Lambda, The Ultimate TA

Using a Proof Assistant to Teach Programming Language Foundations

ICFP 2009

Benjamin C. Pierce
University of Pennsylvania

A Wake-Up Call

From a recent email exchange with a textbook editor...

SIGCSE = ACM Special Interest Group on Computer Science Education

"At SIGCSE this year, someone mentioned to me that the programming languages course is in danger of disappearing from the CS curriculum. Is there any truth to this? I also heard there was talk about this at a recent SIGPLAN meeting. Is the course in danger at your school?"

Is The Sky Falling?

Is The Sky Falling?

- I don’t think so

Is The Sky Falling?

- I don't think so
- There are a lot of great ideas in our community, and their impact in the wider world is increasing, not decreasing
  - Witness Haskell, F#, Scala, ... (not to mention many bits of Java and C#)

Is The Sky Falling?

- I don't think so
- There are a lot of great ideas in our community, and their impact in the wider world is increasing, not decreasing
  - Witness Haskell, F#, Scala, ... (not to mention many bits of Java and C#)
- However, I do think we're not doing a good enough job on packaging our ideas in a form that seems optimally relevant or compelling to our students

Is The Sky Falling?

- I don't think so
- There are a lot of great ideas in our community, and their impact in the wider world is increasing, not decreasing
  - Witness Haskell, F#, Scala, ... (not to mention many bits of Java and C#)
- However, I do think we're not doing a good enough job on packaging our ideas in a form that seems optimally relevant or compelling to our students

Innovate or die...

One Small Step

One Small Step

From...

Theory of PL
for PL geeks

One Small Step

From...

Theory of PL
for PL geeks

To...

Software Foundations
for the masses

What / Why?

- What belongs in a course on “Software Foundations for the masses”?
- Why do the masses need to know it?

My List

My List

Logic

- Inductively defined relations
- Inductive proof techniques

My List

Logic

- Inductively defined relations
- Inductive proof techniques

Functional Programming

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

My List

Logic

- Inductively defined relations
- Inductive proof techniques

Functional Programming

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

PL Theory

- Precise description of program structure and behavior
  - operational semantics
  - lambda-calculus
- Program correctness
  - Hoare Logic
- Types

My List

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, ...

PL Theory

- Precise description of program structure and behavior
  - operational semantics
  - lambda-calculus
- Program correctness
  - Hoare Logic
- Types

My List

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
- Program correctness
  - Hoare Logic
- Types

My List

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
- Program correctness
  - Hoare Logic
- Types

- Language design is a pervasive activity
- Program meaning and correctness are pervasive concerns
- 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 appears to be a really bad assumption.

My List (II)

Proof!

- The ability to recognize and construct rigorous mathematical arguments

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...

My List (II)

Proof!

- The ability to recognize and construct rigorous mathematical arguments

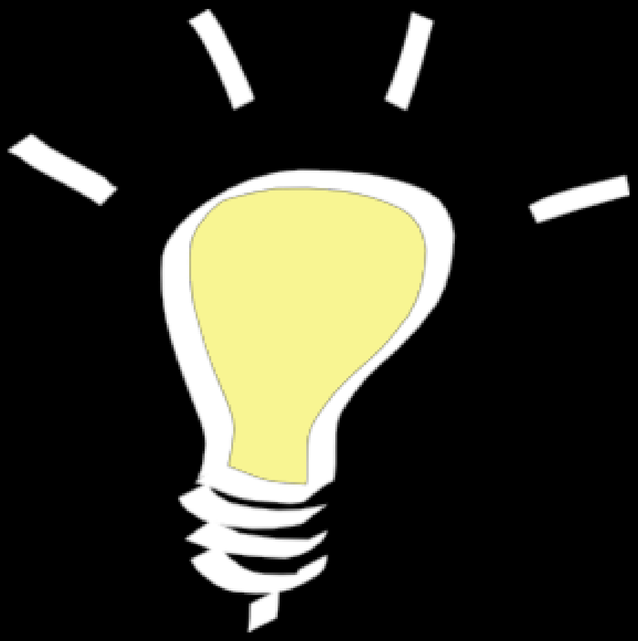
Sine qua non...

But...

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

Requires an instructor-intensive feedback loop

automated proof assistant



automated proof assistant

=

automated proof assistant

=

one TA per student

One Giant Leap!

- 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

One Giant Leap!

- 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

$\Rightarrow$ Restructure entire course around the idea of proof

What is a Proof?

formal vs. informal

formal vs. informal

plausible
vs.
deductive

formal vs. informal

plausible
vs.
deductive

inductive vs. deductive

formal vs. informal

plausible
vs.
deductive

inductive vs. deductive

careful vs. rigorous

formal vs. informal

plausible
vs.
deductive

inductive vs. deductive

detailed vs. formal

careful vs. rigorous

explanation vs. proof

formal vs. informal

plausible
vs.
deductive

inductive vs. deductive

detailed vs. formal

careful vs. rigorous

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 <u>understanding</u>

vs.

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

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 <u>understanding</u>

vs.

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

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

Varieties of “Certainty Proofs”

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

Varieties of “Certainty Proofs”

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

instructions for writing...

Varieties of "Certainty Proofs"

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

instructions for writing...

program for constructing...

Varieties of “Certainty Proofs”

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

Varieties of “Certainty Proofs”

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

concentrate here

Varieties of "Certainty Proofs"

teach by example

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

concentrate here

Varieties of "Certainty Proofs"

teach by example

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

mostly ignore

concentrate here

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 Software Foundations Course

Parameters

- 40-70 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 -- solutions <u>not</u> easily available)

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"
- Automation

Choosing My Poison

I chose Coq

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

Choosing My Poison

I chose Coq

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

Choosing My Poison

I chose Coq

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

Choosing My Poison

I chose Coq

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

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

Overview

- Basic functional programming (and fundamental Coq tactics)
- Logic (and more Coq tactics)
- While programs and Hoare Logic
- Simply typed lambda-calculus
- Records and subtyping

Interactive session in lecture

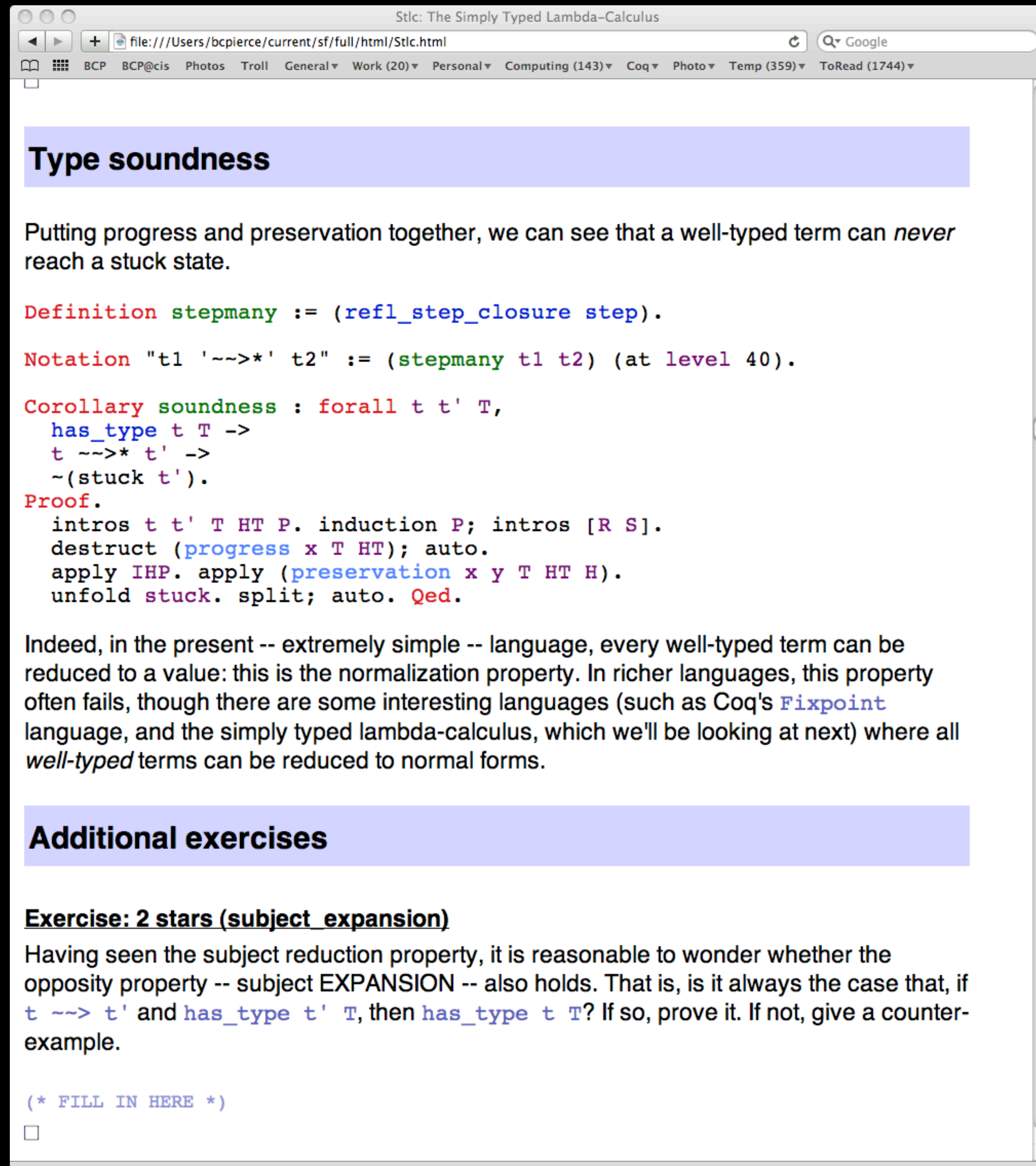
local

```
(** ** 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 variants for easier reading*


Stlc: The Simply Typed Lambda-Calculus

file:///Users/bcpierce/current/sf/full/html/Stlc.html

BCP BCP@cis Photos Troll General▾ Work (20)▾ Personal▾ Computing (143)▾ Coq▾ Photo▾ Temp (359)▾ ToRead (1744)▾

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

Outcomes

Old (Paper-and-Pencil) Syllabus

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

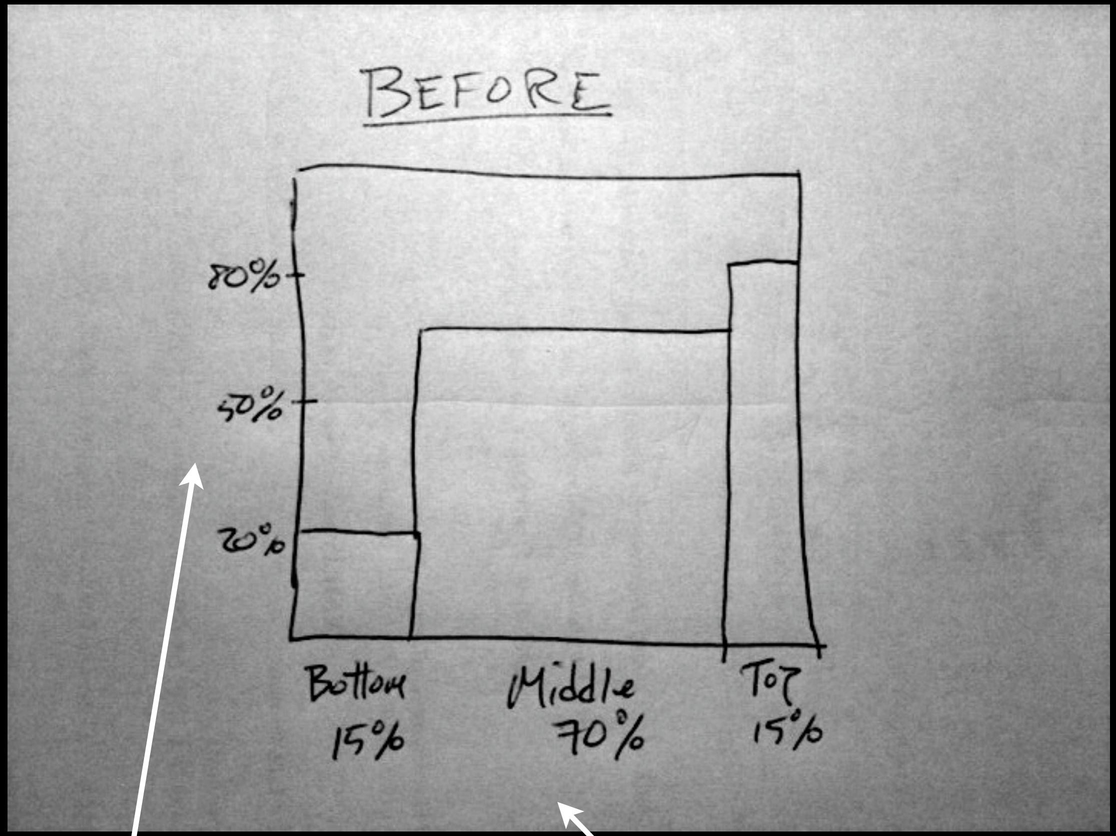
New Syllabus

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

New Syllabus

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

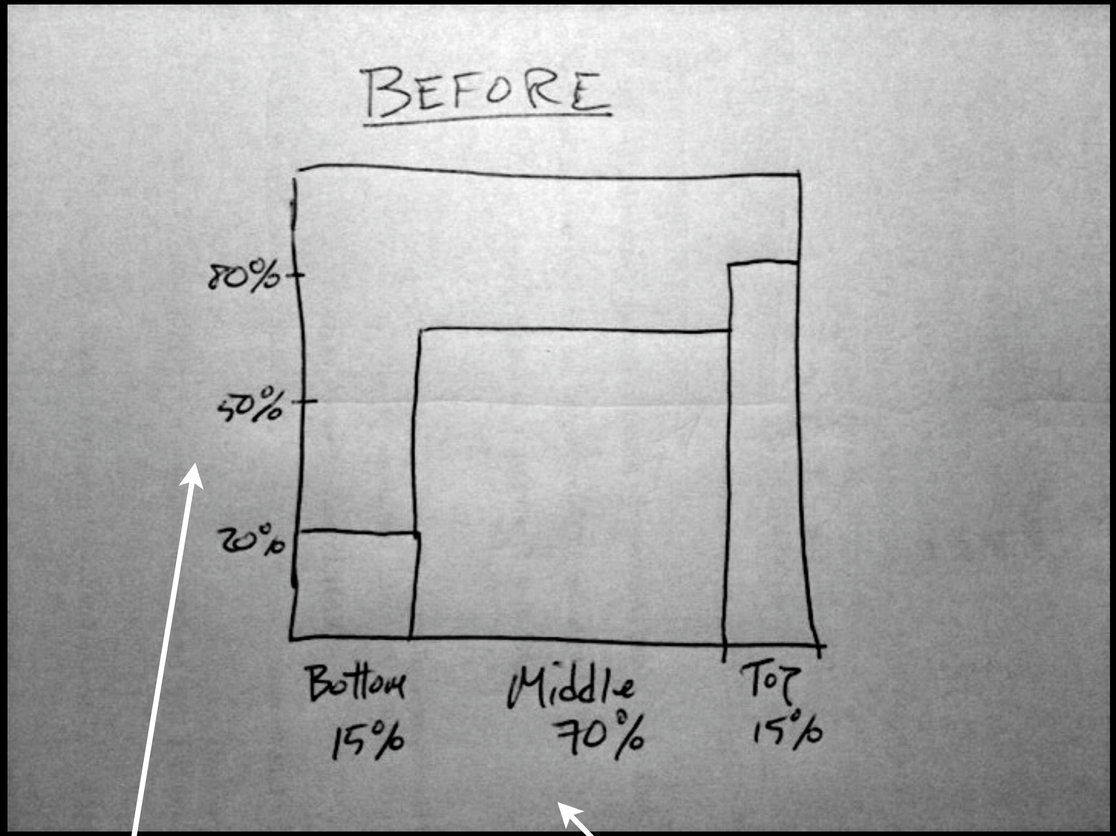
The Fear

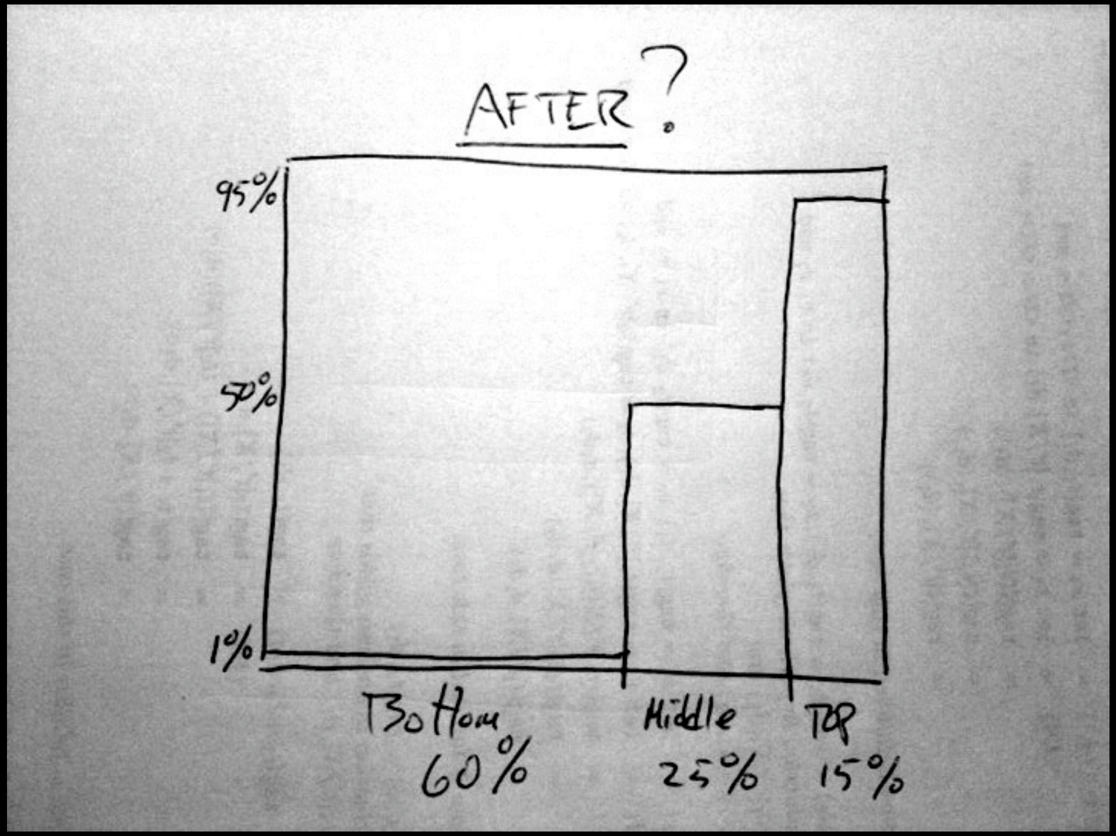


Comprehension

Preparation / aptitude

The Fear





Comprehension

Preparation / aptitude

The Actuality

- Bottom 15% does *not* turn into 60%
- Middle 70% learn at least as much about PL, and they get a solid grasp of what induction means
- Top 15% really hone their understanding, both of proofs and of PL theory
- Most students perform better on paper exams

The Video-Game Effect

- Concrete confirmation of the correctness of each proof step is nice
- Getting Coq to say "Proof complete" is <u>extremely satisfying</u>

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

Bottom Line

- The course can still be improved
- But the way it works for the students is very encouraging even as it stands

Oops, forgot one thing...

Oops, forgot one thing...

There is 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!

The Software Foundations Courseware

What It Is

- A pretty-well-thought-out stylistic framework and some tool support for building formalized instructional material
- One semester's worth of fairly finished lectures, homework, and solutions

Status

- The course has been taught twice at Penn, and once each at Maryland, UCSD, Purdue, and Portland State
- Being taught at Maryland, Lehigh, Iowa, and Princeton this semester, and at Penn (and hopefully some other places!) in the Spring
- Notes (minus solutions) are publicly available as Coq scripts and HTML files:

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

- Instructors who want to use the material in their own courses can obtain read/write access to the SVN repository by emailing me.

What's Next

- Our plans for this year:
  - polish existing material
  - experiment with `ssreflect` package
  - consider replacing subtyping by references (and maybe a stack machine)
- Contributions welcome!
  - Exceptions, etc.
  - Other languages (FJ, ...)
  - More advanced type systems, ...
  - Program analysis
  - More / deeper aspects of Coq

  - Translating the whole thing to another prover...? Sure!

Guided Tour

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.
```

Coq's internal functional language is pretty much like core ML, Haskell, etc., except that only structural recursion is allowed

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.

Notation "x :: y" := (cons x y)
                     (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x , .. , y ]" := (cons x .. (cons y []) ..).
Notation "x ++ y" := (app x y)
                     (at level 60, right associativity).
```

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

("tactic" = command in a proof script, causing Coq to make some step of reasoning)

Programming with Propositions

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

```
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 <u>formally</u> is even harder
- What to do?

The POPLMark Tarpit

- Dealing carefully with variable binding is hard; doing it <u>formally</u> 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 formally 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

Parting Thoughts

Is Coq The Ultimate TA?

Pros:

- Can really build everything we need from scratch
- Curry-Howard
  - Proving = programming
- Good automation

Cons:

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

> 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)

Is <u>Some</u> Proof Assistant The Ultimate TA?

- For students with less mathematical preparation, emphatically yes
  - better motivation, better performance
- But there are some caveats:
  - making up new material is hard
  - needs of formalization significantly shape choice and presentation of material
  - important to remember who's boss

(hint: it's not you)

Back To That Wake-Up Call...

Back To That Wake-Up Call...

- Did we address the original concern?

Back To That Wake-Up Call...

- Did we address the original concern?
- Of course not.
  - This course is theoretical and mainly focused at the graduate level
  - For pure undergrad courses, we surely need something different

Back To That Wake-Up Call...

- Did we address the original concern?
- Of course not.
  - This course is <u>theoretical</u> and mainly <u>focused at the graduate level</u>
  - For pure undergrad courses, we surely need something different
- Indeed,

Back To That Wake-Up Call...

- Did we address the original concern?
- Of course not.
  - This course is theoretical and mainly focused at the graduate level
  - For pure undergrad courses, we surely need something different
- Indeed,
- But to succeed, we need to make better connections with the rest of the curriculum...

Back To That Wake-Up Call...

- Did we address the original concern?
- Of course not.
  - This course is <u>theoretical</u> and mainly <u>focused at the graduate level</u>
  - For pure undergrad courses, we surely need something different
- Indeed,
- But to succeed, we need to make better connections with the rest of the curriculum...

  ...and come to terms with the fact that real-world software construction has changed a lot since we last looked carefully!

In Particular

In Particular

We’re missing a huge opportunity for promoting our ideas...

In Particular

We're missing a huge opportunity for promoting our ideas...

There is a window of opportunity for someone to make $$$ by writing "The Book" for CS1 (intro programming / first year CS)

- Using F#
- GUI-based
- Emphasizing "scripting" examples (using .NET libraries)

Thanks!

SF courseware co-authors:
Chris Casinghino and Michael Greenberg

Additional contributions:
Jeff Foster, Ranjit Jhala, Greg Morrisett, Andrew Tolmach

Good ideas:
Andrew Appel (and many others!)

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

> There is strictly speaking no such thing as a mathematical proof; we can, in the last analysis, do nothing but point...
>
> Hardy, 1928