A scalable static analysis framework for reliable program development exploiting incrementality and modularity

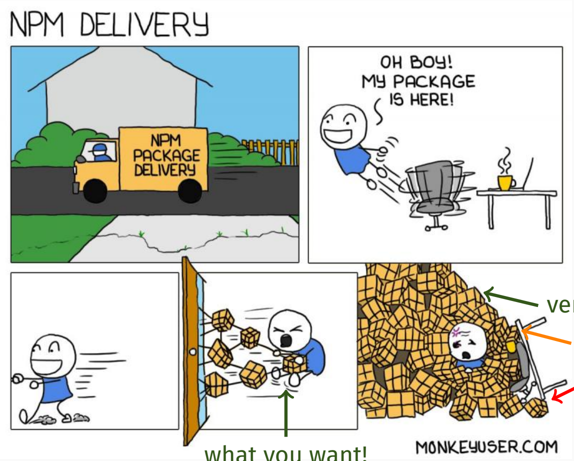
Isabel García Contreras

Universidad Politécnica de Madrid
IMDEA Software Institute

PhD Thesis Defense
July 21st, 2021

Why analyze/verify software?

A motivation for us programmers



verified library

random github project

ML-generated code

what you want!

Introduction

Context: analyzing/verifying software projects during development to:

- detect and report 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.

Challenges addressed in this thesis:

- performance: incremental and modular analysis to take advantage of localized changes,
    - application: *on-the-fly* assertion checking,
- precision: allowing the programmer to guide the analysis,
- incomplete code: manual specification + reanalysis, and
- precision: incomplete abstract intepretations.

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) )
```

*actual in-Emacs footage

Basics

Constraint Logic Programs/Horn Clauses

$$Head_k \leftarrow B_{k,1}, \ldots, B_{k,n_k}$$

We use **Prolog** syntax (“`:-`” instead of “$\leftarrow$”):

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

Abstract Interpretation [Cousot & Cousot POPL’77]

Simulates the execution of the program using an **abstract domain** D_α, simpler than the contrete one. Guarantees:

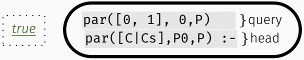
- analysis termination, provided that D_α meets some conditions,
- results are **safe approximations** of the concrete semantics.

Concrete 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).
```

Top-down AND tree of `:- par([0, 1], 0,P).` [Bruynooghe JLP'91]



Concrete 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).
```

Top-down AND tree of `:- par([0, 1], 0,P).` [Bruynooghe JLP'91]

true

`par([0, 1], 0,P)` }query
`par([C|Cs],P0,P) :-` }head

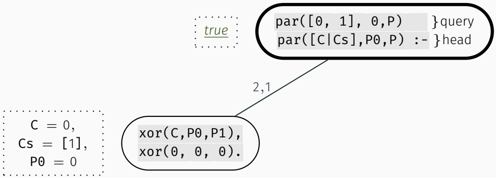
C = 0,
Cs = [1],
P0 = 0

Concrete 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).
```

Top-down AND tree of `:- par([0, 1], 0,P).` [Bruynooghe JLP'91]

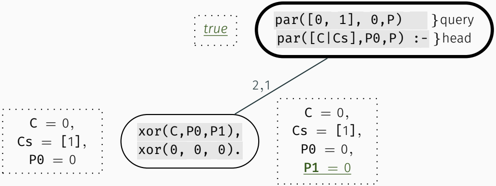


Concrete semantics – AND trees

```
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).
```

Top-down AND tree of `:- par([0, 1], 0,P).` [Bruynooghe JLP'91]

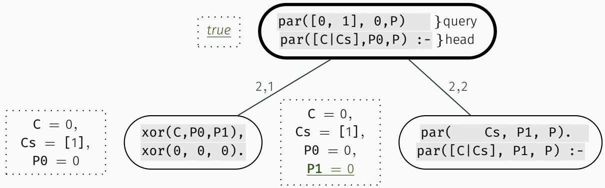


Concrete semantics – AND trees

```
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).
```

Top-down AND tree of `:- par([0, 1], 0,P).` [Bruynooghe JLP'91]

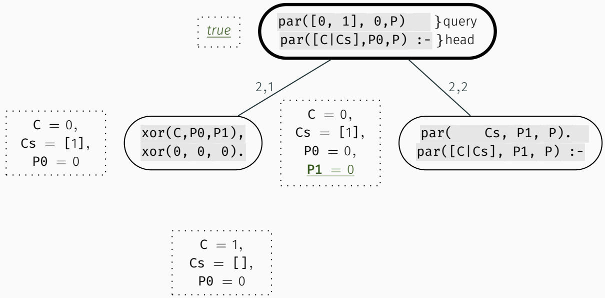


Concrete semantics – AND trees

```
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).
```

Top-down AND tree of `:- par([0, 1], 0,P).` [Bruynooghe JLP'91]

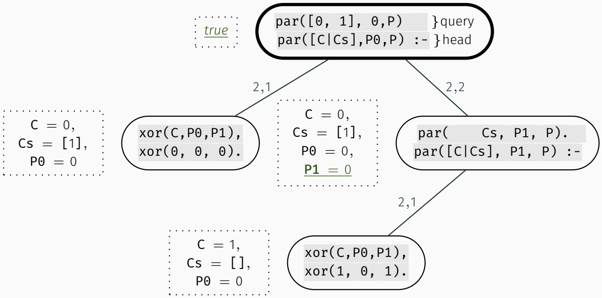


Concrete semantics – AND trees

```
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).
```

Top-down AND tree of `:- par([0, 1], 0,P).` [Bruynooghe JLP'91]

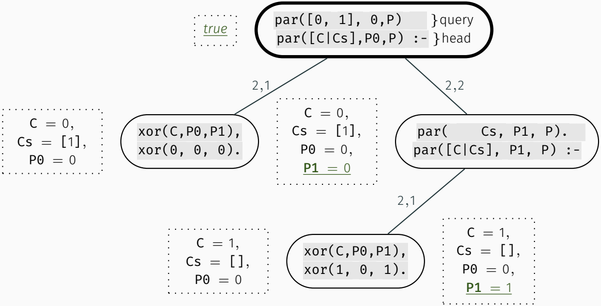


Concrete semantics – AND trees

```
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).
```

Top-down AND tree of `:- par([0, 1], 0,P).` [Bruynooghe JLP'91]

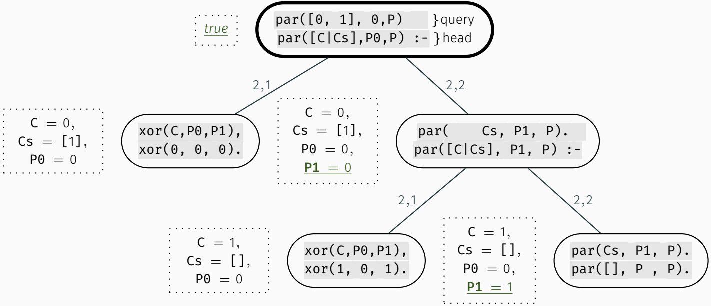


Concrete semantics – AND trees

```
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).
```

Top-down AND tree of `:- par([0, 1], 0,P).` [Bruynooghe JLP'91]

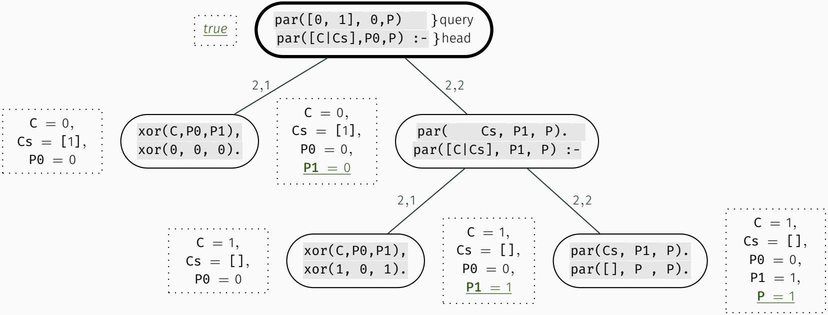


Concrete semantics – AND trees

```
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).
```

Top-down AND tree of `:- par([0, 1], 0,P).` [Bruynooghe JLP'91]

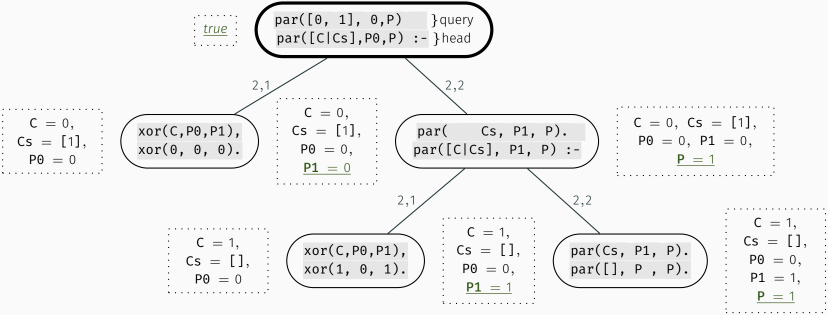


Concrete semantics – AND trees

```
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).
```

Top-down AND tree of `:- par([0, 1], 0,P).` [Bruynooghe JLP'91]

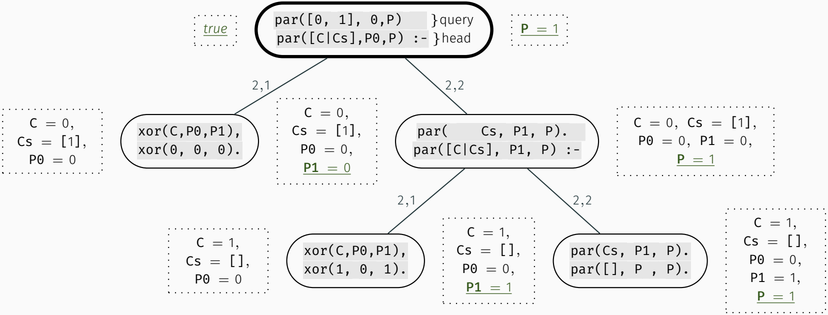


Concrete semantics – AND trees

```
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).
```

Top-down AND tree of `:- par([0, 1], 0,P).` [Bruynooghe JLP'91]



Abstract semantics

A PLAI **analysis graph** ($\mathscr{A}$) has a set of **nodes** $\langle A, \lambda^c \rangle \mapsto \lambda^s$ for every potentially reachable predicate, where: [NACLP'89, TOPLAS'00]

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

λ^c and λ^s are values of some abstract domain D_α.

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:

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

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

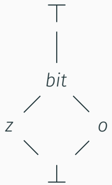
$\langle \texttt{par(L,P0,P)}, (P0/-) \rangle \mapsto \bot$

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

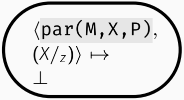
Building an analysis graph

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

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



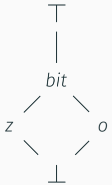
Initial query $\langle par(M, X, P), (X/z)\rangle$



Building an analysis graph

```
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).
```



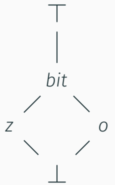
Initial query $\langle par(M, X, P), (X/z)\rangle$

$\langle$`par(M,X,P)`, $(X/_z)\rangle \mapsto (X/_z, P/_z)$

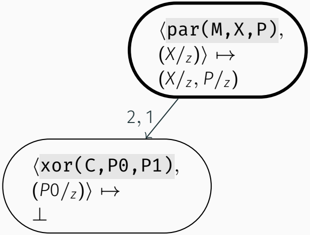
Building an analysis graph

```
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).
```



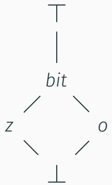
Initial query $\langle par(M, X, P), (X/z)\rangle$



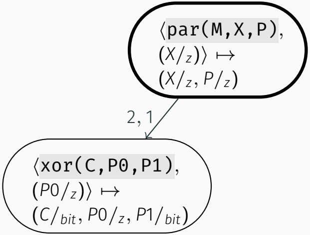
Building an analysis graph

```
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).
```



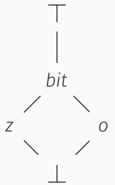
Initial query $\langle par(M, X, P), (X/z)\rangle$



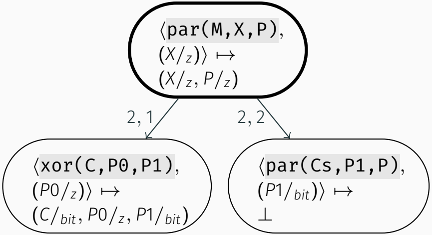
Building an analysis graph

```
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).
```



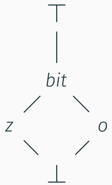
Initial query $\langle par(M, X, P), (X/z)\rangle$



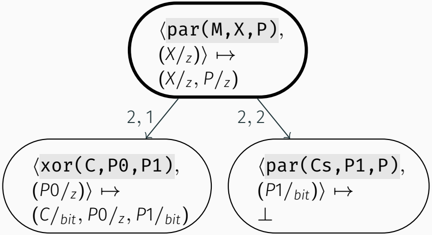
Building an analysis graph

```
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).
```



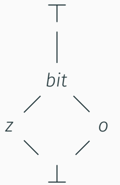
Initial query $\langle par(M, X, P), (X/z)\rangle$



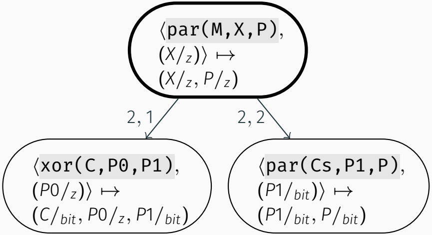
Building an analysis graph

```
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).
```



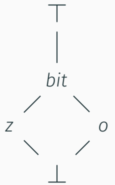
Initial query $\langle par(M, X, P), (X/z)\rangle$



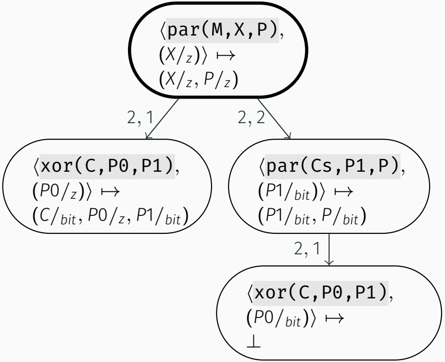
Building an analysis graph

```
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).
```



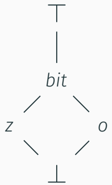
Initial query $\langle par(M, X, P), (X/z)\rangle$



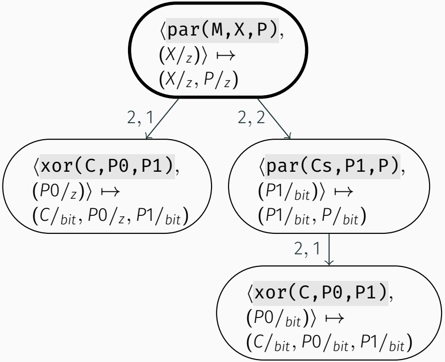
Building an analysis graph

```
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).
```



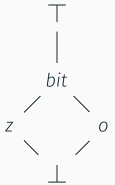
Initial query $\langle par(M, X, P), (X/z)\rangle$



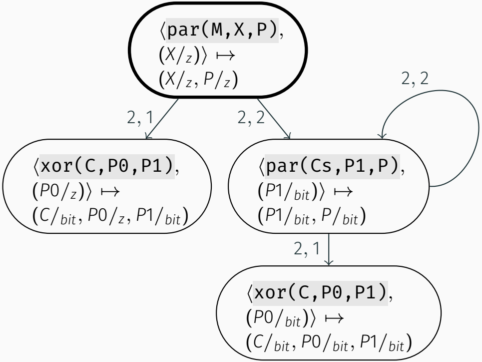
Building an analysis graph

```
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).
```



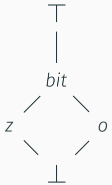
Initial query $\langle par(M, X, P), (X/z)\rangle$



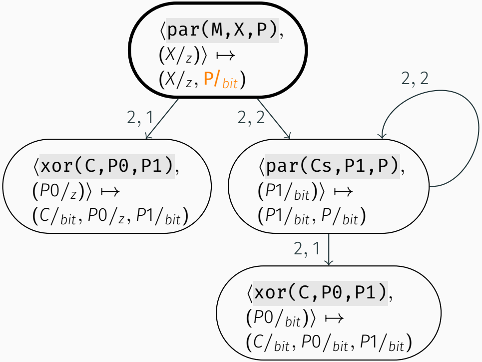
Building an analysis graph

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

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

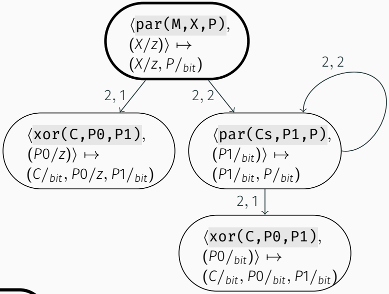


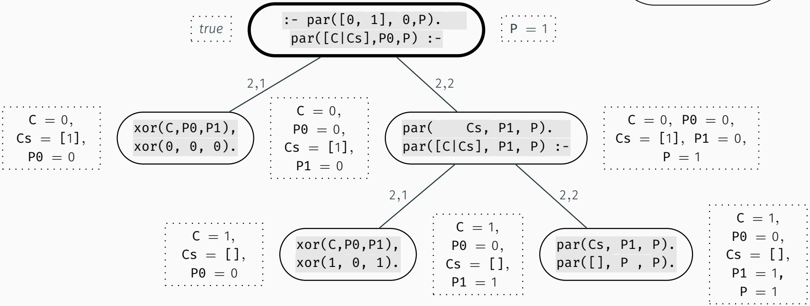
Initial query $\langle par(M, X, P), (X/z)\rangle$



Analysis correctness

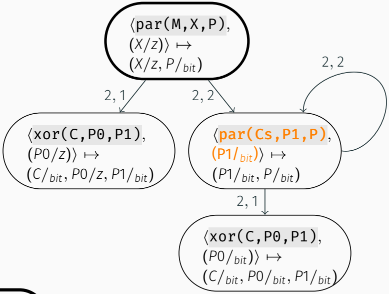
An analysis graph $\mathscr{A}$ is **correct** for a program P and a set of queries Q if it approximates all the calls, answers, and dependencies in the concrete semantics $[\![P]\!]_{\mathscr{Q}}$ (a set of AND trees).

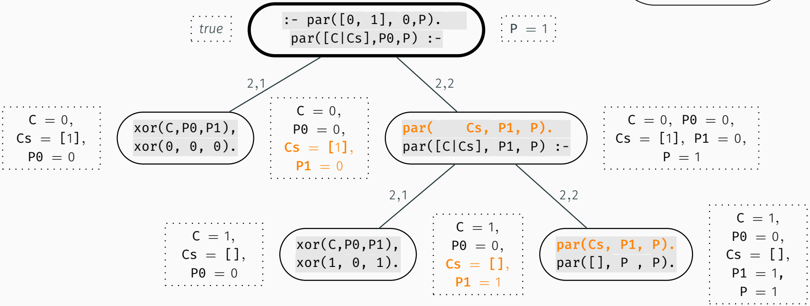




Analysis correctness

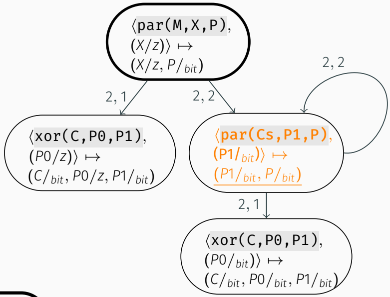
An analysis graph $\mathscr{A}$ is **correct** for a program P and a set of queries Q if it approximates all the **calls**, answers, and dependencies in the concrete semantics $[\![P]\!]_{\mathcal{Q}}$ (a set of AND trees).

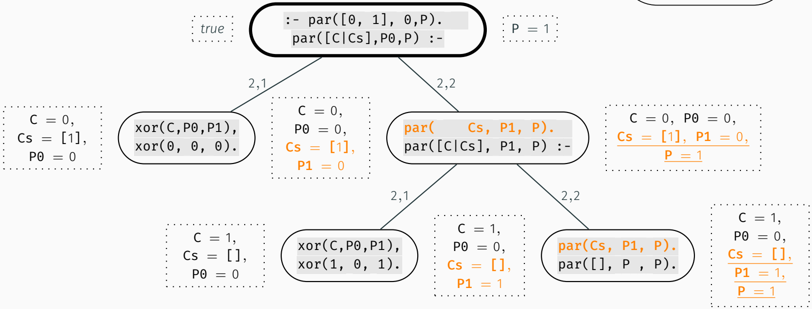




Analysis correctness

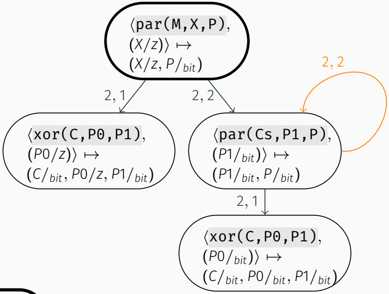
An analysis graph $\mathscr{A}$ is **correct** for a program P and a set of queries Q if it approximates all the calls, **answers**, and dependencies in the concrete semantics $[\![P]\!]_{\mathcal{Q}}$ (a set of AND trees).

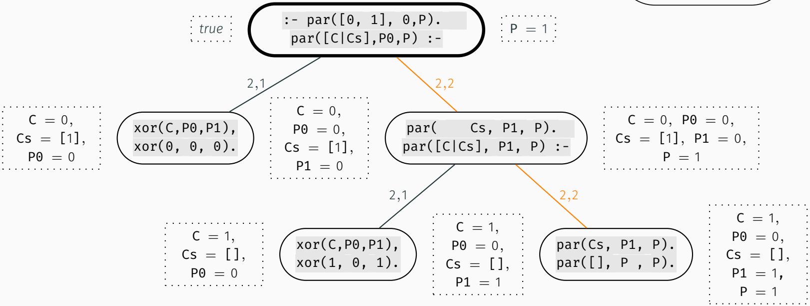




Analysis correctness

An analysis graph $\mathscr{A}$ is **correct** for a program P and a set of queries Q if it approximates all the calls, answers, and **dependencies** in the concrete semantics $[\![P]\!]_{\mathscr{Q}}$ (a set of AND trees).





Analysis graphs

Two levels of abstraction of program execution:

- control: unbounded number of AND trees as a graph,
- data: parametric abstraction by providing domain operations.

Properties:

- interprocedural: each node contains a summary of the behavior of a predicate,
- multivariance: distinguish different abstract call patterns for
  - precision – differentiate contexts (for optimizations/verification),
  - efficiency – localize recomputation,
- path approximations: the edges in a path of the graph abstract the call stack and ordered literals form a regular approximation of all previously called predicates,

Note that analysis graphs may be used to analyze imperative programs, either via translation to Horn clauses [LOPSTR07] or directly [FTfJP07].

Incremental analysis algorithm

[TOPLAS'00]

Input	$\mathcal{Q}_\alpha$: initial abstract queries. P': target program (changed). Δ: clauses that changed from P to P'. $\mathcal{A}$: analysis results of P.
Output	a correct analysis graph for P' and $\mathcal{Q}_\alpha$.

The algorithm is based on:

- events to trigger (re)analysis at the level of literals,
- when abstractly executing a predicate call, a subgraph is reused if possible,
- for abstract domains requiring widening:
  - successive increasing calls turned into a cycle in the graph,
  - successive increasing successes are generalized (widened),
  - the order in event processing affects the abstract values in analysis result.

Adding clauses

Expand the graph for the new possible calls (and update answers).

Deleting clauses

For precision: remove subgraphs and recompute.

General conditions when restarting an analysis

The following conditions justify all incremental algorithms in the thesis, as well as the adding and deleting clauses.

Starting from a correct partial analysis

If an analysis graph correctly abstracts all the behaviors represented in calls of its node, it can be reused to obtain a correct and precise analysis.

Starting from any partial analysis

An analysis graph can be reused to obtain a correct and precise analysis if for any node:

- it correctly abstracts all the behaviors represented by its call, or
- it is scheduled to be reanalyzed.

Key: when reusing an analysis result, for efficiency, subgraphs are never (re)checked, only the success of its root node is reused when abstractly executing a literal.

Incremental and modular analysis

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 module by module until an intermodular fixpoint is reached again.

So far, in abstract interpretation:

- fine-grain (clause-level) incremental analysis for non-modular programs.
  [ICLP’95, SAS’96, Kelly et. al ACSC’97, TOPLAS’00, Albert et. al PEPM’12]
- coarse-grain (module-level) analysis aimed at reducing memory consumption.
  [Codish et. al POPL’93, ENTCS’00, LOPSTR’01, Cousot & Cousot CC’02]

Modular logic programs

Strict module system

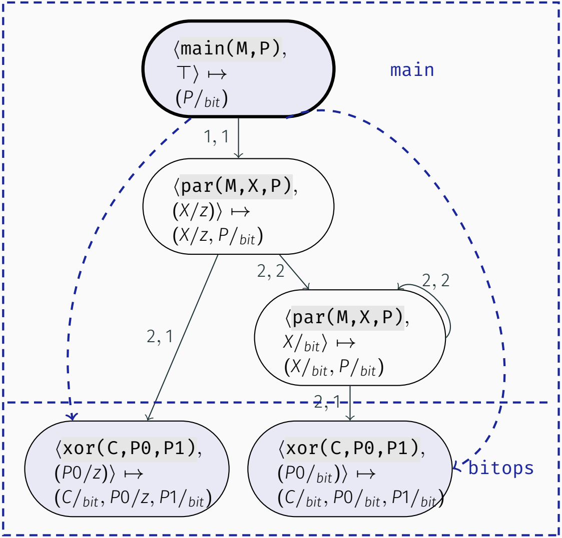
- Modules define an interface of exported and imported predicates.
- Non-exported predicates cannot be seen or used in other modules.

Modular program

```prolog
:- 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).
```

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

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

Analysis 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.

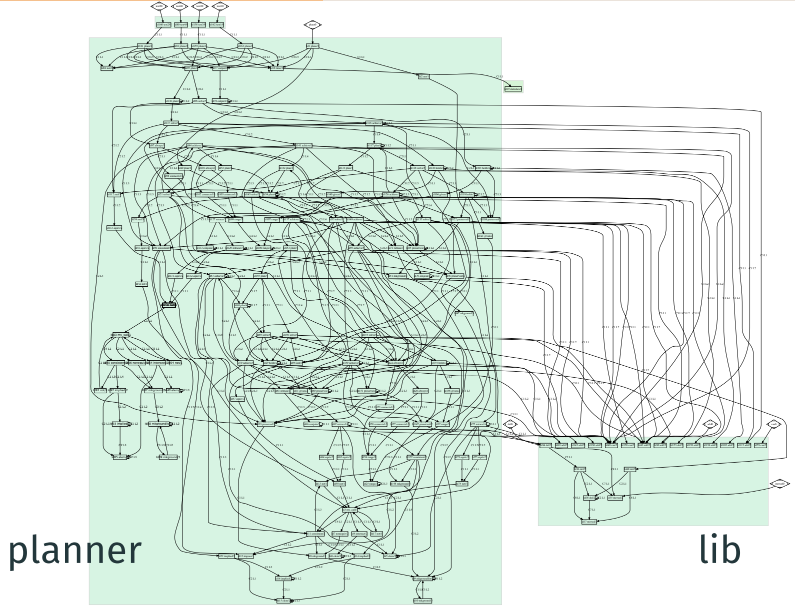
Incremental and modular analysis algorithm

Input	$\mathcal{Q}_\alpha$: initial abstract queries.
	P' ($\{M_i\}$): target program (changed).
	Δ: clauses that changed from **P** to **P'** (split by module).
	$\mathscr{A}$ ($\mathcal{G}$, $\{\mathscr{L}_i\}$): analysis results of **P**.
Output	a correct analysis graph for **P'** and $\mathcal{Q}_\alpha$.

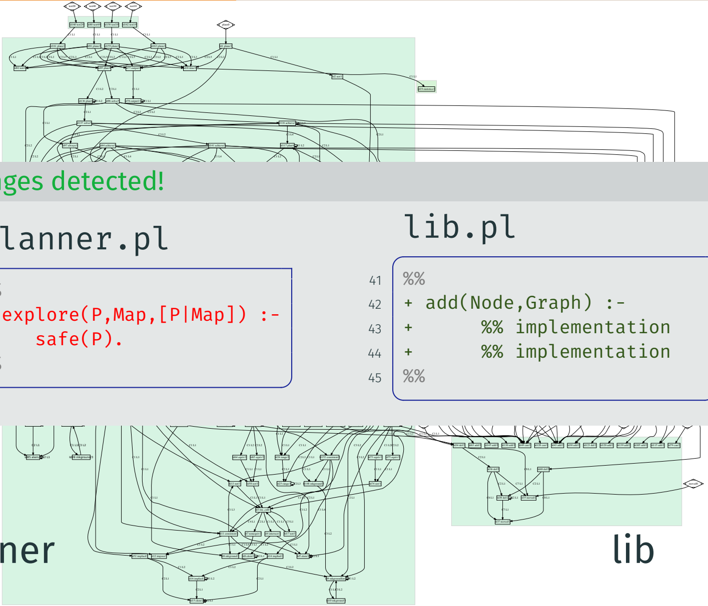
The algorithm:

- assumes as answer $\perp$ when a module has not been analyzed yet,
- (re)starts the analysis of modules using the dependencies in $\mathcal{G}$,
- updates the answers of the imported modules by scheduling new events,
- iterates until an intermodular fixpoint is reached, i.e., the global analysis graph does not change.

Snapshot of analysis graphs



Snapshot of analysis graphs



Changes detected!

planner.pl

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

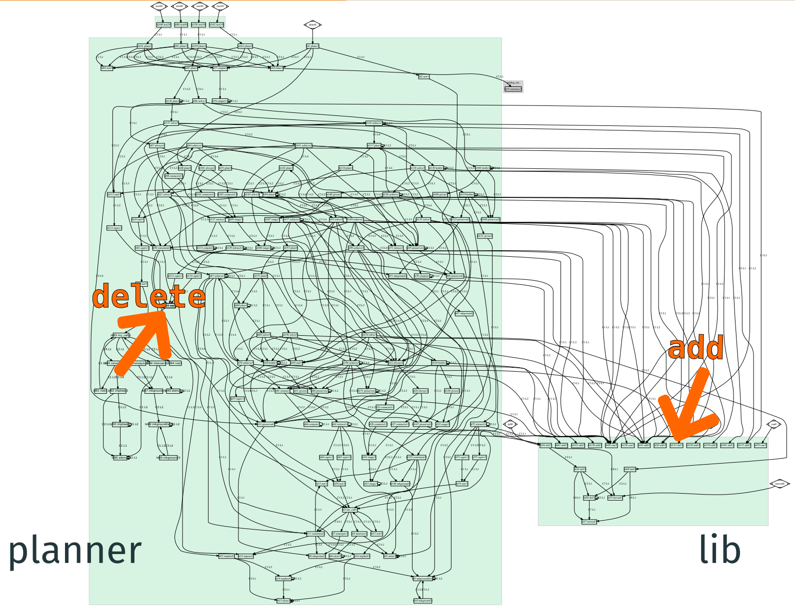
lib.pl

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

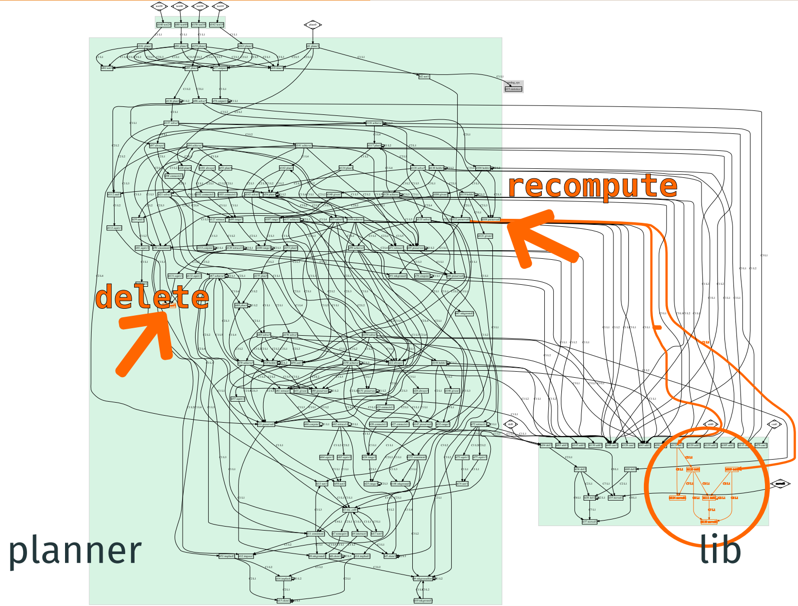
planner

lib

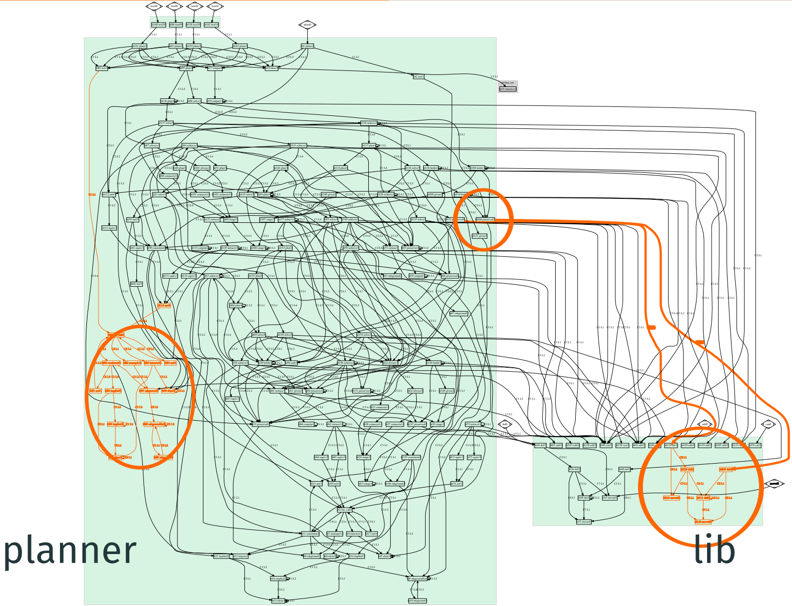
Snapshot of analysis graphs



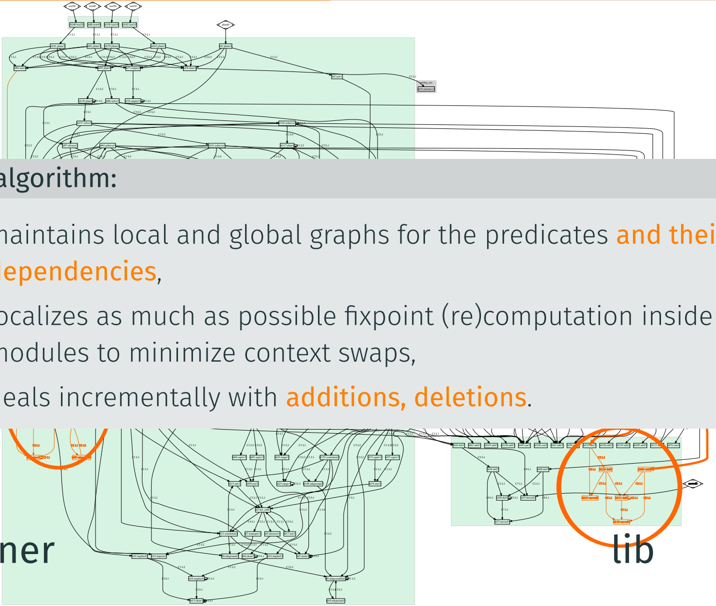
Snapshot of analysis graphs



Snapshot of analysis graphs



Snapshot of analysis graphs



The algorithm:

- maintains local and global graphs for the predicates and their dependencies,
- localizes as much as possible fixpoint (re)computation inside modules to minimize context swaps,
- deals incrementally with additions, deletions.

planner

lib

Fundamental results

Lemma 4.10 (Correctness of IncAnalyze starting from a correct partial analysis)**.** *Let P be a program, Q_α be a set of abstract queries, and fix $q \in Q_\alpha$. Suppose that $\mathscr{A}_0$ is the analysis result $\mathscr{A}_0 = \text{IncAnalyze}(P, Q_\alpha \setminus \{q\}, \emptyset, \emptyset)$. Then the analysis result $\mathscr{A} = \text{IncAnalyze}(P, Q_\alpha, \emptyset, \mathscr{A}_0)$ is correct for P and $\gamma(Q_\alpha)$.*

Theorem 4.11 (Correctness of IncAnalyze starting from a partial analysis)**.** *Let P be a program, Q_α a set of abstract queries, and $\mathscr{A}_0$ a well-formed analysis graph for P. Suppose for all concrete queries $q \in \gamma(Q_\alpha)$, for all nodes n from which there is a path in the concrete execution $q \rightsquigarrow n$ in $[\![P]\!]_Q$, and for all $n_\alpha \in \mathscr{A}_0$ such that $n \in \gamma(n_\alpha)$ either:*

a) $n_\alpha \in Q_\alpha$, or

b) the subgraph with root n_α is correct for P and $\{\gamma(n_\alpha)\}$.

Then $\mathscr{A} = \text{IncAnalyze}(P, Q_\alpha, \emptyset, \mathscr{A}_0)$ is correct for P and $\gamma(Q_\alpha)$.

Theorem 4.12 (Correctness and precision of IncAnalyze95 starting from a partial analysis)**.** *Under the same conditions as Theorem 4.11, if $\mathscr{A}_0 \sqsubseteq \mathscr{A}$, then:*

$$\text{IncAnalyze95}(P, Q_\alpha, \emptyset, \emptyset) = \text{IncAnalyze95}(P, Q_\alpha, \emptyset, \mathscr{A}_0).$$

Lemma 4.13 (Correctness of 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*

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

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

Lemma 4.13 (Precision of 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 $\forall \langle A, \lambda^c \rangle \in \mathscr{L}_0.\mathsf{mod}(A) \in \mathsf{imports}(M)$ correctly and precisely approximates the behavior of the imported predicates. The analysis result*

$$\mathscr{L} = \text{IncAnalyze95}(M, E, \emptyset, \mathscr{L}_0)$$

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

Lemma 5.1 (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*

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

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

Theorem 5.3 (Correctness of ModIncAnalyze)**.** *Let P, P' be modular programs that differ by Δ, Q_α a set of abstract queries, and $\mathscr{A} = \text{ModIncAnalyze}(P, Q_\alpha, \emptyset, (\emptyset, \emptyset))$, then if:*

$$\{\mathcal{G}', \{\mathscr{L}'_{M_i}\}\} = \text{ModIncAnalyze}(P', Q_\alpha, \mathscr{A}, \Delta)$$

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

Lemma 5.4 (Correctness and 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 $\mathscr{L}_M = \text{LocIncAnalyzeI95}(M, E, \mathcal{G}, \emptyset, \emptyset)$. If $\mathcal{G}$ changes to $\mathcal{G}'$ the analysis result:*

$$\text{LocIncAnalyzeI95}(M, E, \mathcal{G}', \mathscr{L}_M, \emptyset) = \text{LocIncAnalyzeI95}(M, E, \mathcal{G}', \emptyset, \emptyset)$$

is the same as analyzing from scratch, i.e., the least correct analysis graph of M, E.

Theorem 5.6 (Correctness and precision of ModIncAnalyze)**.** *Let P and P' be modular programs that differ by Δ, Q_α a set of abstract queries, and $\mathscr{A} = \text{ModIncAnalyzeI95}(P, Q_\alpha, \emptyset, (\emptyset, \emptyset))$, then*

$$\text{ModIncAnalyzeI95}(P', Q_\alpha, \emptyset, (\emptyset, \emptyset)) = \text{ModIncAnalyzeI95}(P', Q_\alpha, \mathscr{A}, \Delta).$$

Fundamental results

Lemma 4.10 (Correctness of IncAnalyze starting from a correct partial analysis). *Let P be a program, Q_α be a set of abstract queries, and fix $q \in Q_\alpha$. Suppose that $\mathscr{A}_0$ is the analysis result $\mathscr{A}_0 = \textsc{IncAnalyze}(P, Q_\alpha \setminus \{q\}, \emptyset, \emptyset)$. Then the analysis result*

$\mathscr{L}_0.\mathsf{mod}(A) \in \mathsf{imports}(M)$ *correctly and precisely approximates the behavior of the imported predicates. The analysis result*

$$\mathscr{L} = \textsc{IncAnalyze95}(M, E, \emptyset, \mathscr{L}_0)$$

Contributions

The results from our incremental, modular analysis are:

- **correct over-approximations** of the AND tree semantics, and
- the **least correct analysis graph** 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).

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

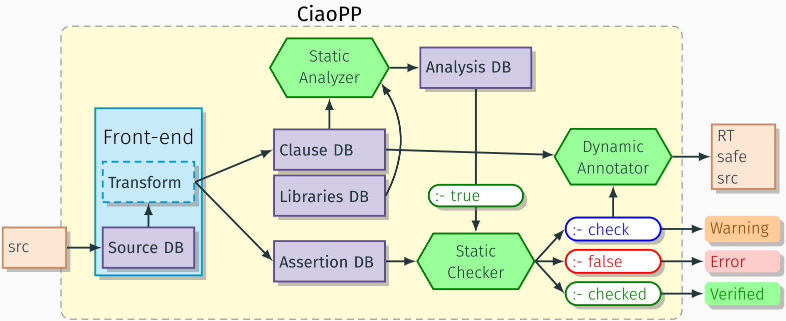
is correct for M and $\gamma(E)$ assuming $\mathscr{L}_0$.

Lemma 4.13 (Precision of 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 $\forall \langle A, \lambda^c \rangle \in$*

Theorem 5.6 (Correctness and precision of ModIncAnalyze). *Let P and P' be modular programs that differ by Δ, Q_α a set of abstract queries, and $\mathscr{A} = \textsc{ModIncAnalyzeI95}(P, Q_\alpha, \emptyset, (\emptyset, \emptyset))$, then*

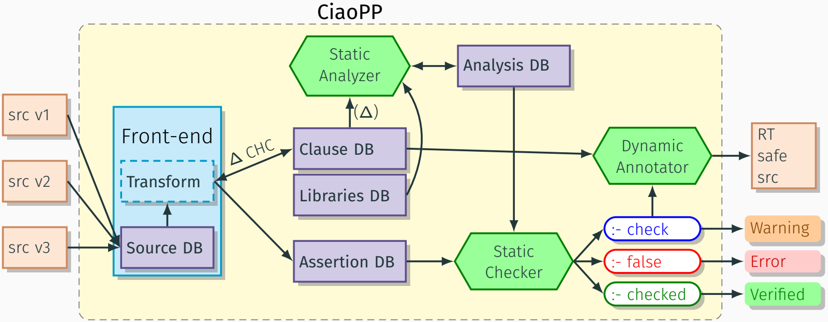
$$\textsc{ModIncAnalyzeI95}(P', Q_\alpha, \emptyset, (\emptyset, \emptyset)) = \textsc{ModIncAnalyzeI95}(P', Q_\alpha, \mathscr{A}, \Delta).$$

Implementation: the `Ciao` model and `CiaoPP` architecture

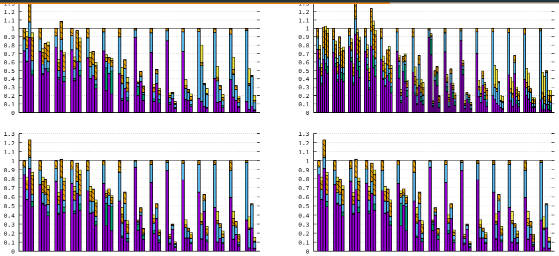


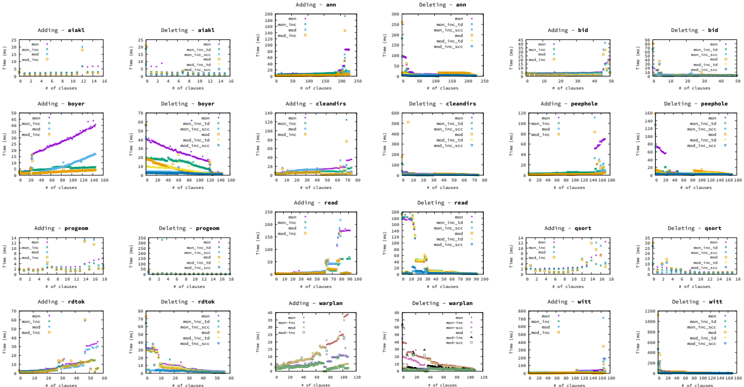
The `Ciao` model is an antecedent to the popular gradual- and hybrid-typing approaches.

Implementation: CiaoPP architecture with incrementality

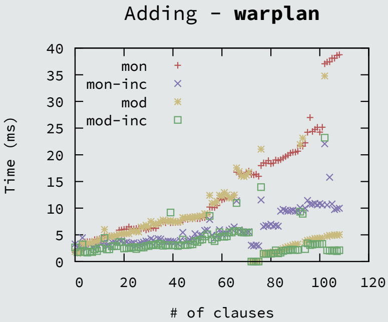


Experimental evaluation



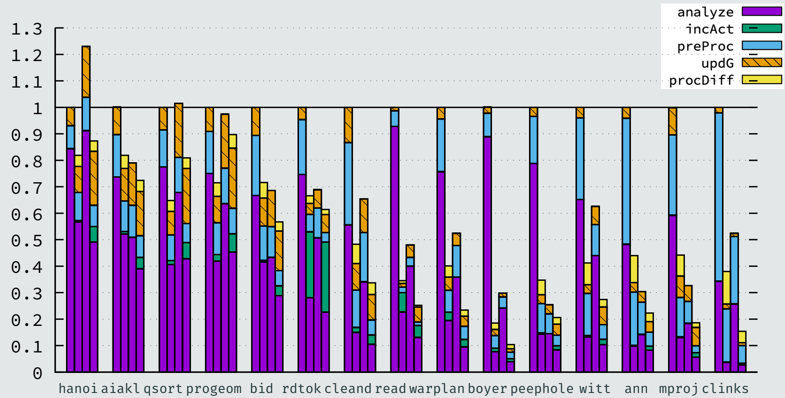


Addition experiment (time in ms) – `def` domain



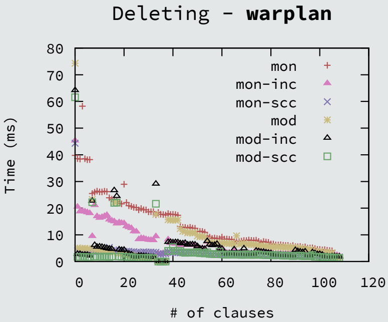
Experimental evaluation

Accumulated normalized time (def) – clause addition



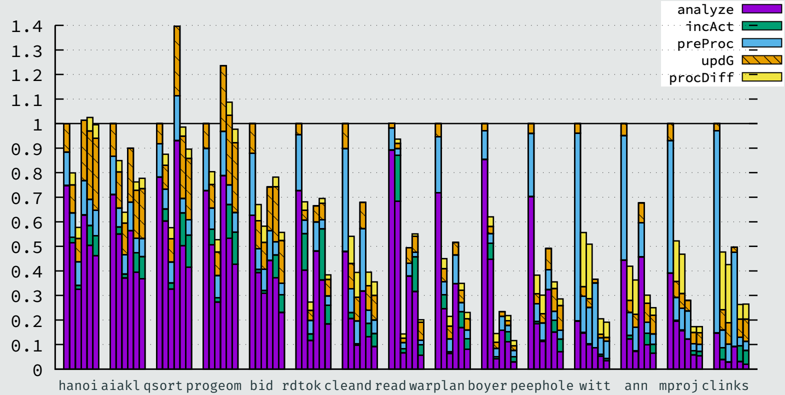
The order inside each set of bars is: |mon|mon_inc|mod|mod_inc|.

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|

Experimental evaluation

Summary

- almost **immediate** response when the changes do not affect the result,
- up to 13× overall speedup w.r.t. the original non-incremental algorithm,
- modular analysis from scratch is improved up to 9×,
- maximum **size** of analysis graphs **reduced**,
- keeping structures for incrementality produces **small overhead**.

Static *on-the-fly* verification in CiaoPP

```prolog
        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) )
```

↖ Part of the parallelizer code.

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)
```

```
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
```

Average assertion checking time (s)

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

	E1			E2			E3		
domain	*noinc*	*inc*	*speedup*	*noinc*	*inc*	*speedup*	*noinc*	*inc*	*speedup*
`pairSh`	2.8	1.6	×1.8	2.9	1.5	×1.9	2.8	1.6	×1.8
`def`	3.0	1.6	×1.9	2.7	1.5	×1.8	2.9	1.7	×1.7
`ShGrC`	18.1	5.1	×3.5	18.3	5.1	×3.6	18.1	4.5	×4.0

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)
```

```
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
```

Average assertion checking time (s) – only changing assertions

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

	E1			E2			E3		
domain	*noinc*	*inc*	*speedup*	*noinc*	*inc*	*speedup*	*noinc*	*inc*	*speedup*
`pairSh`	2.8	1.7	×1.6	2.7	1.6	×1.7	2.9	1.7	×1.7
`def`	3.1	1.5	×2.0	2.9	1.4	×2.0	3.0	1.6	×1.9
`ShGrC`	18.2	2.0	×9.1	18.1	1.9	×9.6	18.2	1.9	×9.6

Guiding the analyzer

Two problems that motivate allowing the user to guide the analyzer:

1. Automatic **approximations** may lead to **imprecise results**:
    - desired **optimizations** cannot be applied,
    - **assertions** cannot be **verified** ("false alarms").
2. Analysis may require excessive **resources** (time or space):

Techniques to optionally annotate program parts to **guide invariants inference**:

Astrée [Cousot et. al ESOP'05] uses at program point:

- *asserts* with properties that have to be verified,
- *known facts* used to refine abstract state.

CiaoPP [ESOP'96] uses assertions that can be qualified with a status:

- *check*: meaning that it needs to be verified,
- *trust*: representing knowledge that the user guarantees to be true (beliefs).

The `Ciao` assertion language

Assertions express abstractions of the behavior of programs.

[ILPSW'97, LP25Y'99]

`pred` assertions

`:- [Status] pred` *Head* `[:` *Pre*`] [=>` *Post*`].`

- *Head*: predicate that the assertion applies to,
- *Pre*: properties about how the predicate is used (hold when called),
- *Post*: properties that hold if Pre holds and the predicate succeeds,
- `Status` qualifies the meaning of assertions.

```
:- trust pred fact(N, R)         => (int(N), R > 0).
:- trust pred fact(N, R) : N > 1 => even(R).
```

Using trust assertions

Trust assertions may be used to:

- **regain precision** during analysis.

Using trust assertions

Trust assertions may be used to:

- **regain precision** during analysis.

```
% (y > 0) % Analyzing with an intervals domain (non relational)
x = y + 2;
% (x > 2, y > 0)
z = x - y;
% (int(z), x > 2, y > 0)
```

Using trust assertions

Trust assertions may be used to:

- **regain precision** during analysis.

```
% (y > 0) % Analyzing with an intervals domain (non relational)
x = y + 2;
% (x > 2, y > 0)
z = x - y;
% (int(z), x > 2, y > 0)
```

But we know `x = y + 2`.

Using trust assertions

Trust assertions may be used to:

- **regain precision** during analysis.

```
1 % (y > 0) % Analyzing with an intervals domain (non relational)
2 x = y + 2;
3 % (x > 2, y > 0)
4 z = x - y;
5 % (int(z), x > 2, y > 0)
```

But we know `x = y + 2`.

```
1 % (y > 0)
2 x = y + 2;
3 % (x > 2, y > 0)
4 z = x - y;
5 % (int(z), x > 2, y > 0)
6 trust(z == 2); % Because of line 2
7 % (z = 2, x > 2, y > 0)
```

Using trust assertions

Trust assertions may be used to:

- **regain precision** during analysis.
- **speed up computation** of analysis.

```
:- trust pred html_escape(S0, S) => (string(S0), string(S)).
html_escape( "``"||S0 , "&ldquo;"||S ) :- !, html_escape(S0, S).
html_escape( "''"||S0 , "&rdquo;"||S ) :- !, html_escape(S0, S).
html_escape( "'" ||S0 , "&quot;" ||S ) :- !, html_escape(S0, S).
html_escape( """"||S0 , "&apos;" ||S ) :- !, html_escape(S0, S).
html_escape(    [X|S0],        [X | S]) :- !,
    character_code(X),
    html_escape(S0, S).
html_escape("","").
```

Using trust assertions

Trust assertions may be used to:

- **regain precision** during analysis.
- **speed up computation** of analysis.
- define **abstract usage or specifications** of libraries or dynamic predicates.

```
:- module(sockets, []).

:- export(receive/2).
:- pred receive(S, M) : (socket(S), var(M)) => list(M, utf8).
:- impl_defined(receive/2).
% receive is written in C
```

Using trust assertions

Trust assertions may be used to:

- **regain precision** during analysis.
- **speed up computation** of analysis.
- define **abstract usage or specifications** of libraries or dynamic predicates.
- (re)define the **language semantics** for abstract domains.

```
:- trust pred '*'(A, B, C) : (int(A), int(B)) => int(C).
:- trust pred '*'(A, B, C) : (flt(A), int(B)) => flt(C).
:- trust pred '*'(A, B, C) : (int(A), flt(B)) => flt(C).
:- trust pred '*'(A, B, C) : (flt(A), flt(B)) => flt(C).
```

Guided analysis

- Precision: assertions are precisely applied during analysis:
    - the abstract success states inferred are covered by the success assertion conditions (if they exist).
    - the abstract call states inferred are covered by the call assertion conditions.
- Correctness modulo assertions: the computed analysis is correct for $P, \mathcal{Q}$ if all conditions are correct.

Reacting incrementally to assertion edits

Why? In generic code assertions play a very important role as a place holder for the code that is not available yet, e.g.:

- yet to be implemented,
- compiled in a different language, or
- linked dynamically.

The contributions are:

- an incremental fixpoint algorithm that reacts to changes in both the program and the assertions, and
- an application of this approach to the scalable analysis of generic programming (based on open predicates).

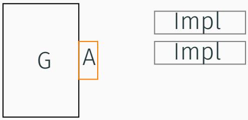
Reacting incrementally to assertion edits

Why? In generic code assertions play a very important role as a place holder for the code that is not available yet, e.g.:

- yet to be implemented,
- compiled in a different language, or
- linked dynamically.

The contributions are:

- an incremental fixpoint algorithm that reacts to changes in both the program and the assertions, and
- an application of this approach to the scalable analysis of generic programming (based on open predicates).



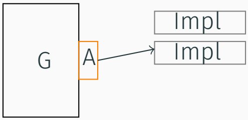
Reacting incrementally to assertion edits

Why? In generic code assertions play a very important role as a place holder for the code that is not available yet, e.g.:

- yet to be implemented,
- compiled in a different language, or
- linked dynamically.

The contributions are:

- an incremental fixpoint algorithm that reacts to changes in both the program and the assertions, and
- an application of this approach to the scalable analysis of generic programming (based on open predicates).



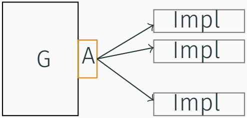
Reacting incrementally to assertion edits

Why? In generic code assertions play a very important role as a place holder for the code that is not available yet, e.g.:

- yet to be implemented,
- compiled in a different language, or
- linked dynamically.

The contributions are:

- an incremental fixpoint algorithm that reacts to changes in both the program and the assertions, and
- an application of this approach to the scalable analysis of generic programming (based on open predicates).



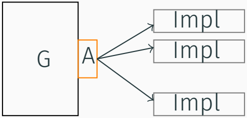
Reacting incrementally to assertion edits

Why? In generic code assertions play a very important role as a place holder for the code that is not available yet, e.g.:

- yet to be implemented,
- compiled in a different language, or
- linked dynamically.

The contributions are:

- an incremental fixpoint algorithm that reacts to changes in both the program and the assertions, and
- an application of this approach to the scalable analysis of generic programming (based on open predicates).



(And we also proposed an encoding of generic programming in (Ciao) Prolog.)

Abstract extensionality

How does **the way programs are written** (or transformed) **affect** analysis **precision**?

- **semantically equivalent** programs may exhibit **different properties**.
- **semantically different** programs may **appear identical** when analyzed.
- two classes of programs for a given program P and a given abstraction A – **completeness/incompleteness cliques** ($\mathbb{C}(P, \mathrm{A})/\overline{\mathbb{C}}(P, \mathrm{A})$):
  - $\mathbb{C}(P, \mathrm{A})$ is the class of all variants of P for which the analysis with A is precise (no false alarms)
  - $\overline{\mathbb{C}}(P, \mathrm{A})$ is the class of all variants of P for which analysis with A is imprecise.

  - automatic removal of false alarms is impossible: there is no many-to-one reducibility of $\overline{\mathbb{C}}(P, \mathrm{A})$ to $\mathbb{C}(P, \mathrm{A})$
  - we provide systematic reduction of $\mathbb{C}(P, \mathrm{A})$ into $\overline{\mathbb{C}}(P, \mathrm{A})$ (obfuscation).

Conclusions

Incremental and modular analysis [ICLP TC'18, TAPAS'19, TPLP'21]

- theoretical results of correctness,
- generalized conditions to reuse an analysis graph correctly,
- 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,
- analyzing interactively, on-the-fly is practical! [F-IDE'21, ICLP'21]

Conclusions

Guided analysis [LOPSTR'18]

- we extended the algorithm to use trusted assertions,
- we showed how safely assuming assertions affects the analysis result,
- we provided means to detect incompatible assertions.

Incremental analysis with assertions [LOPSTR'19]

- we extended the context-sensitive analysis algorithm to react incrementally to fine grain changes in (multivariant) assertions,
- we showed an application for generic code, to efficiently specialize the analysis result as implementations become available,
- we provided a syntax to build generic programs in Prolog using traits.

Conclusions

Abstract extensionality [POPL'20]

- non-trivial abstract interpretation always unveils implicitly also properties concerning the way programs are written,
- automatic **semantic** code **obfuscation**,
- systematic **removal of false alarms** is **impossible**,
- the class of all programs that are incomplete for a given non-trivial abstraction is **Turing complete**.

Future work

- Applications of incremental and modular analysis:
  - improving performance when combining analysis with other techniques (e.g., transformations),
  - semantic code search,
  - parallel fixpoint computation,
- Heuristics for automatic configuration of incrementality settings.
- Amenability of abstract domains to incrementality.
- Incrementality-aware transformations (from other source languages).

A scalable static analysis framework for reliable program development exploiting incrementality and modularity

Isabel García Contreras

Universidad Politécnica de Madrid
IMDEA Software Institute

PhD Thesis Defense
July 21st, 2021