Proof Assistant as Teaching Assistant

A View from the Trenches

ITP 2010

Benjamin C. Pierce
University of Pennsylvania

An Experiment

in Pedagogy

Goal

Goal

From...

Teaching theorem proving as a topic in its own right

Goal

From...

Teaching theorem proving as a topic in its own right

To...

Theorem prover as a *framework* for teaching something else

A “software foundations” course for students from a broad range of backgrounds

Parameters

- Taught yearly at Penn
- 30-70 students
- Semi-required course for masters and PhD students
- Mix of undergraduates, MSE students, and PhD students (mostly not studying PL)
- 13 weeks, 23 lectures (80 minutes each), plus 3 review sessions and 3 exams
- Weekly homework assignments (~10 hours each)

A “Software Foundations” Syllabus

(for the masses)

Logic

- Inductively defined relations
- Inductive proof techniques

$$\frac{\text{logic}}{\text{software engineering}} = \frac{\text{calculus}}{\text{EE, civil, mechanical, ...}}$$

Functional Programming

- programs as data, polymorphism, recursion, ...

- FPLs are going mainstream (Haskell, Scala, F#, ...)
- Individual FP ideas are already mainstream
  - mutable state = bad (e.g. for concurrency)
  - polymorphism = good (for reusability)
  - higher-order functions = useful
  - ...

PL Theory

- Precise description of program structure and behavior
  - operational semantics
  - lambda-calculus

- Language design is a pervasive activity

- Program correctness
  - Hoare Logic

- Program meaning and correctness are pervasive concerns

- Types

- Types are a pervasive technology

Oops, forgot one thing...

- The difficulty with teaching many of these topics is that they presuppose the ability to read and write mathematical proofs
- In a course for arbitrary computer science students, this turns out to be a really bad assumption

My List (II)

Proof!

- The ability to recognize and construct rigorous mathematical arguments

Sine qua non...

My List (II)

Proof!

- The ability to recognize and construct rigorous mathematical arguments

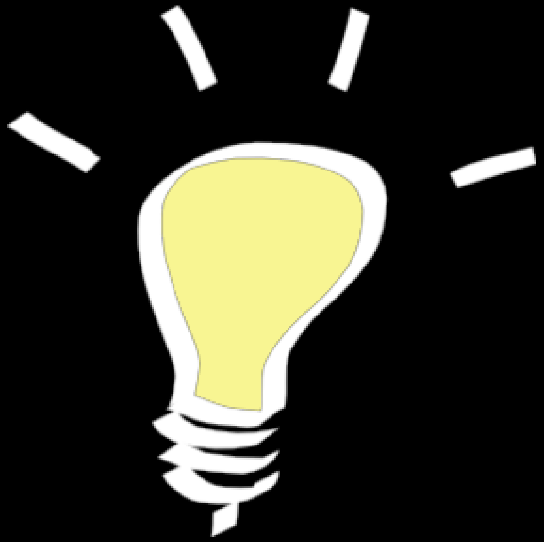
Sine qua non...

But...

Very hard to teach these skills effectively in a large class (while teaching anything else)

Requires an instructor-intensive feedback loop

A Bright Idea...



automated proof assistant

=

one TA per student

...With Major Consequences!

- Using a proof assistant completely shapes the way ideas are presented
  - Working "against the grain" is a *really bad idea*
- Learning to drive a proof assistant is a significant intellectual challenge

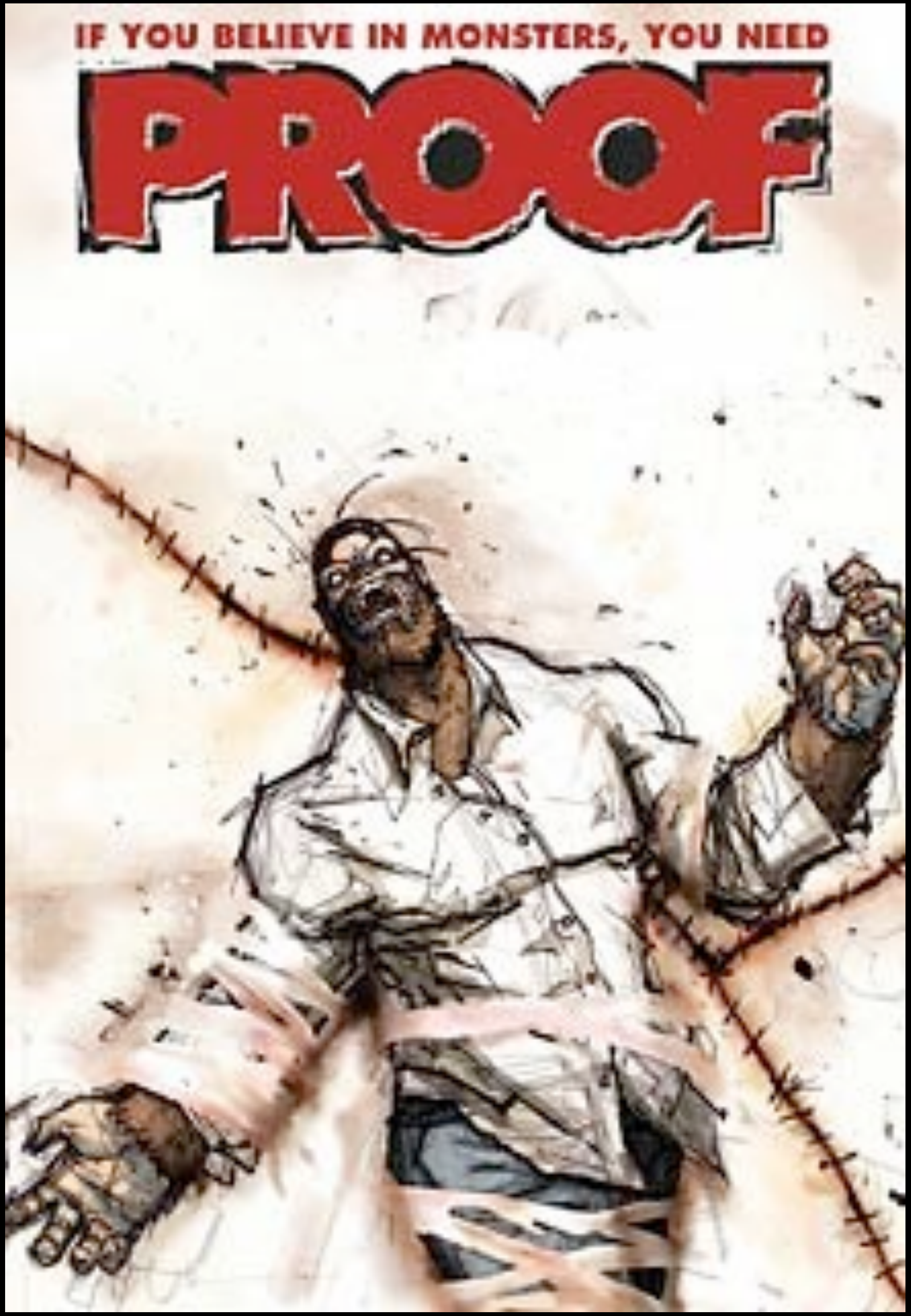
...With Major Consequences!

- Using a proof assistant completely shapes the way ideas are presented
  - Working "against the grain" is a *really bad idea*
- Learning to drive a proof assistant is a significant intellectual challenge

⇒ Restructure entire course
around the idea of proof

Any Questions?

Let’s talk…

What is  ?

explanation vs. proof

formal vs. informal

plausible
vs.
deductive

inductive vs. deductive

detailed vs. formal

intuition vs. knowledge

careful vs. rigorous

A Useful Distinction

Proofs optimized for conveying understanding

vs.

Proofs optimized for conveying certainty

A Useful Distinction

Very hard to teach! ↘

Proofs optimized for conveying <u>understanding</u>

vs.

Proofs optimized for conveying <u>certainty</u>

A Useful Distinction

Very hard to teach!

But addressed in lots of other courses

Proofs optimized for conveying <u>understanding</u>

vs.

Proofs optimized for conveying <u>certainty</u>

A Useful Distinction

Very hard to teach!

But addressed in lots of other courses

Proofs optimized for conveying understanding

vs.

Proofs optimized for conveying certainty

Critically needed for doing PL

A Useful Distinction

Very hard to teach!

But addressed in lots of other courses

Proofs optimized for conveying <u>understanding</u>

vs.

Proofs optimized for conveying <u>certainty</u>

<u>Not</u> adequately addressed elsewhere in the curriculum

Critically needed for doing PL

A Useful Distinction

Very hard to teach!

But addressed in lots of other courses

Proofs optimized for conveying <u>understanding</u>

vs.

Proofs optimized for conveying <u>certainty</u>

Possible to teach
(with tool support!)

<u>Not</u> adequately addressed
elsewhere in the curriculum

Critically needed for doing PL

A Spectrum of "Certainty Proofs"

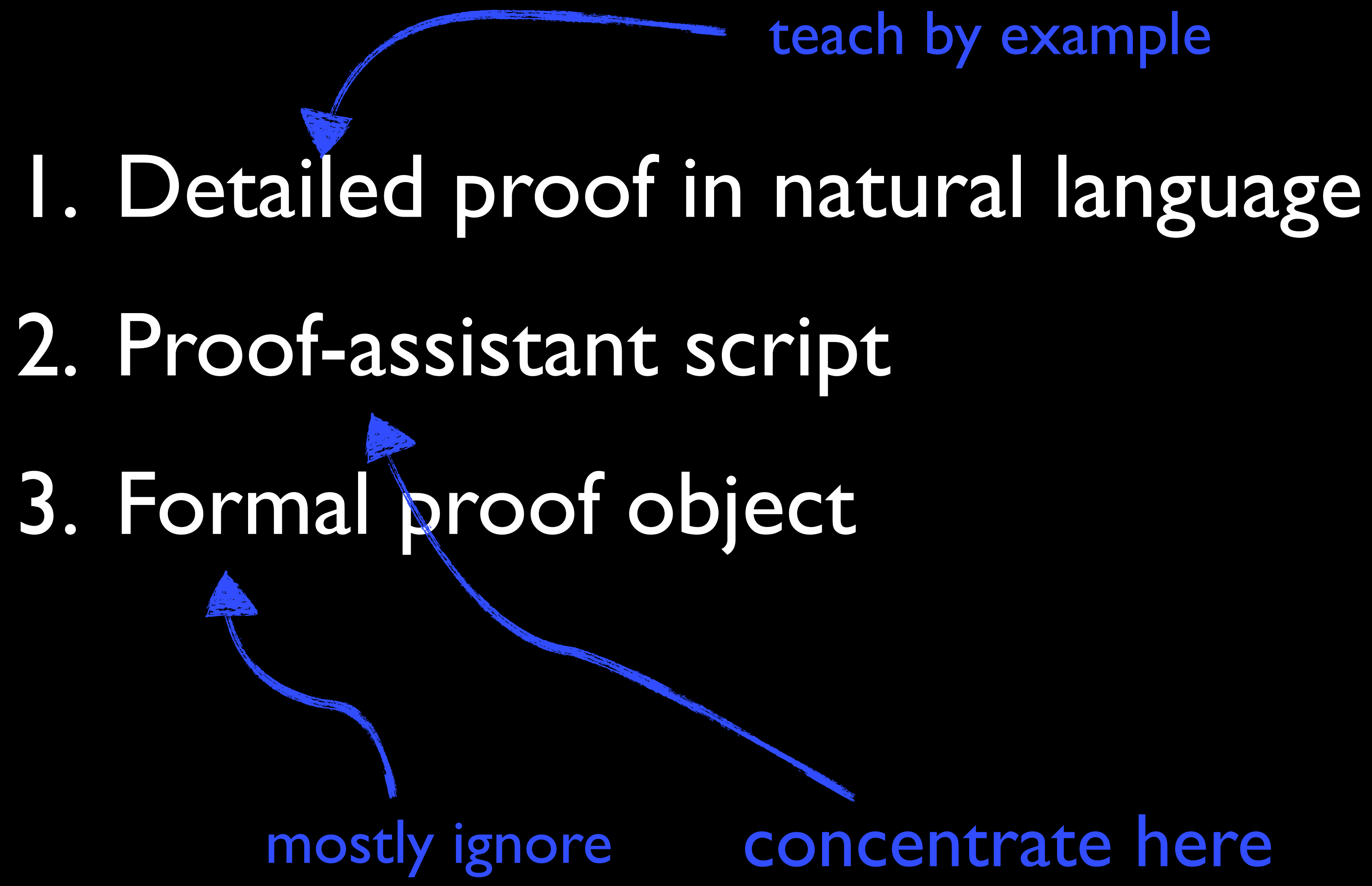
1. Detailed proof in natural language
2. Proof-assistant script
3. Formal proof object

instructions for writing...

program for constructing...

> "Certainty" is far from being a sign of success, it is only a symptom of lack of imagination, of conceptual poverty. It produces smug satisfaction and prevents the growth of knowledge. — Lakatos

A Spectrum of “Certainty Proofs”



teach by example

1. Detailed proof in natural language
2. Proof-assistant script
3. Formal proof object

mostly ignore

concentrate here

> “Certainty” is far from being a sign of success, it is only a symptom of lack of imagination, of conceptual poverty. It produces smug satisfaction and prevents the growth of knowledge. — Lakatos

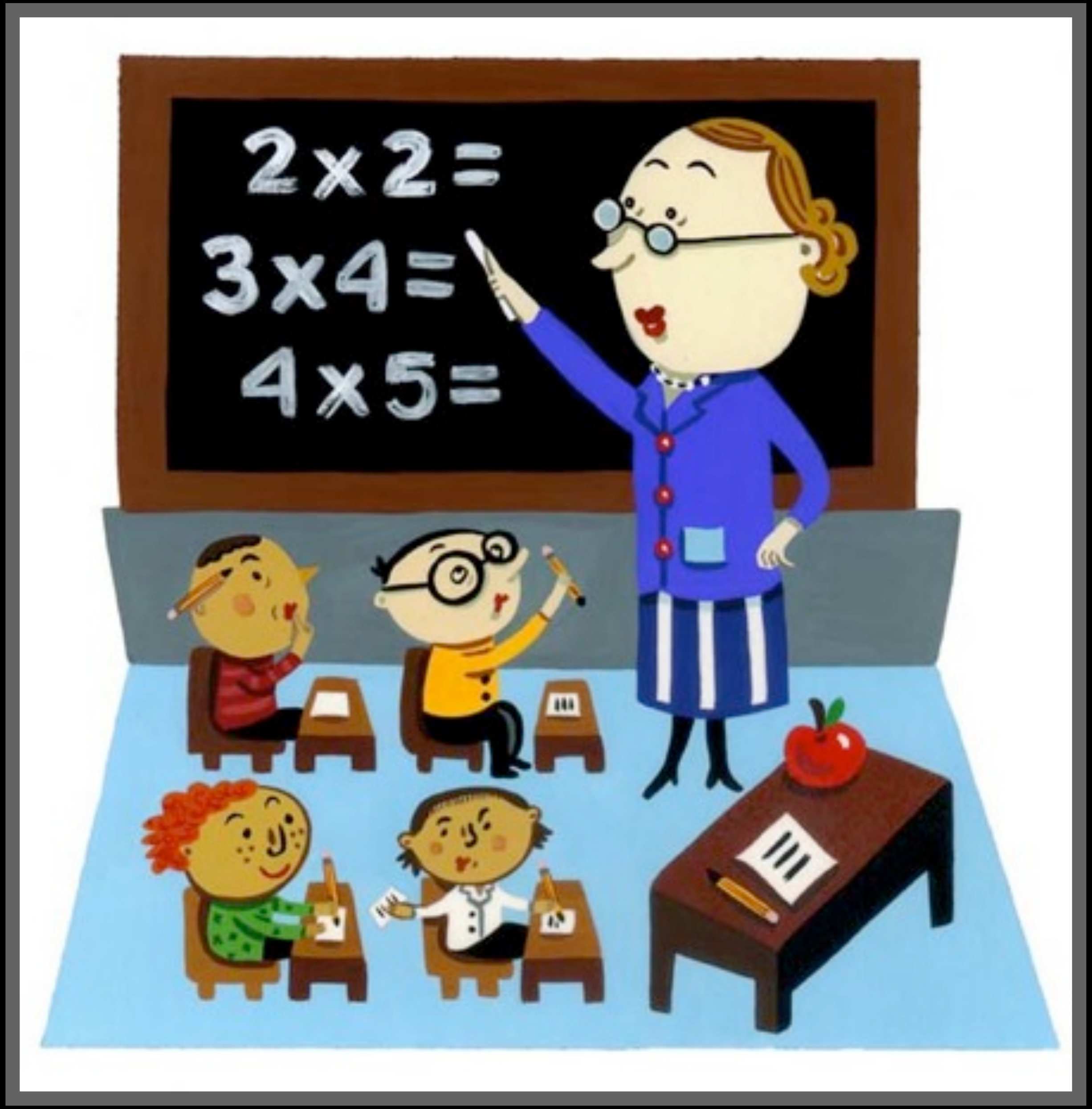
Goals

(ideally)

We would ^ like students to be able to

1. write correct definitions
2. make useful / interesting claims about them
3. verify their correctness (and find bugs)
4. write clear proofs demonstrating their correctness

The Course



Choosing One's Poison

Many proof assistants have been used to teach programming languages...

(usually to a narrower audience)

Isabelle
HOL
Coq
Tutch
SASyLF
Agda
ACL2
etc.

None is perfect

Choosing My Poison

Choosing My Poison

I chose Coq

Choosing My Poison

I chose Coq

- Curry-Howard gives a nice story, from FP through “programming with propositions”

Choosing My Poison

I chose Coq

- Curry-Howard gives a nice story, from FP through "programming with propositions"
- Mature tool

Choosing My Poison

I chose Coq

- Curry-Howard gives a nice story, from FP through "programming with propositions"
- Mature tool
- Automation

Choosing My Poison

I chose Coq

- Curry-Howard gives a nice story, from FP through “programming with propositions”
- Mature tool
- Automation
- Familiarity

Choosing My Poison

I chose Coq

- Curry-Howard gives a nice story, from FP through "programming with propositions"
- Mature tool
- Automation
- Familiarity
- Local expertise

Choosing My Poison

I chose Coq

- Curry-Howard gives a nice story, from FP through “programming with propositions”
- Mature tool
- Automation
- Familiarity
- Local expertise
- 

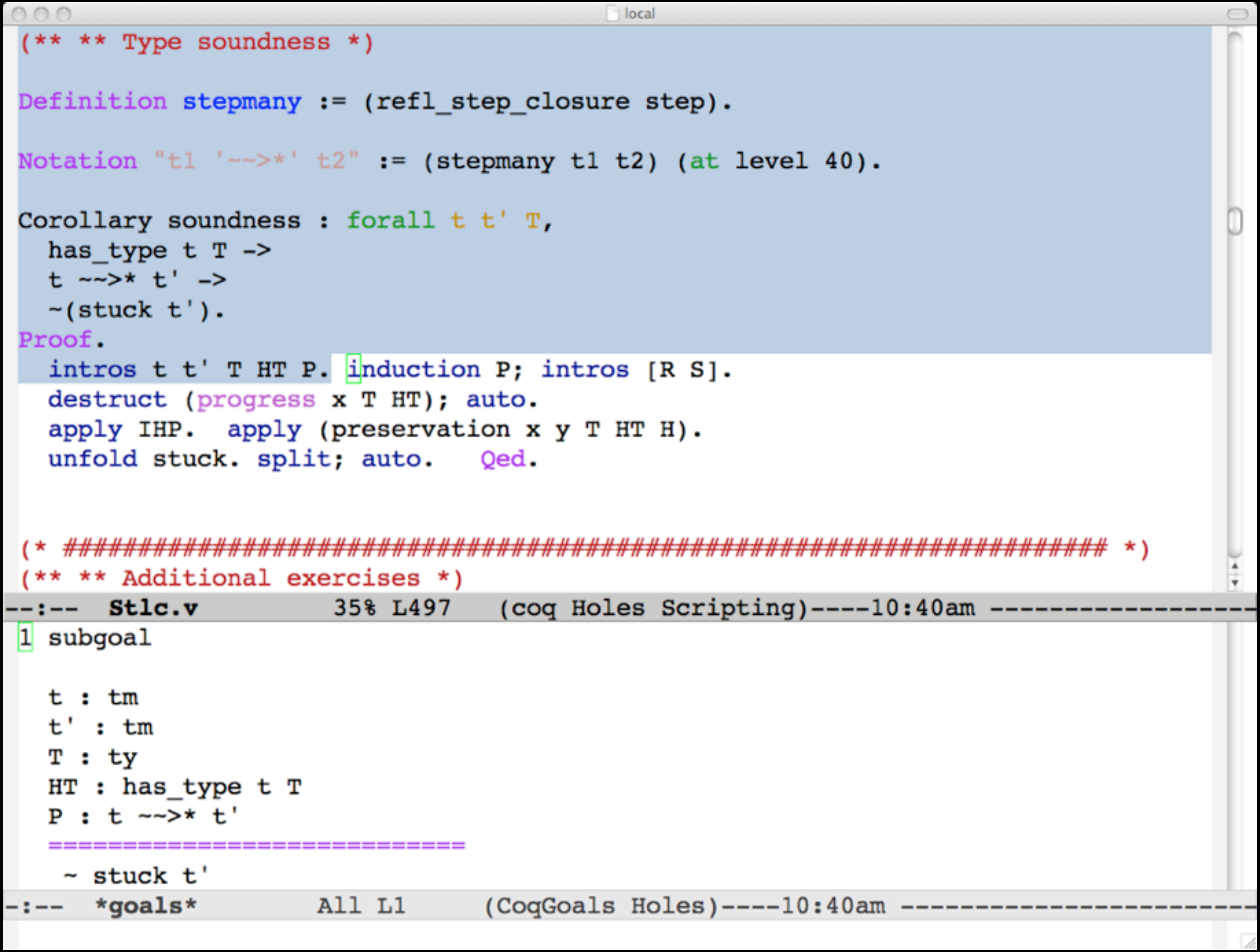
Choosing My Poison

I chose Coq

- Curry-Howard gives a nice story, from FP through "programming with propositions"
- Mature tool
- Automation
- Familiarity
- Local expertise
- 

And now that we've got the hard part out of the way...

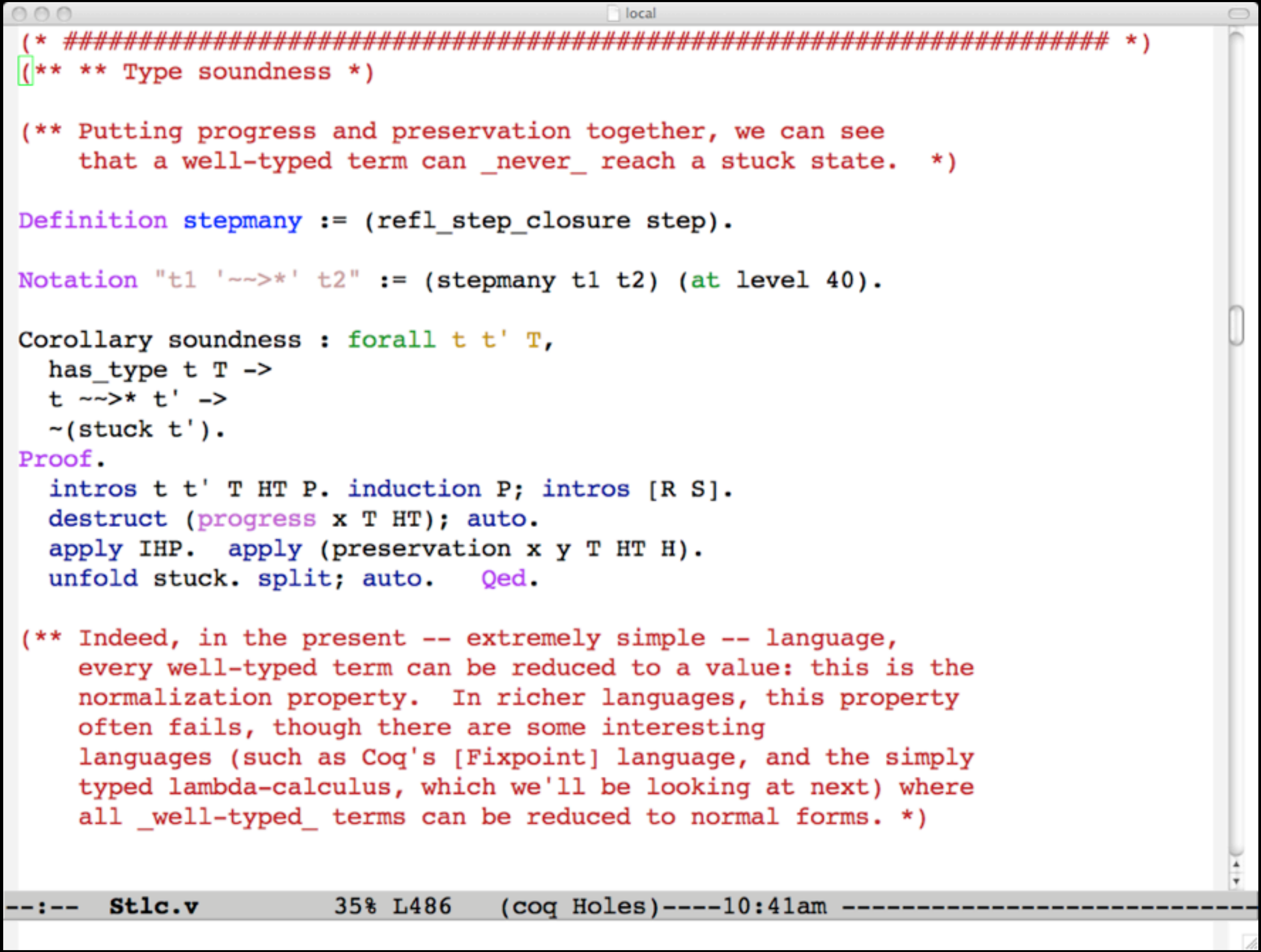
Interactive session in early lectures



```
(** ** Type soundness *)

Definition stepmany := (refl_step_closure step).

Notation "t1 '~~>*' t2" := (stepmany t1 t2) (at level 40).

Corollary soundness : forall t t' T,
  has_type t T ->
  t ~~>* t' ->
  ~(stuck t').
Proof.
  intros t t' T HT P. induction P; intros [R S].
  destruct (progress x T HT); auto.
  apply IHP.  apply (preservation x y T HT H).
  unfold stuck. split; auto.   Qed.


(* ###################################################################### *)
(** ** Additional exercises *)
--:--  Stlc.v          35% L497    (coq Holes Scripting)----10:40am ----------------
```

```
1 subgoal

  t : tm
  t' : tm
  T : ty
  HT : has_type t T
  P : t ~~>* t'
  ============================
   ~ stuck t'
-:--  *goals*         All L1     (CoqGoals Holes)----10:40am ---------------------
```

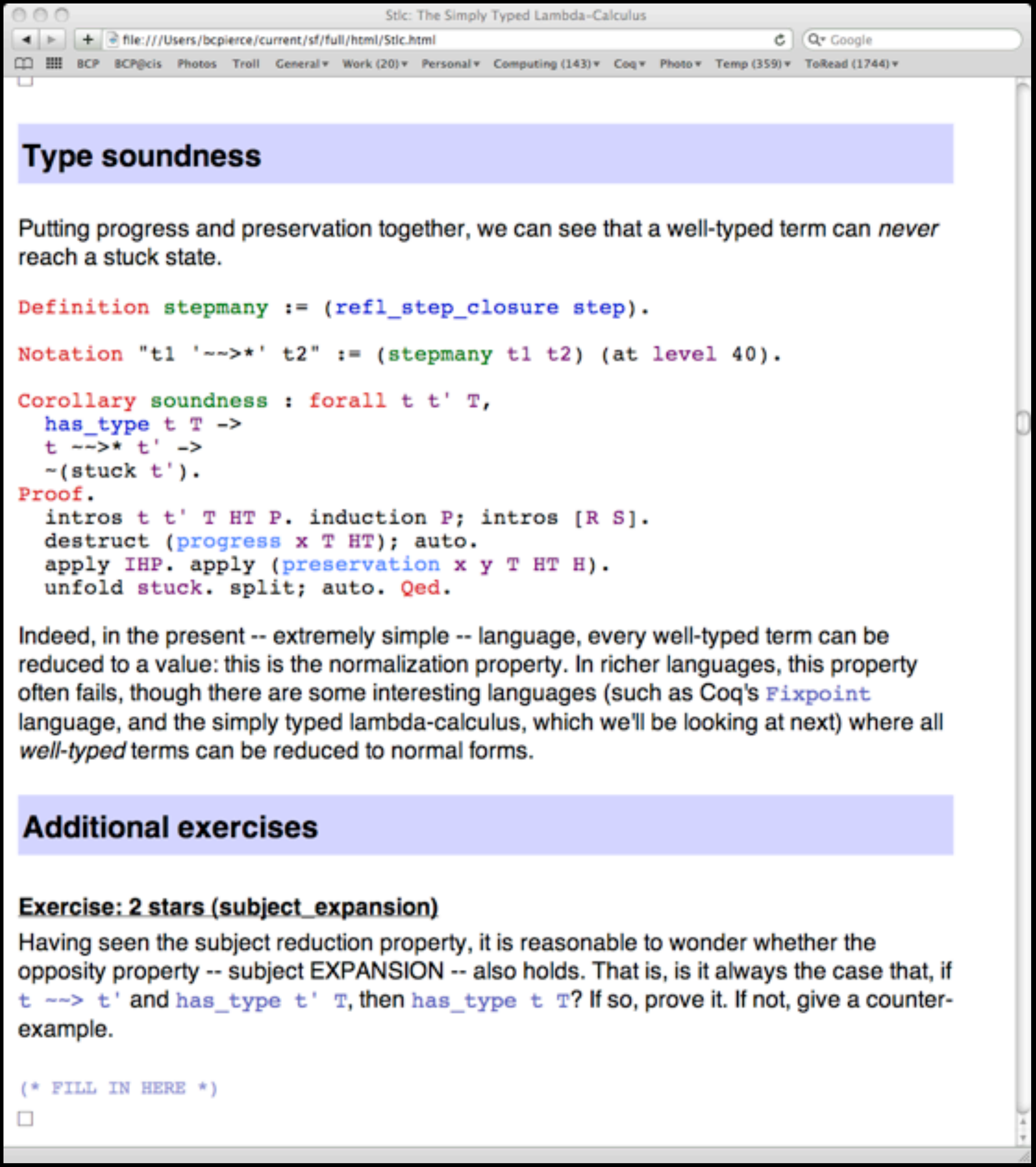
Expanded version for handouts and homework assignments



```
(* ###################################################################### *)
(** ** Type soundness *)

(** Putting progress and preservation together, we can see
    that a well-typed term can _never_ reach a stuck state.  *)

Definition stepmany := (refl_step_closure step).

Notation "t1 '~~>*' t2" := (stepmany t1 t2) (at level 40).

Corollary soundness : forall t t' T,
  has_type t T ->
  t ~~>* t' ->
  ~(stuck t').
Proof.
  intros t t' T HT P. induction P; intros [R S].
  destruct (progress x T HT); auto.
  apply IHP.  apply (preservation x y T HT H).
  unfold stuck. split; auto.   Qed.

(** Indeed, in the present -- extremely simple -- language,
    every well-typed term can be reduced to a value: this is the
    normalization property.  In richer languages, this property
    often fails, though there are some interesting
    languages (such as Coq's [Fixpoint] language, and the simply
    typed lambda-calculus, which we'll be looking at next) where
    all _well-typed_ terms can be reduced to normal forms. *)
```

```
--:--  Stlc.v         35% L486    (coq Holes)----10:41am ------------------------------
```

Typeset version for easier reading*



☐

Type soundness

Putting progress and preservation together, we can see that a well-typed term can *never* reach a stuck state.

```
Definition stepmany := (refl_step_closure step).

Notation "t1 '-->*' t2" := (stepmany t1 t2) (at level 40).

Corollary soundness : forall t t' T,
  has_type t T ->
  t -->* t' ->
  ~(stuck t').
Proof.
  intros t t' T HT P. induction P; intros [R S].
  destruct (progress x T HT); auto.
  apply IHP. apply (preservation x y T HT H).
  unfold stuck. split; auto. Qed.
```

Indeed, in the present -- extremely simple -- language, every well-typed term can be reduced to a value: this is the normalization property. In richer languages, this property often fails, though there are some interesting languages (such as Coq's `Fixpoint` language, and the simply typed lambda-calculus, which we'll be looking at next) where all *well-typed* terms can be reduced to normal forms.

Additional exercises

Exercise: 2 stars (subject_expansion)

Having seen the subject reduction property, it is reasonable to wonder whether the opposity property -- subject EXPANSION -- also holds. That is, is it always the case that, if `t --> t'` and `has_type t' T`, then `has_type t T`? If so, prove it. If not, give a counter-example.

```
(* FILL IN HERE *)
```

☐

*... in a web browser, with an index and hyperlinks to definitions

And check out:
Narrating Formal Proof,
Carst Tankink, Herman Geuvers and James McKinna, at UITP on Thursday...

Guided Tour

Course Overview

- Basic functional programming (and fundamental Coq tactics)
- Logic (and more Coq tactics)
- While programs and Hoare Logic
- Simply typed lambda-calculus
- References and store typing
- Subtyping

Cold Start

Start from bare, unadorned Coq

- No libraries
- Just inductive definitions, structural recursion, and (dependent, polymorphic) functions

Basics

Inductively define booleans, numbers, etc. Recursively define functions over them.

```
Inductive nat : Type :=
  | O : nat
  | S : nat -> nat.

Fixpoint plus (n : nat) (m : nat) {struct n} : nat :=
  match n with
    | O => m
    | S n' => S (plus n' m)
  end.
```

Restriction to structural recursion is not a big deal, provided we choose examples a bit carefully

Proof by Simplification

A few simple theorems can be proved just by beta-reduction...

```
Theorem plus_0_l : forall n:nat,  plus 0 n = n.

Proof. reflexivity. Qed.
```

Proof by Rewriting

A few more can be proved just by substitution using equality hypotheses.

```
Theorem plus_id_example : forall n m:nat,
  n = m -> plus n n = plus m m.

Proof.
  intros n m. (* move both quantifiers into the context *)
  intros H. (* move the hypothesis into the context *)
  rewrite -> H. (* Rewrite the goal using the hypothesis *)
  reflexivity. Qed.
```

Proof by Case Analysis

More interesting properties require case analysis...

```
Theorem plus_1_neq_0 : forall n,
  beq_nat (plus n 1) 0 = false.

Proof.
  intros n. destruct n as [| n'].
    reflexivity.
    reflexivity. Qed.
```

numeric comparison, returning a boolean

Proof by Induction

... or, more generally, induction

```
Theorem plus_0_r : forall n:nat, plus n 0 = n.

Proof.
  intros n. induction n as [| n'].
  Case "n = 0". reflexivity.
  Case "n = S n'". simpl. rewrite -> IHn'.
       reflexivity.
Qed.
```

Functional Programming

Similarly, we can define (as usual)

- lists, trees, etc.
- polymorphic functions (length, reverse, etc.)
- higher-order functions (map, fold, etc.)
- etc.

```
Inductive list (X:Type) : Type :=
  | nil : list X
  | cons : X -> list X -> list X.
```

Properties of Functional Programs

The handful of tactics we have already seen are enough to prove a a surprising range of properties of functional programs over lists, trees, etc.

```
Theorem map_rev : forall (X Y : Type) (f : X -> Y) (l : list X),
  map f (rev l) = rev (map f l).
```

A Few More Tactics

To go further, we need a few additional tactics...

- inversion
    - e.g., from [x]=[y] derive x=y
- generalizing induction hypotheses
- unfolding definitions

Programming with Propositions

"Coq has another universe, called `Prop`, where the types represent mathematical claims and their inhabitants represent evidence..."

Programming with Propositions

```
Definition true_for_zero (P:nat->Prop) : Prop :=
  P 0.

Definition true_for_n__true_for_Sn (P:nat->Prop) (n:nat) :
Prop :=
  P n -> P (S n).

Definition preserved_by_S (P:nat->Prop) : Prop :=
  forall n', P n' -> P (S n').

Definition true_for_all_numbers (P:nat->Prop) : Prop :=
  forall n, P n.

Definition nat_induction (P:nat->Prop) : Prop :=
     (true_for_zero P)
  -> (preserved_by_S P)
  -> (true_for_all_numbers P).


Theorem our_nat_induction_works : forall (P:nat->Prop),
  nat_induction P.
```

Logic

Familiar logical connectives can be built from Coq's primitive facilities...

```
Inductive and (A B : Prop) : Prop :=
  conj : A -> B -> (and A B).
```

Similarly: disjunction, negation, existential quantification, equality, ...

Inductively Defined Relations

```
Inductive le (n:nat) : nat -> Prop :=
  | le_n : le n n
  | le_S : forall m, (le n m) -> (le n (S m)).

Definition relation (X: Type) := X->X->Prop.

Definition reflexive (X: Type) (R: relation X) :=
  forall a : X, R a a.

Definition preorder (X:Type) (R: relation X) :=
  (reflexive R) /\ (transitive R).
```

Expressions

```
Inductive aexp : Type :=
   | ANum : nat -> aexp
   | APlus : aexp -> aexp -> aexp
   | AMinus : aexp -> aexp -> aexp
   | AMult : aexp -> aexp -> aexp.

Fixpoint aeval (e : aexp) {struct e} : nat :=
  match e with
   | ANum n => n
   | APlus a1 a2 => plus (aeval a1) (aeval a2)
   | AMinus a1 a2 => minus (aeval a1) (aeval a2)
   | AMult a1 a2 => mult (aeval a1) (aeval a2)
  end.
```

(Similarly boolean expressions)

Optimization

```
Fixpoint optimize_0plus (e:aexp) {struct e} : aexp :=
  match e with
  | ANum n => ANum n
  | APlus (ANum 0) e2 => optimize_0plus e2
  | APlus e1 e2 => APlus (optimize_0plus e1) (optimize_0plus e2)
  | AMinus e1 e2 => AMinus (optimize_0plus e1) (optimize_0plus e2)
  | AMult e1 e2 => AMult (optimize_0plus e1) (optimize_0plus e2)
  end.
```

```
Theorem optimize_0plus_sound: forall e,
  aeval (optimize_0plus e) = aeval e.

Proof.
  intros e. induction e.
  Case "ANum". reflexivity.
  Case "APlus". destruct e1.
    SCase "e1 = ANum n". destruct n.
      SSCase "n = 0". simpl. apply IHe2.
      SSCase "n <> 0". simpl. rewrite IHe2. reflexivity.
    SCase "e1 = APlus e1_1 e1_2".
      simpl. simpl in IHe1. rewrite IHe1. rewrite IHe2. reflexivity.
    SCase "e1 = AMinus e1_1 e1_2".
      simpl. simpl in IHe1. rewrite IHe1. rewrite IHe2. reflexivity.
    SCase "e1 = AMult e1_1 e1_2".
      simpl. simpl in IHe1. rewrite IHe1. rewrite IHe2. reflexivity.
  Case "AMinus".
    simpl. rewrite IHe1. rewrite IHe2. reflexivity.
  Case "AMult".
    simpl. rewrite IHe1. rewrite IHe2. reflexivity. Qed.
```

Automation

At this point, we begin introducing some simple automation facilities.

(As we go on further and proofs become longer, we gradually introduce more powerful forms of automation.)

```
Theorem optimize_0plus_sound'': forall e,
  aeval (optimize_0plus e) = aeval e.
Proof.
  intros e.
  induction e;
    (* Most cases follow directly by the IH *)
    try (simpl; rewrite IHe1; rewrite IHe2; reflexivity);
    (* ... or are immediate by definition *)
    try (reflexivity).
  (* The interesting case is when e = APlus e1 e2. *)
  Case "APlus".
    destruct e1;
      try (simpl; simpl in IHe1; rewrite IHe1; rewrite IHe2; reflexivity).
    SCase "e1 = ANum n". destruct n.
      SSCase "n = 0". apply IHe2.
      SSCase "n <> 0". simpl. rewrite IHe2. reflexivity. Qed.
```

While Programs

```
Inductive com : Type :=
   | CSkip : com
   | CAss  : id -> aexp -> com
   | CSeq  : com -> com -> com
   | CIf  : bexp -> com -> com -> com
   | CWhile : bexp -> com -> com.
```

```
Notation "'SKIP'" :=
  CSkip.
Notation "c1 ; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "l '::=' a" :=
  (CAss l a) (at level 60).
Notation "'WHILE' b 'DO' c 'LOOP'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IF' e1 'THEN' e2 'ELSE' e3" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
```

With a bit of notation hacking...

```
Definition factorial : com :=
  Z ::= !X;
  Y ::= A1;
  WHILE BNot (!Z === A0) DO
    Y ::= !Y *** !Z;
    Z ::= !Z --- A1
  LOOP.
```

Program Equivalence

```
Definition cequiv (c1 c2 : com) : Prop :=
  forall (st st':state), (c1 / st ~~> st') <-> (c2 / st ~~> st').
```

Definitions and basic properties

- “program equivalence is a congruence”

Case study: constant folding

Hoare Logic

Assertions

Hoare triples

Weakest preconditions

Proof rules

- Proof rule for assignment
- Rules of consequence
- Proof rule for SKIP
- Proof rule for ;
- Proof rule for conditionals
- Proof rule for loops

Using Hoare Logic to reason about programs

- e.g. correctness of factorial program

Small-Step Operational Semantics

At this point we switch from big-step to small-step style (and, for good measure, show their equivalence).

Types

Fundamentals

- Typed arithmetic expressions

Simply typed lambda-calculus

Properties

- Free variables
- Substitution
- Preservation
- Progress
- Uniqueness of types

Typechecking algorithm

The POPLMark Tarpit

The POPLMark Tarpit

- Dealing carefully with variable binding is hard; doing it formally is even harder

The POPLMark Tarpit

- Dealing carefully with variable binding is hard; doing it formally is even harder
- What to do?

The POPLMark Tarpit

- Dealing carefully with variable binding is hard; doing it formally is even harder
- What to do?
  - DeBruijn indices?

The POPLMark Tarpit

- Dealing carefully with variable binding is hard; doing it formally is even harder
- What to do?
  - DeBruijn indices?
  - Locally Nameless?

The POPLMark Tarpit

- Dealing carefully with variable binding is hard; doing it <u>formally</u> is even harder
- What to do?
    - DeBruijn indices?
    - Locally Nameless?
    - Switch to Isabelle? Twelf?

The POPLMark Tarpit

- Dealing carefully with variable binding is hard; doing it formally is even harder
- What to do?
  - DeBruijn indices?
  - Locally Nameless?
  - Switch to Isabelle? Twelf?
  - Finesse the problem!

A Cheap Solution

A Cheap Solution

- Observation: If we only ever substitute closed terms, then capture-incurring and capture-avoiding substitution behave the same.

A Cheap Solution

- Observation: If we only ever substitute closed terms, then capture-incurring and capture-avoiding substitution behave the same.
- Second observation [Tolmach]: Replacing the standard weakening+permutation with a "context invariance" lemma makes this presentation *very* clean.

A Cheap Solution

- Observation: If we only ever substitute closed terms, then capture-incurring and capture-avoiding substitution behave the same.
- Second observation [Tolmach]: Replacing the standard weakening+permutation with a “context invariance” lemma makes this presentation *very* clean.
- Downside: Doesn’t work for System F

Subtyping

- Records
- Subtyping relation
- Properties

Outcomes

The Fear

Old syllabus:

- inductive definitions
- operational semantics
- untyped λ-calculus
- simply typed λ-calculus
- references
- exceptions
- records and subtyping
- Featherweight Java

New syllabus

- Coq

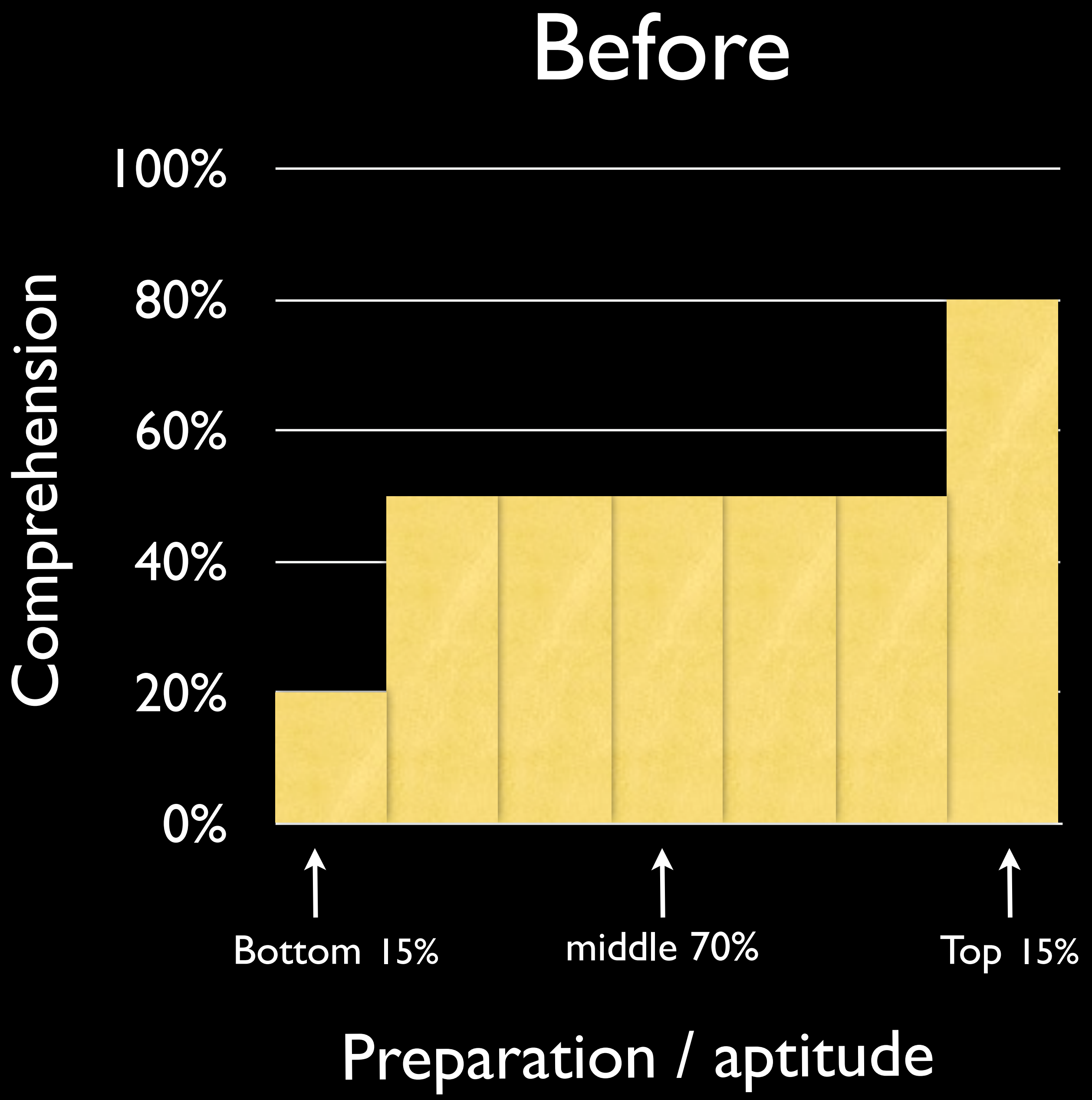
The Actuality

Old syllabus:

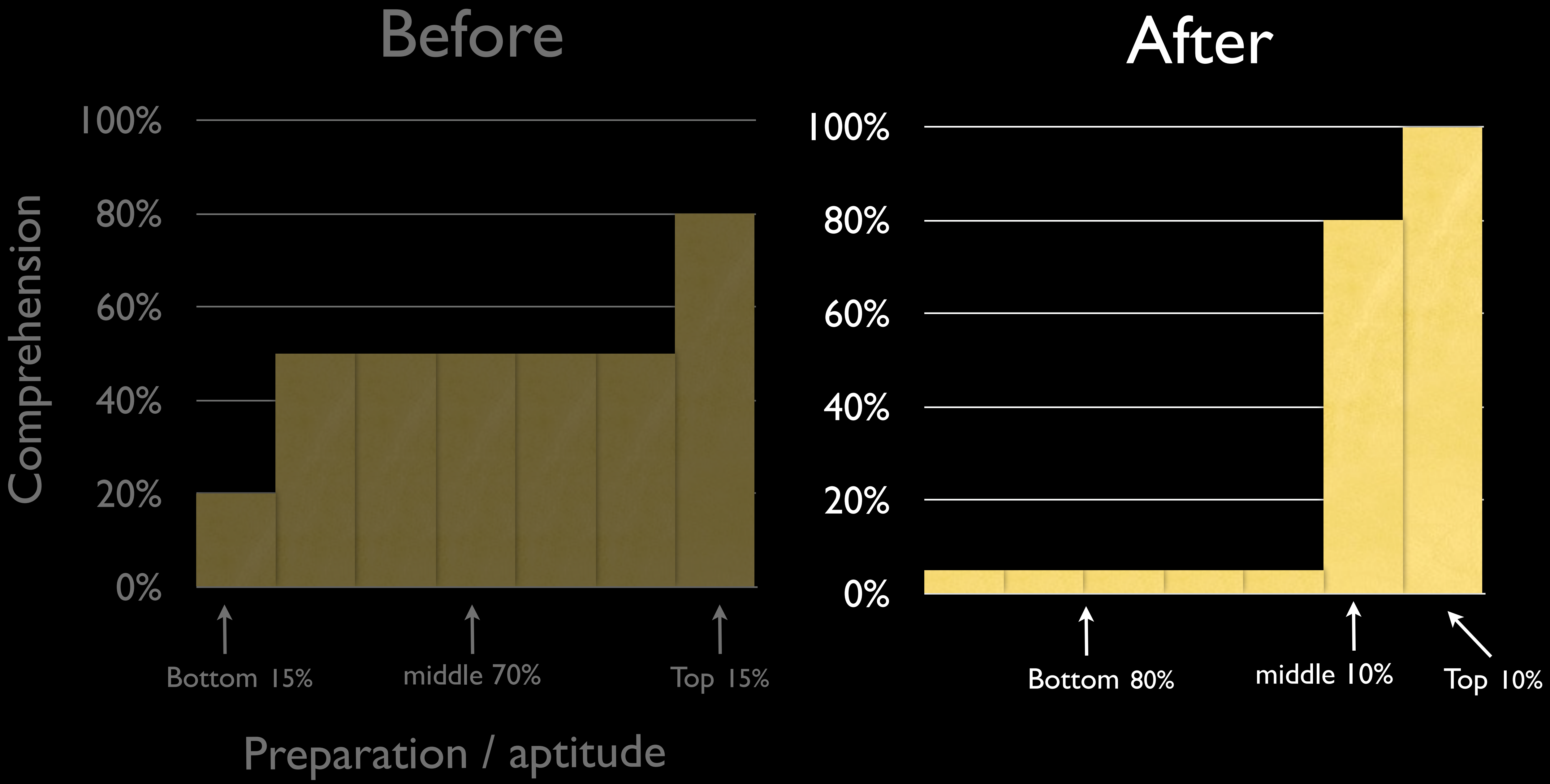
- inductive definitions
- operational semantics
- ~~untyped λ-calculus~~
- simply typed λ-calculus
- references
- ~~exceptions~~
- records and subtyping
- ~~Featherweight Java~~
- functional programming
- logic (and Curry-Howard)
- `while` programs
- program equivalence
- Hoare Logic
- Coq

New syllabus

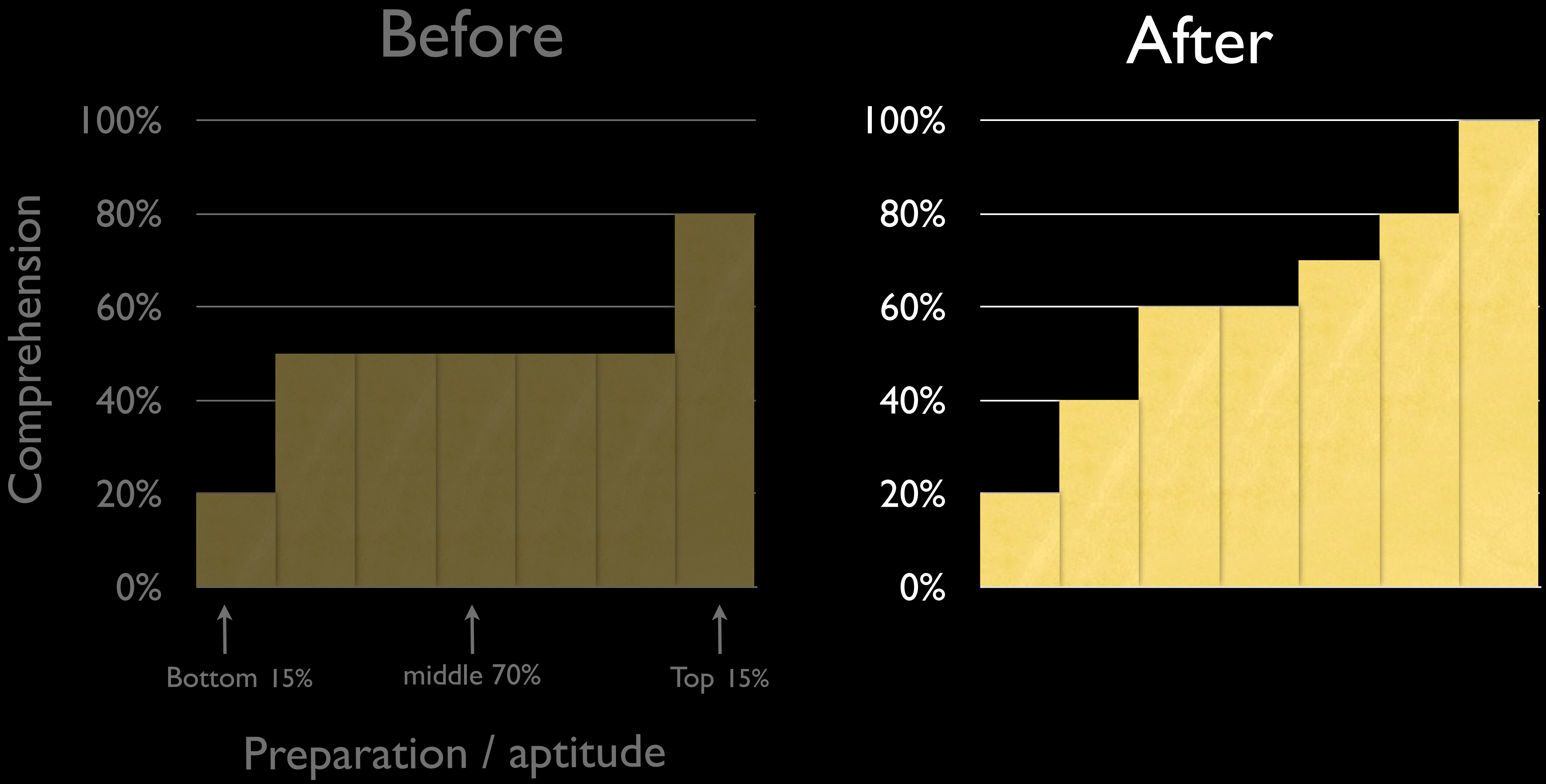
The Fear


Before
100%
80%
60%
40%
20%
0%
Comprehension
Bottom 15%
middle 70%
Top 15%
Preparation / aptitude

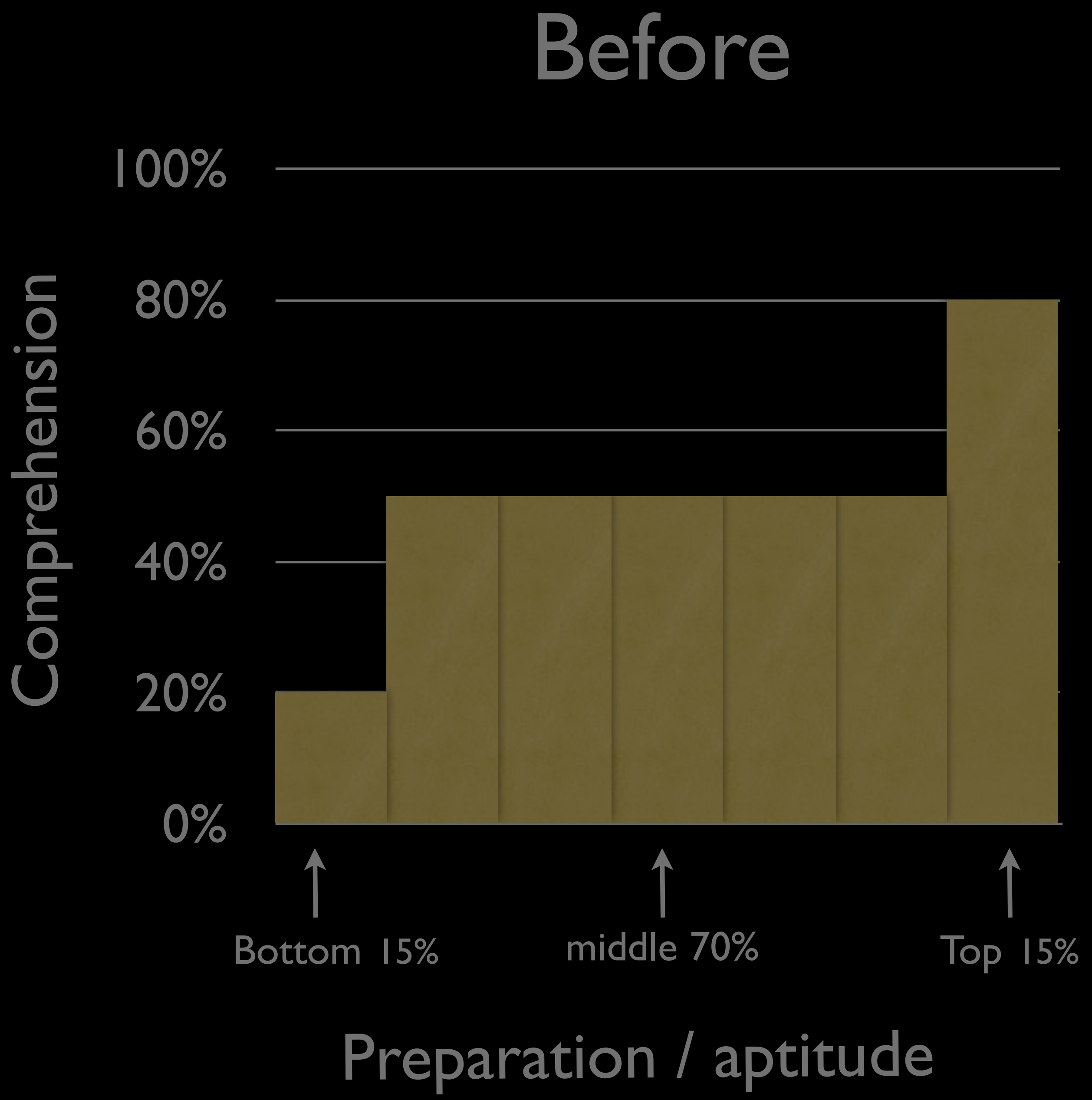
The Fear

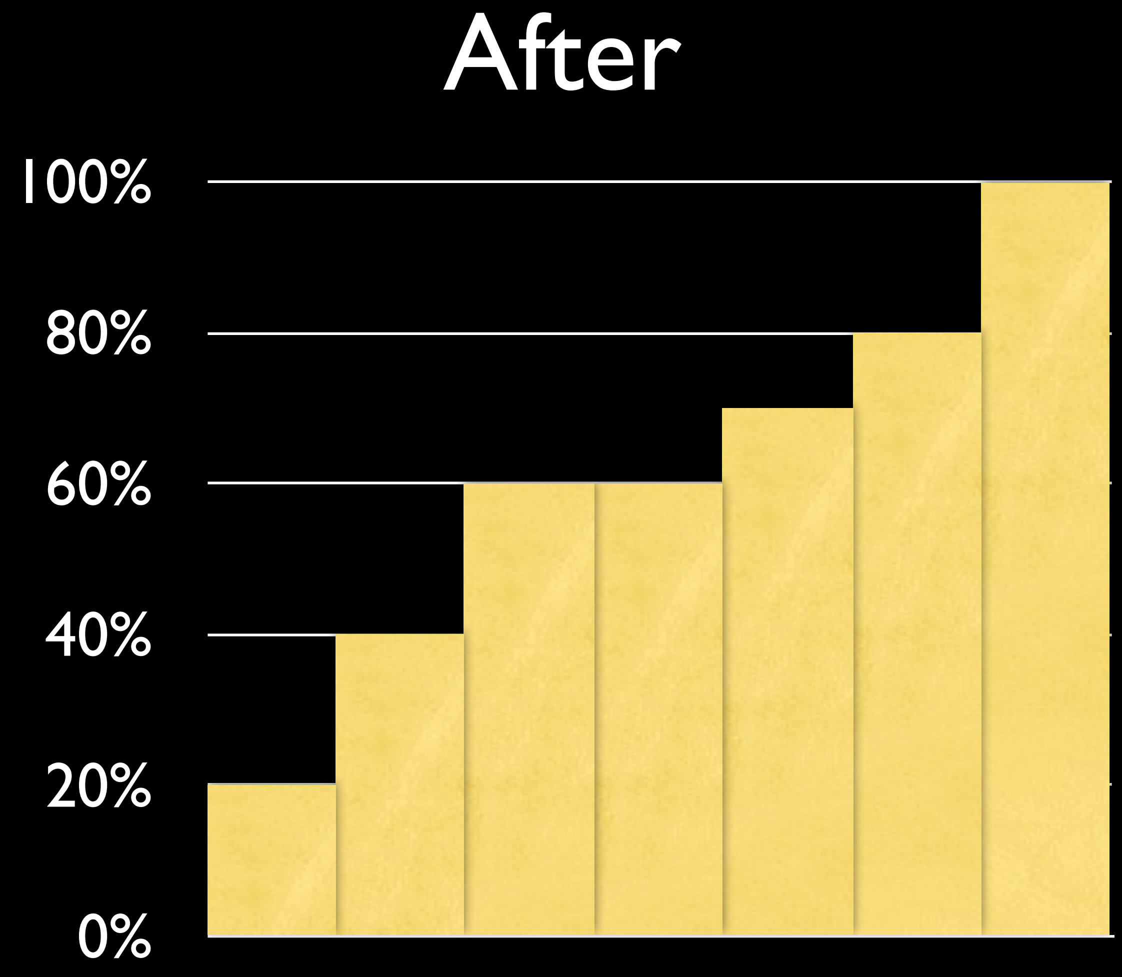

Before
100%
80%
60%
40%
20%
0%
Comprehension
Bottom 15%
middle 70%
Top 15%
Preparation / aptitude
After
100%
80%
60%
40%
20%
0%
Bottom 80%
middle 10%
Top 10%

The Actuality



The Actuality





in fact, students typically *performed better on* ***paper*** *exams* than in pre-Coq offerings of the course

What About Those Goals?

We would like students to be able to

1. write correct definitions
2. make useful / interesting claims about them
3. verify their correctness
    1. by hand
    2. by writing proof scripts
4. write clear proofs of their correctness

What About Those Goals?

pretty well

We would like students to be able to

1. write correct definitions
2. make useful / interesting claims about them
3. verify their correctness
    1. by hand
    2. by writing proof scripts
4. write clear proofs of their correctness

What About Those Goals?

pretty well

pretty well

We would like students to be able to

1. write correct definitions
2. make useful / interesting claims about them
3. verify their correctness
    1. by hand
    2. by writing proof scripts
4. write clear proofs of their correctness

What About Those Goals?

pretty well

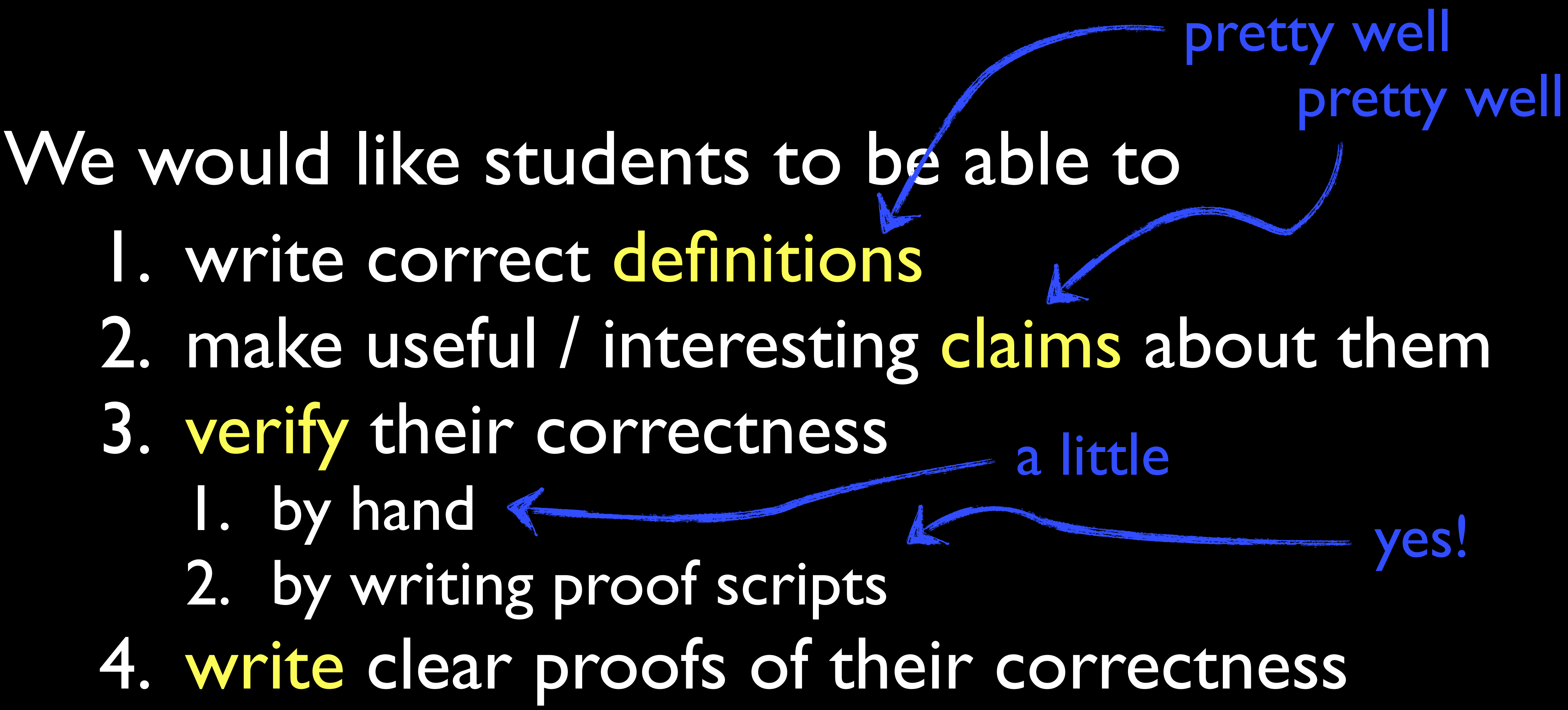
pretty well

We would like students to be able to

1. write correct definitions
2. make useful / interesting claims about them
3. verify their correctness
    1. by hand
    2. by writing proof scripts
4. write clear proofs of their correctness

yes!

What About Those Goals?



pretty well

pretty well

We would like students to be able to

1. write correct definitions
2. make useful / interesting claims about them
3. verify their correctness
    1. by hand
    2. by writing proof scripts
4. write clear proofs of their correctness

a little

yes!

What About Those Goals?

pretty well

pretty well

We would like students to be able to

1. write correct definitions
2. make useful / interesting claims about them
3. verify their correctness
   1. by hand
   2. by writing proof scripts
4. write clear proofs of their correctness

a little

yes!

imperfectly

One small catch...

Making up lectures and homeworks takes between one and two orders of magnitude more work for the instructor than a paper-and-pencil presentation of the same material!

Is Coq The Ultimate TA?

Pros:

- Can really build everything we need from scratch
- Curry-Howard → nice unifying story
    - Proving = programming

Is Coq The Ultimate TA?

Pros:

- Can really build everything we need from scratch
- Curry-Howard → nice unifying story
    - Proving = programming

Cons:

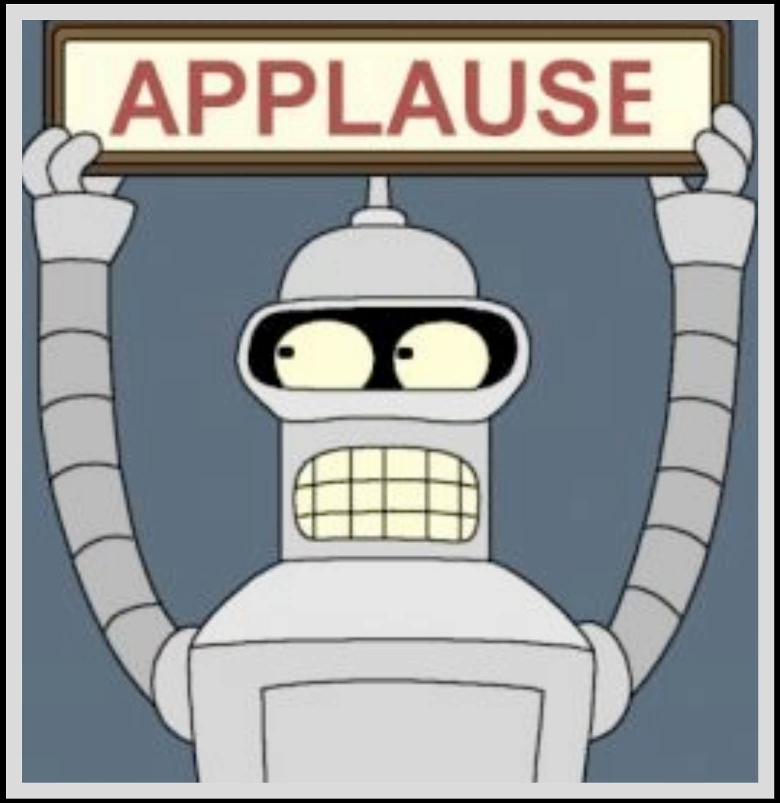
- Curry-Howard
    - Proving = programming → deep waters
    - Constructive logic can be confusing to students
- Annoyances
    - Lack of animation facilities
    - "User interface"
        - Notation facilities

> `My Coq proof scripts do not have the conciseness and elegance of Jérôme Vouillon's.  Sorry, I've been using Coq for only 6 years...`
>
> `– Leroy (2005)`

Bottom Line...

Bottom Line...

It works!



Want to

?

Use Our Materials

- The course has been taught successfully at several places (Penn three times, Maryland, Portland State, Princeton, UCSD, Purdue, and the Oregon PL Summer School...)
- Full text of the notes (minus solutions) are publicly available as Coq scripts and HTML files:

http://www.cis.upenn.edu/~bcpierce/sf

Improve Our Materials

Textbook model

- fixed (small) set of authors
- printed on paper
- limited scope
- new version every couple of years

OSS model

- electronic distribution
- many contributors (around a core group)
- extensible
- new versions as needed

If you are teaching from these materials and want *write access* to the SVN repo, just email me

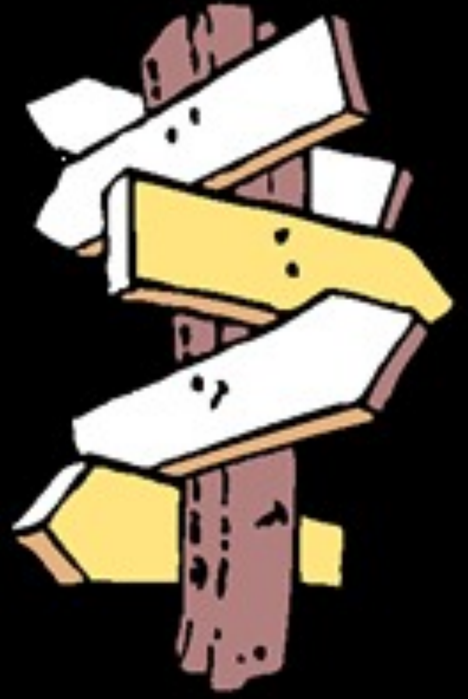
Adapt Our Materials

- Think this course would work better in Isabelle, Agda, ACL2, ...?
- Go for it!

Ignore Our Materials

and do it your own way!

- The Software Foundations course is an existence proof
- Plenty of room for competing efforts



What Next?

WORLD DOMINATION

Soon you will all bow before me.

Thin End of the Wedge:
Compilers

- Verified compilers are becoming a hot topic
  - Impressive recent achievements
  - Easy to see why it's important
- Beautiful expositions exist
  - e.g. Xavier Leroy's lecture notes from 2010 OPLSS
- Looks like a wonderful way to teach compilers

The Big Game:
Undergrad Discrete Math

Similar issues:

- Students come into discrete math courses (at least in the U.S.) with little or no idea of "what is a proof"
- Insufficient instructor resources to give every student continuous feedback

The Big Game:
Undergrad Discrete Math

Similar issues:

- Students come into discrete math courses (at least in the U.S.) with little or no idea of "what is a proof"
- Insufficient instructor resources to give every student continuous feedback

But not identical!

- Much less time — must keep overhead lower
- *Informal* proof skills equally important
- Broader range of relevant math (number theory, graph theory, discrete probability...)

Thank you!

SF courseware co-authors:
Chris Casinghino, Michael Greenberg, Vilhelm Sjöberg, Brent Yorgey

More contributors:
Andrew W. Appel, Jeffrey Foster, Michael Hicks, Ranjit Jhala, Greg Morrisett, Leonid Spesivtsev, and Andrew Tolmach

http://www.cis.upenn.edu/~bcpierce/sf/