# Regression Proving with Dependent Types: Theory and Practice

Karl Palmskog

`https://setoid.com`

The University of Texas at Austin

Joint work with Ahmet Celik, Chenguang Zhu, and Milos Gligoric

# Regression Proving with Dependent Types: Practice and Some Theory

Karl Palmskog

https://setoid.com

The University of Texas at Austin

Joint work with Ahmet Celik, Chenguang Zhu, and Milos Gligoric

# “Scale Changes Everything”

| Project | Year | Assistant | Check Time | LOC |
|---|---|---|---|---|
| 4-Color Theorem | 2005 | Coq | tens of mins | 60k |
| Odd Order Theorem | 2012 | Coq | tens of mins | 150k |
| Kepler Conjecture | 2015 | HOL Light | days | 500k |
| CompCert | 2009 | Coq | tens of mins | 40k |
| seL4 | 2009 | Isabelle/HOL | hours | 200k |
| Cogent BilbyFS | 2016 | Isabelle/HOL | days | 14k |
| Verdi Raft | 2016 | Coq | tens of mins | 50k |

# Proof Engineering Can Help

*"[T]he activity of construction, maintenance, documentation and presentation of large formal proof developments."*

—David Aspinall

# Proof Engineering Can Help

*"[T]he activity of construction, maintenance, documentation and presentation of large formal proof developments."*

—David Aspinall

## This talk

1. techniques for faster checking of *evolving* projects (**for** Coq)
2. formalization and verification of these techniques (**in** Coq)

# Our Working Analogy: Proofs $\sim$ Tests

- tests are “partial functional specifications” of programs
- proofs represent many, usually an infinite number of, tests
- does not fit all projects in mathematics well

```
Fixpoint app {A} (l m:list A)
:= match l with
   | [] ⇒ m
   | a :: l’ ⇒ a :: app l’ m
   end.
```

1. Coq function

# Our Working Analogy: Proofs ∼ Tests

- tests are "partial functional specifications" of programs
- proofs represent many, usually an infinite number of, tests
- does not fit all projects in mathematics well

```
Fixpoint app {A} (l m:list A)
:= match l with
   | [] ⇒ m
   | a :: l' ⇒ a :: app l' m
   end.
```

1. Coq function

```
Lemma asoc: ∀ A (l m n:list A),
app l(app m n) = app(app l m) n.
Proof.
induction l; intros; auto.
simpl; rewrite IHl; auto.
Qed.
```

2. Coq lemma

# Our Working Analogy: Proofs $\sim$ Tests

- tests are "partial functional specifications" of programs
- proofs represent many, usually an infinite number of, tests
- does not fit all projects in mathematics well

```
Fixpoint app {A} (l m:list A)
:= match l with
   | [] ⇒ m
   | a :: l' ⇒ a :: app l' m
   end.
```

1. Coq function

```
Lemma asoc: ∀ A (l m n:list A),
app l(app m n) = app(app l m) n.
Proof.
induction l; intros; auto.
simpl; rewrite IHl; auto.
Qed.
```

2. Coq lemma

```
let test_app_assoc ctxt =
 assert_equal
  (app [1] (app [2] [3]))
  (app (app [1] [2]) [3])
```

3. OCaml test

# 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 much later (for seemingly unrelated proof)

# 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 much 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 much later (for seemingly unrelated test)

# Basic Techniques For More Efficient Regression Proving

## Proof selection: check only proofs affected by changes

- file/module selection
- asynchronous proof checking

Examples: Make, Isabelle [ITP '14]

# Basic Techniques For More Efficient Regression Proving

## Proof selection: check only proofs affected by changes

- file/module selection
- asynchronous proof checking

Examples: Make, Isabelle [ITP '14]

## Proof parallelization: leverage multi-core hardware

- parallel checking of proofs
- parallel checking of files

Examples: Make, Isabelle [ITP '13], Coq [ITP '15], Lean [CADE '15]

# Our Recent Work on Regression Proving in Practice

- taxonomy of regression proving techniques that leverage **both** selection and parallelism
- implementation of techniques in tool, iCoq, that supports Coq projects (useful for CI, e.g., Travis on GitHub)
- evaluation using iCoq on six open source projects (23 kLOC over 22 revisions per project, on average)

# Regression Proving Modes for Coq (Our Taxonomy)

| **Parallelization** | **Selection** | | |
| --- | --- | --- | --- |
| *Granularity* | *None* | *Files* | *Proofs* |
| File level | `f·none` | `f·file` | N/A |
| Proof level | `p·none` | `p·file` | `p·icoq` |

# Coq Proof-Checking Toolchain

## Legacy Top-Down Proof Checking (1990s)

- `coqc`: compilation of source `.v` files to binary `.vo` files
- `.vo` files contain **specifications and all proofs**
- file-level parallelism via Make

# Coq Proof-Checking Toolchain

## Legacy Top-Down Proof Checking (1990s)

- `coqc`: compilation of source `.v` files to binary `.vo` files
- `.vo` files contain **specifications and all proofs**
- file-level parallelism via Make

## Quick Compilation and Asynchronous Checking (2015)

- `coqc -quick`: compilation of `.v` files to binary `.vio` files
- `.vio` files contain **specifications and proof tasks**
- proof tasks checkable asynchronously in parallel

# Coq Source File Example

```
Require Import List.
Require Import ListUtil.

Import ListNotations.

Fixpoint dedup A A_eq_dec (xs : list A) : list A :=
match xs with
| [] ⇒ []
| x :: xs ⇒
  if in_dec A_eq_dec x xs then dedup A A_eq_dec xs
  else x :: dedup A A_eq_dec xs
end.

Lemma remove_dedup :
 ∀ A A_eq_dec (x : A) xs,
  remove A_eq_dec x (dedup A A_eq_dec xs) =
  dedup A A_eq_dec (remove A_eq_dec x xs).
Proof.
induction xs; intros; auto; simpl.
repeat (try case in_dec; try case A_eq_dec;
 simpl; intuition); auto using f_equal.
- exfalso. apply n0. apply remove_preserve; auto.
- exfalso. apply n. apply in_remove in i; intuition.
Qed.
```

Dedup.v

# Coq Source File Example

```
Require Import List.
Require Import ListUtil.

Import ListNotations.

Fixpoint dedup A A_eq_dec (xs : list A) : list A :=
match xs with
| [] ⇒ []
| x :: xs ⇒
  if in_dec A_eq_dec x xs then dedup A A_eq_dec xs
  else x :: dedup A A_eq_dec xs
end.

Lemma remove_dedup :
 ∀ A A_eq_dec (x : A) xs,
  remove A_eq_dec x (dedup A A_eq_dec xs) =
  dedup A A_eq_dec (remove A_eq_dec x xs).
Proof.
induction xs; intros; auto; simpl.
repeat (try case in_dec; try case A_eq_dec;
 simpl; intuition); auto using f_equal.
- exfalso. apply n0. apply remove_preserve; auto.
- exfalso. apply n. apply in_remove in i; intuition.
Qed.
```

Dedup.v

Require statements expressing file dependencies.

# Coq Source File Example

```
Require Import List.
Require Import ListUtil.

Import ListNotations.

Fixpoint dedup A A_eq_dec (xs : list A) : list A :=
match xs with
| [] ⇒ []
| x :: xs ⇒
  if in_dec A_eq_dec x xs then dedup A A_eq_dec xs
  else x :: dedup A A_eq_dec xs
end.

Lemma remove_dedup :
 ∀ A A_eq_dec (x : A) xs,
  remove A_eq_dec x (dedup A A_eq_dec xs) =
  dedup A A_eq_dec (remove A_eq_dec x xs).
Proof.
induction xs; intros; auto; simpl.
repeat (try case in_dec; try case A_eq_dec;
 simpl; intuition); auto using f_equal.
- exfalso. apply n0. apply remove_preserve; auto.
- exfalso. apply n. apply in_remove in i; intuition.
Qed.
```

Dedup.v

Definition of a recursive function to remove duplicate list elements in Gallina.
Processed by quick-compilation.

# Coq Source File Example

```
Require Import List.
Require Import ListUtil.

Import ListNotations.

Fixpoint dedup A A_eq_dec (xs : list A) : list A :=
match xs with
| [] ⇒ []
| x :: xs ⇒
  if in_dec A_eq_dec x xs then dedup A A_eq_dec xs
  else x :: dedup A A_eq_dec xs
end.

Lemma remove_dedup :
 ∀ A A_eq_dec (x : A) xs,
  remove A_eq_dec x (dedup A A_eq_dec xs) =
  dedup A A_eq_dec (remove A_eq_dec x xs).
Proof.
induction xs; intros; auto; simpl.
repeat (try case in_dec; try case A_eq_dec;
 simpl; intuition); auto using f_equal.
- exfalso. apply n0. apply remove_preserve; auto.
- exfalso. apply n. apply in_remove in i; intuition.
Qed.
```

Dedup.v

Statement (type) of a lemma in Gallina.

# Coq Source File Example

```
Require Import List.
Require Import ListUtil.

Import ListNotations.

Fixpoint dedup A A_eq_dec (xs : list A) : list A :=
match xs with
| [] ⇒ []
| x :: xs ⇒
  if in_dec A_eq_dec x xs then dedup A A_eq_dec xs
  else x :: dedup A A_eq_dec xs
end.

Lemma remove_dedup :
 ∀ A A_eq_dec (x : A) xs,
  remove A_eq_dec x (dedup A A_eq_dec xs) =
  dedup A A_eq_dec (remove A_eq_dec x xs).
Proof.
induction xs; intros; auto; simpl.
repeat (try case in_dec; try case A_eq_dec;
 simpl; intuition); auto using f_equal.
- exfalso. apply n0. apply remove_preserve; auto.
- exfalso. apply n. apply in_remove in i; intuition.
Qed.
```

Dedup.v

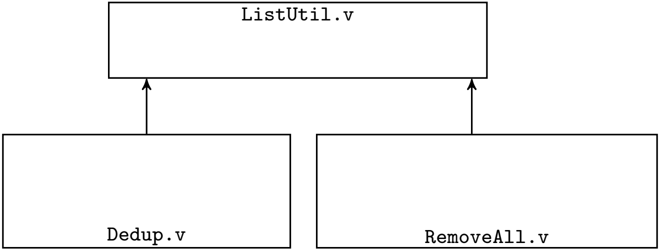
Proof script in Ltac – potentially time-consuming to process. Becomes proof task.

# f•none Mode: File-Level Parallelization, No Selection

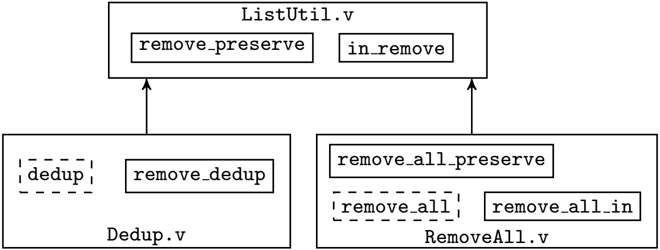
| **Parallelization** | **Selection** | | |
|---|---|---|---|
| *Granularity* | *None* | *Files* | *Proofs* |
| File level | `f•none` | `f•file` | N/A |
| Proof level | `p•none` | `p•file` | `p•icoq` |

- classic mode used in most GitHub projects ("ReproveAll")
- no overhead from proof task management or dep. tracking
- parallelism restricted by file dependency graph

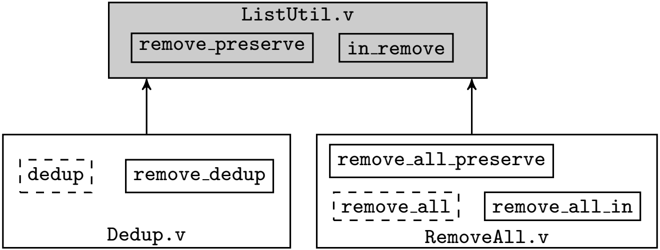
# f•none Mode in Practice



# f·none Mode in Practice

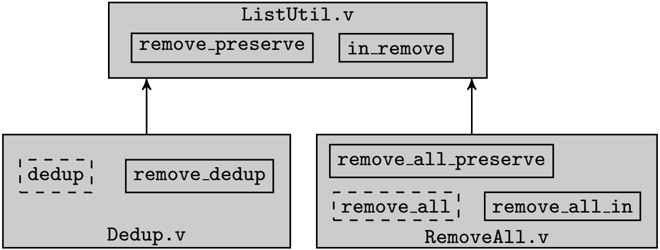


# f•none Mode in Practice




| Phase | Task | Definitions and Lemmas |
|---|---|---|
| 1 | ListUtil.vo | remove_preserve, in_remove |

# f•none Mode in Practice



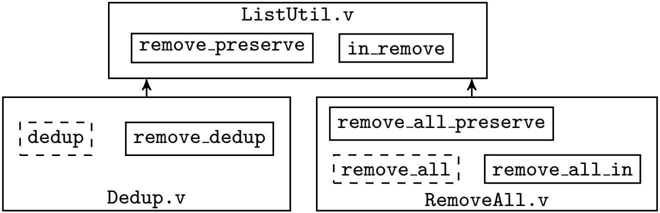

| Phase | Task | Definitions and Lemmas |
|---|---|---|
| 1 | ListUtil.vo | remove_preserve, in_remove |
| 2 | Dedup.vo | dedup, remove_dedup |
| 2 | RemoveAll.vo | remove_all, remove_all_in, remove_all_preserve |

# p•none Mode: Proof-Level Parallelization, No Selection

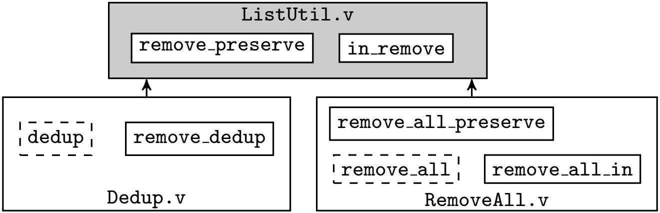
| Parallelization | Selection | | |
|---|---|---|---|
| *Granularity* | *None* | *Files* | *Proofs* |
| File level | f•none | f•file | N/A |
| Proof level | p•none | p•file | p•icoq |

- used in some GitHub Coq projects
- overhead from proof task management
- parallelism (largely) unrestricted by file dependency graph

# p·none Mode in Practice

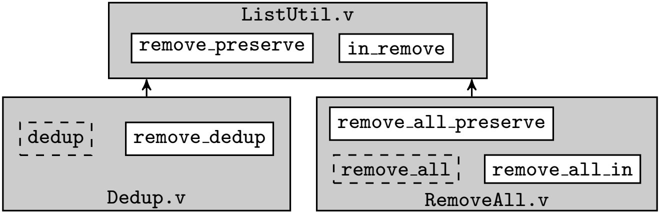


# p·none Mode in Practice



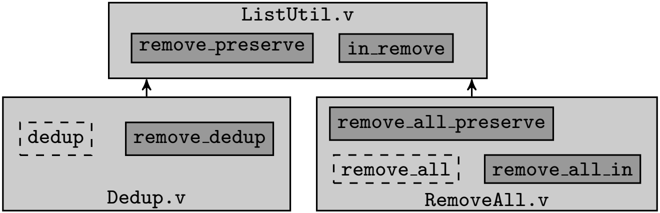

| Phase | Task | Definitions and Lemmas |
|---|---|---|
| 1 | ListUtil.vio | ~~remove_preserve~~, ~~in_remove~~ |

# p•none Mode in Practice




| Phase | Task | Definitions and Lemmas |
|---|---|---|
| 1 | ListUtil.vio | ~~remove_preserve~~, ~~in_remove~~ |
| 2 | Dedup.vio | dedup, ~~remove_dedup~~ |
| 2 | RemoveAll.vio | remove_all, ~~remove_all_in~~, ~~remove_all_preserve~~ |

# p·none Mode in Practice



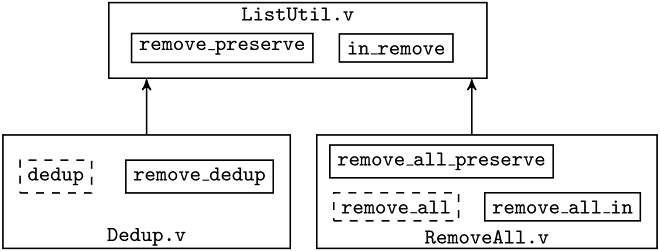

| Phase | Task | Definitions and Lemmas |
|---|---|---|
| 1 | ListUtil.vio | ~~remove_preserve~~, ~~in_remove~~ |
| 2 | Dedup.vio | dedup, ~~remove_dedup~~ |
| 2 | RemoveAll.vio | remove_all, ~~remove_all_in~~, ~~remove_all_preserve~~ |
| 3 | checking | remove_preserve |
| 3 | checking | in_remove |
| 3 | checking | remove_dedup |
| 3 | checking | remove_all_in |
| 3 | checking | remove_all_preserve |

# f•file Mode: File-Level Parallelization, File Selection

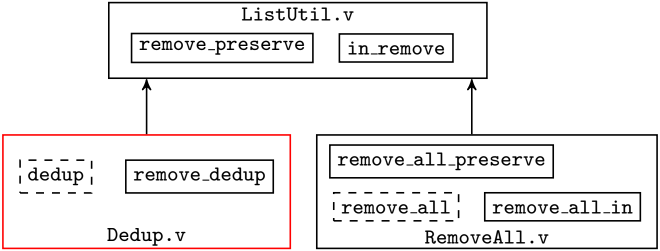
| **Parallelization** | **Selection** | | |
|---|---|---|---|
| *Granularity* | *None* | *Files* | *Proofs* |
| File level | `f•none` | `f•file` | N/A |
| Proof level | `p•none` | `p•file` | `p•icoq` |

- persists file checksums
- overhead from file dependency tracking
- parallelism restricted by file dependency graph

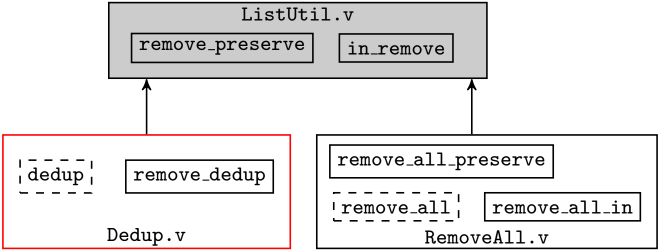
# f•file Mode in Practice



# f•file Mode in Practice

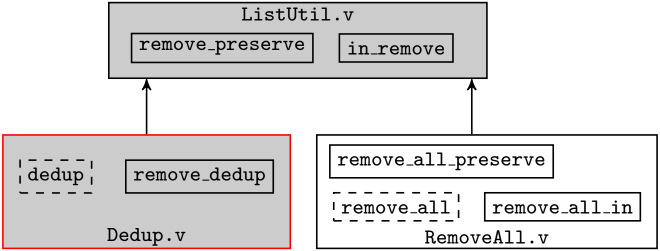


# f•file Mode in Practice




| Phase | Task | Definitions and Lemmas |
|---|---|---|
| 1 | ListUtil.vo | remove_preserve, in_remove |

# f·file Mode in Practice



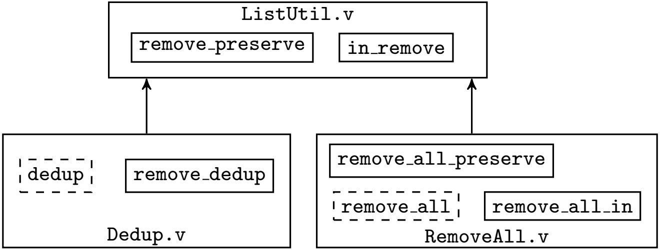

| Phase | Task | Definitions and Lemmas |
|---|---|---|
| 1 | ListUtil.vo | remove_preserve, in_remove |
| 2 | Dedup.vo | dedup, remove_dedup |

# p·file Mode: Proof-Level Parallelism, File Selection

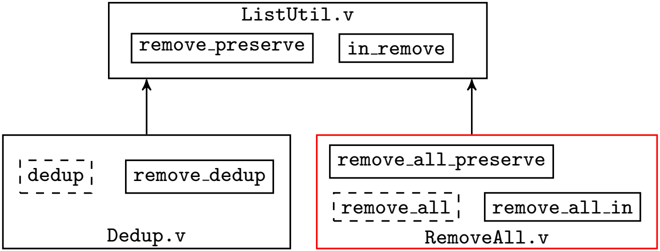
| **Parallelization** | **Selection** | | |
|---|---|---|---|
| *Granularity* | *None* | *Files* | *Proofs* |
| File level | `f·none` | `f·file` | N/A |
| Proof level | `p·none` | `p·file` | `p·icoq` |

- persists file checksums
- overhead from file dependency tracking
- parallelism (mostly) unrestricted by file dependency graph

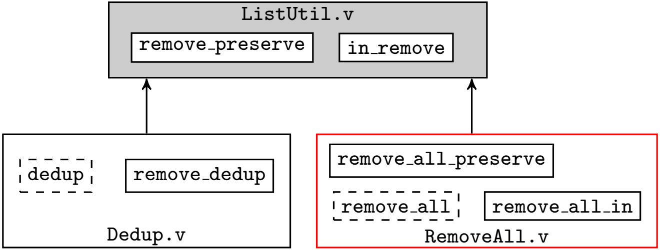
# p·file Mode in Practice



# p·file Mode in Practice

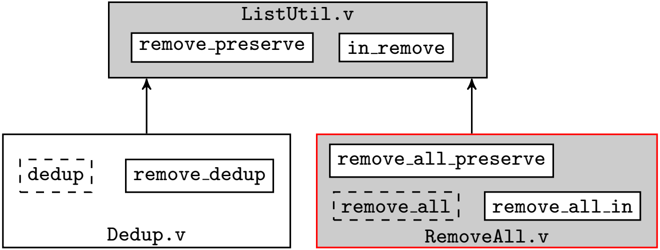


# p·file Mode in Practice



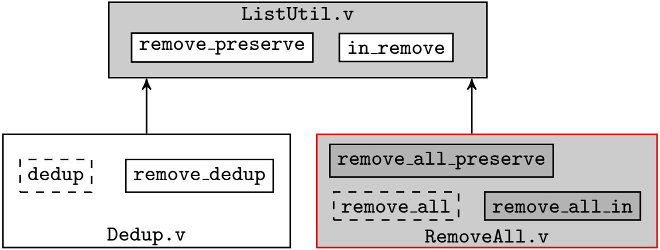

| Phase | Task | Definitions and Lemmas |
|---|---|---|
| 1 | ListUtil.vio | ~~remove_preserve~~, ~~in_remove~~ |

# p·file Mode in Practice




| Phase | Task | Definitions and Lemmas |
|---|---|---|
| 1 | ListUtil.vio | ~~remove_preserve~~, ~~in_remove~~ |
| 2 | RemoveAll.vio | remove_all, ~~remove_all_in~~, ~~remove_all_preserve~~ |

# p·file Mode in Practice



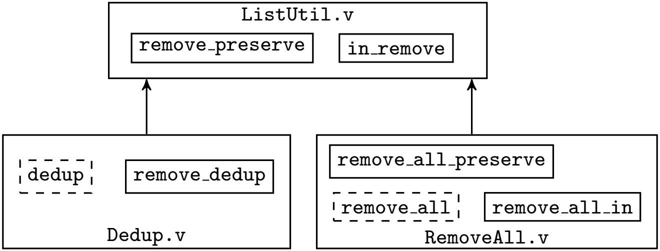

| Phase | Task | Definitions and Lemmas |
|---|---|---|
| 1 | ListUtil.vio | ~~remove_preserve~~, ~~in_remove~~ |
| 2 | RemoveAll.vio | remove_all, ~~remove_all_in~~, ~~remove_all_preserve~~ |
| 3 | checking | remove_all_in |
| 3 | checking | remove_all_preserve |

# p·icoq Mode: Proof-Level Parallelism, Proof Selection

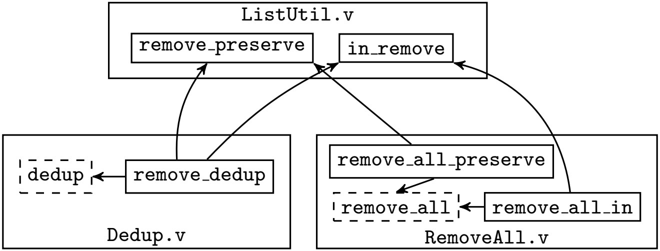
| Parallelization | Selection | | |
|---|---|---|---|
| *Granularity* | *None* | *Files* | *Proofs* |
| File level | f·none | f·file | N/A |
| Proof level | p·none | p·file | p·icoq |

- persists file & proof checksums
- overhead from file & proof dependency tracking
- parallelism (mostly) unrestricted by file dependency graph

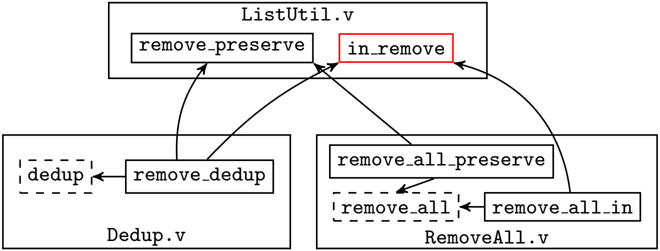
# p·icoq Mode in Practice



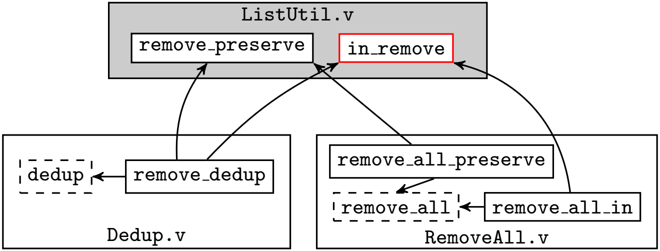
# p·icoq Mode in Practice



# p·icoq Mode in Practice

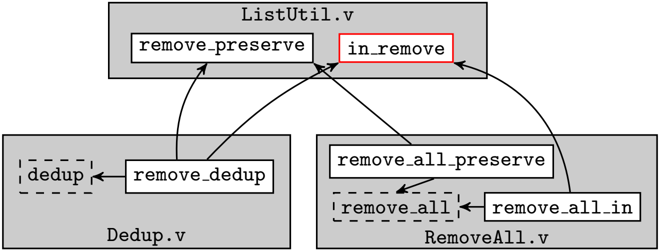


# p·icoq Mode in Practice



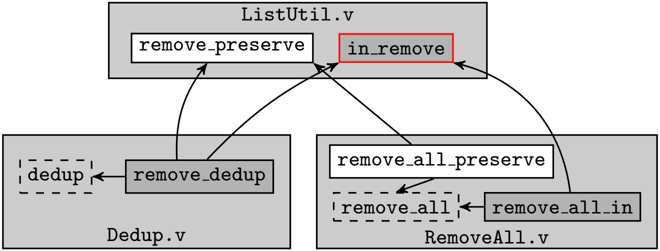

| Phase | Task | Definitions and Lemmas |
|---|---|---|
| 1 | ListUtil.vio | ~~remove_preserve~~, ~~in_remove~~ |

# p·icoq Mode in Practice




| Phase | Task | Definitions and Lemmas |
|---|---|---|
| 1 | ListUtil.vio | ~~remove_preserve~~, ~~in_remove~~ |
| 2 | Dedup.vio | dedup, ~~remove_dedup~~ |
| 2 | RemoveAll.vio | remove_all, ~~remove_all_in~~, ~~remove_all_preserve~~ |

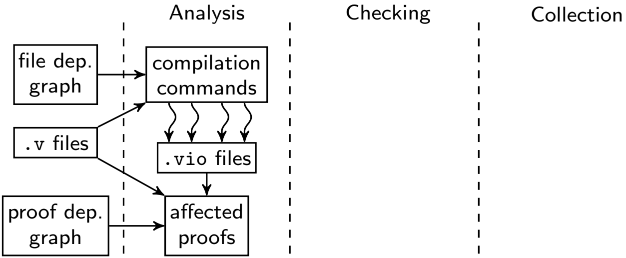
# p·icoq Mode in Practice



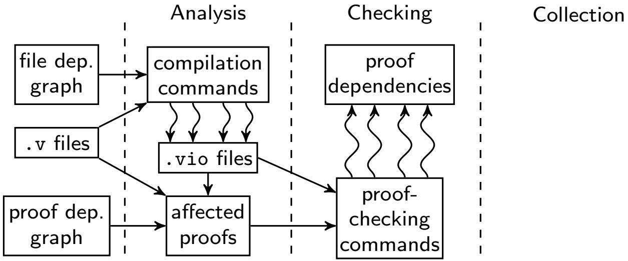

| Phase | Task | Definitions and Lemmas |
|---|---|---|
| 1 | ListUtil.vio | ~~remove_preserve~~, ~~in_remove~~ |
| 2 | Dedup.vio | dedup, ~~remove_dedup~~ |
| 2 | RemoveAll.vio | remove_all, ~~remove_all_in~~, ~~remove_all_preserve~~ |
| 3 | checking | in_remove |
| 3 | checking | remove_dedup |
| 3 | checking | remove_all_in |

# p·icoq Workflow with 4-way Parallelization

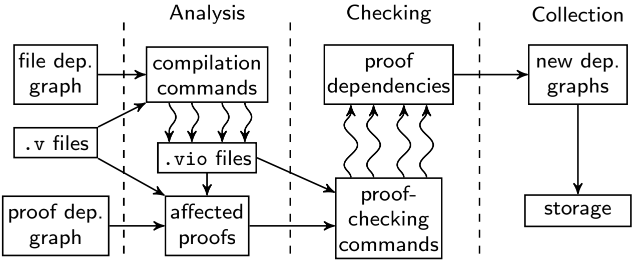
# p·icoq Workflow with 4-way Parallelization



# p·icoq Workflow with 4-way Parallelization



# p·icoq Workflow with 4-way Parallelization



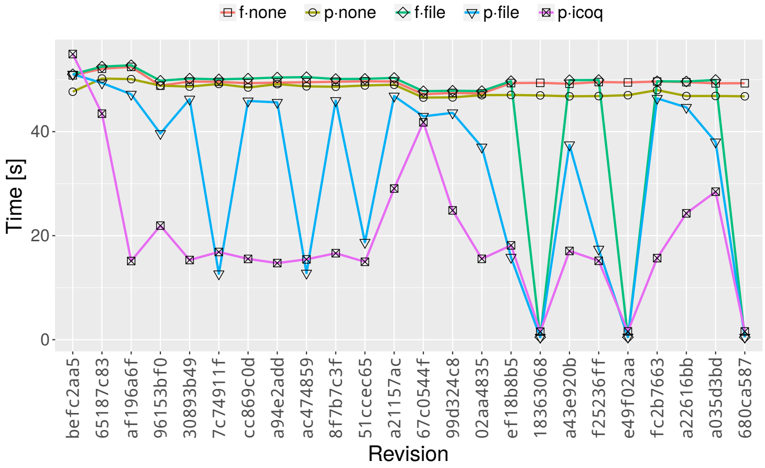
# Evaluation: Open Source Git-Based Projects

| Project | LOC | Domain |
|---|---:|---|
| Coquelicot | 38260 | real number analysis |
| Finmap | 5661 | finite sets and maps |
| Flocq | 24786 | floating-point arithmetic |
| Fomegac | 2637 | formal system metatheory |
| Surface Effects | 9621 | functional programming languages |
| Verdi | 56147 | distributed systems |
| $\sum$ | 137112 | |
| Avg. | 22852.00 | |

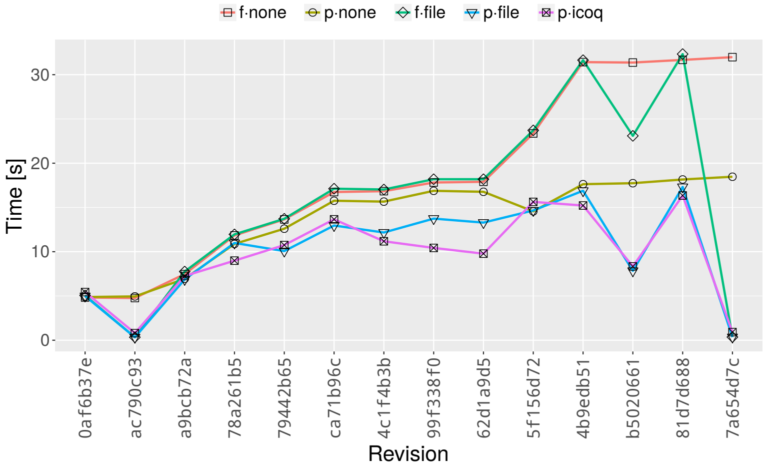
# Evaluation: Open Source Git-Based Projects

| **Project** | **LOC** | **#Revs.** | **#Files** | **#Proof Tasks** |
|---|---:|---:|---:|---:|
| Coquelicot | 38260 | 24 | 29 | 1660 |
| Finmap | 5661 | 23 | 4 | 959 |
| Flocq | 24786 | 23 | 40 | 943 |
| Fomegac | 2637 | 14 | 13 | 156 |
| Surface Effects | 9621 | 24 | 15 | 289 |
| Verdi | 56147 | 24 | 222 | 2756 |
| $\sum$ | 137112 | 132 | 323 | 6763 |
| Avg. | 22852.00 | 22.00 | 53.83 | 1127.16 |

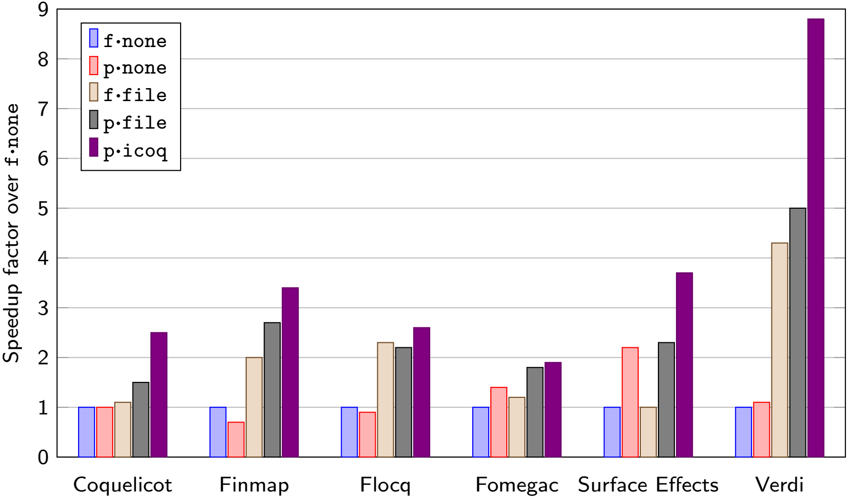
# Results with 4-way Parallelization: Coquelicot



# Results with 4-way Parallelization: Fomegac

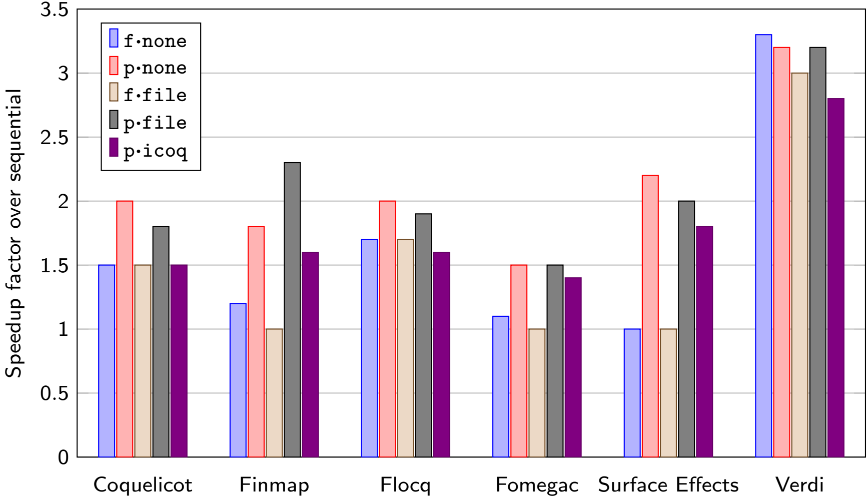


# Speedups over `f·none` for 4-way Parallel Checking




"How much faster modes are than the default mode, for each project"

# Speedups from Sequential to 4-way Parallel Checking




“Effect of parallelism on each mode and project”

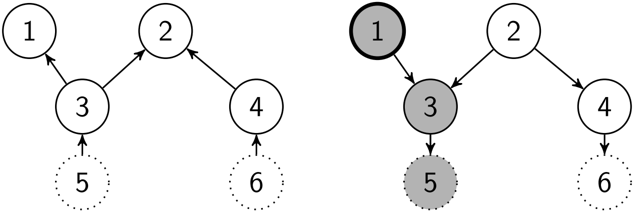
# Our Recent Work on the Theory of Regression Proving

We want to prove our regression proving techniques correct!

First steps:

- model of change impact analysis in Coq using MathComp
- practical tool, Chip, extracted from Coq code
- evaluation of Chip for regression testing and build tools

# Impact Analysis, Abstractly




(a) (b)

# Formal Model, Informally

- finite sets of vertices $V, V'$ where $V \subseteq V'$
- set $A$ of artifacts with decidable equality
- functions $f : V \to A$ and $f' : V' \to A$
- dependency graphs $g$ and $g'$ for vertices in $V$ and $V'$
- set $N \subseteq V'$ of checkable vertices
- operation *check* on vertices, with distinguishable results $R$

# Formal Model, Informally

**Modified Vertices**

A vertex $v \in V$ is <u>modified</u> whenever $f(v) \neq f'(v)$.

**Impacted Vertices**

A vertex $v \in V$ is <u>impacted</u> ($v \in I$) if it is reachable from some modified vertex in $g^{-1}$.

**Fresh Vertices**

A vertex $v \in V'$ is <u>fresh</u> ($v \in F$) whenever $v \notin V$.

We check all vertices in the set $(I \cup F) \cap N$.

# Encoding in Coq using MathComp (Sketch)

```
Variable (A : eqType).
Variables (V' : finType) (P : pred V').
Definition V := sig_finType P.
Variables (f' : V' → A) (f : V → A).

Definition impacted (g : rel V) (m : {set V}) : {set V} :=
\bigcup_( x | x \in m) [set y | connect g x y].

Definition impacted_V' g m := [set (val v) | v in impacted g^-1 m].
Definition fresh_V' := [set v | ~P v].

Definition mod_V := [set v | f v != f' (val v)].
Definition impacted_fresh_V' g := impacted_V' g mod_V :|: fresh_V'.
```

# Correctness Approach

- assume we have all tuples of vertices in $V$ and results of applying *check*
- then, we *check* on all impacted and fresh vertices, and add results and unimpacted-vertex tuples to form set $\mathcal{R}$
- is $\mathcal{R}$ complete: does it contain all checkable vertices in $V'$?
- is $\mathcal{R}$ sound: are all outcomes as if checked from scratch?

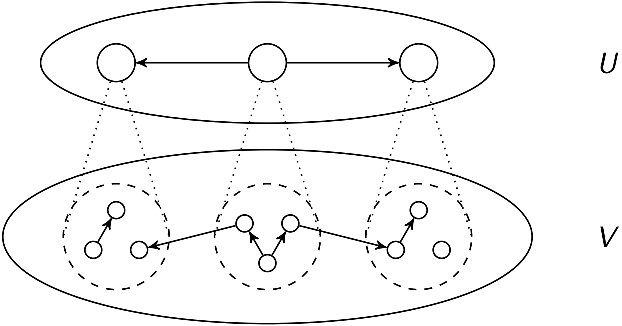
# Correctness in Coq (Sketch)

```
Variable (R : eqType).
Variables (g : rel V) (g' : rel V').
Variables (checkable : pred V) (checkable' : pred V').
Variables (check : V → R) (check' : V' → R).

Variable res_V : seq (V * R).
Hypothesis res_VP : ∀ v r,
 reflect (checkable v ∧ check v = r) ((v,r) \in res_V).

Definition res_unimpacted_V' := [seq (val vr.1, vr.2) |
  vr ← res_V & val vr.1 \notin impacted_V' g mod_V].
Definition res_V' := res_impacted_fresh_V' ++ res_unimpacted_V'.
Definition chk_V' := [seq vr.1 | vr ← res_V'].

Theorem chk_V'_compl : ∀ v, checkable' v → v \in chk_V'.
Theorem chk_V'_sound : ∀ v r, (v, r) \in res_V' →
 checkable' v ∧ check' v = r.
```

# Hierarchical Impact Analysis

- $U$ is set of coarse-grained components ("files")
- $V$ is set of fine-grained components ("proofs")
- $p : U \to 2^V$ is partition of $V$
- $g_\top$ is dep. graph for $U$, $g_\bot$ is dep. graph for $V$
- we can use impact analysis of $U$ and $g_\top$ to analyze $V$ and $g_\bot$

# Hierarchical Impact Analysis, Illustrated



# Hierarchical Strategies

## Overapproximation Strategy (similar to f·file)

- $U'_i$ is set of impacted and fresh vertices in $U'$
- let $V'_p = \bigcup_{u \in U'_i} p'(u)$
- check all checkable vertices in $V'_p$

## Compositional Strategy (similar to p·icoq)

- $U_i$ is set of impacted vertices in $U$
- let $V_p = \bigcup_{u \in U_i} p(u)$
- let $g_p$ be subgraph of $g_\perp$ induced by $V_p$
- perform impact analysis in $g_p$, check resulting vertices

# Tool Implementation and Evaluation

- extracted tool to OCaml from refined Coq code
- integrated with two test selection tools and one build tool
- compared outcomes/times with those for unmodified tools
- outcomes are the same and things run a little slower

# Conclusion

See our iCoq and piCoq papers and recommendations to Coq developers: `https://setoid.com`

Contact us:


- **Karl Palmskog**, `palmskog@utexas.edu`
- Ahmet Celik, `ahmetcelik@utexas.edu`
- Chenguang Zhu, `cgzhu@utexas.edu`
- Milos Gligoric, `gligoric@utexas.edu`