Incremental and Modular Context-Sensitive Analysis

Isabel Garcia-Contreras\textsuperscript{1,2}  Jose F. Morales\textsuperscript{1,2}  Manuel V. Hermenegildo\textsuperscript{1,2}

\textsuperscript{1} IMDEA Software Institute
\textsuperscript{2} T. U. Madrid (UPM)

HCVS: Horn Clauses for Verification and Synthesis — March 28th, 2021
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
class rewrite(clause(H,B),clause(H,P),I,G,Info).

rewrite(clause(H,B), clause(H,P),I,G,Info) :-
  numv(B,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) :- !,
  paraPhrase(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)
).```
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   • Annotate and remove potentially outdated information.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.
Abstract Interpretation

• Simulates the execution of the program using an **abstract domain** $D_\alpha$, simpler than the concrete one.
• Guarantees:
  • Analysis termination, provided that $D_\alpha$ meets some conditions.
  • Results are **safe approximations** of the concrete semantics.

We use **Prolog** syntax for Horn Clauses

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

1. `list([]).`  % fact
2. `list([X|Xs]) :-`  % rule head
3. `list(Xs).`  % rule body
Concrete (Top-down) Semantics – AND trees

1. \text{par}([], P, P).
2. \text{par}([C|Cs], P0, P) :-
   \text{xor}(C, P0, P1),
   \text{par}(Cs, P1, P).
3. \text{xor}(0,0,0).
4. \text{xor}(0,1,1).
5. \text{xor}(1,0,1).
6. \text{xor}(1,1,0).

AND tree of \text{:- par}([0, 1], 0, P):
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.
- $\lambda^c$ is an abstract call to $A$.
- $\lambda^s$ is the abstract answer for $A$ and $\lambda^c$ if it succeeds.

Example:

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

Example nodes:

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

*For any call to \text{par} that succeeds, $P0$ and $P$ are either 1 or 0.*

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

*If \text{par} is called with $P0$ a negative number, it always fails.*

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

Analysis is interprocedural, multivariant, and context sensitive.
Entry:

\[ \text{par}(M,0,P). \]

\[ \text{par}([], P, P). \]
\[ \text{par}([C|Cs], P0, P) : - \]
\[ \quad \text{xor}(C, P0, P1), \]
\[ \quad \text{par}(Cs, P1, P). \]

\[ \text{xor}(0,0,0). \]
\[ \text{xor}(0,1,1). \]
\[ \text{xor}(1,0,1). \]
\[ \text{xor}(1,1,0). \]
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

```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).
```
We have:

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

```
% planner.pl

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

% lib.pl

%  + add(Node,Graph) :-
%   +   % IMPLEMENTATION
%   +   % IMPLEMENTATION
%   +
%   %
```
Snapshot of Analysis Graphs

delete

add

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.
Fundamental results

**Theorem 4** (Correctness of IncAnalyze starting from a partial analysis). Let $P$ be a program, $Q_\alpha$ a set of abstract queries, and $\mathcal{A}_0$ any analysis graph. Let $\mathcal{A} = \text{IncAnalyze}(P, Q_\alpha, \emptyset, \mathcal{A}_0)$. $\mathcal{A}$ 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 $q \leadsto n$ in $[P]Q$, that are abstracted in the analysis $\mathcal{A}_0$ are included in $Q_\alpha$, i.e.:

\[
\forall Q, n.Q \in \gamma(Q_\alpha) \land q \leadsto n \in [P]Q, \forall n_\alpha \in \mathcal{A}_0.n \in \gamma(n_\alpha) \Rightarrow n_\alpha \in Q_\alpha.
\]

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

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

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

\[
\mathcal{L} = \text{IncAnalyze}(M, E, \emptyset, \mathcal{L}_0)
\]

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

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

\[
\mathcal{L} = \text{IncAnalyze}(M, E, \emptyset, \mathcal{L}_0)
\]

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

**Lemma 3** (Correctness updating $\mathcal{L}$ 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 $\mathcal{L}_M$ is correct for $M$ and $\gamma(E)$ assuming $\mathcal{G}$. If $\mathcal{G}$ changes to $\mathcal{G}'$ the analysis result

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

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

**Theorem 10** (Correctness of ModIncAnalyze from scratch). Let $P$ be a modular program, and $Q_\alpha$ a set of abstract queries.

Then, if:

\[
\{\mathcal{G}, \{\mathcal{L}_M\}\} = \text{ModIncAnalyze}(P, Q_\alpha, \emptyset, \emptyset)
\]

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

**Lemma 4** (Precision updating $\mathcal{L}$ 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 $\mathcal{L}_M = \text{LocIncAnalyze}(M, E, \mathcal{G}, \emptyset, \emptyset)$. If $\mathcal{G}$ changes to $\mathcal{G}'$ the analysis result:

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

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

**Theorem 11** (Precision of ModIncAnalyze from scratch). Let $P$ be a modular program and $Q_\alpha$ a set of abstract queries.

The analysis result

\[
\mathcal{A} = \text{ModIncAnalyze}(P, Q_\alpha, \emptyset, \emptyset) = \text{ModAnalyze}(P, Q_\alpha)
\]

such that $\mathcal{A} = \{\mathcal{G}, \{\mathcal{L}_M\}\}$, then $\mathcal{G} = \mathcal{G}'$.

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

\[
\text{ModIncAnalyze}(P', Q_\alpha, \emptyset, \emptyset) = \text{ModIncAnalyze}(P', Q_\alpha, \mathcal{A}, \Delta).
\]
Theorem 4 (Correctness of IncAnalyze starting from a partial analysis). Let $P$ be a program, $Q_\alpha$ a set of abstract queries, and $\mathcal{A}_0$ any analysis graph. Let $\mathcal{A} = \text{IncAnalyze}(P,Q_\alpha,\emptyset,\emptyset)$. $\mathcal{A}$ is correct for $P$ and $\gamma(Q_\alpha)$ if $\mathcal{A}$ is a path in the concrete execution graph. The following hold:

\[ \forall (P',Q_\alpha) \text{ such that } \mathcal{A} = \mathcal{A}(P,Q_\alpha,\emptyset,\emptyset), \text{ then } \mathcal{A} \supseteq \text{IncAnalyze}(P',Q_\alpha,\emptyset,\emptyset). \]

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

\[ \text{ModIncAnalyze}(P,Q_\alpha,\emptyset,\emptyset) = \text{ModIncAnalyze}(P',Q_\alpha,\emptyset,\emptyset). \]

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).

Theorem 4 (Correctness updating $L$ modulo $G$). Let $M$ be a module of program $P$ and $E$ a set of entries. Let $G$ be a previous state of the global analysis graph, if $L_M$ is correct for $M$ and $\gamma(E)$ assuming $G$. If $G$ changes to $G'$ the analysis result

\[ \text{ModIncAnalyze}(P',Q_\alpha,\emptyset,\emptyset) = \text{ModIncAnalyze}(P',Q_\alpha,\emptyset,\emptyset). \]
Addition experiment (time in ms) – def domain

Adding - warplan

Time (ms)

# of clauses
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

Deleting - warplan

Time (ms)

# of clauses
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

Typical properties:
- For Prolog: modes, sharing, determinacy, non-failure, types, cost, ...
- + many others in other supported languages (pointer aliasing, gas, energy, ...).

Average assertion checking time (seconds)

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

<table>
<thead>
<tr>
<th></th>
<th>a-cls</th>
<th>r-cls</th>
<th>t-cls</th>
<th>a-asr</th>
<th>r-asr</th>
<th>t-asr</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-inc analysis</td>
<td>2.0</td>
<td>2.0</td>
<td>1.8</td>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>total</td>
<td>3.0</td>
<td>2.4</td>
<td>3.2</td>
<td>2.9</td>
<td>2.4</td>
<td>2.9</td>
</tr>
<tr>
<td>inc diff</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>(re)analysis</td>
<td>0.4</td>
<td>0.4</td>
<td>0.3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>1.7</td>
<td>1.6</td>
<td>1.3</td>
<td>1.3</td>
<td>1.2</td>
<td>1.2</td>
</tr>
</tbody>
</table>
## Conclusion

### To take home:

- Almost **immediate** response when the changes do not affect the result.
- Up to $13 \times$ speedup w.r.t. the original non-incremental algorithm.
- Being aware of **modular structures** is useful: Up to $2 \times$ speedup when compared with the original incremental algorithm.
- **Modular analysis** from scratch is **improved** up to $9 \times$.
- 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