CS 3110

Functional Programming in Coq

Nate Foster

Spring 2018

Review

Previously in 3110:

- Functional programming
- Modular programming
- Data structures
- Interpreters

Next unit of course: formal methods

Today:

- Proof assistants
- Functional programming in Coq
- Proofs about simple programs

Building reliable software

- Suppose you run a software company

- Suppose you've sunk 30+ person-years into developing the "next big thing":
  - Boeing Dreamliner2 flight controller
  - Autonomous vehicle control software for Tesla
  - Gene therapy DNA tailoring algorithms
  - Super-efficient green-energy power grid controller

- How do you avoid disasters?
  - Turns out software endangers lives
  - Turns out to be impossible to build software

Approaches to validation [lec 11]

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - Static analysis
    ("lint" tools, FindBugs, …)
  - Fuzzers
- Mathematical
  - Sound type systems
  - "Formal" verification

Less formal: Techniques may miss problems in programs

All of these methods should be used!

Even the most formal can still have holes:

- did you prove the right thing?
- do your assumptions match reality?

More formal: eliminate *with certainty* as many problems as possible.

Verification

- In the 1970s, scaled to about tens of LOC
- Now, research projects scale to real software:
  - CompCert: verified C compiler
  - seL4: verified microkernel OS
  - Ynot: verified DBMS, web services
- In another 40 years?

Automated theorem provers

- You give prover a theorem
- Prover searches for:
  - a proof
  - a counterexample
  - or runs out of time
- e.g.,
  - Z3: Microsoft started shipping with device driver developer's kit in Windows 7
  - ACL2: used to verify AMD chip compliance with IEEE floating-point specification, as well as parts of the Java virtual machine

Proof assistants

- You give assistant a theorem
- You and assistant cooperatively find proof
  - Human guides the construction
  - Machine does the low-level details
- e.g.,
  - NuPRL [Prof. Constable, Cornell]: Formalization of mathematics, distributed protocols, security, …
  - Coq: CompCert, Ynot [Dean Morrisett, Cornell], …

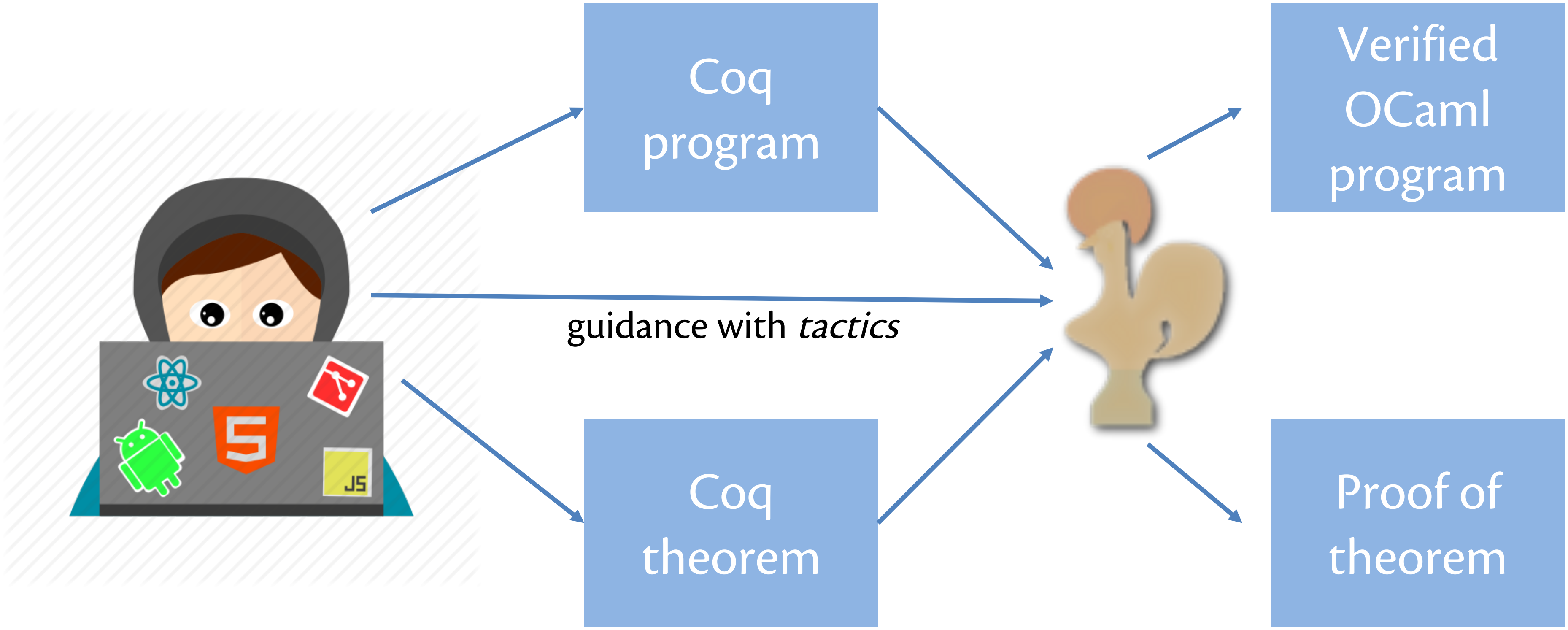
COQ

Coq

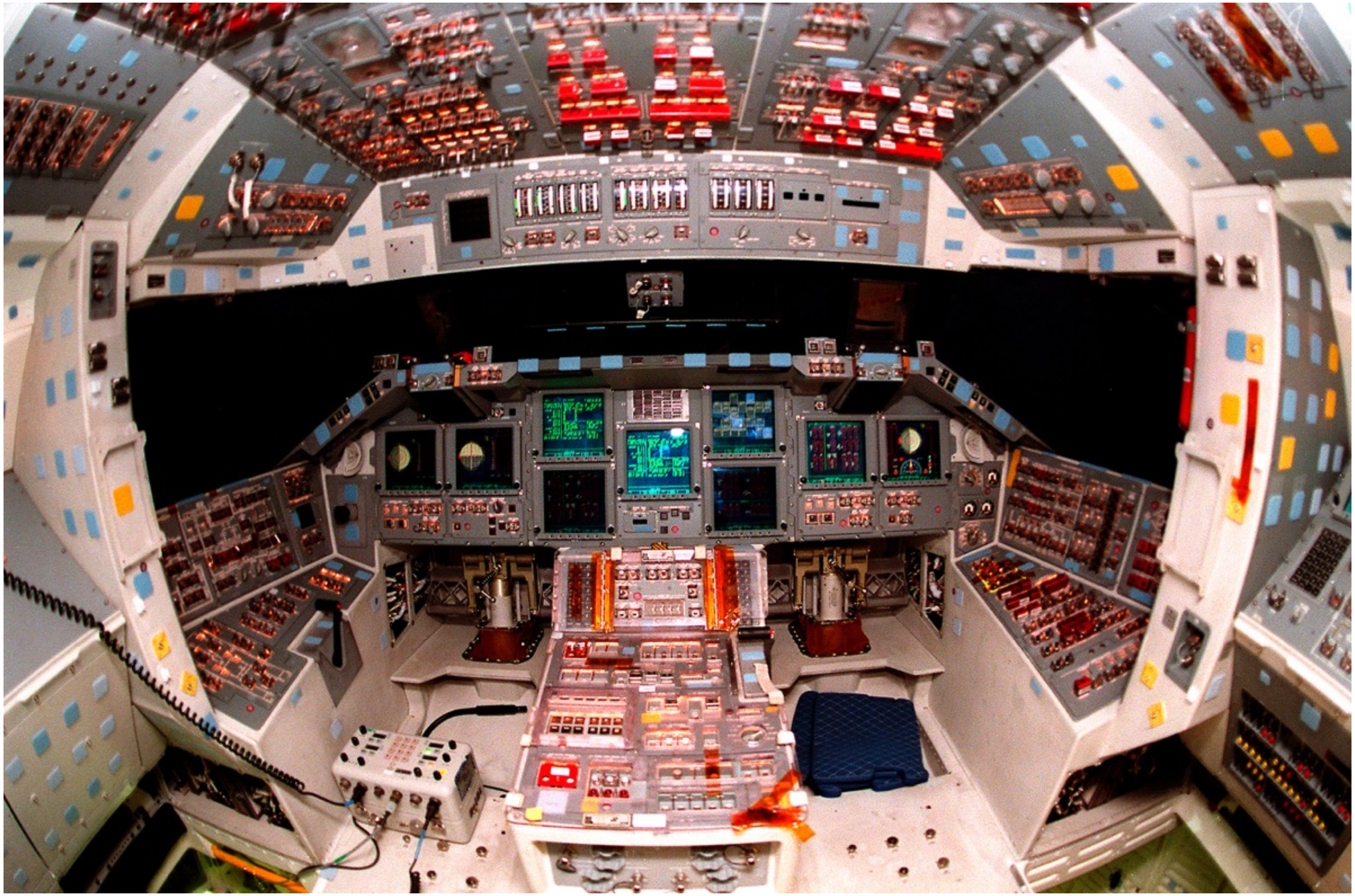
- **1984:** Coquand and Huet implement Coq based on *calculus of inductive constructions*
- **1992:** Coq ported to Caml
- Now implemented in OCaml

Thierry Coquand
1961 –

Coq for program verification



Coq's full system



Subset of Coq we'll use

Our goals

- Write basic functional programs in Coq
  - no side effects, mutability, I/O
- Prove simple theorems in Coq
  - CS 3110 programs: lists, options, trees
  - CS 2800 mathematics: induction, logic

- Non goal: full verification of large programs
- Rather:
  - help you understand what verification involves
  - expose you to the future of functional programming
  - solidify concepts about proof and induction by developing machine-checked proofs

CAUTION: HIGHLY ADDICTIVE

FUNCTIONAL PROGRAMMING IN COQ

Language features

- Anonymous, higher-order functions
- Type inference and annotations
- Pairs
- Lists
- Pattern matching

Commands

- `Let`
- `Check`
- `Print`
- `Compute`
- `Require Import`
- `Locate`
- `Inductive`

THEOREMS ABOUT DAYS

A first theorem

```
Theorem wed_after_tue :
  next_day tue = wed.
```

How we might word proof for a human to read:

- "It's obvious"

OR

- `next_day tue` evaluates to `wed`.
- So we need to show `wed = wed`.
- That follows from the reflexivity of =

OR

- In OCaml, we'd write a test case:
  `assert_equals wed (next_day tue)`

A first theorem

```
Theorem wed_after_tue :
  next_day tue = wed.
Proof.
  auto.
Qed.
```

`auto` is a tactic that searches for a proof; succeeds here because theorem is so easy

Where is the proof?

```
Print wed_after_tue.
wed_after_tue = eq_refl
     : next_day tuesday = wednesday
```

axiom: equality is reflexive
(and expressions may
compute on either side of it)

A first theorem

```
Theorem wed_after_tue :
  next_day tue = wed.
Proof.
  simpl. trivial.
Qed.
```

`simpl` is a tactic that evaluates and simplifies expressions

`trivial` is a tactic that solves trivial equalities

THEOREMS ABOUT DAYS

A second theorem

```
Theorem day_never_repeats :
  forall d, next_day d <> d.
```

Proof. Let d be some day, and proceed by case analysis on what d is.

- If `d` is `sun`, then `next_day d` is `mon`. `sun <> mon` because they are different constructors.
- If `d` is `mon`, then `next_day d` is `tue`. `mon <> tue` because they are different constructors.
- The other cases proceed in the same way.

Or in OCaml, we might write 7 test cases

A second theorem

```
Theorem day_never_repeats :
  forall d, next_day d <> d.
Proof.
  intros d. destruct d.
```

`intros` is a tactic that introduces variables into proof

`destruct` is a tactic that does case analysis

A second theorem

```
Theorem day_never_repeats :
  forall d, next_day d <> d.
Proof.
  intros d. destruct d.
  all: discriminate.
Qed.
```

`all` applies tactic to all subgoals

Upcoming events

- [This week] Design Reviews