CS 3110

Verification in Coq

Nate Foster
Spring 2018

Review

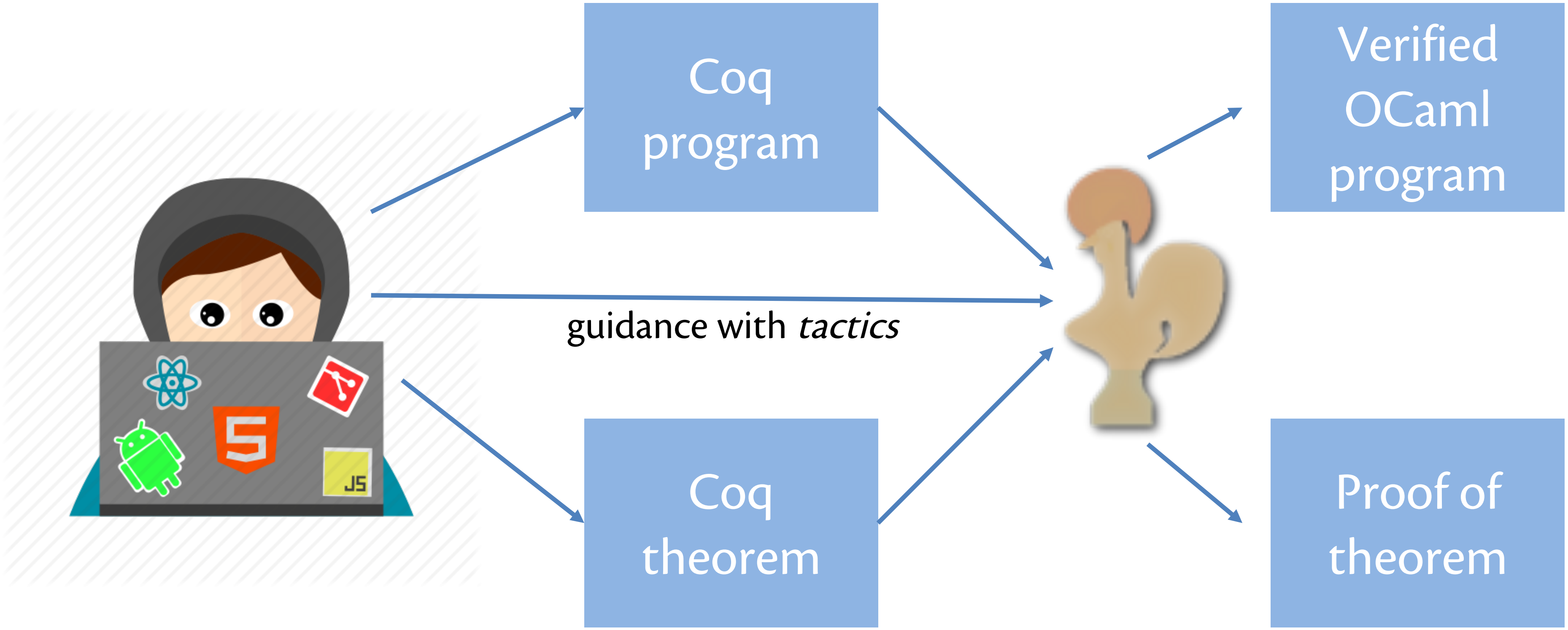
Previously in 3110:

- Functional programming in Coq
- Logic in Coq
- Curry-Howard correspondence (proofs are programs)
- Induction in Coq

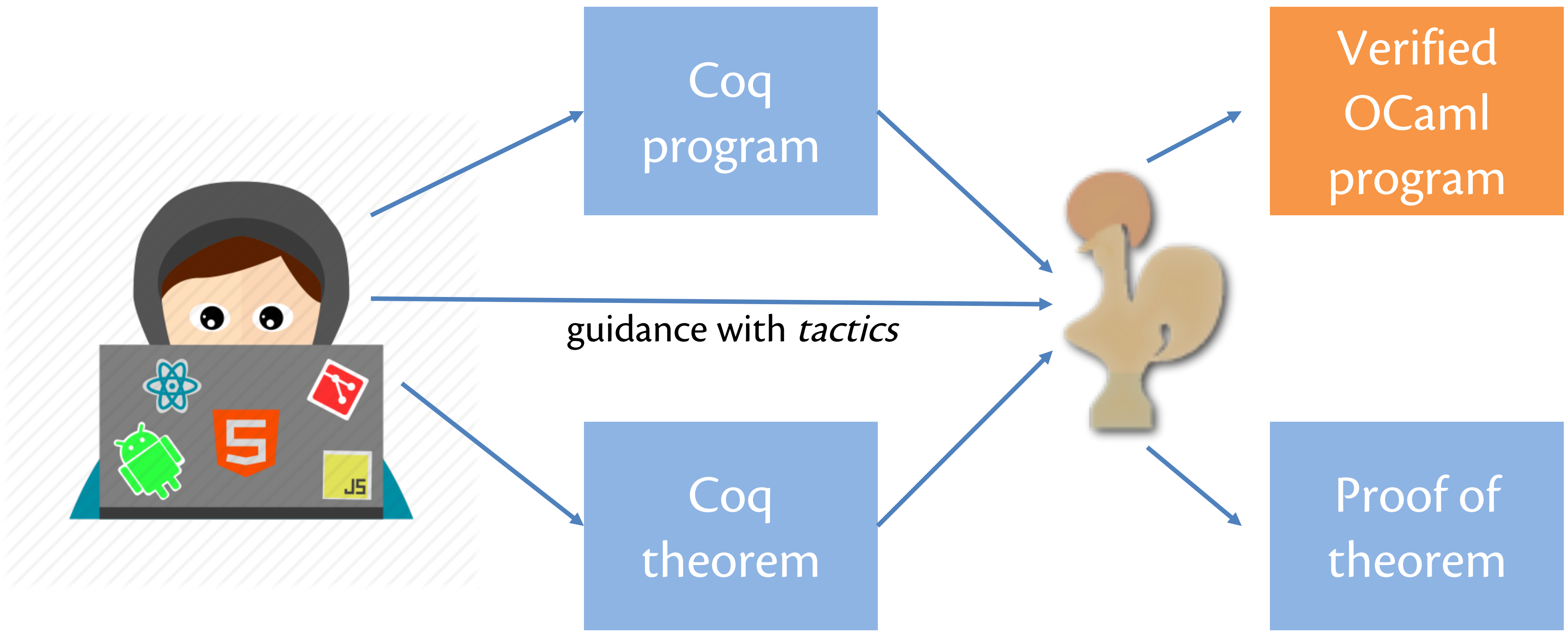
Today: Verification of...

- Functions
- Data structures
- Compilers

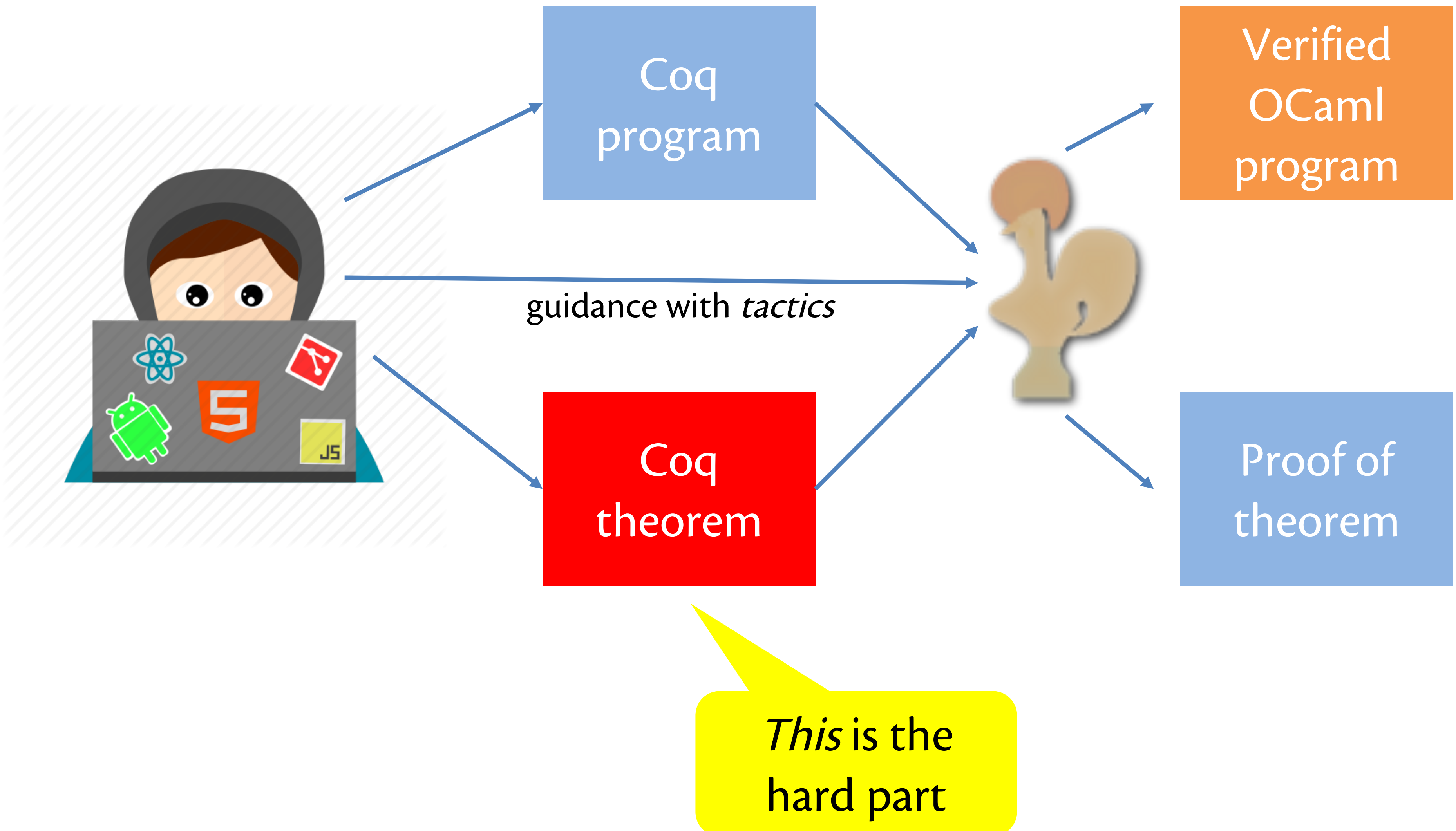
Coq for program verification



Coq for program verification



Coq for program verification



Theorems and test cases

- Do I have the right ones?
- Do I have enough?
- What am I missing?

… there are no great answers to these questions, only methodologies that help

Prove that precondition implies postcondition

VERIFICATION OF A FUNCTION

Factorial

- **Precondition**: `n >= 0`
- **Postcondition**: `fact n = n!`

- **Problem**: how to express `!` in Coq?

Factorial

```
Fixpoint fact (n:nat) :=
  match n with
  | 0 => 1
  | S k => n * (fact k)
  end.

Theorem fact_correct : forall n,
  fact n = fact n.
```

Tail-recursive factorial

```
Fixpoint fact_tr_acc (n:nat) (acc:nat) :=
  match n with
  | 0 => acc
  | S k => fact_tr_acc k (n * acc)
  end.

Definition fact_tr (n:nat) :=
  fact_tr_acc n 1.
```

Precondition: `n >= 0`
Postcondition: `fact_tr n = fact n`

actually unnecessary because `nat` already implies it

Verify factorial

```
Lemma helper : forall (n acc : nat),
  fact_tr_acc n acc = (fact n) * acc.
Proof.
  intros n.
  induction n as [ | k IH]; intros acc.
  - simpl. ring.
  - simpl. rewrite IH. ring.
Qed.

Theorem fact_tr_correct : forall n:nat,
  fact_tr n = fact n.
Proof.
  intros n. unfold fact_tr. rewrite helper. ring.
Qed.
```

Verify factorial

Generalized inductive hyopothesis: not all variables introduced

```
Lemma helper : forall (n acc : nat),
   fact_tr_acc n acc = (fact n) * acc.
Proof.
   intros n.
   induction n as [ | k IH]; intros acc.
   - simpl. ring.
   - simpl. rewrite IH. ring.
Qed.
```

Verify efficient impl equiv. to "obviously correct" inefficient impl.

```
Theorem fact_tr_correct : forall n:nat,
   fact_tr n = fact n.
Proof.
   intros n. unfold fact_tr. rewrite helper. ring.
Qed.
```

`unfold` tactic instantiates definition

Extract verified factorial

```
Extract Inductive nat
  => int [ "0" "succ" ].
Extract Inlined Constant Init.Nat.mul
  => "(*)".
Extraction "fact.ml" fact_tr.
```

Coq `nat` becomes OCaml `int`

Coq * becomes OCaml *

Extract Coq to OCaml

Prove that equations hold for operations

VERIFICATION OF A DATA STRUCTURE

Stack

```
module type Stack = sig
  type 'a t
  val empty    : 'a t
  val is_empty : 'a t -> bool
  val size     : 'a t -> int
  val peek     : 'a t -> 'a option
  val push     : 'a -> 'a t -> 'a t
  val pop      : 'a t -> 'a t option
end
```

Categories of operations

- **Creator:** creates value of type "from scratch" without any inputs of that type
- **Producer:** takes value of type as input and returns value of type as output
- **Observer:** takes value of type as input but does not return value of type as output
- *(**Mutator:** takes value of type as input and mutates the value)*

Stack

```
module type Stack = sig
  type 'a t
  val empty    : 'a t
  val is_empty : 'a t -> bool
  val size     : 'a t -> int
  val peek     : 'a t -> 'a option
  val push     : 'a -> 'a t -> 'a t
  val pop      : 'a t -> 'a t option
end
```

creator

observers

producers

Stack eqn. specification

- `is_empty empty = true`
- `is_empty (push _ _) = false`
- `peek empty = None`
- `peek (push x _) = Some x`
- `size empty = 0`
- `size (push _ s) = 1 + size s`
- `pop empty = None`
- `pop (push _ s) = Some s`

Equational specification

- aka *algebraic specification*
- Set of equations
- Describes interactions between:
  - observers and creators
  - observers and producers
  - producers and creators
  - producers and other producers
- Might not have equation for every possible interaction, because some might not be meaningful

Stack as list

```
Module MyStack.

Definition stack (A:Type) := list A.

Definition empty {A:Type} : stack A :=
  nil.

Definition is_empty {A:Type} (s : stack A)
  : bool
:=
  match s with
  | nil => true
  | _::_ => false
  end.
```

Stack as list

```
Definition push {A:Type} (x : A) (s : stack A)
  : stack A
:=
  x::s.

Definition peek {A:Type} (s : stack A)
  : option A
:=
  match s with
  | nil => None
  | x::_ => Some x
  end.
```

Stack as list

```
Definition pop {A:Type} (s : stack A)
  : option (stack A)
:=
  match s with
  | nil => None
  | _::xs => Some xs
  end.

Definition size {A:Type} (s : stack A)
  : nat
:=
  length s.

End MyStack.
```

Verify stack as list

```
Theorem empty_is_empty : forall (A:Type),
  @is_empty A empty = true.
Proof. auto. Qed.

Theorem push_not_empty : forall (A:Type) (x:A) (s : stack A),
  is_empty(push x s) = false.
Proof. auto. Qed.

Theorem peek_empty : forall (A:Type),
  @peek A empty = None.
Proof. auto. Qed.

Theorem peek_push : forall (A:Type) (x:A) (s : stack A),
  peek(push x s) = Some x.
Proof. auto. Qed.
```

Verify stack as list

```
Theorem pop_empty : forall (A:Type),
  @pop A empty = None.
Proof. auto. Qed.

Theorem pop_push : forall (A:Type) (x:A) (s : stack A),
  pop(push x s) = Some s.
Proof. auto. Qed.

Theorem size_empty : forall (A:Type),
  @size A empty = 0.
Proof. auto. Qed.

Theorem size_push : forall (A:Type) (x:A) (s : stack A),
  size(push x s) = 1 + size s.
Proof. auto. Qed.
```

Extract verified stack

```
Extract Inductive bool => "bool" [ "true" "false" ].
Extract Inductive option => "option" [ "Some" "None" ].
Extract Inductive list => "list" [ "[]" "(::)" ].
Extract Inductive nat => int [ "0" "succ" ].

Extraction "stacks.ml" MyStack.
```

Coq bool, option, list, nat become OCaml equiv.

Prove that meaning is preserved

VERIFICATION OF A COMPILER

Expressions

```
Inductive expr : Type :=
  | Const : nat -> expr
  | Plus : expr -> expr -> expr.

Fixpoint eval_expr (e : expr) : nat :=
  match e with
  | Const n => n
  | Plus e1 e2 =>
    plus (eval_expr e1) (eval_expr e2)
  end.
```

Stack programs

```
Inductive instr : Type :=
  | PUSH : nat -> instr
  | ADD : instr.

Definition prog := list instr.

Definition stack := list nat.
```

Stack programs

```
Fixpoint eval_prog
  (p : prog) (s : stack)
  : option stack
:=
  match p,s with
  | (PUSH n)::p', s =>
    eval_prog p' (n::s)
  | ADD::p', x::y::s' =>
    eval_prog p' ((x+y)::s')
  | nil, s => Some s
  | _, _ => None
  end.
```

Compiler

```
Fixpoint compile (e : expr) : prog :=
  match e with
  | Const n => [PUSH n]
  | Plus e1 e2 =>
    compile e2 ++ compile e1 ++ [ADD]
  end.
```

Verify the compiler

```
Theorem compile_correct :
  forall (e:expr),
    eval_prog (compile e) []
    = Some [eval_expr e].
```

Proof *in lecture code.*

Extract verified compiler

```
Extract Inlined Constant app => "(@)".
Extraction "compiler.ml" compile.
```

Upcoming events

- [Tonight!] Prelim II
- [Wednesday or Thursday] A5 out
- [Friday] Yaron Minsky @ 5:30pm