Ahmet Celik, Karl Palmskog*, Milos Gligoric
The University of Texas at Austin

*Work done while at University of Illinois at Urbana-Champaign
Verification Using Proof Assistants

1. encode definitions in (higher-order) formalism
2. prove propositions \textit{interactively} using powerful \textit{tactics}
3. check soundness of every low-level step

examples: \textbf{Coq}, HOL4, HOL Light, Isabelle/HOL, Lean, Nuprl, ...
Software Development Workflow in Coq

1. Write purely functional program
2. Write specification and prove program correct
3. Extract program to practical language (OCaml, Haskell, ...)
4. Link extracted program to libraries for I/O, communication, ...

Fixpoint alternate l1 l2 :=
match l1 with
| [] ⇒ l2
| h1 :: t1 ⇒
  match l2 with
  | [] ⇒ h1 :: t1
  | h2 :: t2 ⇒
    h1 :: h2 :: alternate t1 t2
  end
end.

1. Coq program
Software Development Workflow in Coq

1. Write purely functional program
2. Write specification and prove program correct
3. Extract program to practical language (OCaml, Haskell, ...)
4. Link extracted program to libraries for I/O, communication, ...

Fixpoint alternate l1 l2 :=
  match l1 with
  | [] ⇒ l2
  | h1 :: t1 ⇒
    match l2 with
    | [] ⇒ h1 :: t1
    | h2 :: t2 ⇒
      h1 :: h2 :: alternate t1 t2
  end
end.

Inductive alt :=
| alt_nil : forall l, alt [] l l
| alt_step : forall a l t1 t2,
  alt l t1 t2 →
  alt (a :: t1) l (a :: t2).

Lemma alt_alternate :
  forall l1 l2 l3,
  alt l1 l2 l3 →
  alternate l1 l2 = l3.
Proof.
(* omitted proof script ... *)
Qed.

1. Coq program
2. Coq spec/proof
Software Development Workflow in Coq

1. Write purely functional program
2. Write specification and prove program correct
3. Extract program to practical language (OCaml, Haskell, ...)
4. Link extracted program to libraries for I/O, communication, ...

| Fixpoint alternate l1 l2 :=
|   match l1 with
|   | [] ⇒ l2
|   | h1 :: t1 ⇒
|   |   match l2 with
|   |     | [] ⇒ h1 :: t1
|   |     | h2 :: t2 ⇒
|   |     |     h1 :: h2 :: alternate t1 t2
|   end
|   end. |

| Inductive alt :=
|   | alt_nil : forall l,
|   |     alt [] l l
|   | alt_step : forall a l t1 t2,
|   |     alt l t1 t2 →
|   |     alt (a :: t1) l (a :: t2).
| Lemma alt_alternate :
|     forall l1 l2 l3,
|     alt l1 l2 l3 →
|     alternate l1 l2 = l3.
| Proof.
| (*) omitted proof script ... *)
| Qed. |

| let rec alternate l1 l2 =
|   match l1 with
|   | [] ⇒ l2
|   | h1 :: t1 ⇒
|   |     (match l2 with
|   |       | [] ⇒ h1 :: t1
|   |       | h2 :: t2 ⇒
|   |       |     h1 :: (h2 ::
|   |       |     (alternate t1 t2)))

1. Coq program  2. Coq spec/proof  3. OCaml program
Software Development Workflow in Coq

1. Write purely functional program
2. Write specification and prove program correct
3. Extract program to practical language (OCaml, Haskell, ...)
4. Link extracted program to libraries for I/O, communication, ...

Fixpoint alternate l1 l2 :=
match l1 with
| [] ⇒ l2
| h1 :: t1 ⇒
  match l2 with
  | [] ⇒ h1 :: t1
  | h2 :: t2 ⇒
    h1 :: h2 :: alternate t1 t2
  end
end.

Inductive alt :=
| alt_nil : forall l,
  alt [] 1 l
| alt_step : forall a l t1 t2,
  alt l t1 t2 →
  alt (a :: t1) l (a :: t2).

Lemma alt_alternate :
forall l1 l2 l3,
alt l1 l2 l3 →
alternate l1 l2 = l3.
Proof.
(* omitted proof script ... *)
Qed.

let rec alternate l1 l2 =
match l1 with
| [] → l2
| h1 :: t1 →
  (match l2 with
   | [] → h1 :: t1
   | h2 :: t2 →
     h1 :: (h2 ::
       (alternate t1 t2))))

1. Coq program  2. Coq spec/proof  3. OCaml program
### Some Large-Scale Verification Projects

<table>
<thead>
<tr>
<th>Project</th>
<th>Year</th>
<th>Assistant</th>
<th>Check Time</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-Color Theorem</td>
<td>2005</td>
<td>Coq</td>
<td>tens of mins</td>
<td>60k</td>
</tr>
<tr>
<td>Odd Order Theorem</td>
<td>2012</td>
<td>Coq</td>
<td>hours</td>
<td>150k</td>
</tr>
<tr>
<td>Kepler Conjecture</td>
<td>2015</td>
<td>HOL Light</td>
<td>days</td>
<td>500k</td>
</tr>
<tr>
<td>CompCert</td>
<td>2009</td>
<td>Coq</td>
<td>tens of mins</td>
<td>40k</td>
</tr>
<tr>
<td>seL4</td>
<td>2009</td>
<td>Isabelle/HOL</td>
<td>hours</td>
<td>200k</td>
</tr>
<tr>
<td>Verdi Raft</td>
<td>2016</td>
<td>Coq</td>
<td>tens of mins</td>
<td>50k</td>
</tr>
</tbody>
</table>

*problem: long proof checking times*
Problem: Regression Proving in Evolving Projects

Typical **proving** scenario:

1. change **definition** or **lemma statement**
2. begin process of **re-checking** all **proofs**
3. **checking** fails hours later (for seemingly unrelated **proof**)
Problem: Regression Proving in Evolving Projects

Typical **proving** scenario:

1. change definition or lemma statement
2. begin process of re-checking all proofs
3. checking fails hours later (for seemingly unrelated proof)

Typical **testing** scenario:

1. change method statements or method signature
2. begin process of re-running all tests
3. testing fails hours later (for seemingly unrelated test)
Regression Test Selection (RTS)

A regression test selection technique chooses, from an existing test set, tests that are deemed necessary to validate modified software.

Rothermel and Harrold, ACM TOSEM 6, 2 ’97
Regression Test Selection (RTS)

A regression **test** selection technique chooses, from an existing **test set**, **tests** that are deemed necessary to **validate** modified **software**.

Rothermel and Harrold, ACM TOSEM 6, 2 ’97

A regression **proof** selection technique chooses, from an existing **proof set**, **proofs** that are deemed necessary to **verify** modified **theories**.
Our iCoq tool:

- ... implements regression proof selection for Coq projects
- ... tracks dependencies between Coq definitions and proofs
- ... checks only affected proofs after each change
Our iCoq tool:
- ... implements regression proof selection for Coq projects
- ... tracks dependencies between Coq definitions and proofs
- ... checks only affected proofs after each change

Tool phases:

1. **analysis**: locate proofs affected by changes
2. **execution**: emit and run proof-checking commands
3. **collection**: find dependencies of modified definition/proofs

Key idea: maintain file and proof dependency graphs
iCoq Workflow

- Analysis
- Checking
- Collection
iCoq Workflow

- Analysis
  - file dep. graph
  - source files
  - proof dep. graph

- Checking

- Collection
iCoq Workflow

- File dependency graph
- Source files
- Proof dependency graph
- Compilation commands
- Compiled files
- Affected proofs

Analysis

Checking

Collection
iCoq Workflow

Analysis

file dep. graph

source files

proof dep. graph

compilation commands

compiled files

affected proofs

Checking

proof dependencies

checking commands

Collection

new dep. graphs

storage

8 / 1
iCoq Workflow

- Analysis
  - file dep. graph
  - source files
  - proof dep. graph
  - compilation commands
  - compiled files
  - affected proofs

- Checking
  - proof dependencies
  - checking commands

- Collection
  - new dep. graphs
  - storage
Example, revision 1

```
Require Export List. Export ListNotations.

Fixpoint alternate l1 l2 : list nat :=
  match l1 with
  | [] ⇒ l2 | h1 :: t1 ⇒
    match l2 with
    | [] ⇒ h1 :: t1
    | h2 :: t2 ⇒
      h1 :: h2 :: alternate t1 t2
    end
  end.

Inductive alt : list nat → list nat → list nat → Prop :=
  | alt_nil : ∀ l, alt [] l l
  | alt_step : ∀ a l t1 t2, alt l t1 t2
    ⇒
    alt (a :: t1) l (a :: t2).

Lemma alt_alternate :
  ∀ l1 l2 l3, alt l1 l2 l3
  ⇒ alternate l1 l2 = l3.
Proof.
(* ... omitted proof script ... *)
Qed.
```

```
Require Import Alternate.

Lemma alt_exists :
  ∀ l1 l2, exists l3, alt l1 l2 l3.
Proof.
induction l1; intros; destruct l2.
- exists []. apply alt_nil.
- exists (n :: l2). apply alt_nil.
- exists (a :: l1). apply alt_step.
  apply alt_nil.
- specialize(IHl1 l2).
  destruct IHl1. exists (a :: n :: x).
  repeat apply alt_step. auto.
Qed.
```

```
Require Import Alternate.

Require Export List. Export ListNotations.

Fixpoint alternate l1 l2 : list nat :=
  match l1 with
  | [] ⇒ l2 | h1 :: t1 ⇒
    match l2 with
    | [] ⇒ h1 :: t1
    | h2 :: t2 ⇒
      h1 :: h2 :: alternate t1 t2
    end
  end.

Inductive alt : list nat → list nat → list nat → Prop :=
  | alt_nil : ∀ l, alt [] l l
  | alt_step : ∀ a l t1 t2, alt l t1 t2
    ⇒
    alt (a :: t1) l (a :: t2).

Lemma alt_alternate :
  ∀ l1 l2 l3, alt l1 l2 l3
  ⇒ alternate l1 l2 = l3.
Proof.
(* ... omitted proof script ... *)
Qed.
```

```
Require Import Alternate.

Lemma alt_exists :
  ∀ l1 l2, exists l3, alt l1 l2 l3.
Proof.
induction l1; intros; destruct l2.
- exists []. apply alt_nil.
- exists (n :: l2). apply alt_nil.
- exists (a :: l1). apply alt_step.
  apply alt_nil.
- specialize(IHl1 l2).
  destruct IHl1. exists (a :: n :: x).
  repeat apply alt_step. auto.
Qed.
```

```
Require Import Alternate.

Require Export List. Export ListNotations.

Fixpoint alternate l1 l2 : list nat :=
  match l1 with
  | [] ⇒ l2 | h1 :: t1 ⇒
    match l2 with
    | [] ⇒ h1 :: t1
    | h2 :: t2 ⇒
      h1 :: h2 :: alternate t1 t2
    end
  end.

Inductive alt : list nat → list nat → list nat → Prop :=
  | alt_nil : ∀ l, alt [] l l
  | alt_step : ∀ a l t1 t2, alt l t1 t2
    ⇒
    alt (a :: t1) l (a :: t2).

Lemma alt_alternate :
  ∀ l1 l2 l3, alt l1 l2 l3
  ⇒ alternate l1 l2 = l3.
Proof.
(* ... omitted proof script ... *)
Qed.
```

```
Require Import Alternate.

Lemma alt_exists :
  ∀ l1 l2, exists l3, alt l1 l2 l3.
Proof.
induction l1; intros; destruct l2.
- exists []. apply alt_nil.
- exists (n :: l2). apply alt_nil.
- exists (a :: l1). apply alt_step.
  apply alt_nil.
- specialize(IHl1 l2).
  destruct IHl1. exists (a :: n :: x).
  repeat apply alt_step. auto.
Qed.
```
Require Export List. Export ListNotations.

Fixpoint alternate l1 l2 : list nat :=
match l1 with
| [] => l2 | h1 :: t1 =>
  match l2 with
  | [] => h1 :: t1
  | h2 :: t2 => h1 :: h2 :: alternate t1 t2
end
end.

Inductive alt : list nat -> list nat -> list nat -> Prop :=
| alt_nil : forall l, alt [] l l
| alt_step : forall a l t1 t2,
  alt l t1 t2 =>
  alt (a :: t1) l (a :: t2).

Lemma alt_alternate :
  forall l1 l2 l3, alt l1 l2 l3 => alternate l1 l2 = l3.
Proof.
(* ... omitted proof script ... *)
Qed.

Require Import Alternate.

Lemma alt_exists :
  forall l1 l2, exists l3, alt l1 l2 l3.
Proof.
induction l1; intros; destruct l2.
- exists []. apply alt_nil.
- exists (n :: l2). apply alt_nil.
- exists (a :: l1). apply alt_step.
  apply alt_nil.
- specialize(IHl1 l2).
  destruct IHl1. exists (a :: n :: x).
  repeat apply alt_step. auto.
Qed.

AltLem.v

file dependency graph
A Regression Proof Selection Tool For Coq

Example, revision 1

Require Export List. Export ListNotations.

Fixpoint alternate l1 l2 : list nat :=
match l1 with
| [] → l2 | h1 :: t1 ⇒
  match l2 with
  | [] ⇒ h1 :: t1
  | h2 :: t2 ⇒
    h1 :: h2 :: alternate t1 t2
end
end.

Inductive alt : list nat → list nat → list nat → Prop :=
| alt_nil : forall l, alt [] l l
| alt_step : forall a l t1 t2,
  alt l t1 t2
  ⇒
  alt (a :: l1) l (a :: t2).

Lemma alt_alternate :
forall l1 l2 l3, alt l1 l2 l3
⇒
alternate l1 l2 = l3.
Proof. (* ... omitted proof script ... *) Qed.

Require Import Alternate.

Lemma alt_exists :
forall l1 l2, exists l3, alt l1 l2 l3.
Proof.
induction l1; intros; destruct l2.
- exists []. apply alt_nil.
- exists (n :: l2). apply alt_nil.
- exists (a :: l1). apply alt_step.
  apply alt_nil.
- specialize(IHl1 l2).
  destruct IHl1. exists (a :: n :: x).
  repeat apply alt_step. auto.
Qed.

AltLem.v

proof dependency graph
Example, revision 1

**Require Export List. Export ListNotations.**

**Fixpoint alternate l1 l2 : list nat :=**

match l1 with
| [] ⇒ l2 | h1 :: t1 ⇒
  match l2 with
  | [] ⇒ h1 :: t1 | h2 :: t2 ⇒
    h1 :: h2 :: alternate t1 t2
  end
end.

**Inductive alt : list nat → list nat → list nat → Prop :=**

| alt_nil : forall l, alt [] l l |
| alt_step : forall a l t1 t2, alt l t1 t2 → alt (a :: t1) l (a :: t2).

**Lemma alt_alternate :**

forall l1 l2 l3, alt l1 l2 l3 → alternate l1 l2 = l3.

**Proof.**

(* ... omitted proof script ... *)

Qed.

**Require Import Alternate.**

**Lemma alt_exists :**

forall l1 l2, exists l3, alt l1 l2 l3.

**Proof.**

induction l1; intros; destruct l2.
- exists []. apply alt_nil.
- exists (n :: l2). apply alt_nil.
- exists (a :: l1). apply alt_step.
  apply alt_nil.
- specialize(IHl1 l2).
  destruct IHl1. exists (a :: n :: x).
  repeat apply alt_step. auto.

Qed.

**AltLem.v**

**Programmer now changes alternate**

**Alternate.v**
Example, revision 2

Require Export List. Export ListNotations.

Fixpoint alternate l1 l2 : list nat :=
  match l1, l2 with
  | [], _ => l2
  | _, [] => l1
  | h1 :: t1, h2 :: t2 => h1 :: h2 :: alternate t1 t2
  end.

Inductive alt : list nat -> list nat -> list nat -> Prop :=
  | alt_nil : forall l, alt [] l l
  | alt_step : forall a l t1 t2, alt l t1 t2 -> alt (a :: t1) l (a :: t2).

Lemma alt_alternate :
  forall l1 l2 l3, alt l1 l2 l3 -> alternate l1 l2 = l3.
  Proof.
  (* ... omitted proof script ... *)
  Qed.

Require Import Alternate.

Lemma alt_exists :
  forall l1 l2, exists l3, alt l1 l2 l3.
  Proof.
  induction l1; intros; destruct l2.
  - exists []. apply alt_nil.
  - exists (n :: l2). apply alt_nil.
  - exists (a :: l1). apply alt_step.
    apply alt_nil.
  - specialize(IHl1 l2).
    destruct IHl1. exists (a :: n :: x).
    repeat apply alt_step. auto.
  Qed.

Change creates new revision
Fixpoint alternate l1 l2 : list nat :=
match l1, l2 with
| [], _ ⇒ l2
| _, [] ⇒ l1
| h1 :: t1, h2 :: t2 ⇒ h1 :: h2 :: alternate t1 t2
end.

Inductive alt : list nat → list nat → list nat → Prop :=
| alt_nil : forall l, alt [] l l
| alt_step : forall a l t1 t2, alt l t1 t2 → alt (a :: t1) l (a :: t2).

Lemma alt_alternate : forall l1 l2 l3, alt l1 l2 l3 → alternate l1 l2 = l3.
Proof.
(* ... omitted proof script ... *)
Qed.

Require Import Alternate.
Lemma alt_exists :
forall l1 l2, exists l3, alt l1 l2 l3.
Proof.
induction l1; intros; destruct l2.
- exists[]. apply alt_nil.
- exists (n :: l2). apply alt_nil.
- exists (a :: l1). apply alt_step.
  apply alt_nil.
  specialize(IHl1 l2).
  destruct IHl1. exists (a :: n :: l2).
  repeat apply alt_step. auto.
Qed.

Require Export List. Export ListNotations.

File dependency graph
Require Export List. Export ListNotations.

Fixpoint alternate l1 l2 : list nat :=
  match l1, l2 with
  | [], _ ⇒ l2
  | _, [] ⇒ l1
  | h1 :: t1, h2 :: t2 ⇒
    h1 :: h2 :: alternate t1 t2
  end.

Inductive alt : list nat → list nat → list nat → Prop :=
  | alt_nil : forall l, alt [] l l
  | alt_step : forall a l t1 t2,
    alt l t1 t2
    →
    alt (a :: t1) l (a :: t2).

Lemma alt_alternate :
  forall l1 l2 l3, alt l1 l2 l3
  →
  alternate l1 l2 = l3.
  Proof.
  (* ... omitted proof script ... *)
  Qed.

Require Import Alternate.

Lemma alt_exists :
  forall l1 l2, exists l3, alt l1 l2 l3.
  Proof.
  induction l1; intros; destruct l2.
  - exists []. apply alt_nil.
  - exists (n :: l2). apply alt_nil.
  - exists (a :: l1). apply alt_step.
    apply alt_nil.
  - specialize(IHl1 l2).
    destruct IHl1. exists (a :: n :: x).
    repeat apply alt_step. auto.
  Qed.

AltLem.v

file dependency graph

Alternate.v --→ List.v

AltLem.v

AltLem.v
Example, revision 2

```
Require Export List. Export ListNotations.

Fixpoint alternate l1 l2 : list nat :=
match l1, l2 with
| [], _ ⇒ l2
| _, [] ⇒ l1
| h1 :: t1, h2 :: t2 ⇒ h1 :: h2 :: alternate t1 t2
end.

Inductive alt : list nat → list nat → list nat → Prop :=
| alt_nil : forall l, alt [] l l
| alt_step : forall a l t1 t2, alt l t1 t2
  → alt (a :: t1) l (a :: t2).

Lemma alt_alternate :
  forall l1 l2 l3, alt l1 l2 l3
  → alternate l1 l2 = l3.
  Proof. (* ... omitted proof script ... *) Qed.

Require Import Alternate.

Lemma alt_exists :
  forall l1 l2, exists l3, alt l1 l2 l3.
  Proof. induction l1; intros; destruct l2.
  - exists []. apply alt_nil.
  - exists (n :: l2). apply alt_nil.
  - exists (a :: l1). apply alt_step.
    apply alt_nil.
  - specialize(IHl1 l2).
    destruct IHl1. exists (a :: n :: x).
    repeat apply alt_step. auto.
  Qed.
```

```
Require Export List. Export ListNotations.

Fixpoint alternate l1 l2 : list nat :=
match l1, l2 with
| [], _ ⇒ l2
| _, [] ⇒ l1
| h1 :: t1, h2 :: t2 ⇒ h1 :: h2 :: alternate t1 t2
end.

Inductive alt : list nat → list nat → list nat → Prop :=
| alt_nil : forall l, alt [] l l
| alt_step : forall a l t1 t2, alt l t1 t2
  → alt (a :: t1) l (a :: t2).

Lemma alt_alternate :
  forall l1 l2 l3, alt l1 l2 l3
  → alternate l1 l2 = l3.
  Proof. (* ... omitted proof script ... *) Qed.
```

```
Require Import Alternate.

Lemma alt_exists :
  forall l1 l2, exists l3, alt l1 l2 l3.
  Proof. induction l1; intros; destruct l2.
  - exists []. apply alt_nil.
  - exists (n :: l2). apply alt_nil.
  - exists (a :: l1). apply alt_step.
    apply alt_nil.
  - specialize(IHl1 l2).
    destruct IHl1. exists (a :: n :: x).
    repeat apply alt_step. auto.
  Qed.
```

```
Alternate.v

AltLem.v

proof dependency graph
```
**Require Export List. Export ListNotations.**

**Fixpoint alternate l1 l2 : list nat :=**

- match l1, l2 with
  - | [], _ ⇒ l2
  - | _, [] ⇒ l1
  - | h1 :: t1, h2 :: t2 ⇒ h1 :: h2 :: alternate t1 t2
end.

**Inductive alt : list nat → list nat → list nat → Prop :=**

- | alt_nil : forall l, alt [] l l
- | alt_step : forall a l t1 t2, alt l t1 t2 → alt (a :: t1) l (a :: t2).

**Lemma alt_alternate :**

- | forall l1 l2 l3, alt l1 l2 l3 → alternate l1 l2 = l3.

**Proof.**

(* ... omitted proof script ... *)

Qed.

**Require Import Alternate.**

**Lemma alt_exists :**

- | forall l1 l2, exists l3, alt l1 l2 l3.

**Proof.**

induction l1; intros; destruct l2.

- | exists []. apply alt_nil.
- | exists (n :: l2). apply alt_nil.
- | exists (a :: l1). apply alt_step.
  - apply alt_nil.
- | specialize(IHl1 l2).
  - destruct IHl1. exists (a :: n :: x).
  - repeat apply alt_step. auto.

Qed.

**AltLem.v**

**proof dependency graph**
Example, revision 2

Require Export List. Export ListNotations.

Fixpoint alternate l1 l2 : list nat :=
match l1, l2 with
| [], _ ⇒ l2
| _, [] ⇒ l1
| h1 :: t1, h2 :: t2 ⇒ h1 :: h2 :: alternate t1 t2
end.

Inductive alt : list nat → list nat → list nat → Prop :=
| alt_nil : forall l, alt [] l l
| alt_step : forall a l t1 t2,
  alt l t1 t2 →
  alt (a :: t1) l (a :: t2).

Lemma alt_alternate :
  forall l1 l2 l3, alt l1 l2 l3 → alternate l1 l2 = l3.
Proof.
(* ... omitted proof script ... *)
Qed.

Require Import Alternate.

Lemma alt_exists :
  forall l1 l2, exists l3, alt l1 l2 l3.
Proof.
induction l1; intros; destruct l2.
- exists []. apply alt_nil.
- exists (n :: l2). apply alt_nil.
- exists (a :: l1). apply alt_step.
  apply alt_nil.
- specialize(IHl1 l2).
  destruct IHl1. exists (a :: n :: x).
  repeat apply alt_step. auto.
Qed.

AltLem.v

iCoq checks proof of alt_alternate
... but not proof of alt_exists!
## iCoq Components

<table>
<thead>
<tr>
<th>Component</th>
<th>Lang.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coqast</td>
<td>OCaml</td>
<td>compute digests of proof ASTs</td>
</tr>
<tr>
<td>coqdepends</td>
<td>OCaml</td>
<td>extract dependencies from proof ASTs</td>
</tr>
<tr>
<td>coqdigest</td>
<td>OCaml</td>
<td>compute digests of proof scripts</td>
</tr>
<tr>
<td>icoqc</td>
<td>OCaml</td>
<td>proof-checking dependency extraction</td>
</tr>
<tr>
<td>icoqgr</td>
<td>Java/bash</td>
<td>construct/maintain dependency graphs</td>
</tr>
</tbody>
</table>
Components and the iCoq Workflow

- **Analysis**
  - compilation commands
  - compiled files
  - affected proofs
  - icoqgr, coqast, coqdigest

- **Checking**
  - proof dependencies
  - checking commands
  - icoqgr, icoqc

- **Collection**
  - new dep. graphs
  - storage
  - icoqgr, coqdepends

File dep. graph → compilation commands → compiled files → affected proofs
Source files → compilation commands
Proof dep. graph → compilation commands

Check proof dependencies with icoqgr, icoqc
<table>
<thead>
<tr>
<th>Project</th>
<th>Area</th>
<th>LOC</th>
<th>#Revisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flocq</td>
<td>floating-point math</td>
<td>25k</td>
<td>24</td>
</tr>
<tr>
<td>UniMath</td>
<td>math foundations</td>
<td>43k</td>
<td>24</td>
</tr>
<tr>
<td>Verdi</td>
<td>distributed systems</td>
<td>54k</td>
<td>24</td>
</tr>
</tbody>
</table>
## Evaluation: Open Source Git-Based Projects

<table>
<thead>
<tr>
<th>Project</th>
<th>Area</th>
<th>LOC</th>
<th>#Revisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flocq</td>
<td>floating-point math</td>
<td>25k</td>
<td>24</td>
</tr>
<tr>
<td>UniMath</td>
<td>math foundations</td>
<td>43k</td>
<td>24</td>
</tr>
<tr>
<td>Verdi</td>
<td>distributed systems</td>
<td>54k</td>
<td>24</td>
</tr>
</tbody>
</table>

For both checking-from-scratch and when using iCoq, measured:

1. total & average \#proofs checked
2. total & average proof checking time in seconds

... for all revisions of each project
### Reduction in #proofs to check using iCoq

<table>
<thead>
<tr>
<th>Project</th>
<th>Total</th>
<th>iCoq</th>
<th>$P_{sel}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flocq</td>
<td>22482</td>
<td>2164</td>
<td>N/A</td>
</tr>
<tr>
<td>Avg.</td>
<td>936.75</td>
<td>90.16</td>
<td>9.62</td>
</tr>
<tr>
<td>UniMath</td>
<td>17754</td>
<td>853</td>
<td>N/A</td>
</tr>
<tr>
<td>Avg.</td>
<td>739.75</td>
<td>35.54</td>
<td>4.85</td>
</tr>
<tr>
<td>Verdi</td>
<td>65413</td>
<td>4458</td>
<td>N/A</td>
</tr>
<tr>
<td>Avg.</td>
<td>2725.54</td>
<td>185.75</td>
<td>6.80</td>
</tr>
</tbody>
</table>

$P_{sel}$: proof selection percentage
## Reduction in checking time using iCoq

<table>
<thead>
<tr>
<th>Project</th>
<th>( \sum )</th>
<th>Avg.</th>
<th>coq_makefile</th>
<th>iCoq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flocq</td>
<td>888.36</td>
<td>37.01</td>
<td>303.71</td>
<td>12.65</td>
</tr>
<tr>
<td>UniMath</td>
<td>12882.46</td>
<td>536.76</td>
<td>3742.88</td>
<td>155.95</td>
</tr>
<tr>
<td>Verdi</td>
<td>32528.57</td>
<td>1355.35</td>
<td>3379.37</td>
<td>140.80</td>
</tr>
</tbody>
</table>

end-to-end time in seconds, including all phases
## Ratios of Total Times

<table>
<thead>
<tr>
<th>Project</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flocq</td>
<td>2.92</td>
</tr>
<tr>
<td>UniMath</td>
<td>3.44</td>
</tr>
<tr>
<td>Verdi</td>
<td>9.62</td>
</tr>
</tbody>
</table>

bottom line: speedups up to $10 \times$ vs. checking from scratch
Parallelism and Selection?

- iCoq evaluation uses *sequential* proof checking
- selection is *orthogonal* to parallelization
- in ISSTA ’18 paper, we do combined selection/parallelization
- speedups are up to $28 \times$
Conclusion

- **tool**, iCoq, for Coq regression proof selection, giving
  - **speedup** of up to $10 \times$

Contact us:

- Ahmet Celik (ahmetcelik@utexas.edu)
- **Karl Palmskog** (palmskog@utexas.edu)
- Milos Gligoric (gligoric@utexas.edu)

Resources:

- Proof selection technique paper in ASE ’17
- Website: [http://cozy.ece.utexas.edu/icoq/](http://cozy.ece.utexas.edu/icoq/)
- GitHub: [https://github.com/proofengineering/icoq](https://github.com/proofengineering/icoq)

This work was partially supported by the US National Science Foundation under Grants Nos. CCF-1438982, CCF-1566363, and CCF-1652517, and a Google Faculty Research Award.
Coq v8.5 Asynchronous Proof-Checking Toolchain

- newly-added toolchain can produce .vio files without proofs
- .vio files contain proof tasks, checked asynchronously

Legacy workflow:
- `AltLem.v`
- `coqc` with `coqc -quick`
- `AltLem.vo` with `coqc -check-vio-tasks`

New workflow:
- `AltLem.v` with `coqc -quick`
- `AltLem.vio` with `coqc -check-vio-tasks`
The under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task.

Yang et al., PLDI ’11

No bugs were found in the distributed protocols of verified systems, despite that we specifically searched for protocol bugs and spent more than eight months in this process.

Fonseca et al., EuroSys ’17
Sources of Overhead

- quick-compilation
- asynchronous proof checking book-keeping
- change analysis (hashing)
- dependency extraction

ICOQ targeted at large-scale Coq projects
Safety for RPS Techniques

<table>
<thead>
<tr>
<th>Definition (RTS Safety)</th>
<th>For every possible change to a project, the technique never omits to run a test affected by the change.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition (RPS Safety?)</td>
<td>For every possible change to a project, the technique never omits to check a proof affected by the change.</td>
</tr>
</tbody>
</table>
Potential iCoq Issues with Safety

- tactic language and plugin dependencies
- universe constraints
- proofs in parameterized modules
in use for over 40 years, mostly in academia

- can yield higher confidence than testing, model checking, ...
- expensive to apply (expertise, time, opportunity cost, ...)

Proof Assistants In Perspective

- Automath: 1968
- SLCF: 1972
- ELCF: 1978
- Nuprl: 1984
- Coq: 1985
- Isabelle: 1986
- HOL: 1988
- HOL Light: 1996
- Lean: 2015
Historical Obstacles to RPS

- LCF proof assistants required use of read-eval-print loop
- proofs could not be processed out-of-order
- all interaction was synchronous

# coqtop
Welcome to Coq 8.5.3 (October 2017)

Coq < Lemma alt_exists : forall l1 l2, exists l3, alt l1 l2 l3.
1 subgoal

=============
forall l1 l2 : list nat, exists l3 : list nat, alt l1 l2 l3

alt_exists < induction l1; intros; destruct l2.