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

NPM DELIVERY

OH BOY! MY PACKAGE IS HERE!

what you want!

verified library

random github project

ML-generated code
**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 interpretations.
Motivation – (incremental) static on-the-fly verification

*actual in-Emacs footage*
Constraint Logic Programs/Horn Clauses

\[ Head_k \leftarrow B_{k,1}, \ldots, B_{k,n_k} \]

We use Prolog syntax (“\( \leftarrow \)” instead of “\( \leftarrow \)”:)

1. `list([]).` % fact
2. `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 \( D_\alpha \), simpler than the concrete one. Guarantees:

- analysis termination, provided that \( D_\alpha \) 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(0,1,1).
xor(1,0,1).
xor(1,1,0).

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

Top-down AND tree of - par([0,1], 0,P).

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

[Bruynooghe JLP’91]
Concrete semantics – AND trees

1. \( \text{par}([], P, P) \).
2. \( \text{par}([C \mid Cs], P_0, P) \) :- \( \text{xor}(C, P_0, P_1) \), \( \text{par}(Cs, P_1, P) \).
3. \( \text{xor}(0,0,0) \).
4. \( \text{xor}(0,1,1) \).
5. \( \text{xor}(1,0,1) \).
6. \( \text{xor}(1,1,0) \).

Top-down AND tree of \( \text{par}([0, 1], 0, P) \).

\( \begin{align*}
\text{par}([0, 1], 0, P) \quad \text{query} \\
\text{true} \\
\text{par}([C \mid Cs], P_0, P) \quad \text{head}
\end{align*} \)

\( C = 0, \)  
\( Cs = [1], \)  
\( P_0 = 0 \)
Concrete semantics – AND trees

Top-down AND tree of \(- \text{par}([0, 1], 0, P).\)

[Bruynooghe JLP’91]
Concrete semantics – AND trees

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

Top-down AND tree of :- `par([θ, 1], θ,P).

[Bruynooghe JLP’91]
Concrete semantics – AND trees

Top-down AND tree of 

\[ \text{par}([0, 1], 0, P) \]

[Bruynooghe JLP’91]
Concrete semantics – AND trees

1. `par([], P, P).
2. `par([C|Cs], P0, P) :- ←
3. `xor(C, P0, P1),
4. `par(Cs, P1, P). ←

5. `xor(0,0,0).
6. `xor(0,1,1).
7. `xor(1,0,1).
8. `xor(1,1,0).

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

```
par([0, 1], 0, P).
par([C|Cs], P0, P) :-
  xor(C, P0, P1),
  par(Cs, P1, P).
```

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

```
C = 0, Cs = [1], P0 = 0
xor(C, P0, P1),
 xor(0, 0, 0).
```

```
C = 0, Cs = [1], P0 = 0, P1 = 0
par(Cs, P1, P).
par([C|Cs], P1, P) :-
```

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

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

```
C = 0, Cs = [1], P0 = 0
xor(C, P0, P1),
 xor(0, 0, 0).
```

```
C = 0, Cs = [1], P0 = 0, P1 = 0
par(Cs, P1, P).
par([C|Cs], P1, P) :-
```

```
C = 1, Cs = [], P0 = 0
```
Concrete semantics – AND trees

1. \( \text{par}([], P, P). \)
2. \( \text{par}([C|Cs], P_0, P) : - \)
   \( \text{xor}(C, P_0, P_1), \quad \leftarrow \)
   \( \text{par}(Cs, P_1, P). \)
3. \( \text{xor}(0,0,0). \)
4. \( \text{xor}(0,1,1). \)
5. \( \text{xor}(1,0,1). \quad \leftarrow \)
6. \( \text{xor}(1,1,0). \)

Top-down AND tree of \( : - \text{par}([0, 1], 0, P). \) [Bruynooghe JLP’91]
Concrete semantics – AND trees

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

5. \( \text{xor}(0,0,0) \).
6. \( \text{xor}(0,1,1) \).
7. \( \text{xor}(1,0,1) \) :-
8. \( \text{xor}(1,1,0) \).

Top-down AND tree of \( :- \text{par}([0, 1], 0, P) \). [Bruynooghe JLP’91]
Concrete semantics – AND trees

1. `par([], P, P). ←`
2. `par([C|Cs], P0, P) :-
   xor(C, P0, P1),
   par(Cs, P1, P). ←`
3. `xor(0, 0, 0).`
4. `xor(0, 1, 1).`
5. `xor(1, 0, 1).`
6. `xor(1, 1, 0).`

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

[Bruynooghe JLP’91]
Concrete semantics – AND trees

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

Top-down AND tree of \text{- 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

\[
\begin{align*}
1. & \quad \text{par}([], P, P). \\
2. & \quad \text{par}([\text{C}|\text{Cs}], P0, P) :\quad \leftarrow \\
3. & \quad \text{xor}(\text{C}, P0, P1), \\
4. & \quad \text{par}(\text{Cs}, P1, P).
\end{align*}
\]

Top-down AND tree of \(-\text{par}([0, 1], \theta, P)\).

[Bruynooghe JLP’91]
Abstract semantics

A PLAI analysis graph (\(\mathcal{G}\)) 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 \(A\), and
- \(\lambda^s\) is the abstract answer for \(A\) and \(\lambda^c\) if any call succeeds.

\(\lambda^c\) and \(\lambda^s\) are values of some abstract domain \(D_\alpha\).

Example

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

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

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

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

If \text{par} is called with \(P0\) a negative number, it always fails.
Building an analysis graph

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

Initial query \( \langle \text{par}(M, X, P), (X/z) \rangle \)

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

Diagram:
```
\[ \begin{array}{c}
\text{bit} \\
\text{z} \\
\text{o} \\
\text{⊥}
\end{array} \]
```

```
\langle \text{par}(M, X, P), (X/z) \rangle \mapsto \text{⊥}
```
Building an analysis graph

Initial query \( \langle \text{par}(M, X, P), (X/z) \rangle \)

\[
\begin{align*}
1 & \quad \text{par}([], P, P). \quad \leftarrow \\
2 & \quad \text{par}([C|Cs], P0, P) :- \\
3 & \quad \quad \text{xor}(C, P0, P1), \\
4 & \quad \quad \text{par}(Cs, P1, P). \\
5 & \quad \text{xor}(0,0,0). \\
6 & \quad \text{xor}(0,1,1). \\
7 & \quad \text{xor}(1,0,1). \\
8 & \quad \text{xor}(1,1,0). \\
\end{align*}
\]
Building an analysis graph

Initial query \(\langle \text{par}(M, X, P), (X/z)\rangle\)

1. \(\text{par}([], P, P).\)
2. \(\text{par}([\text{C}|\text{Cs}], P0, P) \leftarrow\)
3. \(\text{xor}(C, P0, P1),\)
   \(\text{par}(\text{Cs}, P1, P).\)
4. \(\text{par}(\text{Cs}, P1, P).\)
5. \(\text{xor}(0,0,0).\)
6. \(\text{xor}(0,1,1).\)
7. \(\text{xor}(1,0,1).\)
8. \(\text{xor}(1,1,0).\)
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 ⟨par(M, X, P), (X/z)⟩

⟨par(M, X, P), (X/z)⟩ \leftrightarrow
  ⟨X/z, P/z⟩

⟨xor(C, P0, P1), (P0/z)⟩ \leftrightarrow
  ⟨C/bit, P0/z, P1/bit⟩
Building an analysis graph

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

```
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).
```

Diagram:
```
\[ \begin{array}{c}
\langle \text{par}(M, X, P), (X/z) \rangle \\
(2,1) \quad (2,2)
\end{array} \]
```

```
\langle \text{par}(Cs, P1, P), (P1/bit) \rangle \\
\langle \text{par}(M, X, P), (X/z) \rangle \\
\langle \text{par}(Cs, P1, P), (P1/bit) \rangle \\
\langle \text{par}(M, X, P), (X/z) \rangle
```

Building an analysis graph

1. `par([], P, P). ←`
2. `par([C|Cs], P0, P) :- ← xor(C, P0, P1), par(Cs, P1, P).`
3. `xor(0,0,0). ← xor(0,1,1). ← xor(1,0,1). ← xor(1,1,0).`
4. Initial query \( \langle \text{par}(M, X, P), (X/z) \rangle \)
5. \( \langle \text{par}(M, X, P), (X/z) \rangle \leftarrow (X/z, P/z) \)
6. \( \langle \text{xor}(C, P0, P1), (P0/z) \rangle \leftarrow (C/bit, P0/z, P1/bit) \)
7. \( \langle \text{par}(Cs, P1, P), (P1/bit) \rangle \leftarrow \)
8. \( \top \)
Building an analysis graph

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

Initial query \( \langle \text{par}(M, X, P), (X/z) \rangle \)
Building an analysis graph

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

5. \texttt{xor(0,0,0).}
6. \texttt{xor(0,1,1).}
7. \texttt{xor(1,0,1).}
8. \texttt{xor(1,1,0).}

Initial query \(\langle \text{par}(M, X, P), (X/z) \rangle\)
Building an analysis graph

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

1. $\text{par}([], P, P)$.
2. $\text{par}([C|Cs], P0, P) \leftarrow$ xor($C$, $P0$, $P1$), $\text{par}(Cs, P1, P)$.
3. xor(0, 0, 0).
4. xor(0, 1, 1).
5. xor(1, 0, 1).
6. xor(1, 1, 0).
7. $\top$
Building an analysis graph

1. `par([], P, P).
2. `par([C|Cs], P0, P) :-
xor(C, P0, P1),
   `par(Cs, P1, P). ←
3. `xor(0,0,0).
4. `xor(0,1,1).
5. `xor(1,0,1).
6. `xor(1,1,0).

Initial query: `par(M, X, P), (X/z)"
Building an analysis graph

Initial query \(\langle \text{par}(M, X, P), (X/z) \rangle\)

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

5  xor(0,0,0).
6  xor(0,1,1).
7  xor(1,0,1).
8  xor(1,1,0).
```
An analysis graph $\mathcal{A}$ 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 $\llbracket P \rrbracket \mathcal{Q}$ (a set of AND trees).
Analysis correctness

An analysis graph $\mathcal{A}$ 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 $\mathcal{J}$ (a set of AND trees).
Analysis correctness

An analysis graph \( \mathcal{A} \) 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 $\mathcal{A}$ 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 $[P]_Q$ (a set of AND trees).
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

<table>
<thead>
<tr>
<th>Input</th>
<th>$\mathcal{Q}_\alpha$: initial abstract queries.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$P'$: target program (changed).</td>
</tr>
<tr>
<td></td>
<td>$\Delta$: clauses that changed from $P$ to $P'$.</td>
</tr>
<tr>
<td></td>
<td>$\mathcal{A}$: analysis results of $P$.</td>
</tr>
<tr>
<td>Output</td>
<td>a correct analysis graph for $P'$ and $\mathcal{Q}_\alpha$.</td>
</tr>
</tbody>
</table>

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.
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

```
:- 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).
```
We have:

- a **global analysis graph** $G$: call dependencies among imported/exported predicates.
- a **local analysis graph** $L_M$ per module $M$: limited to the predicates used in $M$. 

**Analysis graphs for incremental and modular analysis**
Incremental and modular analysis algorithm

Input
- $\mathcal{Q}_\alpha$: initial abstract queries.
- $P' (\{M_i\})$: target program (changed).
- $\Delta$: clauses that changed from $P$ to $P'$ (split by module).
- $\mathcal{A}(G, \{L_i\})$: analysis results of $P$.

Output
- a correct analysis graph for $P'$ and $\mathcal{Q}_\alpha$.

The algorithm:
- assumes as answer $\bot$ when a module has not been analyzed yet,
- (re)starts the analysis of modules using the dependencies in $G$,
- 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
Changes detected!

**planner.pl**

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

**lib.pl**

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

The diagrams and text indicate that changes have been detected in the code, specifically in the `planner.pl` and `lib.pl` files. The changes are highlighted with arrows and boxes indicating the affected lines of code.
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.
Lemma 4.10 (Correctness of IncAnalyze starting from a correct partial analysis). Let $P$ be a program, $Q_{\alpha}$ be a set of abstract queries, and fix $q \in Q_{\alpha}$. Suppose that $\mathcal{A}_0$ is the analysis result $\mathcal{A}_0 = \text{IncAnalyze}(P, Q_{\alpha}, \emptyset, \emptyset)$. Then the analysis result $\mathcal{A} = \text{IncAnalyze}(P, Q_{\alpha}, \emptyset, \mathcal{A}_0)$ is correct for $P$ and $\gamma(Q_{\alpha})$.

Theorem 4.11 (Correctness of IncAnalyze starting from a partial analysis). Let $P$ be a program, $Q_{\alpha}$ a set of abstract queries, and $\mathcal{A}_0$ 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 $q \sim n$ in $[P]Q$, and for all $n_{\alpha} \in \mathcal{A}_0$ such that $n \in \gamma(n_{\alpha})$ either:

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

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

Then $\mathcal{A} = \text{IncAnalyze}(P, Q_{\alpha}, \emptyset, \mathcal{A}_0)$ 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 $\mathcal{A}_0 \subseteq \mathcal{A}$, then:

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

Lemma 4.13 (Correctness of IncAnalyze modulo imported predicates). Let $M$ be a module of program $P$, $E$ a set of abstract queries. Let $\mathcal{L}_0$ 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 4.13 (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 $\forall (A, \lambda^c) \in \mathcal{L}_0.\text{mod}(A) \in \text{imports}(M)$ correctly and precisely approximates the behavior of the imported predicates. The analysis result

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

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

Lemma 5.1 (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 5.3 (Correctness of ModIncAnalyze). Let $P, P'$ be modular programs that differ by $\Delta$, $Q_{\alpha}$ a set of abstract queries, and $\mathcal{A} = \text{ModIncAnalyze}(P, Q_{\alpha}, \emptyset, (\emptyset, \emptyset))$, then:

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

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

Lemma 5.4 (Correctness and 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 $\mathcal{L}_M = \text{LocIncAnalyze95}(M, E, \mathcal{G}, (\emptyset, \emptyset))$. If $\mathcal{G}$ changes to $\mathcal{G}'$, the analysis result:

$\text{LocIncAnalyze95}(M, E, \mathcal{G}', \mathcal{L}_M, \emptyset) = \text{LocIncAnalyze95}(M, E, \mathcal{G}', \emptyset, \emptyset)$

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 $\Delta$, $Q_{\alpha}$ a set of abstract queries, and $\mathcal{A} = \text{ModIncAnalyze95}(P, Q_{\alpha}, (\emptyset, \emptyset))$, then:

$\text{ModIncAnalyze95}(P', Q_{\alpha}, (\emptyset, \emptyset)) = \text{ModIncAnalyze95}(P', Q_{\alpha}, \mathcal{A}, \Delta)$.
Lemma 4.10 (Correctness of IncAnalyze starting from a correct partial analysis). Let $P$ be a program, $Q_\alpha$ be a set of abstract queries, and fix $q \in Q_\alpha$. Suppose that $A_0$ is the analysis result $A_0 = \text{IncAnalyze}(P, Q_\alpha \setminus \{q\}, \emptyset, \emptyset)$. Then the analysis result $A = \text{IncAnalyze}(P, Q_\alpha, \emptyset, (\emptyset, \emptyset))$ is correct for $P$ and $\gamma(Q_\alpha)$. The analysis result $L = \text{IncAnalyze}_95(M, E, \emptyset, L_0)$ is the least analysis graph for $M$ and $\gamma(E)$ assuming $L_0$.\[\]

**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). \[\]

**Theorem 5.6** (Correctness and precision of ModIncAnalyze). Let $P$ and $P'$ be modular programs that differ by $\Delta$, $Q_\alpha$ a set of abstract queries, and $A = \text{ModIncAnalyze}_95(P, Q_\alpha, \emptyset, (\emptyset, \emptyset))$, then
\[
\text{ModIncAnalyze}_95(P', Q_\alpha, \emptyset, (\emptyset, \emptyset)) = \text{ModIncAnalyze}_95(P', Q_\alpha, A, \Delta).
\]
The **Ciao** model is an antecedent to the popular gradual- and hybrid-typing approaches.
Implementation: CiaoPP architecture with incrementality

```
:- check
:- false
:- checked
```

Source DB

Clause DB

Libraries DB

Assertion DB

Static Analyzer

Analysis DB

Dynamic Annotator

RT safe src

Warning

Error

Verified

Front-end

Transform

Source DB

CiaoPP

src v1

src v2

src v3

\(\Delta\)

\(\Delta\ CHC\)
Experimental evaluation
Experimental evaluation

Addition experiment (time in ms) – def domain

Adding - warplan

Time (ms)

# of clauses

0 10 20 30 40

0 5 10 15 20 25 30 35 40

Adding - read

Time (ms)

# of clauses

0 20 40 60 80

0 5 10 15 20 25 30 35 40

Adding - cleandirs

Time (ms)

# of clauses

0 20 40 60 80

0 5 10 15 20 25 30 35 40

Adding - witt

Time (ms)

# of clauses

0 20 40 60 80

0 5 10 15 20 25 30 35 40

Adding - bid

Time (ms)

# of clauses

0 100 200 300 400

0 10 20 30 40 50 60 70 80 90 100

Adding - qsort

Time (ms)

# of clauses

0 20 40 60 80

0 10 20 30 40 50 60 70 80 90

Adding - rdtok

Time (ms)

# of clauses

0 20 40 60 80

0 5 10 15 20 25 30 35 40
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

Deleting - warplan

# of clauses

Time (ms)
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 $13 \times$ overall speedup w.r.t. the original non-incremental algorithm,
- modular analysis from scratch is improved up to $9 \times$, 
- 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(Is,P,Indep,Gnd),
  add_annotations(Is,P,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],Ind,Gnd).
```

Average assertion checking time (s)

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

<table>
<thead>
<tr>
<th></th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain</td>
<td>noinc</td>
<td>inc</td>
<td>speedup</td>
</tr>
<tr>
<td>pairSh</td>
<td>2.8</td>
<td>1.6</td>
<td>×1.8</td>
</tr>
<tr>
<td>def</td>
<td>3.0</td>
<td>1.6</td>
<td>×1.9</td>
</tr>
<tr>
<td>ShGrC</td>
<td>18.1</td>
<td>5.1</td>
<td>×3.5</td>
</tr>
</tbody>
</table>
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),!.
```

Verifying assertion:

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

Average assertion checking time (s) – only changing assertions

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

<table>
<thead>
<tr>
<th>domain</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>noinc</td>
<td>inc</td>
<td>speedup</td>
</tr>
<tr>
<td>pairSh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>def</td>
<td>3.1</td>
<td>1.5</td>
<td>x2.0</td>
</tr>
<tr>
<td>ShGrC</td>
<td>18.2</td>
<td>2.0</td>
<td>x9.1</td>
</tr>
</tbody>
</table>
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.

1. :- trust pred fact(N, R) => (int(N), R > 0).
2. :- 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.

```
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)
```
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.

```
% (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 \( x = y + 2 \).

```
% (y > 0)
x = y + 2;
% (x > 2, y > 0)
z = x - y;
% (int(z), x > 2, y > 0)
trust(z == 2);  % Because of line 2
% (z = 2, x > 2, y > 0)
```
Using **trust** assertions

**Trust** assertions may be used to:

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

```prolog
:- 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("","").
```
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.

```prolog
:- 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.

```plaintext
1  :- trust pred '*'(A, B, C) : (int(A), int(B)) => int(C).
2  :- trust pred '*'(A, B, C) : (flt(A), int(B)) => flt(C).
3  :- trust pred '*'(A, B, C) : (int(A), flt(B)) => flt(C).
4  :- 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 $P, Q$ 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.)
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 $(C(P, A)/\overline{C}(P, A))$:
  - $C(P, A)$ is the class of all variants of $P$ for which the analysis with $A$ is precise (no false alarms)
  - $\overline{C}(P, A)$ 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 $\overline{C}(P, A)$ to $C(P, A)$
- we provide systematic reduction of $C(P, A)$ into $\overline{C}(P, A)$ (obfuscation).
## Conclusions

**Incremental and modular analysis**

- **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 $13 \times$ 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**,
- analyzing **interactively, on-the-fly** is practical!

---

[ICLP TC’18, TAPAS’19, TPLP’21]

[F-IDE’21, ICLP’21]
Conclusions

<table>
<thead>
<tr>
<th>Guided analysis</th>
<th>[LOPSTR’18]</th>
</tr>
</thead>
<tbody>
<tr>
<td>• we extended the algorithm to use trusted assertions,</td>
<td></td>
</tr>
<tr>
<td>• we showed how safely assuming assertions affects the analysis result,</td>
<td></td>
</tr>
<tr>
<td>• we provided means to detect incompatible assertions.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Incremental analysis with assertions</th>
<th>[LOPSTR’19]</th>
</tr>
</thead>
<tbody>
<tr>
<td>• we extended the context-sensitive analysis algorithm to react incrementally to fine grain changes in (multivariant) assertions,</td>
<td></td>
</tr>
<tr>
<td>• we showed an application for generic code, to efficiently specialize the analysis result as implementations become available,</td>
<td></td>
</tr>
<tr>
<td>• we provided a syntax to build generic programs in Prolog using traits.</td>
<td></td>
</tr>
</tbody>
</table>
Conclusions

Abstract extensionality

- 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