Fast Checking of Coq Proofs, in Theory and Practice

Karl Palmskog

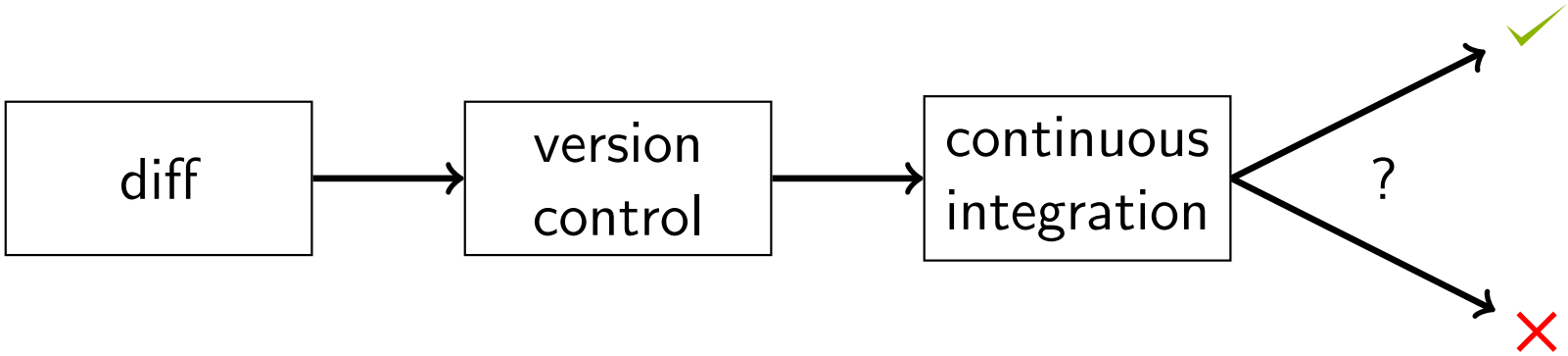
https://setoid.com

Joint work with Ahmet Celik and Milos Gligoric

Proof Asssistant Projects and Checking Time

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

Building and Checking in Coq Code Development



- development platforms: GitHub, GitLab, ...
- build systems: Make, Dune, ...
- scalability: size of change vs. size of codebase

Our Recent Work

1. techniques for faster checking of *evolving* projects, for Coq
   - locally and in continuous integration systems
   - evaluation on histories of large-scale projects such as UniMath

2. formalization and verification of these techniques, in Coq
   - theory using MathComp/SSReflect
   - extraction to OCaml

Basic Ideas For Faster Proving

Proof selection: check only proofs affected by changes

- file/module selection
- asynchronous proof checking

Examples: Make, Dune, Isabelle

Proof parallelization: leverage multi-core hardware

- parallel checking of proofs
- parallel checking of files

Examples: Make, Dune, Isabelle, Coq, Lean

Coq Source File Running Examples

```
Require Import List.
Import ListNotations.

Lemma remove_preserve : ∀ A A_eq_dec (x y : A) xs,
  x ≠ y → In y xs → In y (remove A_eq_dec x xs).
Proof.
induction xs; simpl; intros.
- intuition.
- case A_eq_dec; intros.
  + apply IHxs; subst; intuition.
  + intuition; subst; left; auto.
Qed.

Lemma in_remove : ∀ A A_eq_dec (x y : A) xs, In y (remove A_eq_dec x xs) → In y xs.
Proof.
induction xs; simpl; intros; auto.
destruct A_eq_dec; simpl in *; intuition.
Qed.
```

ListUtil.v

Coq Source File Running Examples

```
Require Import List ListUtil.
Import ListNotations.

Fixpoint dedup A A_eq_dec xs : 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 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 Import List ListUtil.
Import ListNotations.

Fixpoint remove_all A A_eq_dec rm l : list A :=
match rm with
| [] ⇒ l
| d :: ds ⇒
  let l' := remove A_eq_dec d l in
  remove_all A A_eq_dec ds l'
end.

Lemma remove_all_in : ∀ A A_eq_dec ds l x,
  In x (remove_all A A_eq_dec ds l) → In x l.
Proof.
induction ds; simpl; intros; intuition.
eauto using in_remove.
Qed.

Lemma remove_all_preserve :
  ∀ A A_eq_dec ds l x,
  ∼ In x ds → In x l →
  In x (remove_all A A_eq_dec ds l).
Proof.
induction ds; simpl; intros;
 intuition auto using remove_preserve.
Qed.
```

RemoveAll.v

Transparent vs. Opaque Proofs

```
Lemma remove_all_in : ∀ A A_eq_dec ds l x,
  In x (remove_all A A_eq_dec ds l) → In x l.
Proof.
induction ds; simpl; intros; intuition.
eauto using in_remove.
Qed.
```

- Proofs that end in `Qed` (vs. `Define`) are opaque
- Opaque proofs not generally accessible, not kept in memory
- Only opaque proofs can become "proof tasks"
- Problem: lemmas inside sections not fully defined in isolation
  - need `Proof using` annotations
  - ... or `Set Default Proof Using` declarations

Coq Proof-Checking Toolchain (1990s)

- coqc: compilation of source `.v` files to binary `.vo` files
- `.vo` files contain **constant types and bodies**
- file-level parallelism via Make (`coq_makefile`) or Dune

Commands:

```
coqc ListUtil.v
coqc Dedup.v
coqc RemoveAll.v
```

Results:

```
ListUtil.vo Dedup.vo RemoveAll.vo
```

Coq Quick Compilation Toolchain (2015)

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

Commands:

```
coqc -vio ListUtil.v
coqc -vio Dedup.v
coqc -vio RemoveAll.v
```

Results:

```
ListUtil.vio Dedup.vio RemoveAll.vio
```

Coq Vos/Vok Compilation Toolchain (2019)

- `coqc -vos`: compilation of `.v` files to binary `.vos` files
- `.vos` files contain **constant types**
- proofs not checkable at all

Commands:

```
coqc -vos ListUtil.v
coqc -vos Dedup.v
coqc -vos RemoveAll.v
```

Results:

```
ListUtil.vos Dedup.vos RemoveAll.vos
```

Coq Dependencies

- file dependencies obtained from `.v` files by `coqdep`
- term dependencies obtained via the `coq-dpdgraph` plugin
- but terms must be built before they can be analyzed!
- idea: keep track of checksums and dependencies of "previous" files and terms

`https://github.com/coq-community/coq-dpdgraph/`

Regression Proving Modes for Coq (Our Taxonomy)

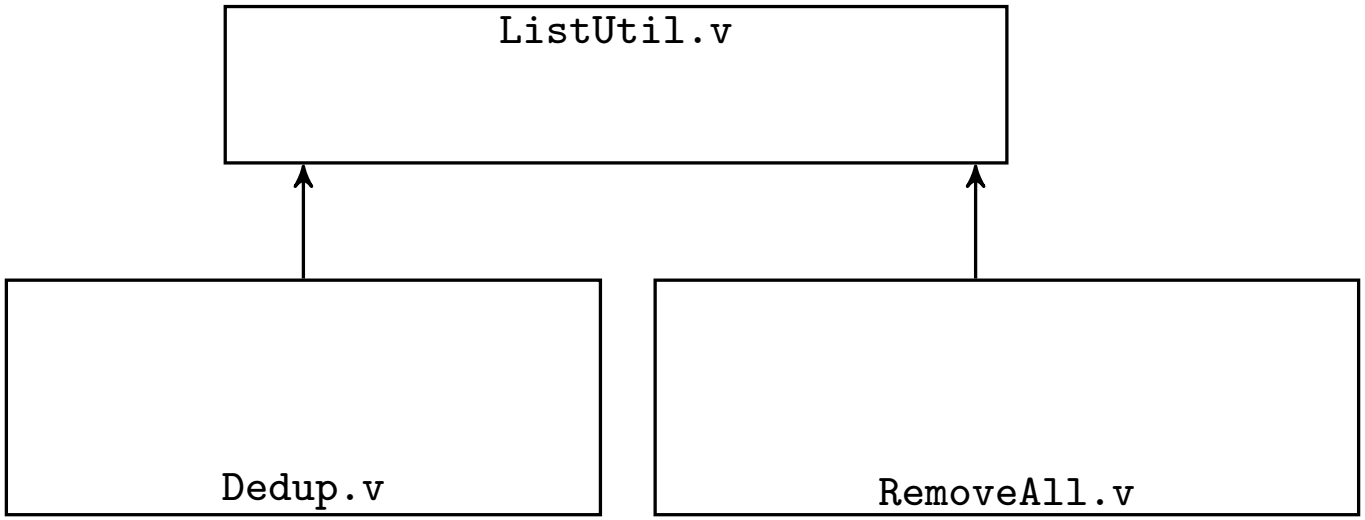
Parallelization	**Selection**		
Granularity	*None*	*Files*	*Proofs*
File level	`f·none`	`f·file`	N/A
Proof level	`p·none`	`p·file`	`p·icoq`

`f·none` Mode: File-Level Parallelization, No Selection

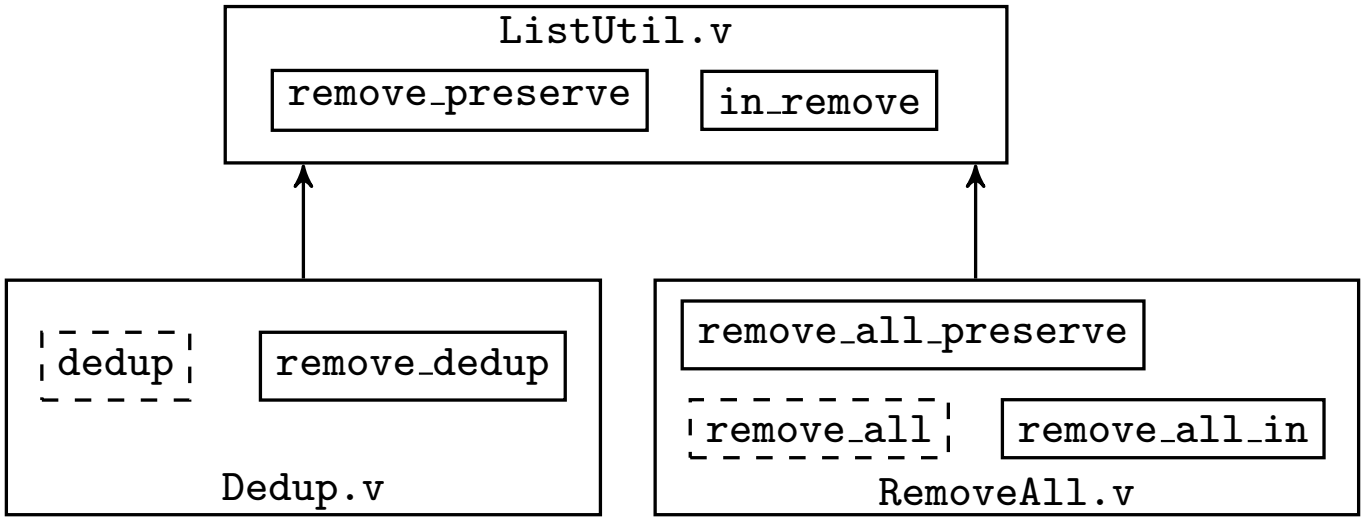
Parallelization	**Selection**		
Granularity	*None*	*Files*	*Proofs*
File level	`f·none`	`f·file`	N/A
Proof level	`p·none`	`p·file`	`p·icoq`

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

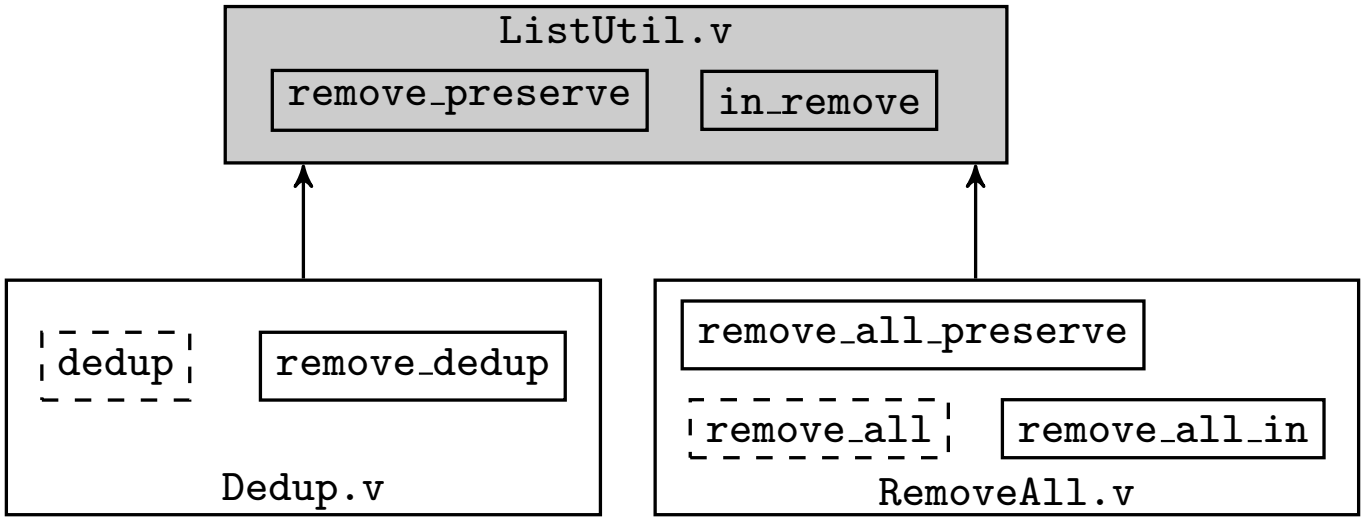
f·none Mode in Practice



f·none Mode in Practice

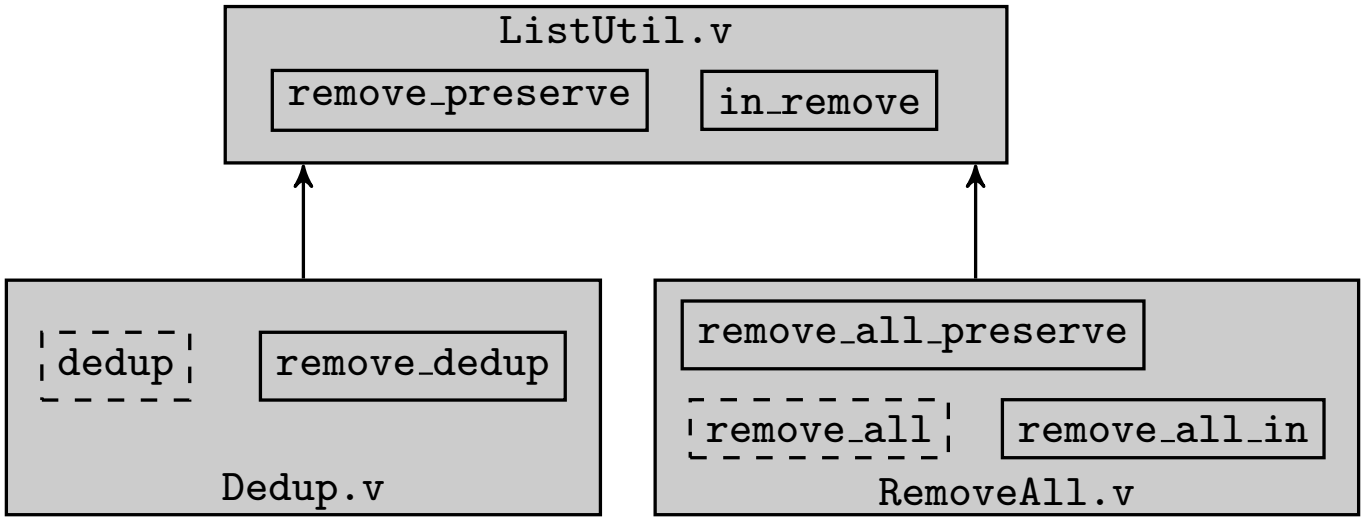


f·none Mode in Practice



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

f·none Mode in Practice



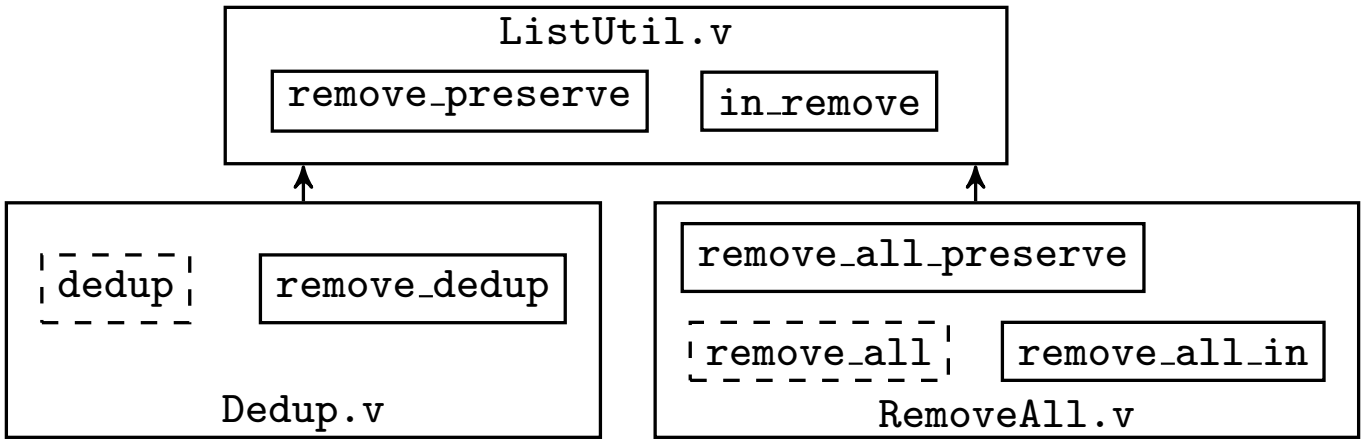
Phase	Task	Definitions and Lemmas
1	ListUtil.vo	remove_preserve, in_remove
2	Dedup.vo	dedup, remove_dedup
2	RemoveAll.vo	remove_all, remove_all_in, remove_all_preserve

p·none Mode: Proof-Level Parallelization, No Selection

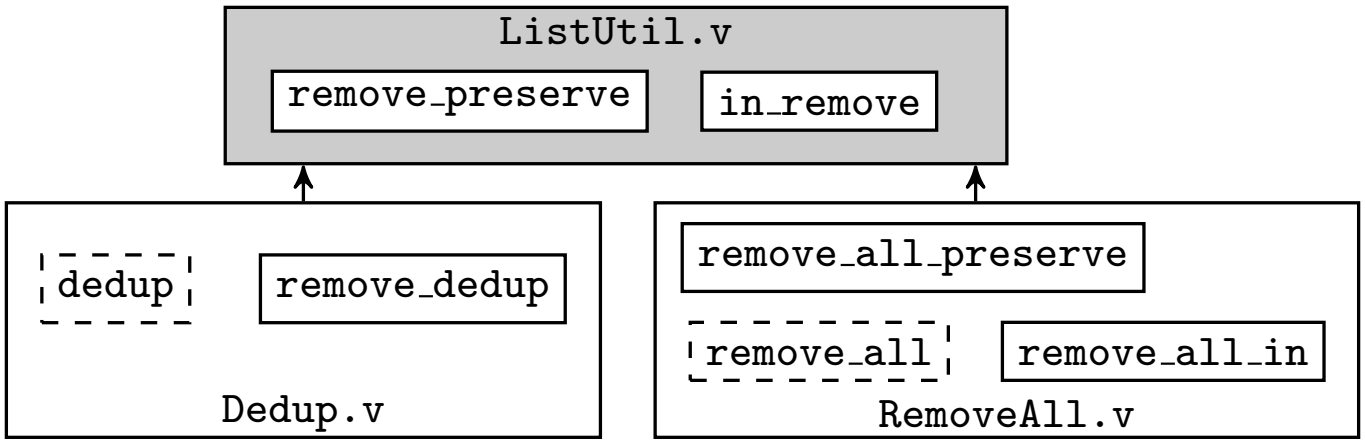
Parallelization	**Selection**		
Granularity	*None*	*Files*	*Proofs*
File level	`f·none`	`f·file`	N/A
Proof level	`p·none`	`p·file`	`p·icoq`

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

p•none Mode in Practice

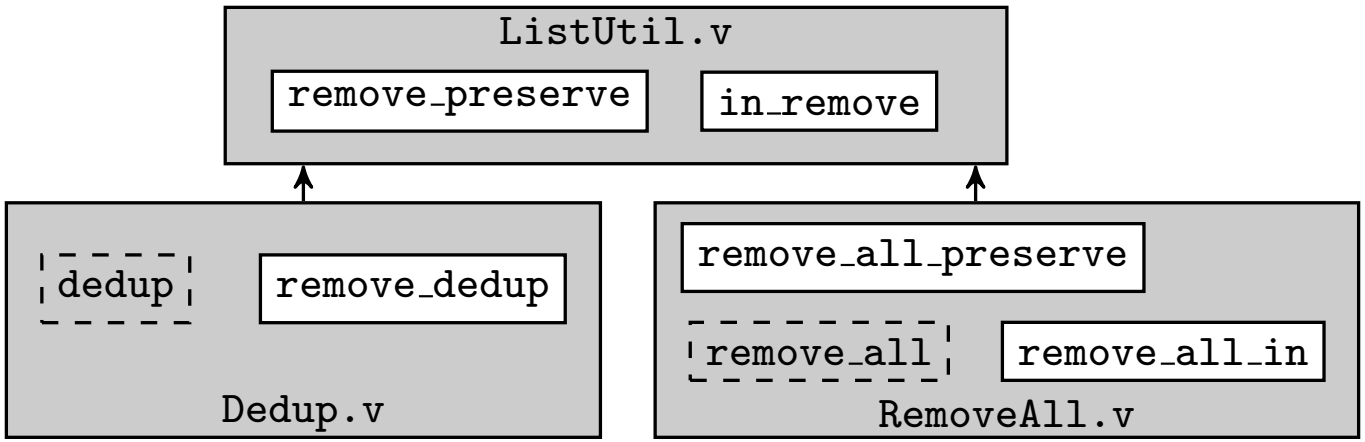


p·none Mode in Practice



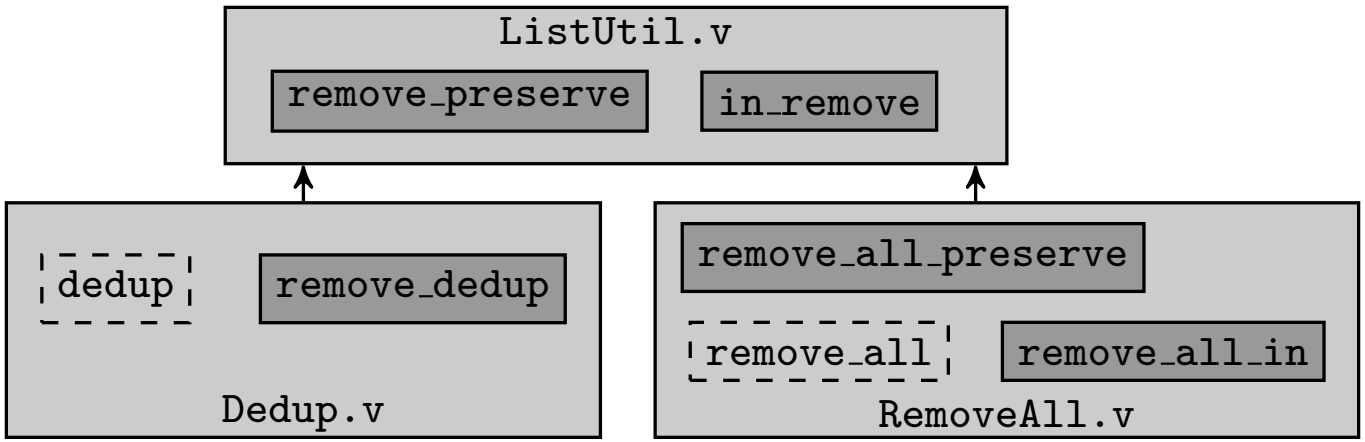
Phase	Task	Definitions and Lemmas
1	ListUtil.vio	~~remove_preserve~~, ~~in_remove~~

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



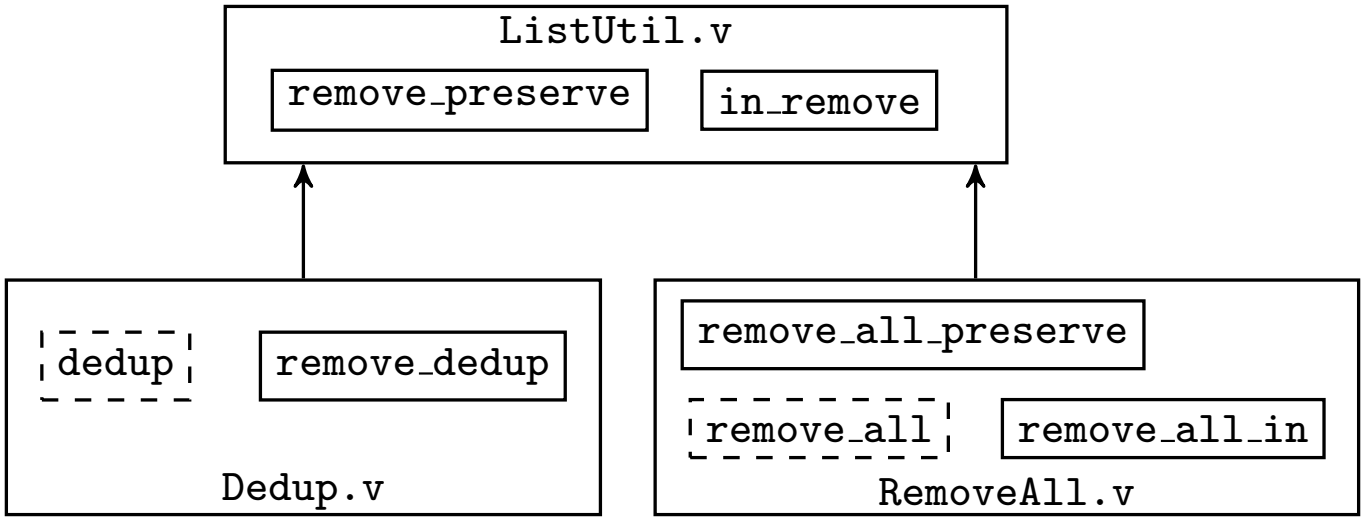
Phase	Task	Definitions and Lemmas
1	ListUtil.vio	~~remove_preserve~~, ~~in_remove~~
2	Dedup.vio	dedup, ~~remove_dedup~~
2	RemoveAll.vio	remove_all, ~~remove_all_in~~, ~~remove_all_preserve~~
3	checking	remove_preserve
3	checking	in_remove
3	checking	remove_dedup
3	checking	remove_all_in
3	checking	remove_all_preserve

`f·file` Mode: File-Level Parallelization, File Selection

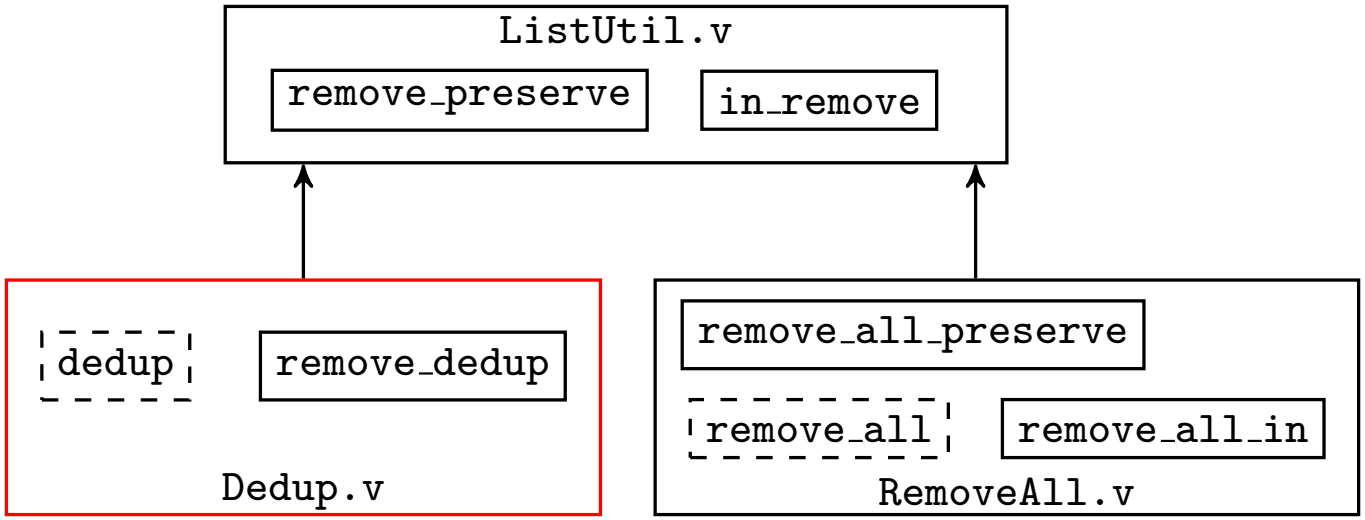
Parallelization	**Selection**		
Granularity	*None*	*Files*	*Proofs*
File level	`f·none`	`f·file`	N/A
Proof level	`p·none`	`p·file`	`p·icoq`

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

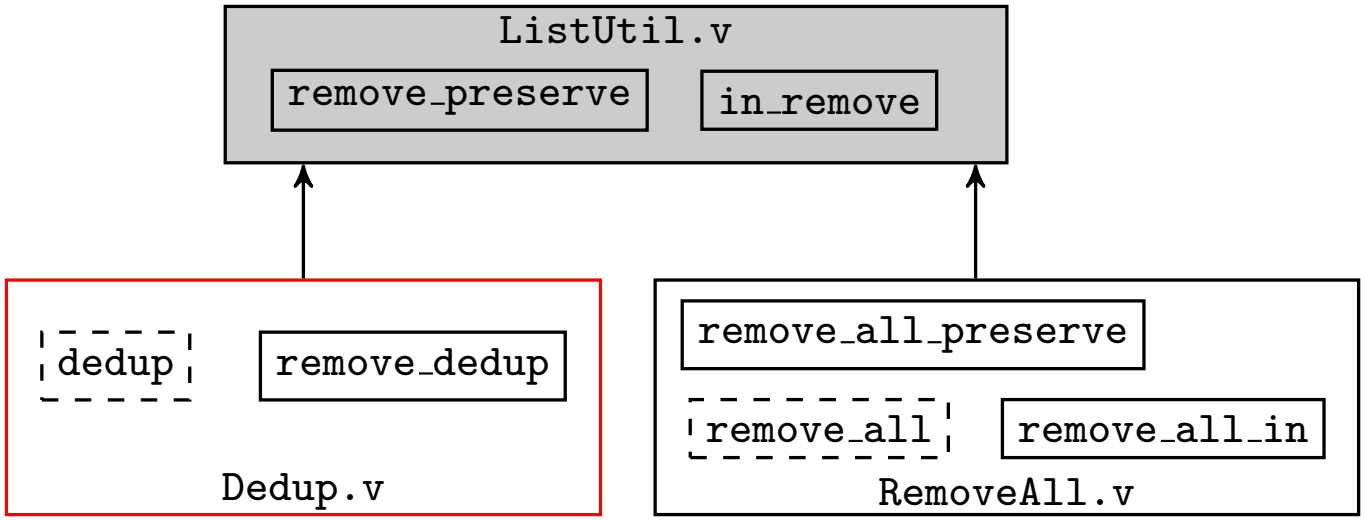
f·file Mode in Practice



f·file Mode in Practice

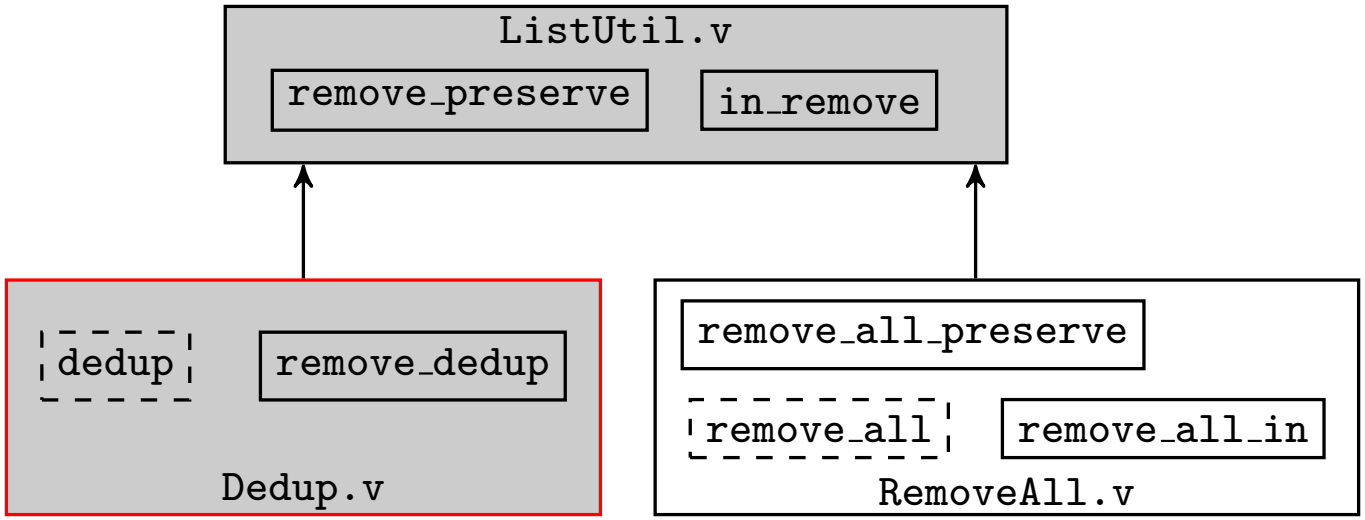


f·file Mode in Practice



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

f·file Mode in Practice



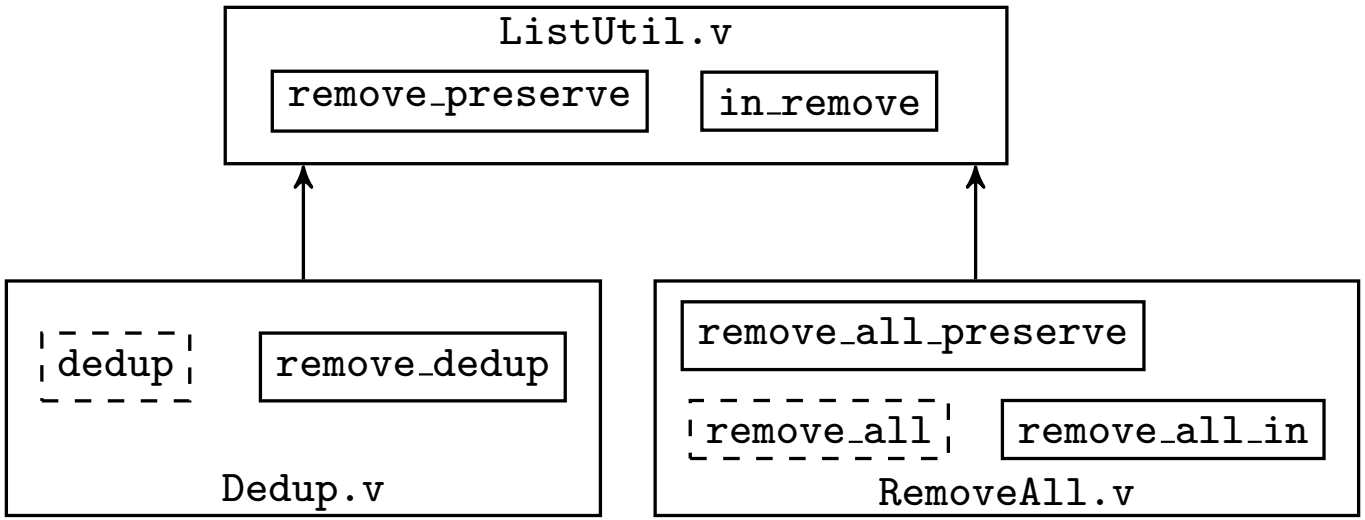
Phase	Task	Definitions and Lemmas
1	ListUtil.vo	remove_preserve, in_remove
2	Dedup.vo	dedup, remove_dedup

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

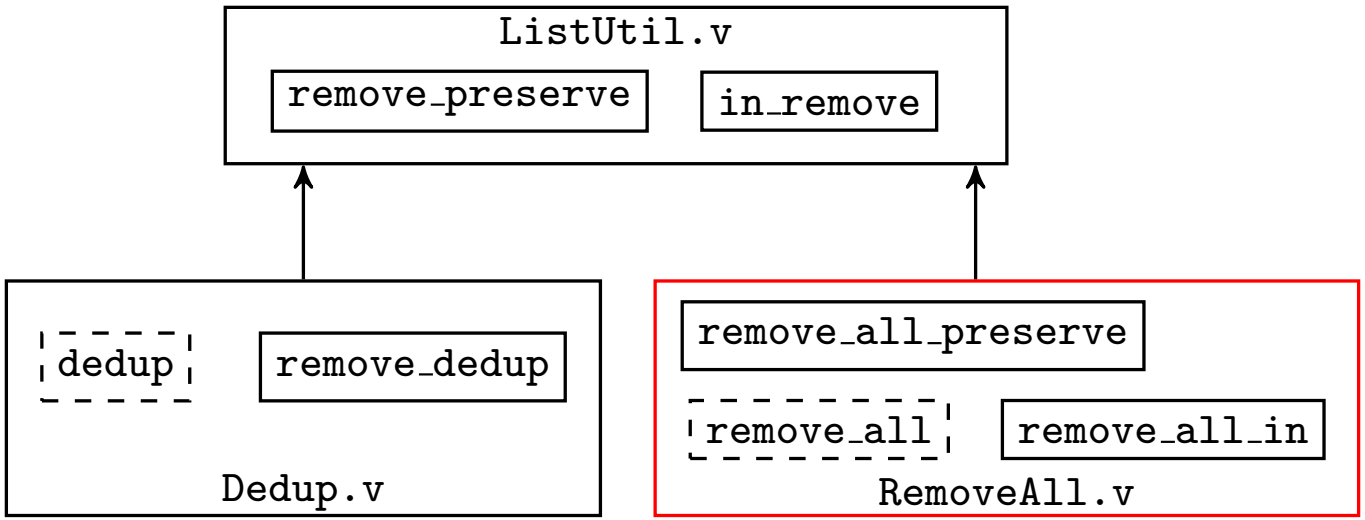
Parallelization	**Selection**		
Granularity	*None*	*Files*	*Proofs*
File level	`f·none`	`f·file`	N/A
Proof level	`p·none`	`p·file`	`p·icoq`

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

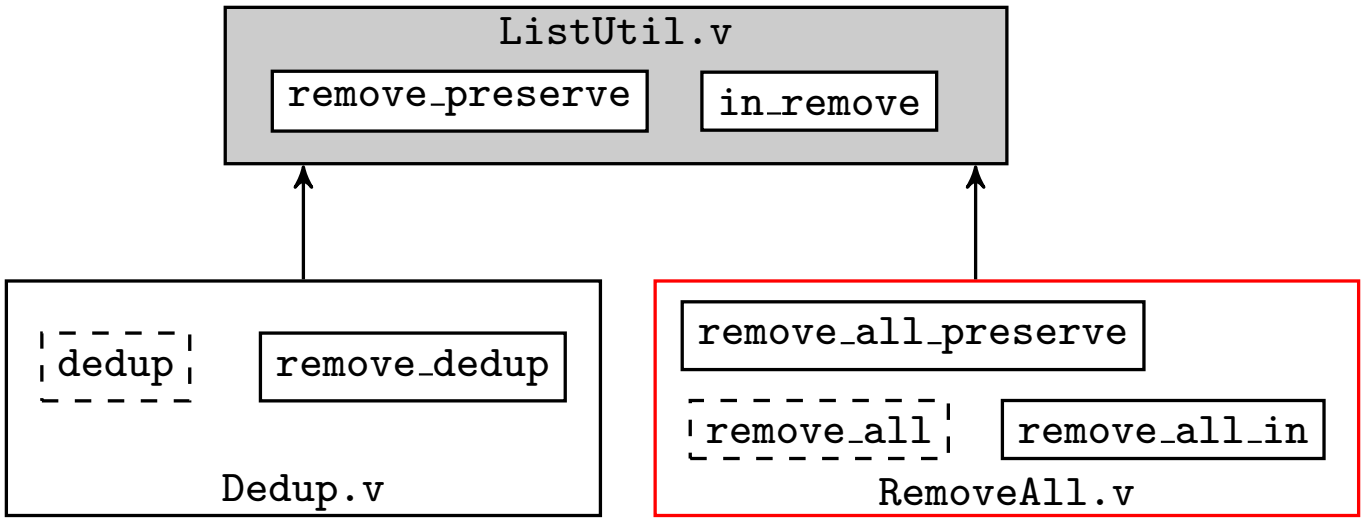
p·file Mode in Practice



p·file Mode in Practice

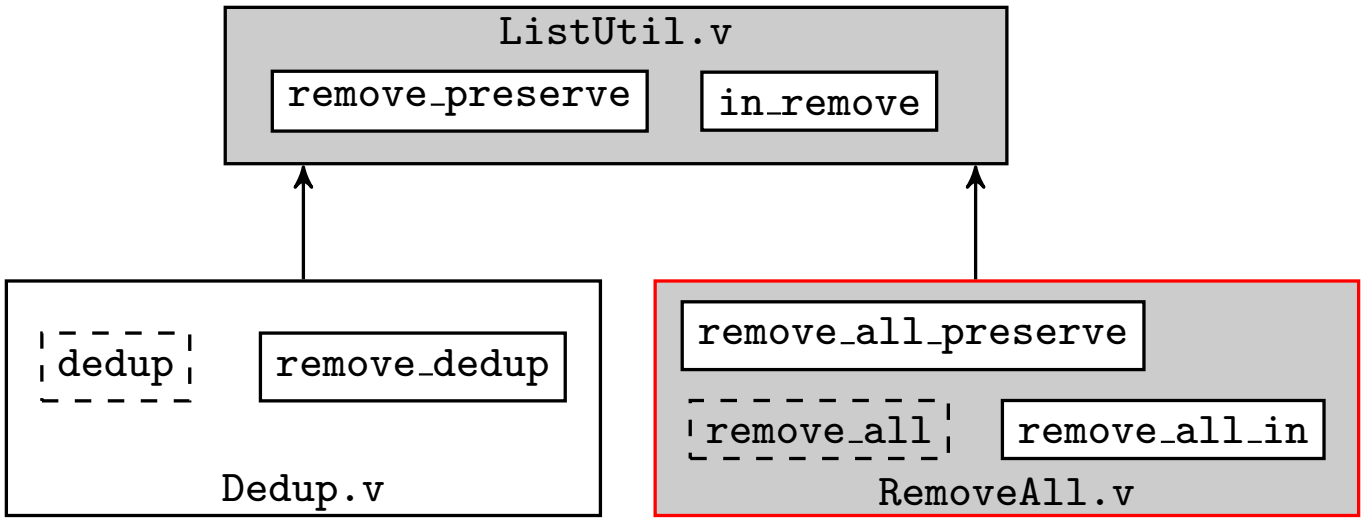


p·file Mode in Practice



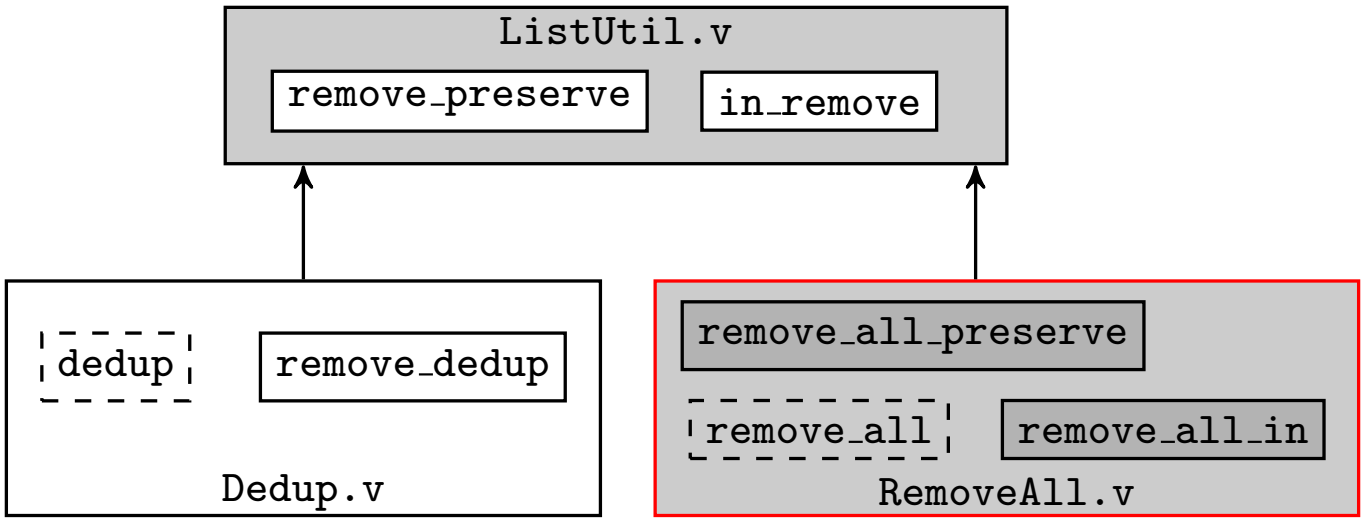
Phase	Task	Definitions and Lemmas
1	ListUtil.vio	~~remove_preserve~~, ~~in_remove~~

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



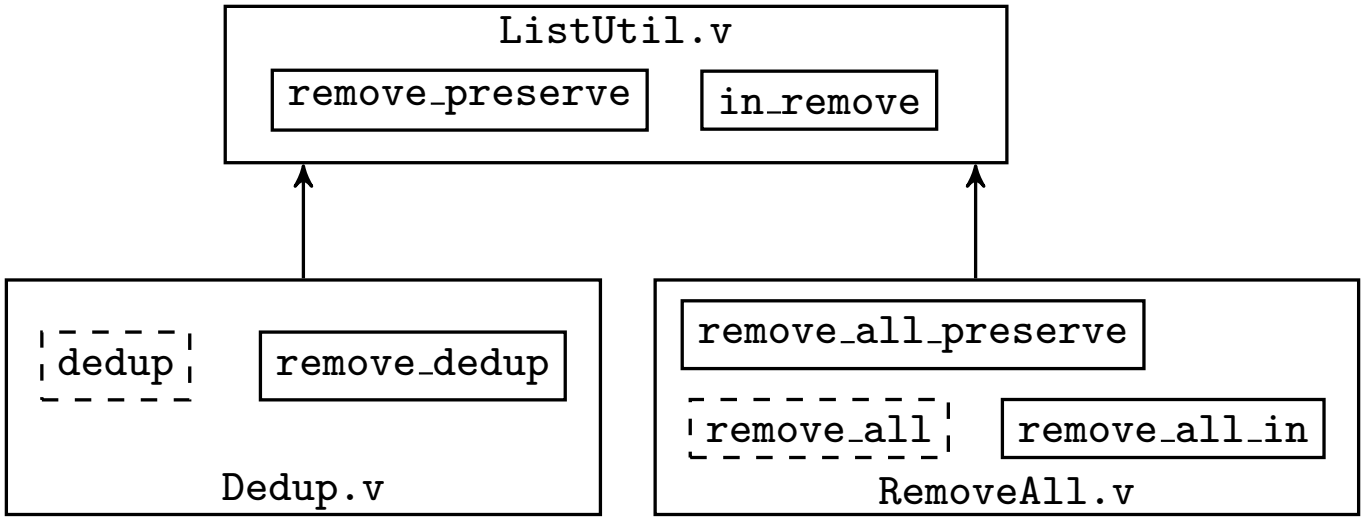
Phase	Task	Definitions and Lemmas
1	ListUtil.vio	~~remove_preserve~~, ~~in_remove~~
2	RemoveAll.vio	remove_all, ~~remove_all_in~~, ~~remove_all_preserve~~
3	checking	remove_all_in
3	checking	remove_all_preserve

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

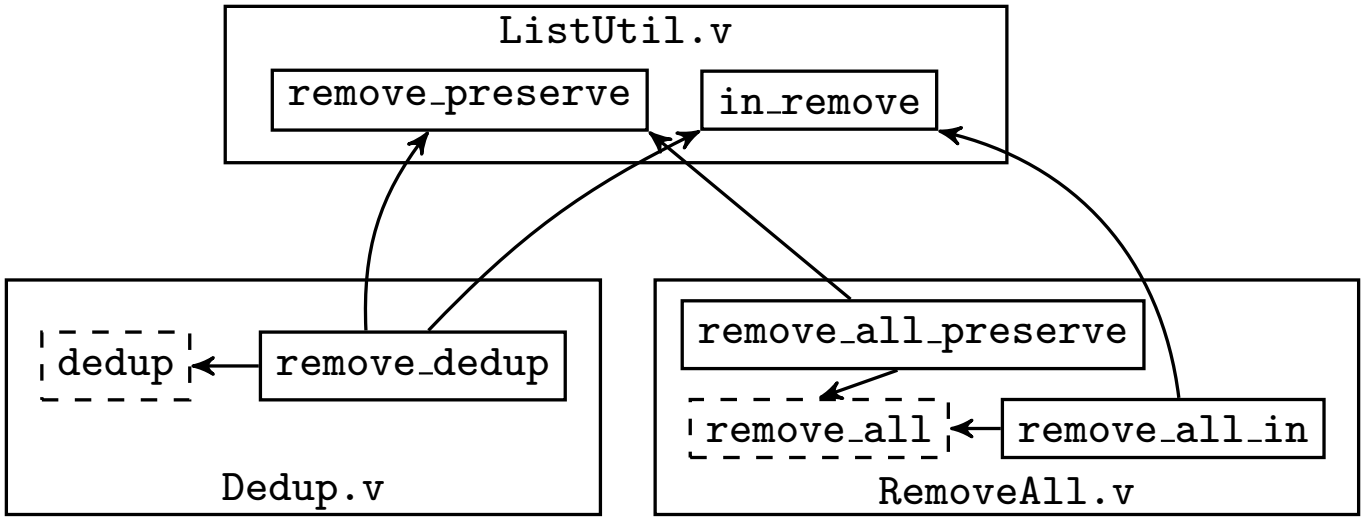
Parallelization	**Selection**		
Granularity	*None*	*Files*	*Proofs*
File level	`f·none`	`f·file`	N/A
Proof level	`p·none`	`p·file`	`p·icoq`

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

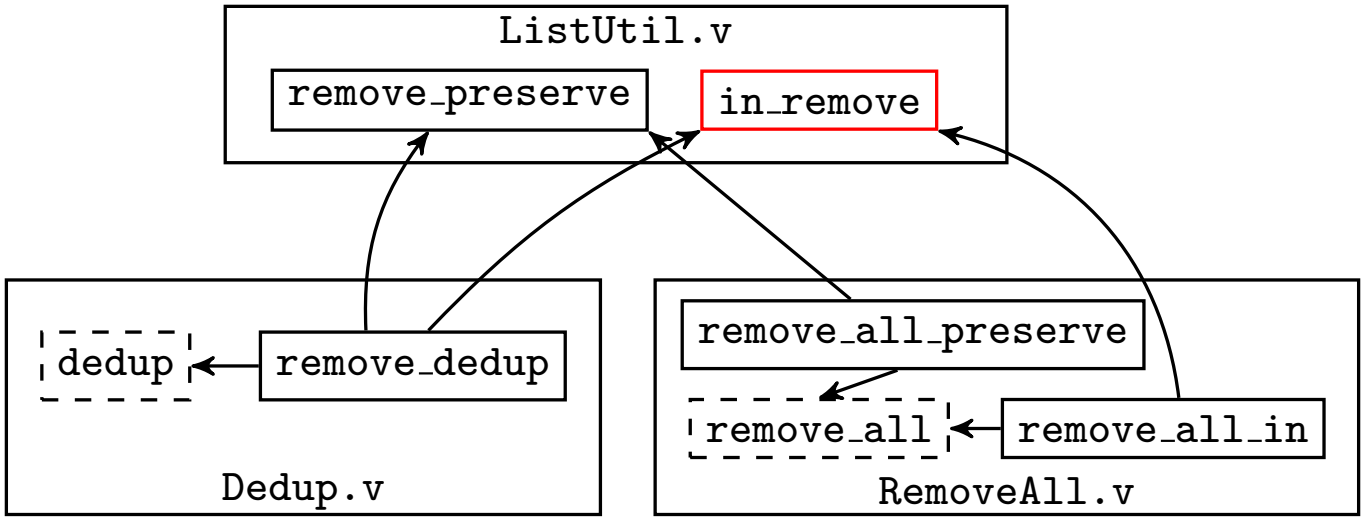
p·icoq Mode in Practice



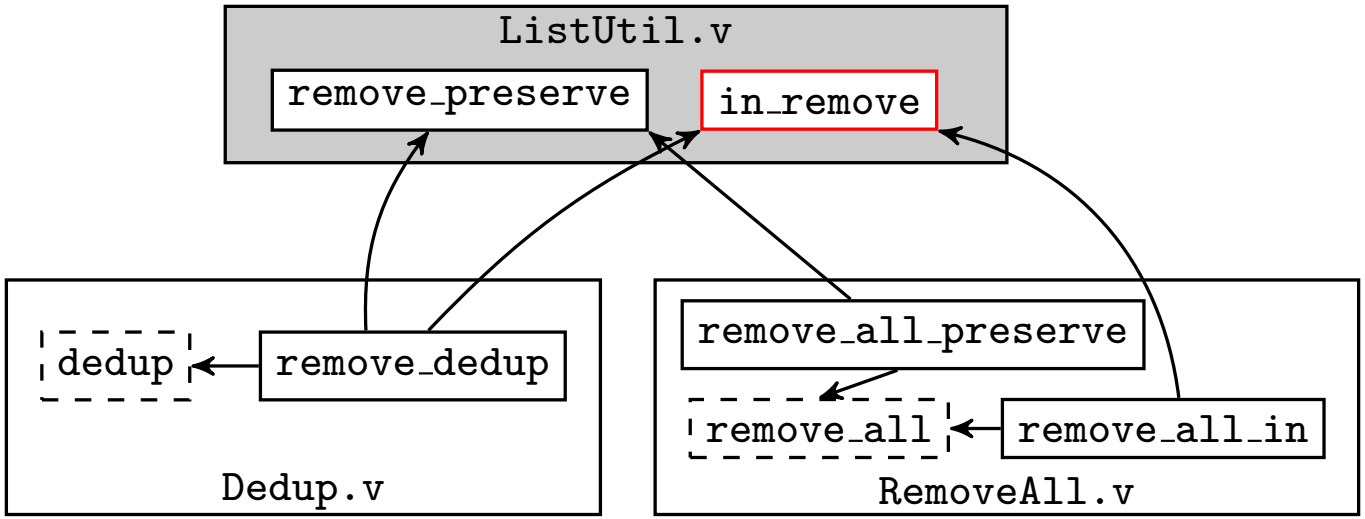
p·icoq Mode in Practice



p·icoq Mode in Practice

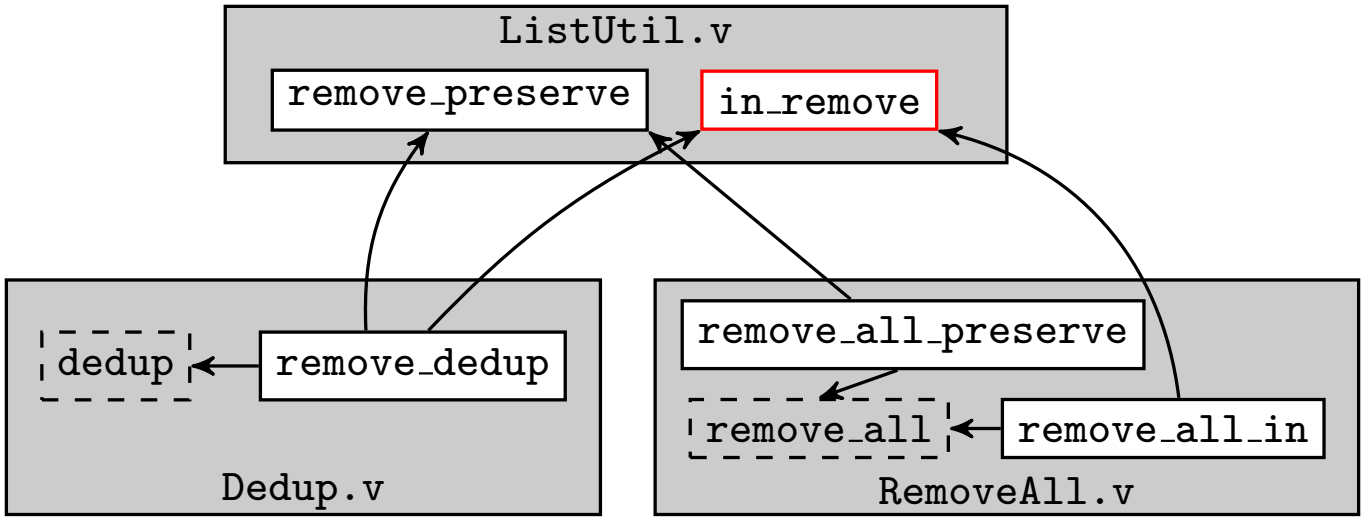


p·icoq Mode in Practice



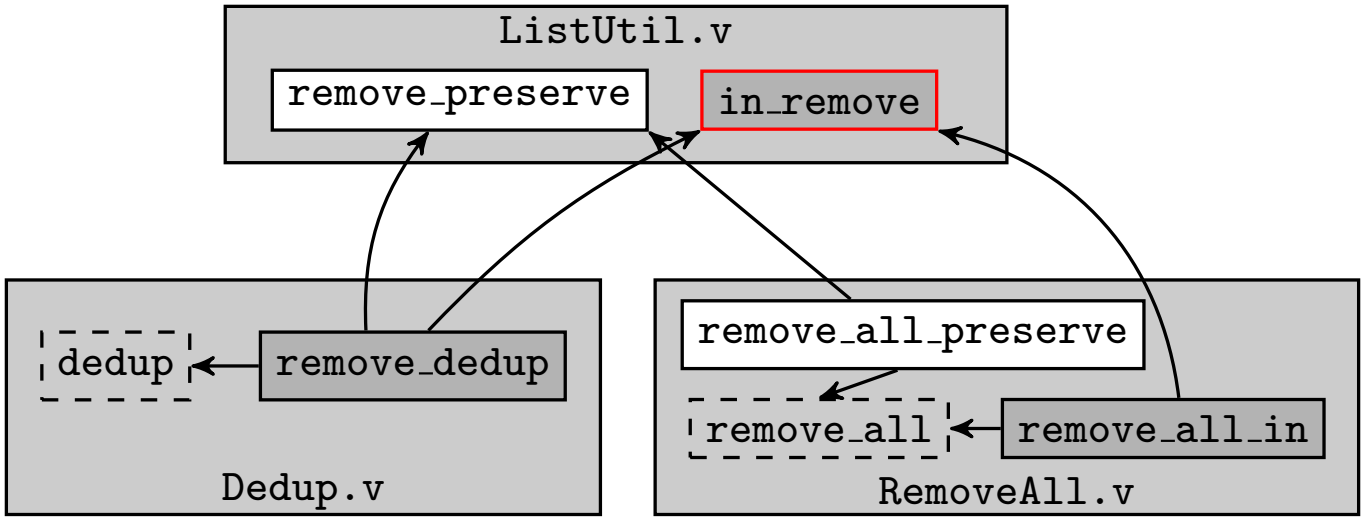
Phase	Task	Definitions and Lemmas
1	ListUtil.vio	~~remove_preserve~~, ~~in_remove~~

p·icoq Mode in Practice



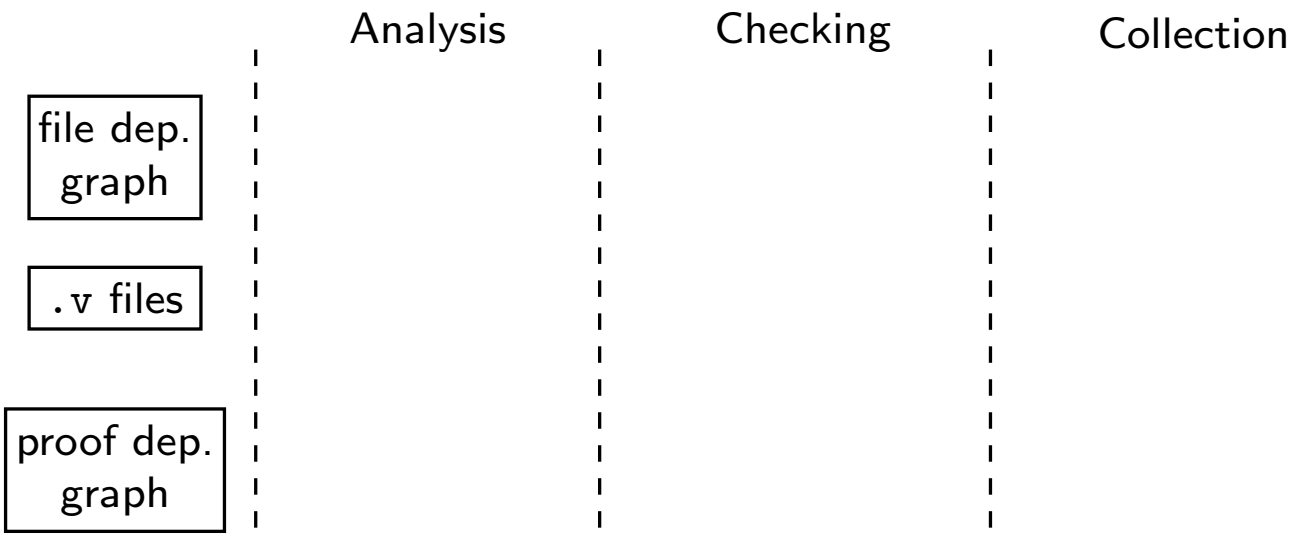
Phase	Task	Definitions and Lemmas
1	ListUtil.vio	~~remove_preserve~~, ~~in_remove~~
2	Dedup.vio	dedup, ~~remove_dedup~~
2	RemoveAll.vio	remove_all, ~~remove_all_in~~, ~~remove_all_preserve~~

p·icoq Mode in Practice

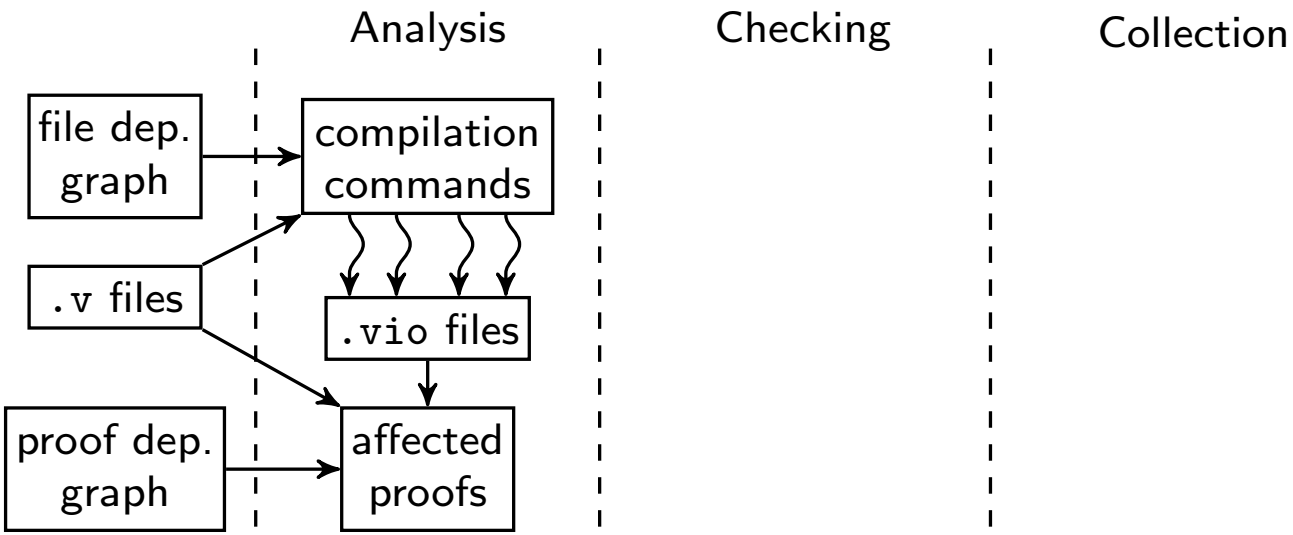


Phase	Task	Definitions and Lemmas
1	ListUtil.vio	~~remove_preserve~~, ~~in_remove~~
2	Dedup.vio	dedup, ~~remove_dedup~~
2	RemoveAll.vio	remove_all, ~~remove_all_in~~, ~~remove_all_preserve~~
3	checking	in_remove
3	checking	remove_dedup
3	checking	remove_all_in

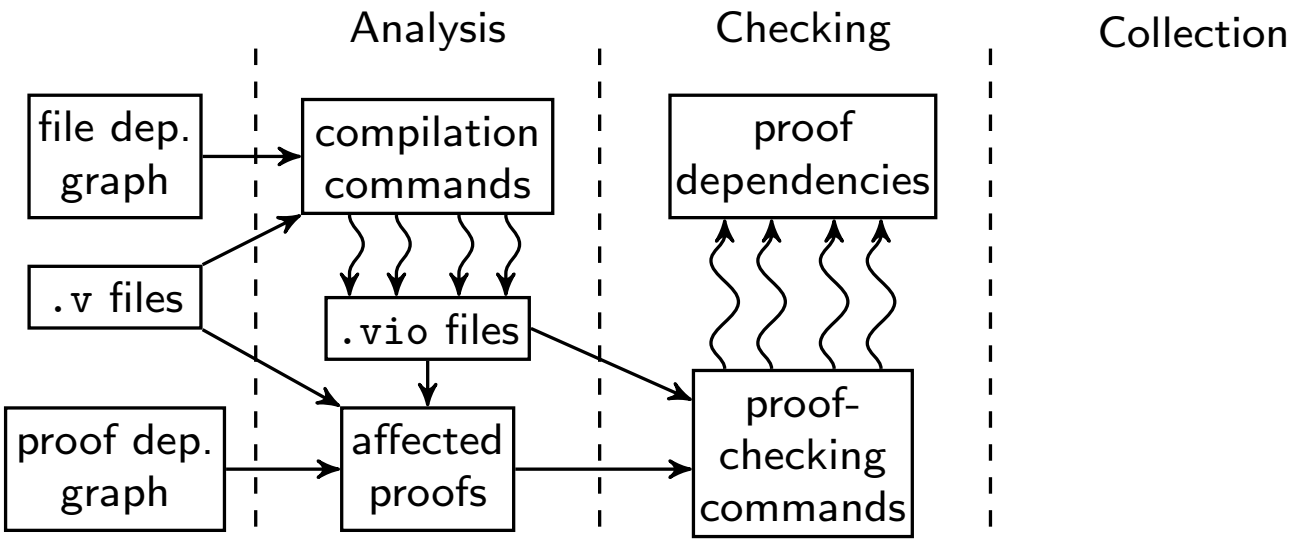
p·icoq Workflow with 4-way Parallelization



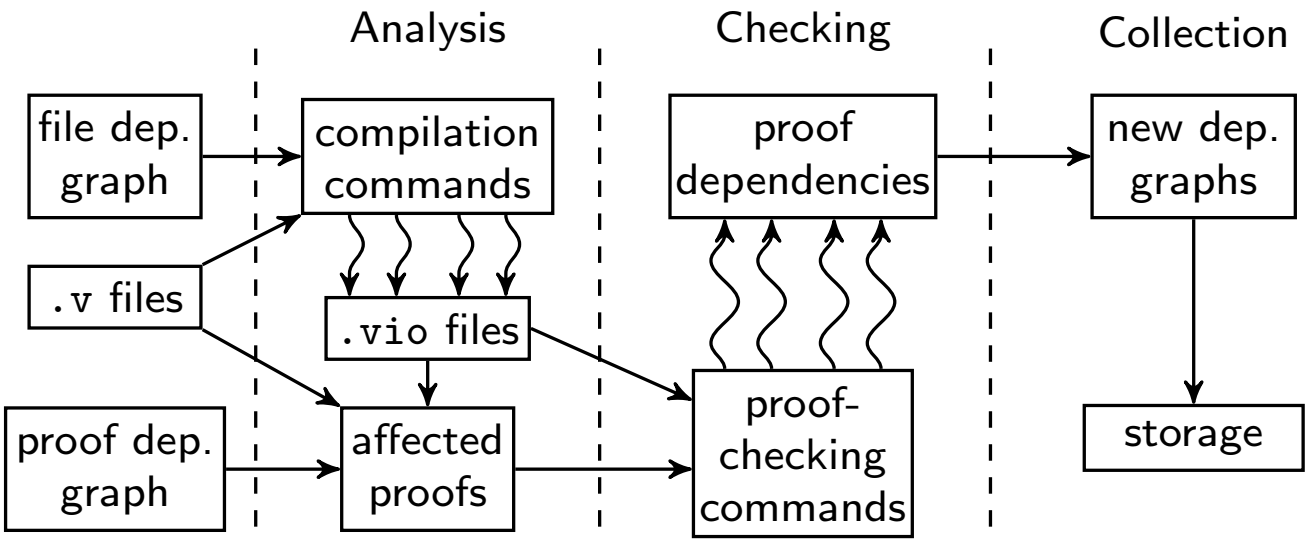
p·icoq Workflow with 4-way Parallelization



p·icoq Workflow with 4-way Parallelization



p·icoq Workflow with 4-way Parallelization



Projects Used for Non-Parallel Evaluation of iCoq

Project	LOC	SHA	URL
Flocq	24786	`4161c990`	gitlab.inria.fr/flocq/flocq
UniMath	43049	`5e525f08`	github.com/UniMath/UniMath
Verdi	53939	`15be6f61`	github.com/uwplse/Verdi

Reduction in #proofs to check using iCoq

Project		**Proofs** iCoq	**Proofs** 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

Reduction proof checking time from scratch

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

Reduction proof checking time locally

Project		LO-Env Time [s] coq_makefile	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

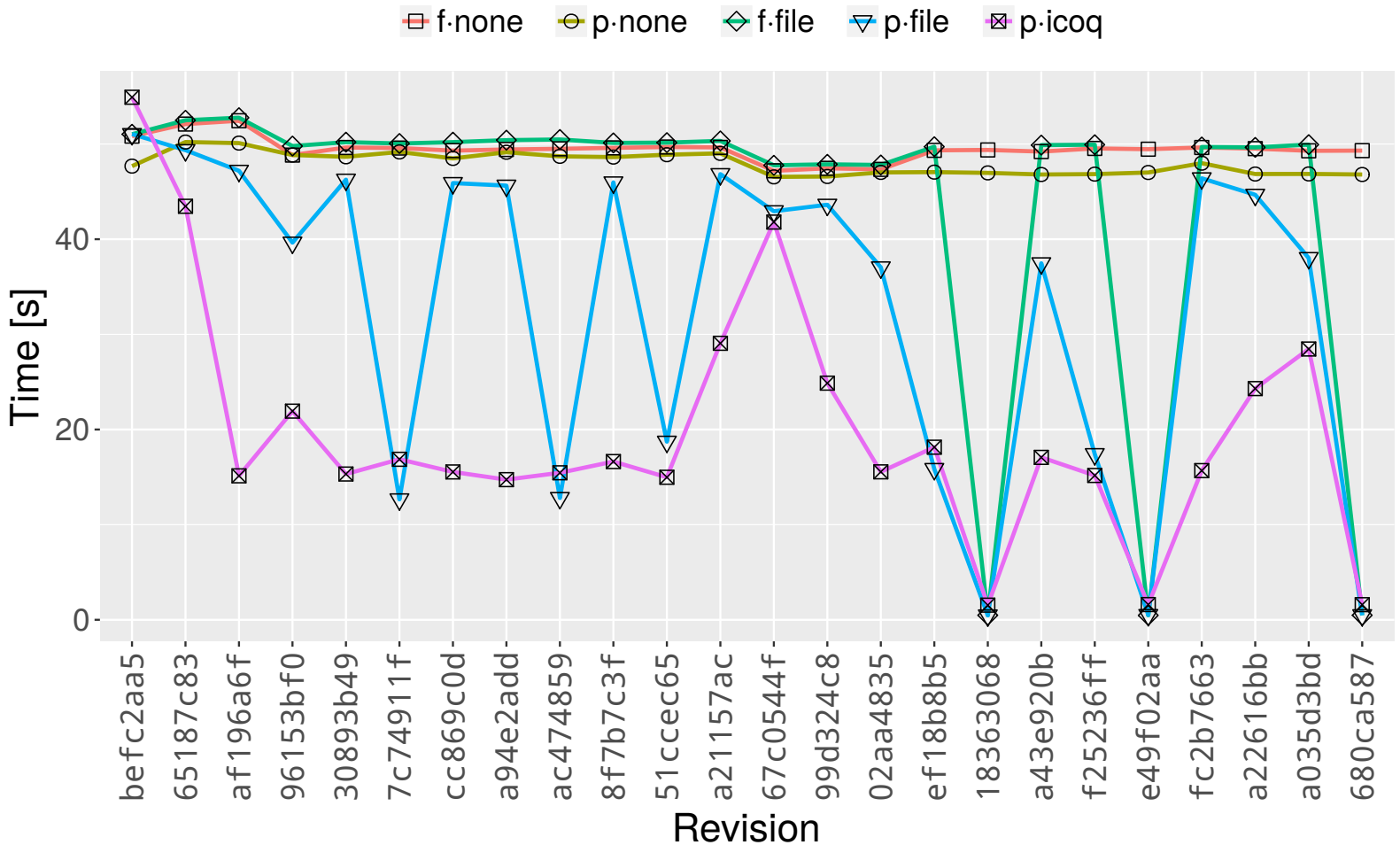
Projects Used for Evaluation of All Modes

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	

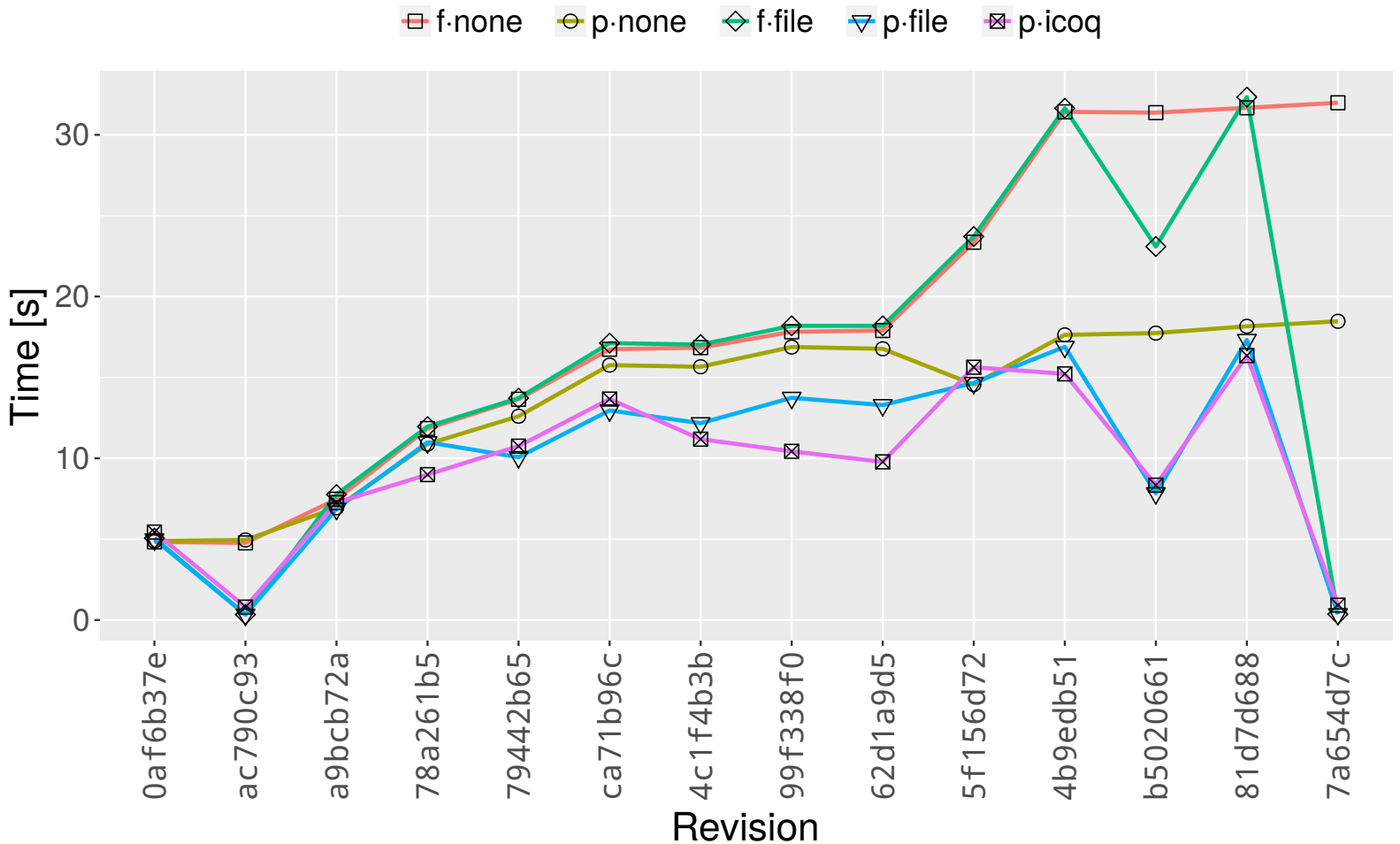
Projects Used for Evaluation of All Modes

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

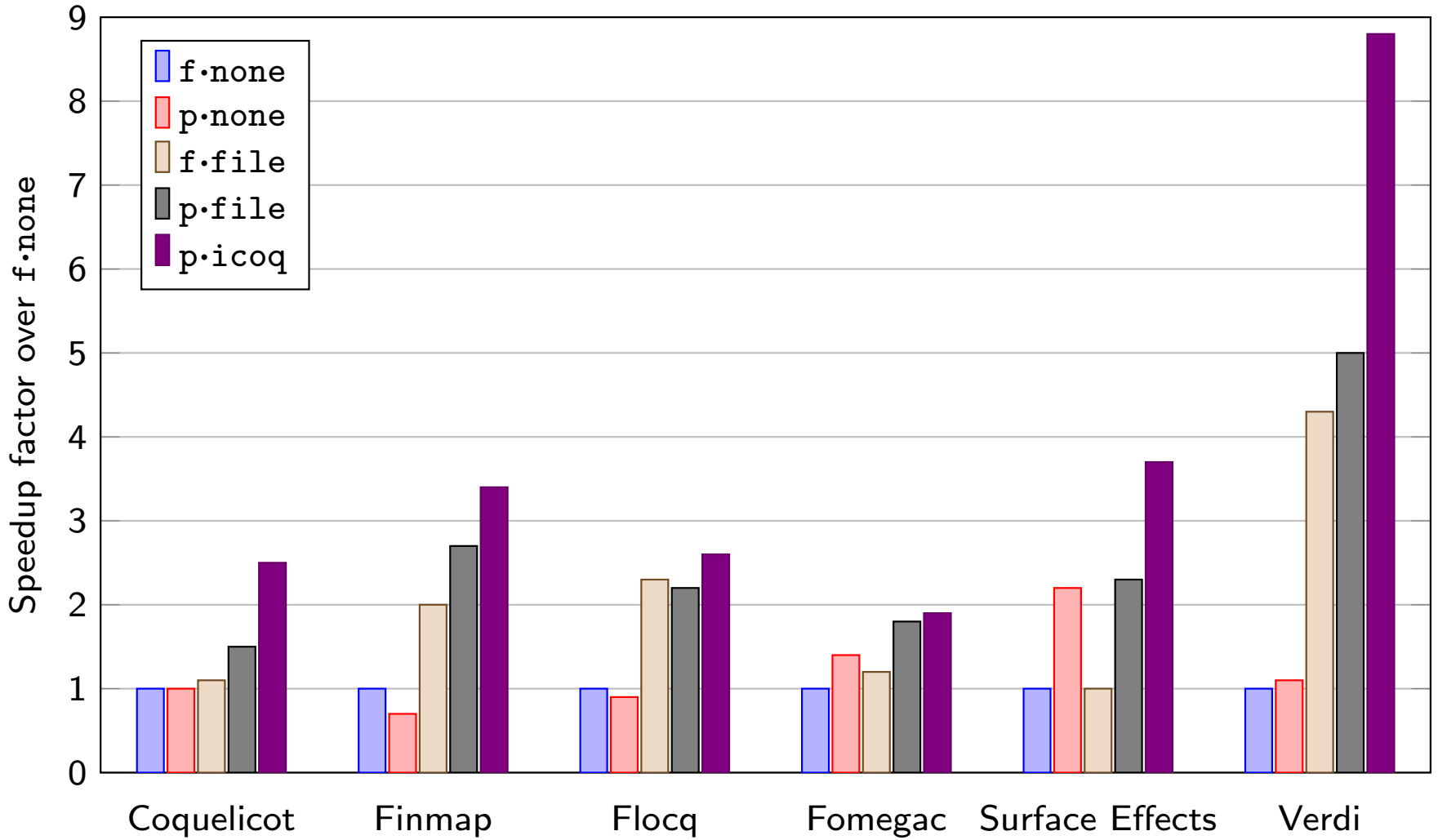
Results with 4-way Parallelization: Coquelicot



Results with 4-way Parallelization: Fomegac

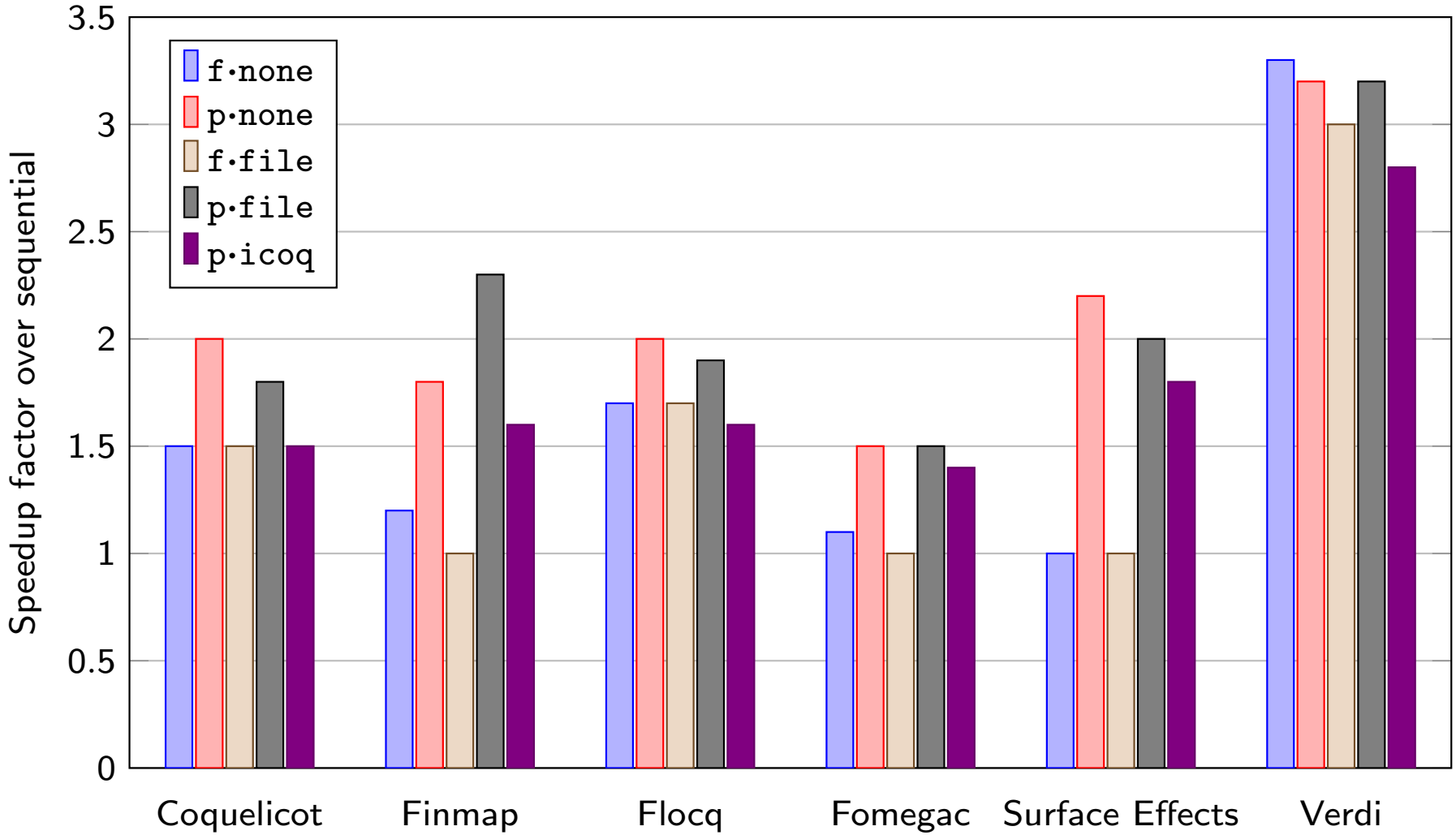


Speedups over `f·none` for 4-way Parallel Checking



"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"

File and proof selection, formally

1. Model of change impact analysis, including hierarchical
2. Library of definitions and proofs in Coq and MathComp
3. Verified practical tool, Chip
4. Evaluation of Chip integrated with build & testing tools

`https://github.com/palmskog/chip`

Change Impact Analysis Model

concept	**notation**	**constraint**	**intuition**
components	V, V'	finite sets s.t. $V \subseteq V'$	"file names"
artifacts	A	finite set	"file contents"
revisions	f, f'	$f : V \to A$, $f' : V' \to A$	"before&after"
dependencies	g, g'	$g : rel\, V$, $g' : rel\, V'$	"before&after"
checkable	E	finite set s.t. $E \subseteq V'$	"test file name"
results	$check$, R	$check(v) \in R$ if $v \in E$	"test runner"

We will generally call $v \in V'$ a vertex and g, g' graphs.

Change Impact Analysis Model, Continued

Modified Vertices

Whenever $f(v) \neq f'(v)$ for some $v \in V$, the artifact for v is **modified** after the revision.

Change Impact Analysis Model, Continued

Modified Vertices

Whenever $f(v) \neq f'(v)$ for some $v \in V$, the artifact for v is **modified** after the revision.

Impacted Vertices

A vertex $v \in V$ is **impacted** if it is reachable from some modified vertex in **inverse** graph g^{-1}.

Change Impact Analysis Model, Continued

Modified Vertices

Whenever $f(v) \neq f'(v)$ for some $v \in V$, the artifact for v is **modified** after the revision.

Impacted Vertices

A vertex $v \in V$ is **impacted** if it is reachable from some modified vertex in **inverse** graph g^{-1}.

Fresh Vertices

A vertex $v \in V'$ is **fresh** whenever $v \notin V$.

Change Impact Analysis Model, Continued

Modified Vertices

Whenever $f(v) \neq f'(v)$ for some $v \in V$, the artifact for v is **modified** after the revision.

Impacted Vertices

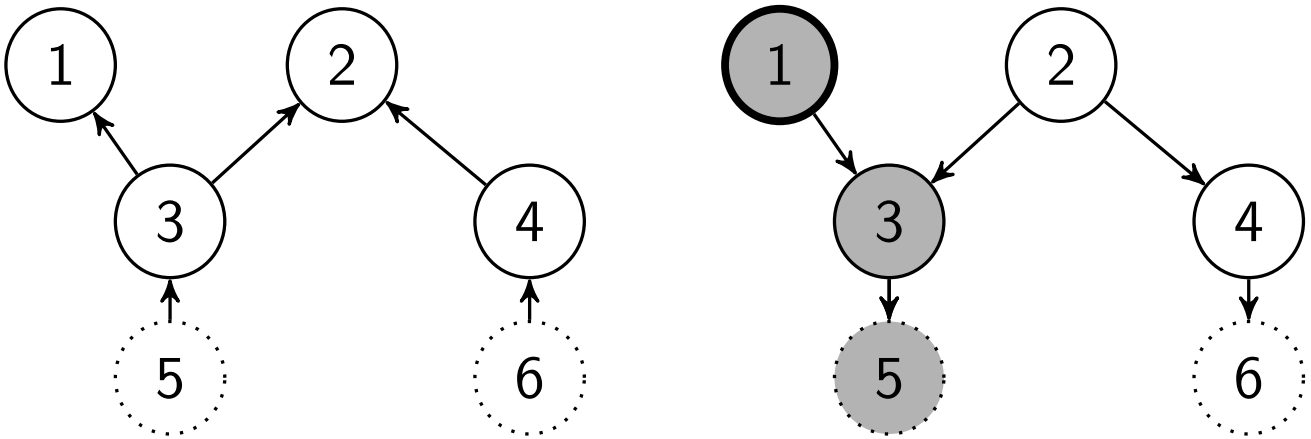
A vertex $v \in V$ is **impacted** if it is reachable from some modified vertex in **inverse** graph g^{-1}.

Fresh Vertices

A vertex $v \in V'$ is **fresh** whenever $v \notin V$.

Key idea: determine all impacted and fresh vertices, then run *check* on executable vertices among those.

Change Impact Analysis Example



- $V = \{1, 2, 3, 4, 5, 6\}$, $V' = V$, $E = \{5, 6\}$
- $f(1) \neq f'(1)$, hence 1 is **modified**
- refl-transitive closure in g^{-1} of 1 is $\{$ 1, 3, 5 $\}$, all **impacted**
- conclusion: only need to run *check*(5) after change

Change Impact Analysis Correctness and Assumptions

Correctness: executing only impacted and fresh vertices that are checkable is **sound** and **complete** for the new revision.

Change Impact Analysis Correctness and Assumptions

Correctness: executing only impacted and fresh vertices that are checkable is **sound** and **complete** for the new revision.

A1: The direct dependencies of a vertex v are the same in both revisions if the artifact of v is the same in both revisions, i.e., if $f(v) = f'(v)$.

Change Impact Analysis Correctness and Assumptions

Correctness: executing only impacted and fresh vertices that are checkable is **sound** and **complete** for the new revision.

A1: The direct dependencies of a vertex v are the same in both revisions if the artifact of v is the same in both revisions, i.e., if $f(v) = f'(v)$.

A2: A vertex v with the same artifact in both revisions is checkable in the new revision if and only if v is checkable in the old revision.

Change Impact Analysis Correctness and Assumptions

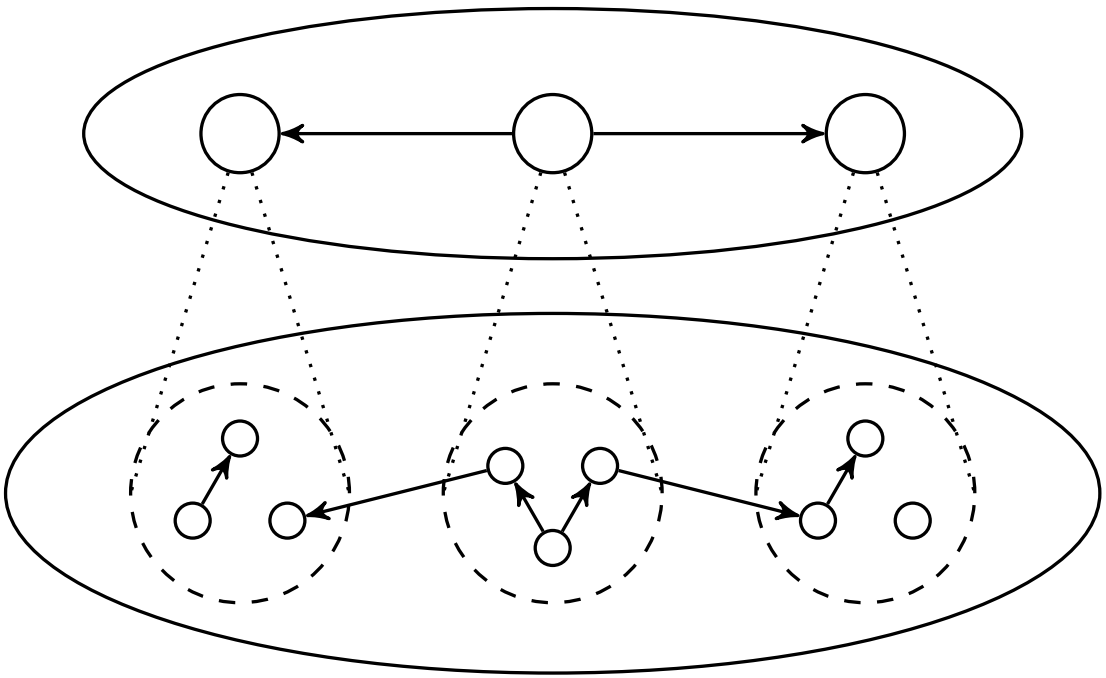
Correctness: executing only impacted and fresh vertices that are checkable is **sound** and **complete** for the new revision.

A1: The direct dependencies of a vertex v are the same in both revisions if the artifact of v is the same in both revisions, i.e., if $f(v) = f'(v)$.

A2: A vertex v with the same artifact in both revisions is checkable in the new revision if and only if v is checkable in the old revision.

A3: The outcome of executing a checkable vertex v is the same in both revisions if the sets of vertices v depends on transitively are the same, and the artifact of each dependency is the same.

Hierarchical Change Impact Analysis



“files”

“proofs”

Hierarchical Change Impact Analysis Model

- U is a set of **coarse-grained** components
- V is a set of **fine-grained** components
- $p : U \to 2^V$ is a **partition** of V
- $g_\top$ is dependency graph for U
- $g_\bot$ is dependency graph for V
- Use change impact analysis of U and $g_\top$ to analyze V and $g_\bot$

Strategies for Hierarchical Change Impact Analysis

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 executable vertices in V'_p

Strategies for Hierarchical Change Impact Analysis

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 executable 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

Strategies for Hierarchical Change Impact Analysis

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 executable 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

Correctness similar to as for basic model!

Hierarchical Change Impact Analysis Assumptions

H1: For all $u, u' \in U$ and $v, v' \in V$, if $u \neq u'$, $g_{\perp}(v, v')$, $v \in p(u)$, and $v' \in p(u')$, then $g_{\top}(u, u')$.

Hierarchical Change Impact Analysis Assumptions

H1: For all $u, u' \in U$ and $v, v' \in V$, if $u \neq u'$, $g_{\perp}(v, v')$, $v \in p(u)$, and $v' \in p(u')$, then $g_{\top}(u, u')$.

H2: For all $u \in U$, if $f_{\top}(u) = f'_{\top}(u)$, then $p(u) = p'(u)$.

Hierarchical Change Impact Analysis Assumptions

H1: For all $u, u' \in U$ and $v, v' \in V$, if $u \neq u'$, $g_{\perp}(v, v')$, $v \in p(u)$, and $v' \in p(u')$, then $g_{\top}(u, u')$.

H2: For all $u \in U$, if $f_{\top}(u) = f'_{\top}(u)$, then $p(u) = p'(u)$.

H3: For all $u \in U$ and $v \in V$, if $f_{\top}(u) = f'_{\top}(u)$ and $v \in p(u)$, then $f_{\perp}(v) = f'_{\perp}(v)$.

Encoding in Coq

Around 2000 lines of specifications, 5000 lines of proofs. Uses finite sets & graphs from the Mathematical Components library.

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

Definition impacted_V' m : {set V'} :=
 [set (val v) |  v in impacted g^-1 m].

Definition fresh_V' : {set V'} := [set v | ~P v].

Definition mod_V : {set V} := [set v | f v != f' (val v)].

Definition impacted_fresh_V' : {set V'} :=
 impacted_V' mod_V :|: fresh_V'.
```

Topological Sorting and Acyclicity

- acyclicity only needed for topological sorting, not change impact analysis
- we extended MathComp graph algorithms originally formalized by Cohen & Théry

```
Definition pdfs (g : V → seq V) p x :=
 if x \notin p.1 then p else
 let p' := foldl (pdfs g) (remove x p.1, p.2) (g x) in
 (p'.1, x :: p'.2).

Definition tseq g := (foldl (pdfs g) (V, [::]) V).2.

Theorem tseq_acyclic_before : ∀ g x y,
 acyclic g → connect g x y → before (tseq g) x y.
```

Evaluation

1. extracted refined executable code to OCaml tool called Chip
2. integrated Chip with:
   - iCoq regression proving tool we developed
   - a Java-based regression testing tool
   - a build tool
3. ran modified tools on open source projects
4. compared the outcomes and running times with those for unmodified tool

Projects Used in the iCoq+Chip Evaluation

Project	LOC	#Proofs	SHA	URL
Flocq	33,544	943	`4161c990`	gitlab.inria.fr/flocq/flocq
StructTact	2,497	187	`8f1bc10a`	github.com/uwplse/StructTact
UniMath	45,638	755	`5e525f08`	github.com/UniMath/UniMath
Verdi	57,181	2,784	`15be6f61`	github.com/uwplse/Verdi

Results for iCoq+Chip

Execution/CIA Time in Seconds

Project		**Total**		**CIA**	
	RecheckAll	**iCoq**	Chip	**iCoq**	Chip
Flocq	1,028.01	313.08	318.19	50.65	53.43
StructTact	45.86	43.90	44.49	14.45	14.98
UniMath	14,989.09	1,910.56	2,026.75	124.79	239.12
Verdi	37,792.07	3,604.23	4,637.27	139.09	1,171.57
Total	53,855.03	5,871.76	7,026.70	328.98	1,479.10

Coq-community Templates

coq community

With our Coq-community templates, you can

- leverage parallellism and (local) file selection in Coq
- use continuous integration on GitHub
- package your project using opam for reuse

`https://github.com/coq-community/templates`

Monorepo vs. Regular Repository

- a monolithic repository hosts multiple packages
- can simplify maintenance and integrating new contributions
- used for Mathematical Components
- continuous integration more complicated
- templates do not work well for monorepos (yet)

`https://github.com/coq-community/hydra-battles/`

Conclusion

Material based on:

- Ahmet Celik, Karl Palmskog, and Milos Gligoric.
  iCoq: Regression Proof Selection for Large-Scale Verification Projects.
  Proceedings of ASE, 2017.
- Karl Palmskog, Ahmet Celik, and Milos Gligoric.
  piCoq: Parallel Regression Proving for Large-Scale Verification Projects.
  Proceedings of ISSTA, 2018.
- Karl Palmskog, Ahmet Celik, and Milos Gligoric.
  Practical Machine-Checked Formalization of Change Impact Analysis.
  Proceedings of TACAS, 2020.

More and contact:

- Website with papers: `https://setoid.com`
- Email: `palmskog@kth.se`
- GitHub: `https://github.com/palmskog`
- The Coq Zulip Chat: `https://coq.zulipchat.com`
- `https://github.com/coq-community/awesome-coq`

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 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.
```

Coq-community on GitHub

`https://github.com/coq-community`

- repositories for long-term maintenance
- best practices and templates (for CIS, READMEs, ...)

Discussion

- component deletion not captured by model (complicates reasoning)
- application: model-based testing of RTS and build tools
- application: end-to-end proof of technique using impact analysis in PL
- application: verified build tool implementation

Depth-First Search (DFS) and Reflexive-Transitive Closures

```
Definition foldl f z s :=
 if s is x :: s' then foldl f (f z x) s' else z.

Definition dfs g s x :=
 if x \in s then s else
 foldl (dfs g) (x :: s) (g x).

Definition connect g x y :=
 y \in dfs g [::] x.

Definition closure g s :=
 foldl (dfs g) [::] s.
```

Graph Paths

```
Definition last x s :=
 if s is x' :: s' then last x' s' else x.

Definition path g x p :=
 if p is y :: p' then y \in g x && path y p' else true.
```

Meaning of Predicates

```
connect g x y
```

$\Leftrightarrow$

```
∃ p. path g x p ∧ y = last x p
```

```
y \in closure g s
```

$\Leftrightarrow$

```
∃ x. x \in s ∧ connect g x y
```

Impactedness

```
Definition impacted g modified :=
 closure g^-1 modified.
```

```
x \in impacted g modified
```

$\Leftrightarrow$

```
∃ y. y \in modified ∧ connect g^-1 y x
```

Impactedness

```
Definition impacted g modified :=
 closure g^-1 modified.
```

```
x \in impacted g modified
```

$\Leftrightarrow$

```
∃ y. y \in modified ∧ connect g x y
```

Acyclicity

```
Definition cycle g p :=
 if p is x :: p' then path g x (p' ++ [x]) else true.

Definition acyclic g :=
 ∀ x p, path g x p → ¬ cycle g (x :: p).
```

Topological Sorts of Acyclic Graphs

```
Definition pdfs g p x :=
 if x \notin p.1 then p else
 let p' := foldl (pdfs g) (remove x p.1, p.2) (g x) in
 (p'.1,  x ::  p'.2).

Definition tseq g :=
 (foldl (pdfs g) (V, [::])   V).2.

Theorem tseq_acyclic_before : ∀ g x y,
 acyclic g → connect g x y → before (tseq g) x y.
```