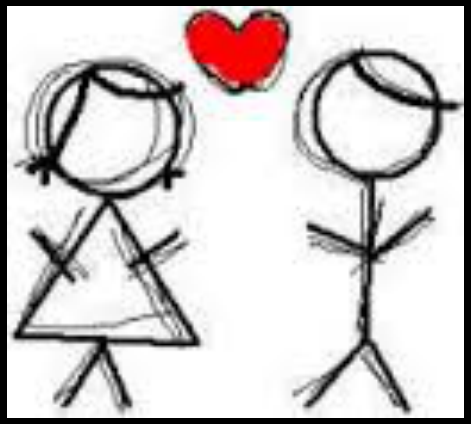
How do types fit into untyped languages? (1988)

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	lar SBA from Contracts
2006	Sam Tobin-H. (2	ts to Programs
2012	Stevie Stri	rst-class Classes
2015	Asum	Classes





Sometimes it is love at first sight.

How can programmers design programs *systematically*?

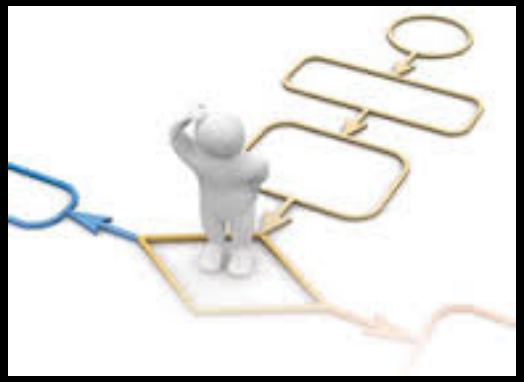
How do you teach 12, 14, 16 year olds programming What benefit does it have? (1995, las

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 *TeachScheme*! — FP and algebra in high schools
c 1995	The Dump (~1,000 pages) — re-focusing on explicit design
96-2001	HtDP/1e — Programs
2002-2005	Designing, implemen /O"
2007-2015	HtDP, 2nd ed. — Progra

An "entertaining" thought

Cormac asked the one critical question

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



Sometimes it develops as a necessity.

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

"Macros"

1985	with Kohlbecker et al	Hygienic Maros
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