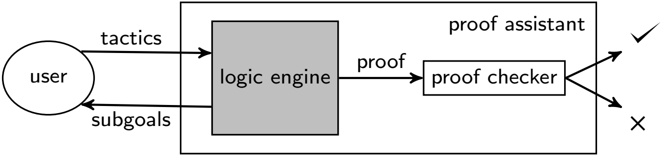
A Regression Proof Selection Tool For Coq

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 **interactively** using powerful **tactics**
3. check soundness of every low-level step



examples: **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.
```

1. Coq program

```
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.
```

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.
```

1. Coq program

```
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.
```

2. Coq spec/proof

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

3. OCaml program

Some Large-Scale Verification Projects

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

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.

iCoq: Regression Proof Selection for Coq

Our iCoq tool:

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

iCoq: Regression Proof Selection for Coq

Our iCoq tool:

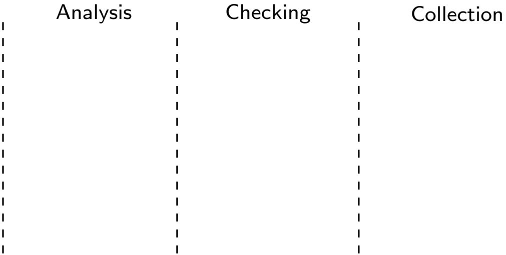
- ... implements regression proof selection for Coq projects
- ... tracks dependencies between Coq definitions and proofs
- ... checks only affected proofs after each change

Tool phases:

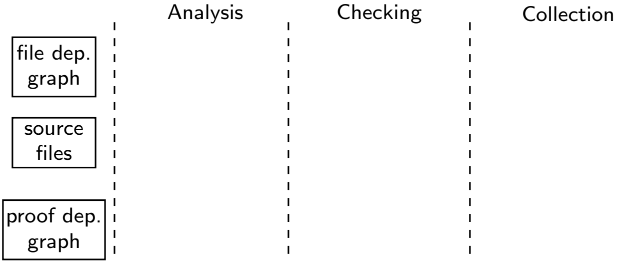
1. **analysis**: locate proofs affected by changes
2. **execution**: emit and run proof-checking commands
3. **collection**: find dependencies of modified definition/proofs

Key idea: maintain file and proof **dependency graphs**

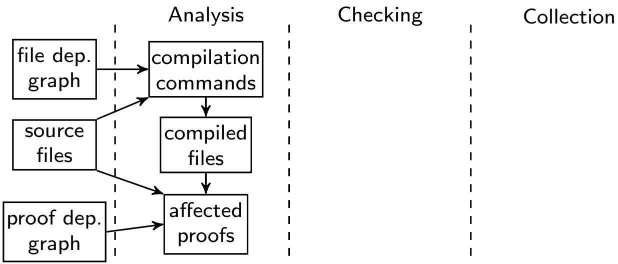
iCoq Workflow



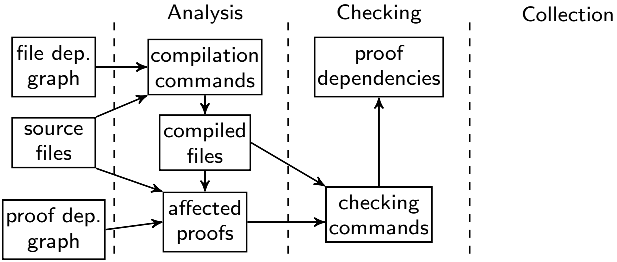
iCoq Workflow



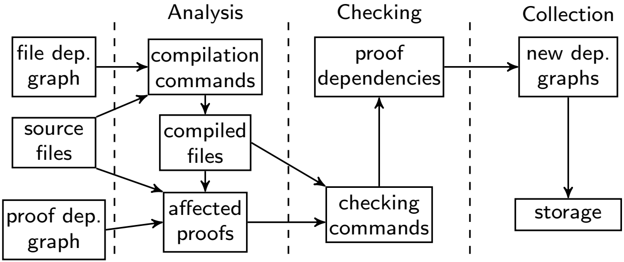
iCoq Workflow



iCoq Workflow


Analysis
Checking
Collection
file dep. graph
source files
proof dep. graph
compilation commands
compiled files
affected proofs
proof dependencies
checking commands

iCoq Workflow



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.
```

Alternate.v

```
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

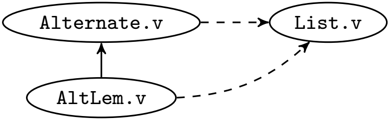
Example, revision 1

```
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.
```

`Alternate.v`

```
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

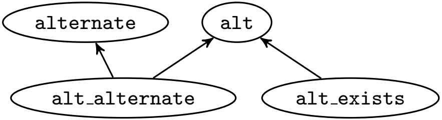
Example, revision 1

```
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.
```

`Alternate.v`

```
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.
```

`Alternate.v`

```
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`

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.
```

`Alternate.v`

```
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`

Change creates new revision

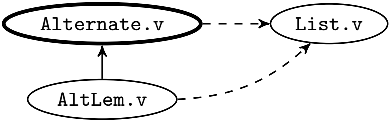
Example, revision 2

```
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.
```

Alternate.v

```
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

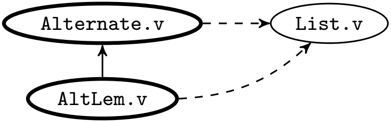
Example, revision 2

```
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.
```

Alternate.v

```
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

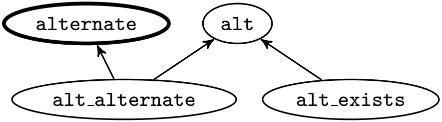
Example, revision 2

```
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.
```

`Alternate.v`

```
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

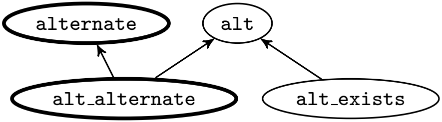
Example, revision 2

```
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.
```

`Alternate.v`

```
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.
```

`Alternate.v`

```
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`

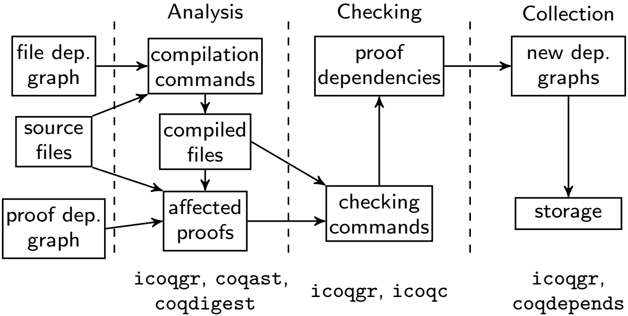
iCoq checks proof of `alt_alternate`

… but not proof of `alt_exists`!

iCoq Components

Component	**Lang.**	**Description**
`coqast`	OCaml	compute digests of proof ASTs
`coqdepends`	OCaml	extract dependencies from proof ASTs
`coqdigest`	OCaml	compute digests of proof scripts
`icoqc`	OCaml	proof-checking dependency extraction
`icoqgr`	Java/bash	construct/maintain dependency graphs

Components and the iCoq Workflow



Analysis

Checking

Collection

file dep. graph

source files

proof dep. graph

compilation commands

compiled files

affected proofs

proof dependencies

checking commands

new dep. graphs

storage

`icoqgr`, `coqast`, `coqdigest`

`icoqgr`, `icoqc`

`icoqgr`, `coqdepends`

Evaluation: Open Source Git-Based Projects

Project	**Area**	**LOC**	**#Revisions**
Flocq	floating-point math	25k	24
UniMath	math foundations	43k	24
Verdi	distributed systems	54k	24

Evaluation: Open Source Git-Based Projects

Project	**Area**	**LOC**	**#Revisions**
Flocq	floating-point math	25k	24
UniMath	math foundations	43k	24
Verdi	distributed systems	54k	24

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

Project		**Proofs** **Total**	 **iCoq**	 P^{sel}
Flocq	$\sum$	22482	2164	N/A
	Avg.	936.75	90.16	9.62
UniMath	$\sum$	17754	853	N/A
	Avg.	739.75	35.54	4.85
Verdi	$\sum$	65413	4458	N/A
	Avg.	2725.54	185.75	6.80

P^{sel}: proof selection percentage

Reduction in checking time using iCoq

Project		**Time [s]** **coq_makefile**	 **iCoq**
Flocq	$\sum$	888.36	303.71
	Avg.	37.01	12.65
UniMath	$\sum$	12882.46	3742.88
	Avg.	536.76	155.95
Verdi	$\sum$	32528.57	3379.37
	Avg.	1355.35	140.80

end-to-end time in seconds, including all phases

Ratios of Total Times

Project	**Ratio**
Flocq	2.92
UniMath	3.44
Verdi	9.62

bottom line: speedups up to 10× 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×

Conclusion

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

Contact us:

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

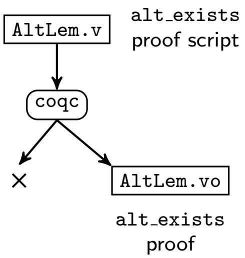
Resources:

- Proof selection technique paper in ASE '17
- Website: `http://cozy.ece.utexas.edu/icoq/`
- GitHub: `https://github.com/proofengineering/icoq`

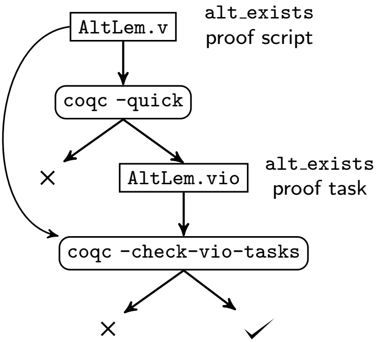
This work was partially supported by the US National Science Foundation under Grants Nos. CCF-1438982, CCF-1566363, and CCF-1652517, and a Google Faculty Research Award.

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



New workflow

Impact of Proof Assistants

> *[T]he 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

Definition (RTS Safety)

For every possible change to a project, the technique never omits to run a test affected by the change.

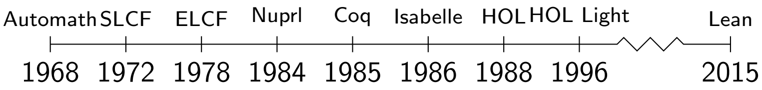
Definition (RPS Safety?)

For every possible change to a project, the technique never omits to check a proof affected by the change.

Potential iCoq Issues with Safety

- tactic language and plugin dependencies
- universe constraints
- proofs in parameterized modules

Proof Assistants In Perspective



Automath	SLCF	ELCF	Nuprl	Coq	Isabelle	HOL	HOL Light	Lean
1968	1972	1978	1984	1985	1986	1988	1996	2015

- in use for over 40 years, mostly in academia
- can yield higher confidence than testing, model checking, ...
- expensive to apply (expertise, time, opportunity cost, ...)

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.
```