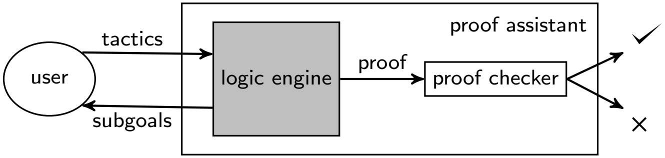
iCoq: Regression Proof Selection for Large-Scale Verification Projects

Karl Palmskog
University of Illinois at Urbana-Champaign

Joint work with Ahmet Celik and Milos Gligoric at UT Austin

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, ...

Some Large-Scale Proof Assistant Projects

Project	**Year**	**Assistant**	**Check Time**	**LOC**
4-Color Theorem	2005	Coq	hours	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.

Our Contribution

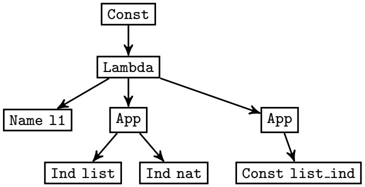
1. propose a regression proof selection (RPS) technique
2. implement RPS technique in tool, iCoq, which supports Coq
3. evaluate iCoq on revision histories of large Coq projects

Coq in a Nutshell

- based on a constructive dependent type theory (CiC)
- definitions + proofs programmed in Gallina/Ltac (`.v` files)
- coqc tool processes `.v` files, outputs terms (`.vo` files)

```
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



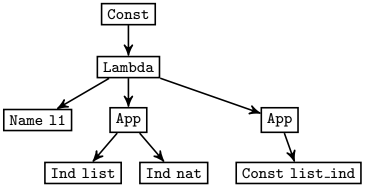
AltLem.vo

Coq in a Nutshell

- based on a constructive dependent type theory (CiC)
- definitions + proofs programmed in Gallina/Ltac (`.v` files)
- coqc tool processes `.v` files, outputs terms (`.vo` files)

```
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



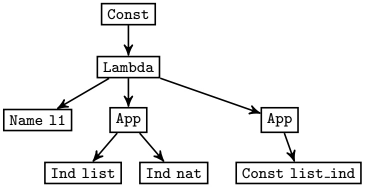
AltLem.vo

Coq in a Nutshell

- based on a constructive dependent type theory (CiC)
- definitions + proofs programmed in Gallina/Ltac (`.v` files)
- coqc tool processes `.v` files, outputs terms (`.vo` files)

```
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`



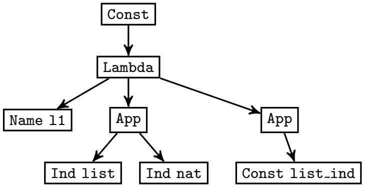
`AltLem.vo`

Coq in a Nutshell

- based on a constructive dependent type theory (CiC)
- definitions + proofs programmed in Gallina/Ltac (`.v` files)
- coqc tool processes `.v` files, outputs terms (`.vo` files)

```
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.
```

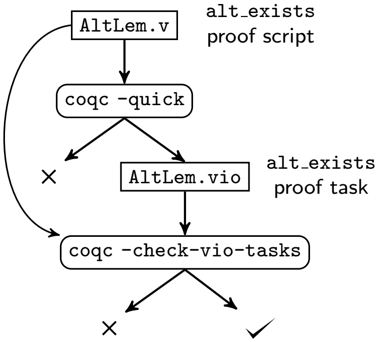
AltLem.v



AltLem.vo

Coq v8.5 Asynchronous Proof-Checking Toolchain

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



Regression Proof Selection Technique

Three phases:

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

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

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

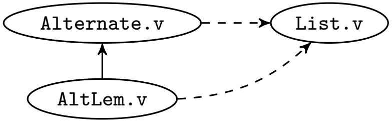
Example, revision 1

```
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


$$\begin{tikzcd}
\text{Alternate.v} \arrow[r, dashed] & \text{List.v} \\
\text{AltLem.v} \arrow[u] \arrow[ur, dashed] &
\end{tikzcd}$$

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

```
# coqc -quick Alternate.v

# coqc -quick AltLem.v

# coqc -check-vio-tasks 0 Alternate.vio

# coqc -check-vio-tasks 0 AltLem.vio
```

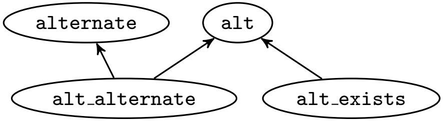
Example, revision 1

```
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`



Identifier 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.

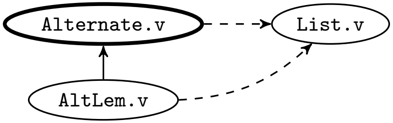
Example, revision 2

```
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


$$\begin{tikzcd} \boxed{\text{Alternate.v}} \arrow[r, dashed] & \text{List.v} \\ \text{AltLem.v} \arrow[u] \arrow[ur, dashed] & \end{tikzcd}$$

File dependency graph

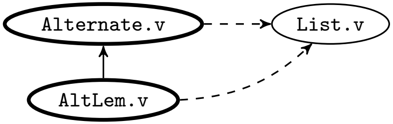
Example, revision 2

```
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


$$\begin{tikzcd}
\text{Alternate.v} \arrow[r, dashed] & \text{List.v} \\
\text{AltLem.v} \arrow[u] \arrow[ur, dashed] &
\end{tikzcd}$$

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

```
# coqc -quick Alternate.v
```

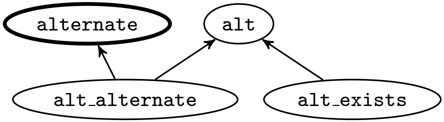
Example, revision 2

```
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.
```

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



Identifier dependency graph

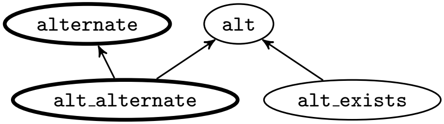
Example, revision 2

```
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



Identifier 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.
```

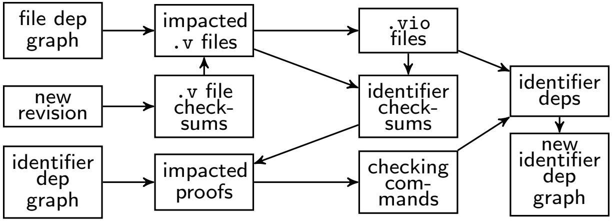
AltLem.v

```
# coqc -check-vio-tasks 0 Alternate.vio
```

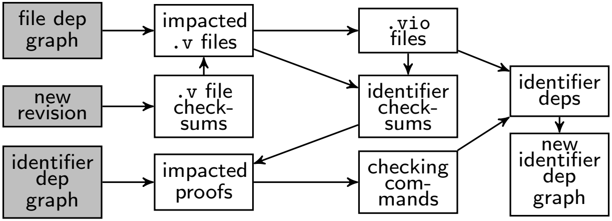
iCoq Components

Component	**Lang.**	**Description**
`coq-ast`	OCaml	compute digests of term ASTs
`coq-depends`	OCaml	extract dependencies from term ASTs
`coq-digest`	OCaml	compute digests of proof scripts
`coqc` extension	OCaml	proof-checking dependency extraction
graph builder	Java	construct/maintain dependency graphs
proof runner	bash	execute proofs

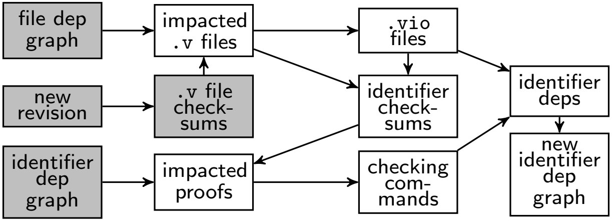
iCoq Workflow



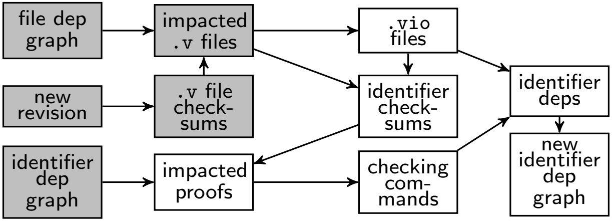
iCoq Workflow



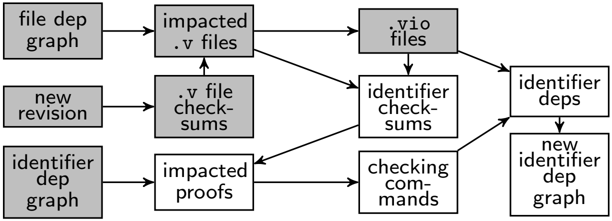
iCoq Workflow



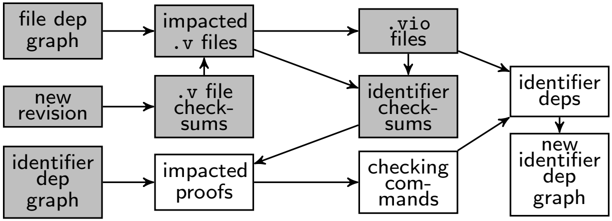
iCoq Workflow



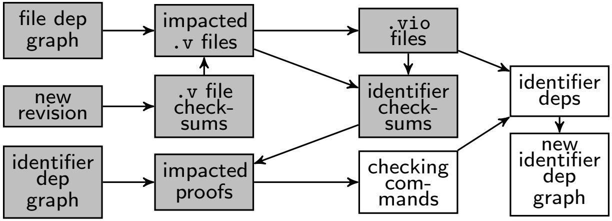
iCoq Workflow



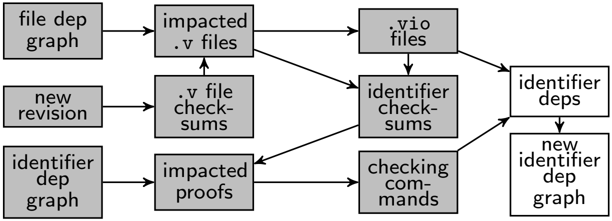
iCoq Workflow



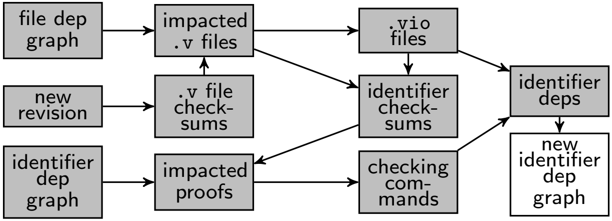
iCoq Workflow



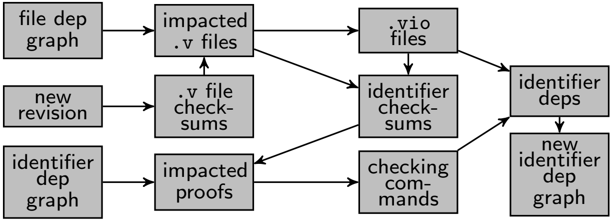
iCoq Workflow



iCoq Workflow



ICoq Workflow



Evaluation Research Questions

- RQ1 How effective is iCoq in terms of reduction in the number of checked proofs?
- RQ2 How effective is iCoq in terms of the proof checking time **inside** a CIS environment?
- RQ3 How effective is iCoq in terms of the proof checking time **outside** a CIS environment?

CIS: Continuous Integration Service (e.g., Travis CI)

Evaluation: Open Source Git-Based Projects

Project	**LOC**	**#Revisions**
CTLTCTL	0.6k	10
InfSeqExt	1.8k	10
StructTact	2.5k	10
WeakUpTo	1.8k	10
Flocq	25k	24
UniMath	43k	24
Verdi	54k	24

Evaluation Environments

1. **CI-Env** uses a CIS to check proofs (from scratch)
2. **LO-Env** uses local machine to check proofs (incrementally)

Experiments run on Intel Core i7-6700 CPU @ 3.40GHz with 16GB of RAM, running Ubuntu 14.04 LTS.

Dependency data always persisted between revision (cached).

RQ1: reduction in #proofs to check using ICoq

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

P^{sel}: proof selection percentage

RQ2: reduction in checking time using iCoq in CI-Env

Project		**CI-Env 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

RQ3: reduction in checking time using iCoq in LO-Env

Project		LO-Env Time [s] coq_makefile	LO-Env Time [s] iCoq
Flocq	$\sum$	297.97	261.62
	Avg.	12.41	10.90
UniMath	$\sum$	3783.52	1692.33
	Avg.	157.64	70.51
Verdi	$\sum$	8157.45	3130.96
	Avg.	339.89	130.45

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

Ratios of Total Times

Project	**CI-Env**	**LO-Env**
Flocq	2.92	1.13
UniMath	3.44	2.23
Verdi	9.62	2.60

Conclusion

- **technique**, regression proof selection, implemented in
- **tool**, iCoq, for Coq projects, giving
- **speedup** of up to 10× in CIS, 3× locally

Contact us:

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

Resources:

- download iCoq: `http://cozy.ece.utexas.edu/icoq`
- Coq components: `https://github.com/proofengineering`

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.