Incremental and Modular Context-Sensitive Analysis

Isabel Garcia-Contreras[1,2] Jose F. Morales[1,2] Manuel V. Hermenegildo[1,2]

[1] IMDEA Software Institute
[2] T. U. Madrid (UPM)

HCVS: Horn Clauses for Verification and Synthesis — March 28th, 2021

Introduction

Context: Analyzing/Verifying software projects during development in order to:

- Detect and report of bugs as early as possible (e.g., on-the-fly, at commit, ...).
- Optimize code and libraries globally for the program being developed.

Problem: Context-sensitive analysis can be quite precise but also expensive, specially for interactive uses.

Make it incremental! – successive changes during development are often comparatively small and localized.

So far, in abstract interpretation, this was achieved by:

- Fine-grain (clause-level) incremental analysis for non-modular programs.

  [SAS'96, TOPLAS'00]
- Coarse-grain (module-level) analysis aimed at reducing memory consumption.

  [ENTCS'00, LOPSTR'01]

We propose an extension of the modular algorithm to react to module changes, and a way to combine it with fine-grain incrementality.

Motivation - (Incremental) Static *On-the-fly* verification

```
        P = B
    ;   rewrite(clause(H,B),clause(H,P),I,G,Info)
    ).

rewrite( clause(H,B), clause(H,P),I,G,Info) :-
    numbervars_2(H,0,Lhv),
    collect_info(B,Info,Lhv,_X,_Y),
    add_annotations(Info,P,I,G),!.

:- pred add_annotations(Info,Phrase,Ind,Gnd)
   : (var(Phrase), indep(Info,Phrase))
   => (ground(Ind), ground(Gnd)).

add_annotations([],[],_,_).
add_annotations([I|Is],[P|Ps],Indep,Gnd)
    add_annotations(I,P,Indep,Gnd),
    add_annotations(Is,Ps,Indep,Gnd).

add_annotations(Info,Phrase,I,G) :- !,
    para_phrase( Info,Code,Type,Vars,I,G),
    make_CGE_phrase( Type,Code,Vars,PCode,I,G),
    (       var(Code),!,
            Phrase = PCode
    ;       Vars = [],!,
            Phrase = Code
    ;       Phrase = (PCode,Code)
    ).
```

```
Verified assertion:
:- check calls add_annotations(Info,Phrase,Ind,Gnd)
   : ( var(Phrase), indep(Info,Phrase) ).
Verified assertion:
:- check success add_annotations(Info,Phrase,Ind,Gnd
)
   : ( var(Phrase), indep(Info,Phrase) )
```

General idea

1. Take “snapshots” of the program sources
   (e.g., at each editor save/pause while developing, each commit, ...).
2. Detect the changes w.r.t. the previous snapshot.
3. Reanalyze:
   - Annotate and remove potentially outdated information.
   - (Re-)Analyze incrementally (only the parts needed) module by module until an intermodular fixpoint is reached again.

→ Recheck assertions/Reoptimize.

Analysis Basics

Abstract Interpretation

- Simulates the execution of the program using an **abstract domain** D_α, simpler than the contrete one.
- Guarantees:
  - Analysis termination, provided that D_α meets some conditions.
  - Results are **safe approximations** of the concrete semantics.

We use `Prolog` syntax for Horn Clauses

$$Head_k \texttt{ :- } B_{k,1}, \ldots, B_{k,n_k}$$

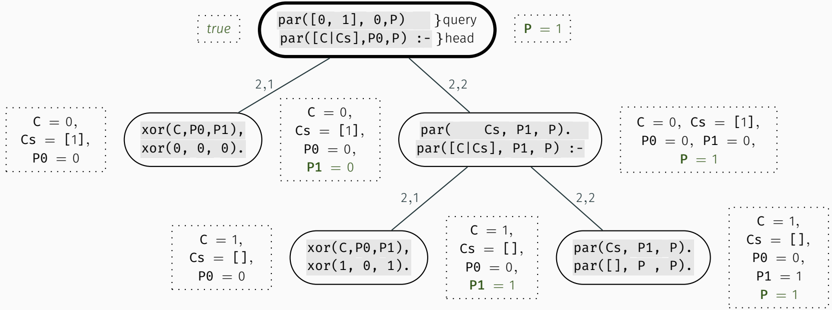
```prolog
list([]).            % fact
list([X|Xs]) :-      % rule head
    list(Xs).        % rule body
```

Concrete (Top-down) Semantics – AND trees

```
par([], P, P).
par([C|Cs], P0, P) :-
    xor(C, P0, P1),
    par(Cs, P1, P).
```

```
xor(0,0,0).
xor(0,1,1).
xor(1,0,1).
xor(1,1,0).
```

AND tree of `:- par([0, 1], 0,P).`:



PLAI Analysis output

A PLAI [NACLP'89] **analysis graph** has a set of **nodes** $\langle A, \lambda^c \rangle \mapsto \lambda^s$ for every potentially reachable predicate, where:

- A is an atom, the predicate identifier.
- λ^c is an abstract call to A.
- λ^s is the abstract answer for A and λ^c if it succeeds.

Example

```
par([], P, P).
par([C|Cs], P0, P) :-
    xor(C, P0, P1),
    par(Cs, P1, P).

xor(0,0,0).
xor(0,1,1).
xor(1,0,1).
xor(1,1,0).
```

Example nodes:

$\langle \mathtt{par}(\mathtt{L}, \mathtt{P0}, \mathtt{P}), \top \rangle \mapsto (P0/_{bit}, P/_{bit})$

For any call to `par` *that succeeds, P0 and P are either 1 or 0.*

$\langle \mathtt{par}(\mathtt{L}, \mathtt{P0}, \mathtt{P}), (P0/-) \rangle \mapsto \bot$

If `par` *is called with P0 a negative number, it always fails.*

Edges: $\langle P, \lambda \rangle_{i,j} \xrightarrow[\lambda^r]{\lambda^p} \langle Q, \lambda' \rangle$, *calling P with λ causes Q to be called with λ'.*

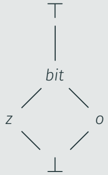
Analysis is **interprocedural**, **multivariant**, and **context sensitive**.

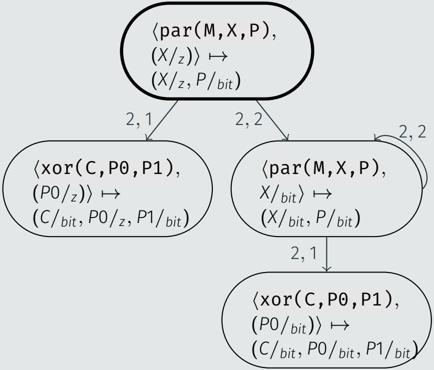
Analysis graph – example

Initial query $\langle par(M, X, P), (X/z)\rangle$

Entry: `:- par(M,0,P).`

```
par([], P, P).
par([C|Cs], P0, P) :-
    xor(C, P0, P1),
    par(Cs, P1, P).

xor(0,0,0).
xor(0,1,1).
xor(1,0,1).
xor(1,1,0).
```





Modular CHC Programs

Strict module system

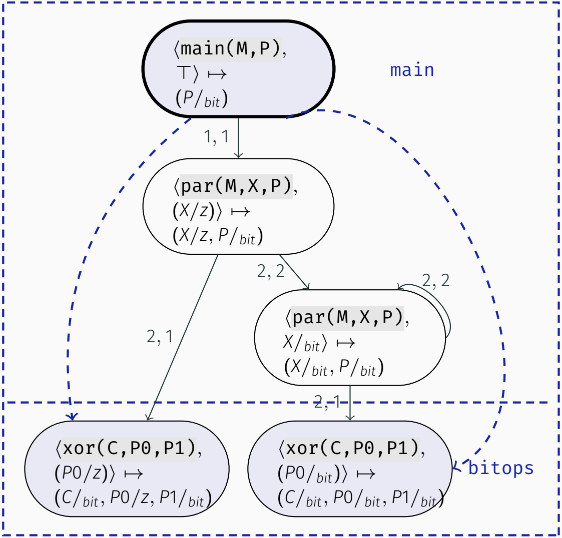
- Modules define an interface of exported and imported predicates.
- Non-exported predicates cannot be seen or used in other modules.

Modular program

```
:- module(main, [main/2]).

:- use_module(bitops, [xor/3]).

main(L,P) :-
    par(L,0,P).

par([], P, P).
par([C|Cs], P0, P) :-
    xor(C, P0, P1),
    par(Cs, P1, P).
```

```
:- module(bitops, [xor/3]).

xor(0,0,0).
xor(0,1,1).
xor(1,0,1).
xor(1,1,0).
```

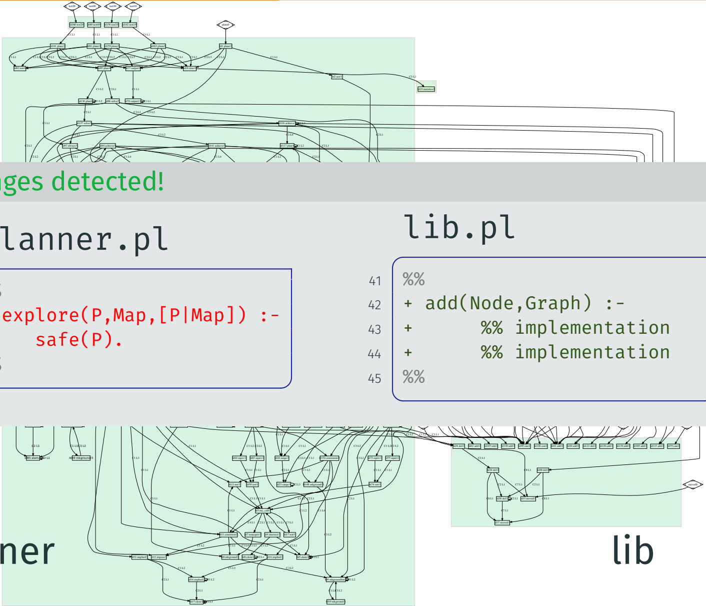
Graphs for Incremental and Modular Analysis



We have:

- A **global analysis graph** $\mathcal{G}$: call dependencies among imported/exported predicates.
- A **local analysis graph** $\mathscr{L}_M$ per module M: limited to the predicates used in M.

Snapshot of Analysis Graphs



Changes detected!

planner.pl

```
%%
- explore(P,Map,[P|Map]) :-
-     safe(P).
%%
```

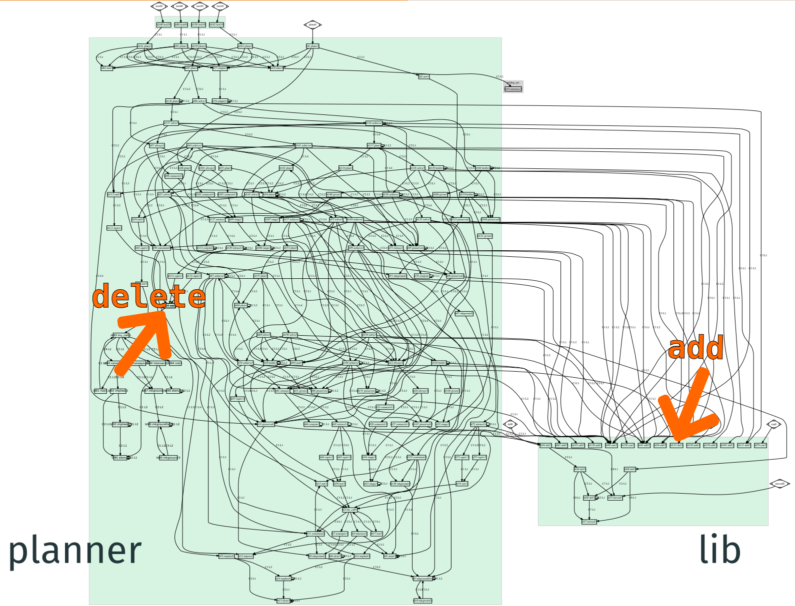
lib.pl

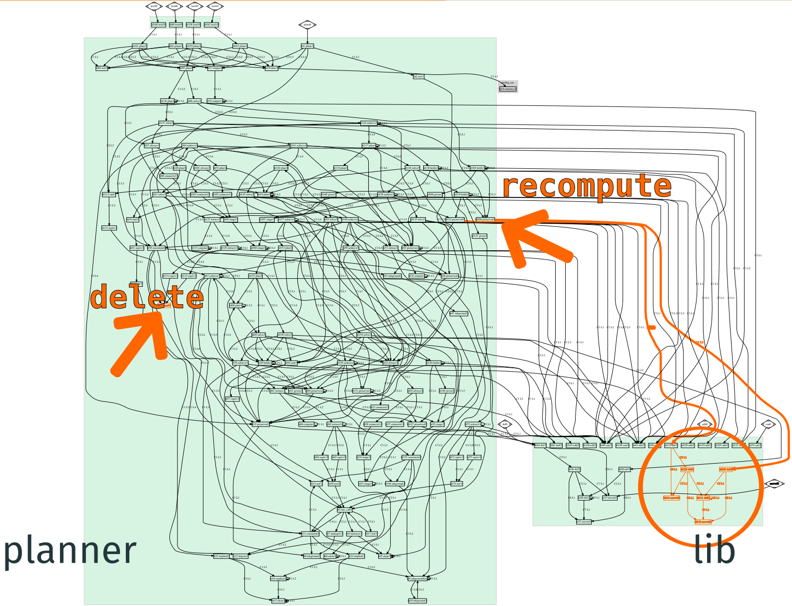
```
%%
+ add(Node,Graph) :-
+       %% implementation
+       %% implementation
%%
```

planner

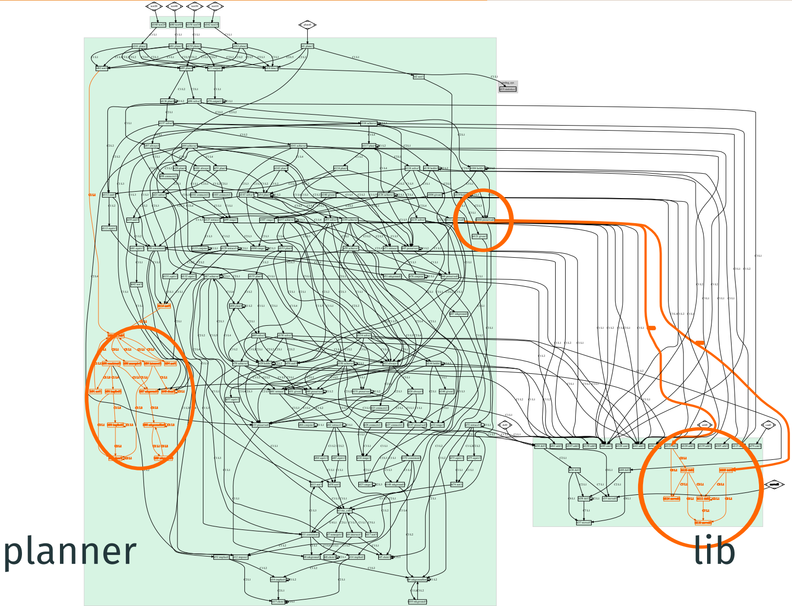
lib

Snapshot of Analysis Graphs

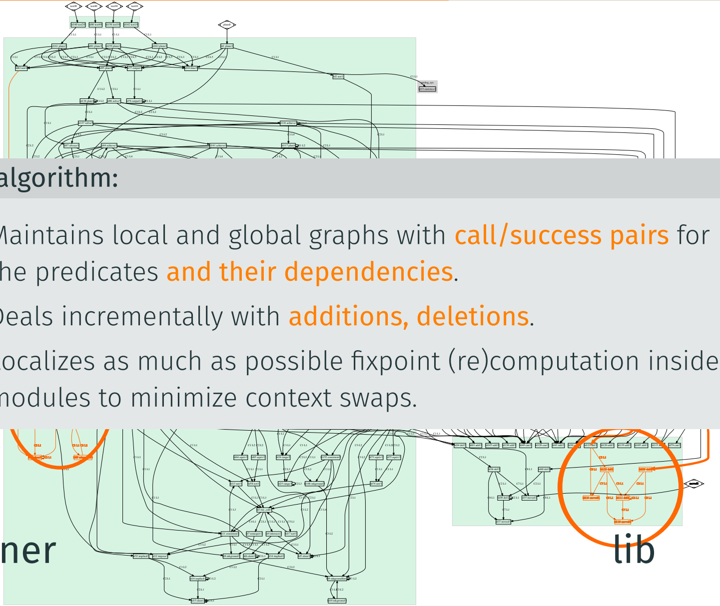



recompute
delete
planner
lib

Snapshot of Analysis Graphs



Snapshot of Analysis Graphs



The algorithm:

- Maintains local and global graphs with call/success pairs for the predicates and their dependencies.
- Deals incrementally with additions, deletions.
- Localizes as much as possible fixpoint (re)computation inside modules to minimize context swaps.

planner

lib

Fundamental results

Theorem 4 (Correctness of $\textsc{IncAnalyze}$ starting from a partial analysis)**.** *Let P be a program, Q_α a set of abstract queries, and $\mathscr{A}_0$* any *analysis graph. Let $\mathscr{A} = \textsc{IncAnalyze}(P, Q_\alpha, \emptyset, \mathscr{A}_0)$. $\mathscr{A}$ is* correct *for P and $\gamma(Q_\alpha)$ if for all concrete queries $q \in \gamma(Q_\alpha)$ all nodes n from which there is a path in the concrete execution $q \rightsquigarrow n$ in $[\![P]\!]_Q$, that are abstracted in the analysis $\mathscr{A}_0$ are included in Q_α, i.e.:*

$$\forall Q, n. Q \in \gamma(Q_\alpha) \land q \rightsquigarrow n \in [\![P]\!]_Q,$$
$$\forall n_\alpha \in \mathscr{A}_0 . n \in \gamma(n_\alpha) \Rightarrow n_\alpha \in Q_\alpha.$$

Theorem 6 (Precision of $\textsc{IncAnalyze}$)**.** *Let P, P' be programs, such that P differs from P' by Δ, let Q_α a set of abstract queries, and $\mathscr{A}_0 = \textsc{IncAnalyze}(P', Q_\alpha, \emptyset, \emptyset)$ an analysis graph. The following hold:*

- *If $\mathscr{A} = \textsc{IncAnalyze}(P, Q_\alpha, \emptyset, \emptyset)$, then $\mathscr{A}$ is the* least program analysis graph *for P and $\gamma(Q_\alpha)$, and*
- $\textsc{IncAnalyze}(P, Q_\alpha, \Delta, \mathscr{A}_0) = \textsc{IncAnalyze}(P, Q_\alpha, \emptyset, \emptyset)$.

Lemma 1 (Correctness of $\textsc{IncAnalyze}$ modulo imported predicates)**.** *Let M be a module of program P, E a set of abstract queries. Let $\mathscr{L}_0$ be an analysis graph such that $\forall \langle A, \lambda^c \rangle \in \mathscr{L}_0.\mathsf{mod}(A) \in \mathsf{imports}(M)$. The analysis result*

$$\mathscr{L} = \textsc{IncAnalyze}(M, E, \emptyset, \mathscr{L}_0)$$

is correct *for M and $\gamma(E)$ assuming $\mathscr{L}_0$.*

Lemma 2 (Precision of $\textsc{IncAnalyze}$ modulo imported predicates)**.** *Let M be a module of program P, E a set of abstract queries. Let $\mathscr{L}_0$ be an analysis graph such that $\forall \langle A, \lambda^c \rangle \in \mathscr{L}_0.\mathsf{mod}(A) \in \mathsf{imports}(M)$ if $\mathscr{L}_0$ contains* the least fixed point *as defined in Theorem 6. The analysis result*

$$\mathscr{L} = \textsc{IncAnalyze}(M, E, \emptyset, \mathscr{L}_0)$$

is the least program analysis graph *for M and $\gamma(E)$ assuming $\mathscr{L}_0$.*

Lemma 3 (Correctness updating $\mathscr{L}$ modulo $\mathcal{G}$)**.** *Let M be a module of program P and E a set of entries. Let $\mathcal{G}$ be a previous state of the global analysis graph, if $\mathscr{L}_M$ is correct for M and $\gamma(E)$ assuming $\mathcal{G}$. If $\mathcal{G}$ changes to $\mathcal{G}'$ the analysis result*

$$\mathscr{L}'_M = \textsc{LocIncAnalyze}(M, E, \mathcal{G}', \mathscr{L}_M, \emptyset)$$

is correct *for M and $\gamma(E)$ assuming $\mathcal{G}$.*

Theorem 10 (Correctness of $\textsc{ModIncAnalyze}$ from scratch)**.** *Let P be a modular program, and Q_α a set of abstract queries. Then, if:*

$$\{\mathcal{G}, \{\mathscr{L}_{M_i}\}\} = \textsc{ModIncAnalyze}(P, Q_\alpha, \emptyset, \emptyset)$$

$\mathcal{G}$ is correct *for P and $\gamma(Q_\alpha)$.*

Lemma 4 (Precision updating $\mathscr{L}$ modulo $\mathcal{G}$)**.** *Let M be a module contained in program P, E a set of entries. Let $\mathcal{G}$ be a previous state of the global analysis graph, if $\mathscr{L}_M = \textsc{LocIncAnalyze}(M, E, \mathcal{G}, \emptyset, \emptyset)$. If $\mathcal{G}$ changes to $\mathcal{G}'$ the analysis result:*

$$\textsc{LocIncAnalyze}(M, E, \mathcal{G}', \mathscr{L}_M, \emptyset) = \textsc{LocIncAnalyze}(M, E, \mathcal{G}', \emptyset, \emptyset)$$

is the same as analyzing from scratch, i.e., the lfp *of M, E.*

Theorem 11 (Precision of $\textsc{ModIncAnalyze}$ from scratch)**.** *Let P be a modular program and Q_α a set of abstract queries. The analysis result*

$$\mathscr{A} = \textsc{ModIncAnalyze}(P, Q_\alpha, \emptyset, \emptyset) = \textsc{ModAnalyze}(P, Q_\alpha)$$

such that $\mathscr{A} = \{\mathcal{G}, \{\mathscr{L}_{M_i}\}\}$, then $\mathcal{G} = \mathcal{G}'$.

Theorem 12 (Precision of $\textsc{ModIncAnalyze}$)**.** *Let P, P' be modular programs that differ by Δ, Q_α a set of queries, and $\mathscr{A} = \textsc{ModIncAnalyze}(P, Q_\alpha, \emptyset, (\emptyset, \emptyset))$, then*

$$\textsc{ModIncAnalyze}(P', Q_\alpha, \emptyset, (\emptyset, \emptyset)) = \textsc{ModIncAnalyze}(P', Q_\alpha, \mathscr{A}, \Delta).$$

Fundamental results

Theorem 4 (Correctness of IncAnalyze starting from a partial analysis). *Let P be a program, Q_α a set of abstract queries, and $\mathscr{A}_0$* any analysis graph. *Let $\mathscr{A} = \textsc{IncAnalyze}(P, Q_\alpha, \emptyset, \mathscr{A}_0)$. $\mathscr{A}$ is* correct *for P and $\gamma(Q_\alpha)$ if*

Lemma 3 (Correctness updating $\mathscr{L}$ modulo $\mathcal{G}$). *Let M be a module of program P and E a set of entries. Let $\mathcal{G}$ be a previous state of the global analysis graph, if $\mathscr{L}_M$ is correct for M and $\gamma(E)$ assuming $\mathcal{G}$. If $\mathcal{G}$ changes to $\mathcal{G}'$ the analysis result*

Contributions

The results from our incremental, modular analysis are:

- Correct over-approximations of the AND tree semantics.
- The most accurate (lfp) if no widening is performed.

Additionally:

- Extended traditional algorithm with widening (not formalized before).
- Split correctness and precision of incremental analysis.
- New results reanalyzing starting from a partial analysis.
- Formalized results of an existing modular algorithm (non incremental).

queries. Let $\mathscr{L}_0$ be an analysis graph such that $\forall \langle A, \lambda^c \rangle \in \mathscr{L}_0.\mathsf{mod}(A) \in \mathsf{imports}(M)$ if $\mathscr{L}_0$ contains the least fixed point *as defined in Theorem 6. The analysis result*

$$\mathscr{L} = \textsc{IncAnalyze}(M, E, \emptyset, \mathscr{L}_0)$$

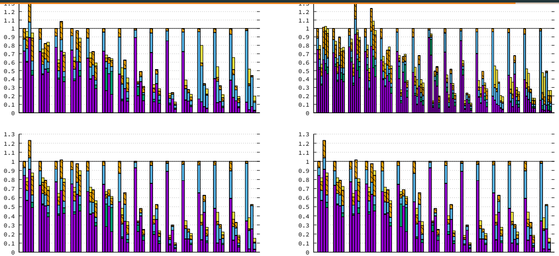
is the least program analysis graph *for M and $\gamma(E)$ assuming $\mathscr{L}_0$.*

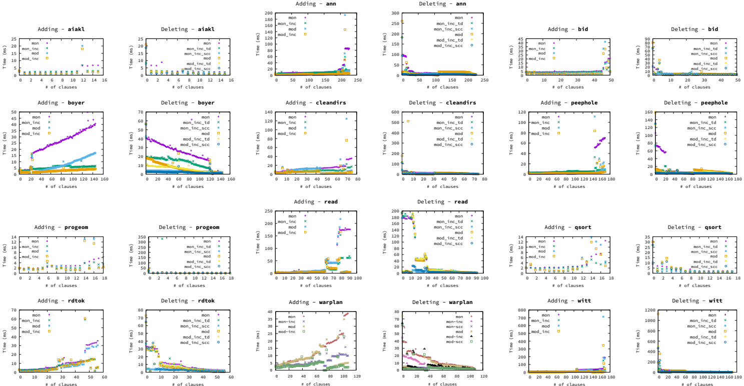
such that $\mathscr{A} = \{\mathcal{G}, \{\mathscr{L}_{M_i}\}\}$, then $\mathcal{G} = \mathcal{G}'$.

Theorem 12 (Precision of ModIncAnalyze). *Let P, P' be modular programs that differ by Δ, Q_α a set of queries, and $\mathscr{A} = \textsc{ModIncAnalyze}(P, Q_\alpha, \emptyset, (\emptyset, \emptyset))$, then*

$$\textsc{ModIncAnalyze}(P', Q_\alpha, \emptyset, (\emptyset, \emptyset)) = \textsc{ModIncAnalyze}(P', Q_\alpha, \mathscr{A}, \Delta).$$

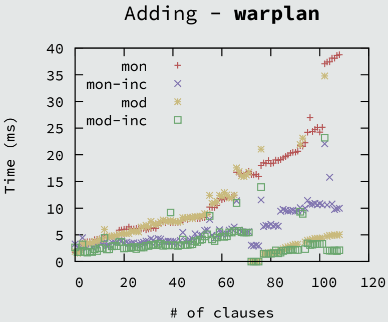
Experimental evaluation





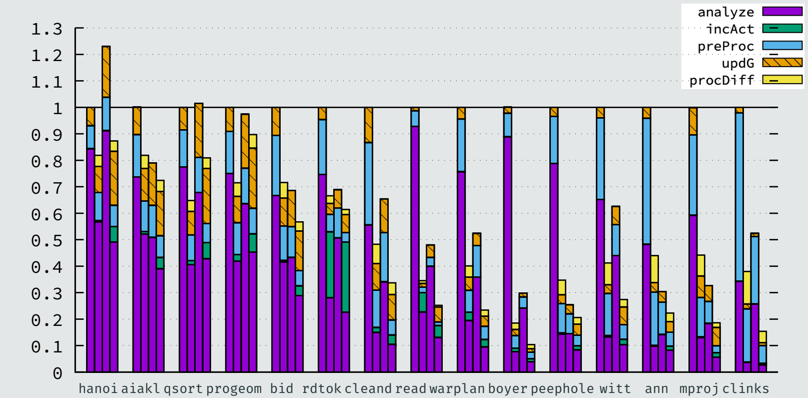
Experimental evaluation

Addition experiment (time in ms) – `def` domain



Experimental evaluation

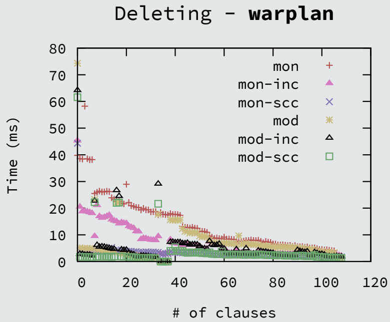
Accumulated normalized time (def) – clause addition



The order inside each set of bars is: |mon|mon_inc|mod|mod_inc|.

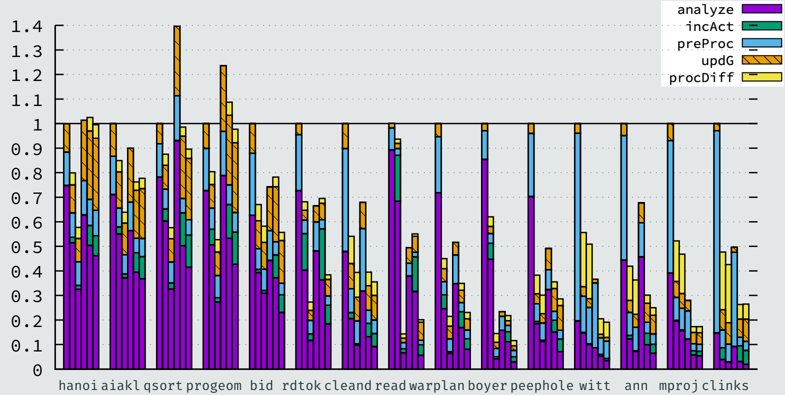
Experimental evaluation

Deletion experiment (time in ms) - `def` domain



Experimental evaluation

Accumulated normalized time (def) – clause deletion



The order inside each set of bars is: |mon|mon_td|mon_scc|mod|mod_td|mod_scc|

The Approach in Action - Static *On-the-fly* verification in CiaoPP

```
        P = B
    ;   rewrite(clause(H,B),clause(H,P),I,G,Info)
    ).

rewrite( clause(H,B), clause(H,P),I,G,Info) :-
    numbervars_2(H,0,Lhv),
    collect_info(B,Info,Lhv,_X,_Y),
    add_annotations(Info,P,I,G),!.

:- pred add_annotations(Info,Phrase,Ind,Gnd)
   : (var(Phrase), indep(Info,Phrase))
   => (ground(Ind), ground(Gnd)).

add_annotations([],[],_,_).
add_annotations([I|Is],[P|Ps],Indep,Gnd) :-
    add_annotations(I,P,Indep,Gnd),
    add_annotations(Is,Ps,Indep,Gnd).

add_annotations(Info,Phrase,I,G) :- !,
    para_phrase( Info,Code,Type,Vars,I,G),
```

```
Verified assertion:
:- check calls add_annotations(Info,Phrase,Ind,Gnd)
   : ( var(Phrase), indep(Info,Phrase) ).
Verified assertion:
:- check success add_annotations(Info,Phrase,Ind,Gnd
)
   : ( var(Phrase), indep(Info,Phrase) )
```

Average assertion checking time (seconds)

Benchmark: `chat-80` port – 5.2*k* LOC across 27 files (`Ciao` Prolog), 20 assertions/experiment.

		a-cls	r-cls	t-cls	a-asr	r-asr	t-asr
`non-inc`	analysis	2.0	2.0	1.8	2.1	2.1	2.1
	total	**3.0**	**2.4**	**3.2**	**2.9**	**2.4**	**2.9**
`inc`	diff	0.1	0.1	0.1	0.1	0.1	0.1
	(re)analysis	0.4	0.4	0.3	–	–	–

Conclusion

To take home:

- Almost **immediate** response when the changes do not affect the result.
- Up to 13× speedup w.r.t. the original non-incremental algorithm.
- Being aware of **modular structures** is useful: Up to 2× speedup when compared with the original incremental algorithm.
- **Modular analysis** from scratch is **improved** up to 9×.
- Keeping structures for incrementality produces **small overhead**.
- Using the analyzer **interactively**, **on the fly** becomes practical.

Future work

- Amenability of abstract domains to incrementality.
- Heuristics for automatic configuration of incrementality settings.
- Applications in the program transformation/partial evaluation context.
- Incrementality-aware transformation (from other source languages).

Thanks!

CiaoPP: `https://github.com/ciao-lang/ciaopp`

Experiments/benchmarks: `https://github.com/ciao-lang/ciaopp_tests/tree/master/tests/incanal`

Full version: `https://doi.org/10.1017/S1471068420000496`