On-the-fly Verification
via (Incremental, Interactive) Abstract Interpretation with CiaoPP and Verifly

Manuel Hermenegildo[1,2] I. García-Contreras[1,2] J. Morales[1,2]
P. López-García[1,3] L. Rustenholz[1,2] D. Ferreiro[1,2] D. Jurjo[1,2]

LOPSTR'23 (w/SPLASH)

Cascais, Portugal – October 23-24, 2023

[1]IMDEA Software Institute
[2]T.U. of Madrid (UPM)
[3]Spanish Research Council (CSIC)

Introduction / overview

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI) [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure). [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure. [ICLP'18, LOPSTR'19, TPLP'21c]
- IDE integration → our `VeriFly` "on-the-fly" verification tool. [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP). [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLDI'90] ... [SAS'20]

Introduction / overview

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI) [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure). [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure. [ICLP'18, LOPSTR'19, TPLP'21c]
- IDE integration → our VeriFly "on-the-fly" verification tool. [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP). [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLOP'90] ... [SAS'20]

Introduction / overview

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI)
  [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure).
  [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure.
  [ICLP'18, LOPSTR'19, TPLP'21c]
- IDE integration → our `VeriFly` "on-the-fly" verification tool.
  [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP).
  [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...)
  [PLDI'90] ... [SAS'20]

Introduction / overview

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI)
  [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure).
  [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure.
  [ICLP'18, LOPSTR'19, TPLP'21c]
- IDE integration → our `VeriFly` "on-the-fly" verification tool.
  [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP).
  [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...)
  [PLDI'90] ... [SAS'20]

Introduction / overview

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the “top-down algorithm,” PLAI) [NACLP’89, MCC’90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure). [SAS’96, TOPLAS’00]
- Extending incremental analysis to exploit much better modular structure. [ICLP’18, LOPSTR’19, TPLP’21c]
- IDE integration → our `VeriFly` “on-the-fly” verification tool. [NASA-FIDE21, TPLP’21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP). [LOPSTR’07, TPLP’18, VPT’20, TPLP’21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLDI’90] ... [SAS’20]

Introduction / overview

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI) [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure). [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure. [ICLP'18, LOPSTR'19, TPLP'21c]
- IDE integration → our `VeriFly` "on-the-fly" verification tool. [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP). [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLDI'90] ... [SAS'20]

Introduction / overview

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI)
  [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure).
  [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure.
  [ICLP'18, LOPSTR'19, TPLP'21c]
- IDE integration → our `VeriFly` "on-the-fly" verification tool.
  [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP).
  [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...)
  [PLDI'90] ... [SAS'20]

Introduction / overview

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI) [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure). [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure. [ICLP'18, LOPSTR'19, TPLP'21c]
- IDE integration → our `VeriFly` "on-the-fly" verification tool. [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP). [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLDI'90] ... [SAS'20]

Introduction / overview

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI)
  [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure).
  [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure.
  [ICLP'18, LOPSTR'19, TPLP'21c]
- IDE integration → our `VeriFly` "on-the-fly" verification tool.
  [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP).
  [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLDI'90] ... [SAS'20]

Brief Introduction to the CiaoPP Framework

90's: mostly statically-typed languages: ML, Haskell — Gödel, Mercury

⇝ Developed the Ciao Prolog language, to provide:
Of course, an excellent Prolog, but, in addition:

- the flexibility / fast prototyping of dynamic languages,
- with the guarantees of static / strongly typed languages.

- Objective: bridge the gap between dynamic and static languages.
(First?) dynamic language with safety assurances.

Brief Introduction to the CiaoPP Framework

90's: mostly statically-typed languages: ML, Haskell — Gödel, Mercury

⇝ Developed the Ciao Prolog language, to provide:
Of course, an excellent Prolog, but, in addition:

- the flexibility / fast prototyping of dynamic languages,
- with the guarantees of static / strongly typed languages.

- Objective: bridge the gap between dynamic and static languages.
  (First?) dynamic language with safety assurances.

Brief Introduction to the CiaoPP Framework

90's: mostly statically-typed languages: ML, Haskell — Gödel, Mercury

⇝ Developed the Ciao Prolog language, to provide:
Of course, an excellent Prolog, but, in addition:

- the flexibility / fast prototyping of dynamic languages,
- with the guarantees of static / strongly typed languages.

- Objective: bridge the gap between dynamic and static languages.
  (First?) dynamic language with safety assurances.

Brief Introduction to the CiaoPP Framework

90's: mostly statically-typed languages: ML, Haskell — Gödel, Mercury

$\rightsquigarrow$ Developed the Ciao Prolog language, to provide:
Of course, an excellent Prolog, but, in addition:

- the flexibility / fast prototyping of dynamic languages,
- with the guarantees of static / strongly typed languages.

- Objective: bridge the gap between dynamic and static languages.
  (First?) dynamic language with safety assurances.

Brief Introduction to the CiaoPP Framework

90's: mostly statically-typed languages: ML, Haskell — Gödel, Mercury

$\rightsquigarrow$ Developed the Ciao Prolog language, to provide:
Of course, an excellent Prolog, but, in addition:

- the flexibility / fast prototyping of dynamic languages,
- with the guarantees of static / strongly typed languages.

- Objective: bridge the gap between dynamic and static languages.
  (First?) dynamic language with safety assurances.

Keys:

- Assertions rather than (traditional) types, and optional.
- Do not restrict the properties $\rightarrow$ accept undecidability.
- Use safe approximations $\rightsquigarrow$ abstract interpretation-based verification.

	Definition	Sufficient condition
P is prt. correct w.r.t. $\mathcal{I}_\alpha$ if	$\alpha([\![P]\!]) \leq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^+} \leq \mathcal{I}_\alpha$
P is complete w.r.t. $\mathcal{I}_\alpha$ if	$\mathcal{I}_\alpha \leq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \leq [\![P]\!]_{\alpha^=}$
P is incorrect w.r.t. $\mathcal{I}_\alpha$ if	$\alpha([\![P]\!]) \not\leq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^=} \not\leq \mathcal{I}_\alpha$, or $[\![P]\!]_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset \wedge [\![P]\!]_\alpha \neq \emptyset$
P is incomplete w.r.t. $\mathcal{I}_\alpha$ if	$\mathcal{I}_\alpha \not\leq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \not\leq [\![P]\!]_{\alpha^+}$

Brief Introduction to the CiaoPP Framework

90's: mostly statically-typed languages: ML, Haskell — Gödel, Mercury

$\rightsquigarrow$ Developed the Ciao Prolog language, to provide:
Of course, an excellent Prolog, but, in addition:

- the flexibility / fast prototyping of dynamic languages,
- with the guarantees of static / strongly typed languages.

- Objective: bridge the gap between dynamic and static languages.
  (First?) dynamic language with safety assurances.

Keys:

- Assertions rather than (traditional) types, and optional.
- Do not restrict the properties $\rightarrow$ accept undecidability.
- Use safe approximations $\rightsquigarrow$ abstract interpretation-based verification.

	Definition	Sufficient condition
P is prt. correct w.r.t. $\mathscr{I}_\alpha$ if	$\alpha([\![P]\!]) \leq \mathscr{I}_\alpha$	$[\![P]\!]_{\alpha^+} \leq \mathscr{I}_\alpha$
P is complete w.r.t. $\mathscr{I}_\alpha$ if	$\mathscr{I}_\alpha \leq \alpha([\![P]\!])$	$\mathscr{I}_\alpha \leq [\![P]\!]_{\alpha^=}$
P is incorrect w.r.t. $\mathscr{I}_\alpha$ if	$\alpha([\![P]\!]) \not\leq \mathscr{I}_\alpha$	$[\![P]\!]_{\alpha^=} \not\leq \mathscr{I}_\alpha$, or $[\![P]\!]_{\alpha^+} \cap \mathscr{I}_\alpha = \emptyset \wedge [\![P]\!]_\alpha \neq \emptyset$
P is incomplete w.r.t. $\mathscr{I}_\alpha$ if	$\mathscr{I}_\alpha \not\leq \alpha([\![P]\!])$	$\mathscr{I}_\alpha \not\leq [\![P]\!]_{\alpha^+}$

Brief Introduction to the CiaoPP Framework

90's: mostly statically-typed languages: ML, Haskell — Gödel, Mercury

$\rightsquigarrow$ Developed the Ciao Prolog language, to provide:
Of course, an excellent Prolog, but, in addition:

- the flexibility / fast prototyping of dynamic languages,
- with the guarantees of static / strongly typed languages.

- Objective: bridge the gap between dynamic and static languages.
  (First?) dynamic language with safety assurances.

Keys:

- Assertions rather than (traditional) types, and optional.
- Do not restrict the properties $\rightarrow$ accept undecidability.
- Use safe approximations $\rightsquigarrow$ abstract interpretation-based verification.

	Definition	Sufficient condition
P is prt. correct w.r.t. $\mathscr{I}_\alpha$ if	$\alpha([\![P]\!]) \leq \mathscr{I}_\alpha$	$[\![P]\!]_{\alpha^+} \leq \mathscr{I}_\alpha$
P is complete w.r.t. $\mathscr{I}_\alpha$ if	$\mathscr{I}_\alpha \leq \alpha([\![P]\!])$	$\mathscr{I}_\alpha \leq [\![P]\!]_{\alpha^=}$
P is incorrect w.r.t. $\mathscr{I}_\alpha$ if	$\alpha([\![P]\!]) \not\leq \mathscr{I}_\alpha$	$[\![P]\!]_{\alpha^=} \not\leq \mathscr{I}_\alpha$, or $[\![P]\!]_{\alpha^+} \cap \mathscr{I}_\alpha = \emptyset \wedge [\![P]\!]_\alpha \neq \emptyset$
P is incomplete w.r.t. $\mathscr{I}_\alpha$ if	$\mathscr{I}_\alpha \not\leq \alpha([\![P]\!])$	$\mathscr{I}_\alpha \not\leq [\![P]\!]_{\alpha^+}$

PLAI (CiaoPP's Generic AI Framework)

- Generic framework: given P (as a set of CHCs) and abstract domain(s), computes $\mathrm{lfp}(S_P^{\alpha}) = [\![P]\!]_{\alpha}$, s.t. $[\![P]\!]_{\alpha}$ safely approximates $[\![P]\!]$.

$\rightarrow$ Essentially efficient, incremental, abstract OLDT resolution algo. for CHC's. It is the original "top-down" algorithm! [NACLP'89]

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
- Characteristics:
  - **Precision:** context-/path-sensitivity (multivariance), prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, widenings, ...
  - Handles mutually recursive methods, library calls, externals, ...
  - Can be **guided** with assertions (*trust* run-time checks, external proofs, etc.)
  - **Modular** (reduced working set) and **incremental** (reuse past analyses).

[NACLP'89, MCC'90, JLP'92, POPL'94, SAS'96, TOPLAS'00, ENTCS'00, FTfJP'07, ICLP'18]

PLAI (CiaoPP's Generic AI Framework)

- Generic framework: given P (as a set of CHCs) and abstract domain(s), computes $\text{lfp}(S_P^{\alpha}) = [\![P]\!]_{\alpha}$, s.t. $[\![P]\!]_{\alpha}$ safely approximates $[\![P]\!]$.

$\rightarrow$ Essentially efficient, incremental, abstract OLDT resolution algo. for CHC's. It is the original "top-down" algorithm! [NACLP'89]

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
- Characteristics:
  - **Precision:** context-/path-sensitivity (multivariance), prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, widenings, ...
  - Handles mutually recursive methods, library calls, externals, ...
  - Can be **guided** with assertions (*trust* run-time checks, external proofs, etc.)
  - **Modular** (reduced working set) and **incremental** (reuse past analyses).

[NACLP'89, MCC'90, JLP'92, POPL'94, SAS'96, TOPLAS'00, ENTCS'00, FTfJP'07, ICLP'18]

PLAI (CiaoPP's Generic AI Framework)

- Generic framework: given P (as a set of CHCs) and abstract domain(s), computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.

$\rightarrow$ Essentially efficient, incremental, abstract OLDT resolution algo. for CHC's. It is the original "top-down" algorithm! [NACLP'89]

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
- Characteristics:
  - **Precision:** context-/path-sensitivity (multivariance), prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, widenings, ...
  - Handles mutually recursive methods, library calls, externals, ...
  - Can be **guided** with assertions (*trust* run-time checks, external proofs, etc.)
  - **Modular** (reduced working set) and **incremental** (reuse past analyses).

[NACLP'89, MCC'90, JLP'92, POPL'94, SAS'96, TOPLAS'00, ENTCS'00, FTfJP'07, ICLP'18]

PLAI (CiaoPP's Generic AI Framework)

- Generic framework: given P (as a set of CHCs) and abstract domain(s), computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.

$\rightarrow$ Essentially efficient, incremental, abstract OLDT resolution algo. for CHC's. It is the original "top-down" algorithm! [NACLP'89]

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
- Characteristics:
  - **Precision:** context-/path-sensitivity (multivariance), prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, widenings, ...
  - Handles mutually recursive methods, library calls, externals, ...
  - Can be **guided** with assertions (*trust* run-time checks, external proofs, etc.)
  - **Modular** (reduced working set) and **incremental** (reuse past analyses).

[NACLP'89, MCC'90, JLP'92, POPL'94, SAS'96, TOPLAS'00, ENTCS'00, FTfJP'07, ICLP'18]

PLAI (CiaoPP's Generic AI Framework)

- Generic framework: given P (as a set of CHCs) and abstract domain(s), computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.

$\rightarrow$ Essentially efficient, incremental, abstract OLDT resolution algo. for CHC's. It is the original "top-down" algorithm! [NACLP'89]

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
- Characteristics:
  - **Precision:** context-/path-sensitivity (multivariance), prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, widenings, ...
  - Handles mutually recursive methods, library calls, externals, ...
  - Can be **guided** with assertions (*trust* run-time checks, external proofs, etc.)
  - **Modular** (reduced working set) and **incremental** (reuse past analyses).

[NACLP'89, MCC'90, JLP'92, POPL'94, SAS'96, TOPLAS'00, ENTCS'00, FTfJP'07, ICLP'18]

PLAI (CiaoPP's Generic AI Framework)

- Generic framework: given P (as a set of CHCs) and abstract domain(s), computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.

→ Essentially efficient, incremental, abstract OLDT resolution algo. for CHC's. It is the original "top-down" algorithm! [NACLP'89]

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
- Characteristics:
  - **Precision:** context-/path-sensitivity (multivariance), prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, widenings, ...
  - Handles mutually recursive methods, library calls, externals, ...
  - Can be **guided** with assertions (*trust* run-time checks, external proofs, etc.)
  - **Modular** (reduced working set) and **incremental** (reuse past analyses).

[NACLP'89, MCC'90, JLP'92, POPL'94, SAS'96, TOPLAS'00, ENTCS'00, FTfJP'07, ICLP'18]

PLAI (CiaoPP's Generic AI Framework)

- Generic framework: given P (as a set of CHCs) and abstract domain(s), computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.

→ Essentially efficient, incremental, abstract OLDT resolution algo. for CHC's. It is the original "top-down" algorithm! [NACLP'89]

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
- Characteristics:
  - **Precision:** context-/path-sensitivity (multivariance), prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, widenings, ...
  - Handles mutually recursive methods, library calls, externals, ...
  - Can be **guided** with assertions (*trust* run-time checks, external proofs, etc.)
  - **Modular** (reduced working set) and **incremental** (reuse past analyses).

[NACLP'89, MCC'90, JLP'92, POPL'94, SAS'96, TOPLAS'00, ENTCS'00, FTfJP'07, ICLP'18]

PLAI (CiaoPP's Generic AI Framework)

- Generic framework: given P (as a set of CHCs) and abstract domain(s), computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.

$\rightarrow$ Essentially efficient, incremental, abstract OLDT resolution algo. for CHC's. It is the original "top-down" algorithm! [NACLP'89]

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
- Characteristics:
  - **Precision:** context-/path-sensitivity (multivariance), prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, widenings, ...
  - Handles mutually recursive methods, library calls, externals, ...
  - Can be **guided** with assertions (*trust* run-time checks, external proofs, etc.)
  - **Modular** (reduced working set) and **incremental** (reuse past analyses).

[NACLP'89, MCC'90, JLP'92, POPL'94, SAS'96, TOPLAS'00, ENTCS'00, FTfJP'07, ICLP'18]

PLAI (CiaoPP's Generic AI Framework)

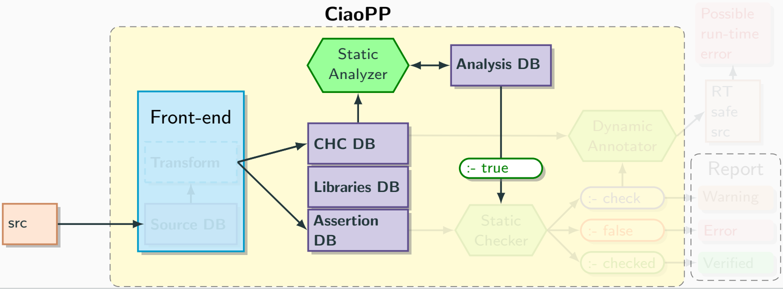
- Generic framework: given P (as a set of CHCs) and abstract domain(s), computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.

→ Essentially efficient, incremental, abstract OLDT resolution algo. for CHC's. It is the original "top-down" algorithm! [NACLP'89]

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
- Characteristics:
  - **Precision:** context-/path-sensitivity (multivariance), prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, widenings, ...
  - Handles mutually recursive methods, library calls, externals, ...
  - Can be **guided** with assertions (*trust* run-time checks, external proofs, etc.)
  - **Modular** (reduced working set) and **incremental** (reuse past analyses).

[NACLP'89, MCC'90, JLP'92, POPL'94, SAS'96, TOPLAS'00, ENTCS'00, FTfJP'07, ICLP'18]

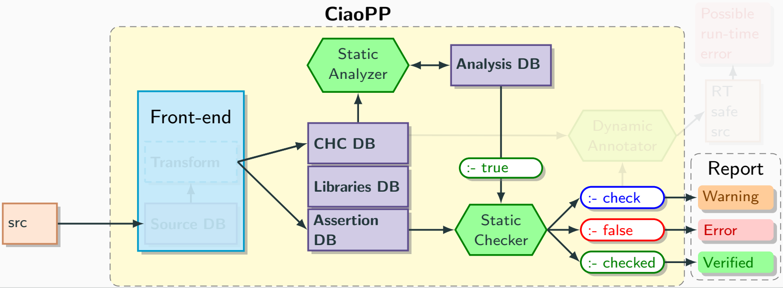
Brief Introduction to the CiaoPP Framework



Analysis

Abstract Interpretation-based, parametric on properties/domains: recursive types/shapes, pointer aliasing, constraints, determinacy, non-failure/exception, cost, sizes, termination, ...

Brief Introduction to the CiaoPP Framework



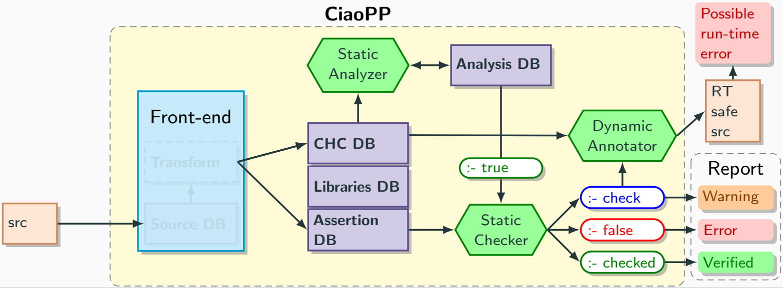
Analysis

Abstract Interpretation-based, parametric on properties/domains: recursive types/shapes, pointer aliasing, constraints, determinacy, non-failure/exception, cost, sizes, termination, ...

Verification

Compares assertions with inferred information; outcome can be verified, error, or warning (cannot verify)

Brief Introduction to the CiaoPP Framework



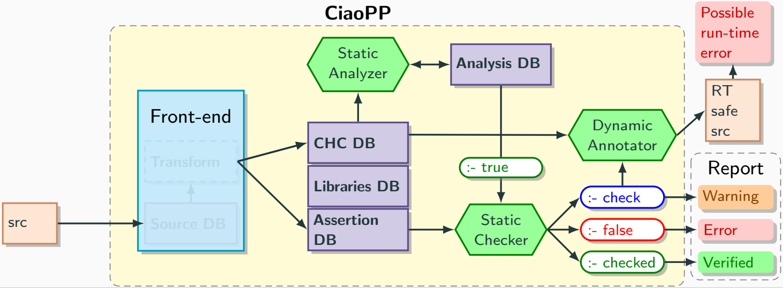
Analysis

Abstract Interpretation-based, parametric on properties/domains: recursive types/shapes, pointer aliasing, constraints, determinacy, non-failure/exception, cost, sizes, termination, ...

Verification

Compares assertions with inferred information; outcome can be verified, error, or warning (cannot verify) → run-time check.

Brief Introduction to the CiaoPP Framework



Analysis

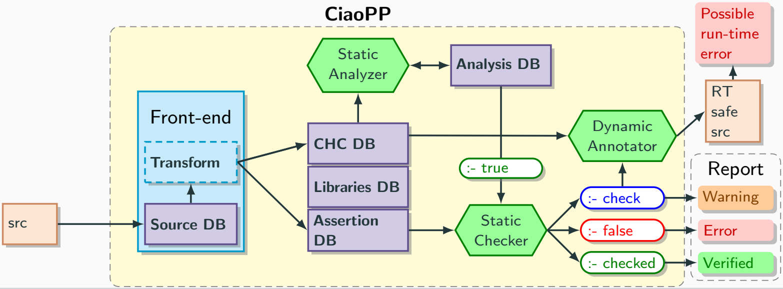
Abstract Interpretation-based, parametric on properties/domains: recursive types/shapes, pointer aliasing, constraints, determinacy, non-failure/exception, cost, sizes, termination, ...

Verification

Compares assertions with inferred information; outcome can be verified, error, or warning (cannot verify) → run-time check.

Proposed in the mid-90's: precursor of gradual- hybrid-typing approaches!

Brief Introduction to the CiaoPP Framework



Analysis

Abstract Interpretation-based, parametric on properties/domains: recursive types/shapes, pointer aliasing, constraints, determinacy, non-failure/exception, cost, sizes, termination, ...

Verification

Compares assertions with inferred information; outcome can be verified, error, or warning (cannot verify) → run-time check.

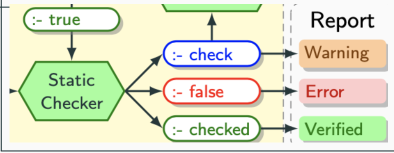
Proposed in the mid-90's: precursor of gradual- hybrid-typing approaches!

Front end

Different source languages supported, by translation to Horn clauses.

Energy Usage Verification

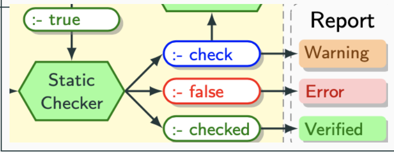
Example: XC Program (FIR Filter), w/Energy Specification [HIP3ES'15, TPLP'18]




```
#pragma check fir(xn, coeffs, state, ELEMENTS) :
          (1 <= ELEMENTS && energy <= 416.0)
#pragma true fir(xn, coeffs, state, ELEMENTS) :
          ( energy >= 3.35*ELEMENTS + 13.96 &&
            energy <= 3.35*ELEMENTS + 14.4 )
#pragma checked fir(xn, coeffs, state, ELEMENTS) :
          (1 <= ELEMENTS && ELEMENTS <= 120 && energy <= 416.1)
#pragma false fir(xn, coeffs, state, ELEMENTS) :
          (121 <= ELEMENTS && energy <= 416.1)

int fir(int xn, int coeffs[], int state[], int ELEMENTS)
{ unsigned int ynl; int ynh;
  ynl = (1<<23); ynh = 0;
  for(int j=ELEMENTS-1; j!=0; j--) {
      state[j] = state[j-1];
      {ynh, ynl} = macs(coeffs[j], state[j], ynh, ynl); }
  state[0] = xn;
  {ynh, ynl} = macs(coeffs[0], xn, ynh, ynl);
  if (sext(ynh,24) == ynh) {
      ynh = (ynh << 8) | (((unsigned) ynl) >> 24);}
  else if (ynh < 0) { ynh = 0x80000000; }
  else { ynh = 0x7fffffff; }
  return ynh; }
```

Energy Usage Verification

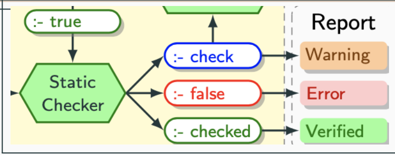
Example: XC Program (FIR Filter), w/Energy Specification [HIP3ES'15, TPLP'18]




```
#pragma check fir(xn, coeffs, state, ELEMENTS) :
          (1 <= ELEMENTS && energy <= 416.0)
#pragma true fir(xn, coeffs, state, ELEMENTS) :
          ( energy >= 3.35*ELEMENTS + 13.96 &&
            energy <= 3.35*ELEMENTS + 14.4 )
#pragma checked fir(xn, coeffs, state, ELEMENTS) :
          (1 <= ELEMENTS && ELEMENTS <= 120 && energy <= 416.1)
#pragma false fir(xn, coeffs, state, ELEMENTS) :
          (121 <= ELEMENTS && energy <= 416.1)

int fir(int xn, int coeffs[], int state[], int ELEMENTS)
{ unsigned int ynl; int ynh;
  ynl = (1<<23); ynh = 0;
  for(int j=ELEMENTS-1; j!=0; j--) {
      state[j] = state[j-1];
      {ynh, ynl} = macs(coeffs[j], state[j], ynh, ynl); }
  state[0] = xn;
  {ynh, ynl} = macs(coeffs[0], xn, ynh, ynl);
  if (sext(ynh,24) == ynh) {
      ynh = (ynh << 8) | (((unsigned) ynl) >> 24);}
  else if (ynh < 0) { ynh = 0x80000000; }
  else { ynh = 0x7fffffff; }
  return ynh; }
```

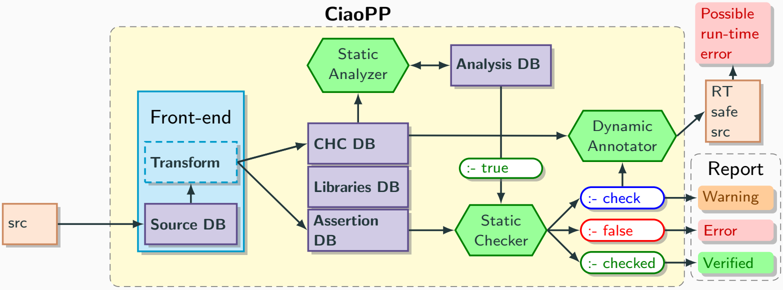
Energy Usage Verification

Example: XC Program (FIR Filter), w/Energy Specification [HIP3ES'15, TPLP'18]

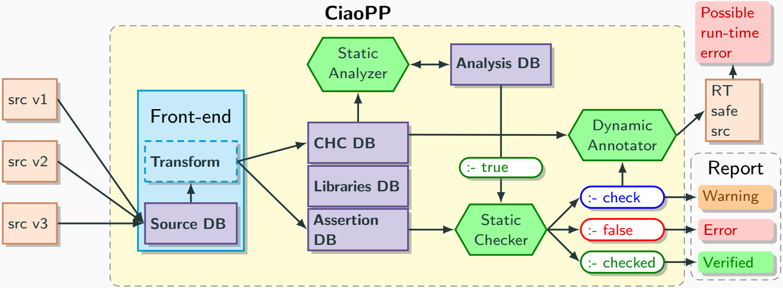



```
#pragma check fir(xn, coeffs, state, ELEMENTS) :
          (1 <= ELEMENTS && energy <= 416.0)
#pragma true fir(xn, coeffs, state, ELEMENTS) :
          ( energy >= 3.35*ELEMENTS + 13.96 &&
            energy <= 3.35*ELEMENTS + 14.4 )
#pragma checked fir(xn, coeffs, state, ELEMENTS) :
          (1 <= ELEMENTS && ELEMENTS <= 120 && energy <= 416.1)
#pragma false fir(xn, coeffs, state, ELEMENTS) :
          (121 <= ELEMENTS && energy <= 416.1)

int fir(int xn, int coeffs[], int state[], int ELEMENTS)
{ unsigned int ynl; int ynh;
  ynl = (1<<23); ynh = 0;
  for(int j=ELEMENTS-1; j!=0; j--) {
      state[j] = state[j-1];
      {ynh, ynl} = macs(coeffs[j], state[j], ynh, ynl); }
  state[0] = xn;
  {ynh, ynl} = macs(coeffs[0], xn, ynh, ynl);
  if (sext(ynh,24) == ynh) {
      ynh = (ynh << 8) | (((unsigned) ynl) >> 24);}
  else if (ynh < 0) { ynh = 0x80000000; }
  else { ynh = 0x7fffffff; }
  return ynh; }
```

Brief Introduction to the CiaoPP Framework

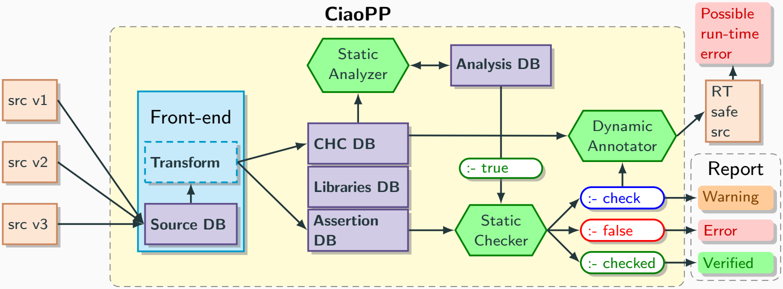


Incremental Analysis/Verification: Basic Idea



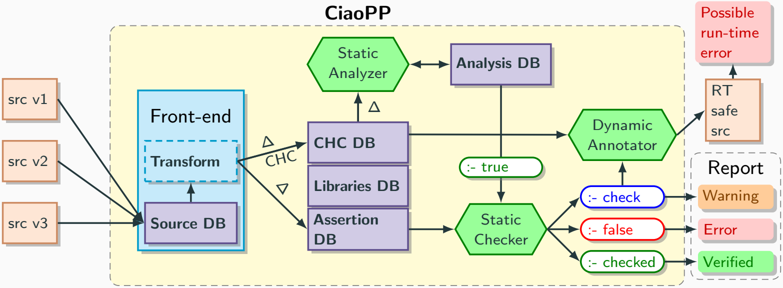
1. Take "snapshots" of the program sources
   (e.g., at each editor save/pause while developing, each commit, ...).
2. Detect the changes w.r.t. the previous snapshot.
3. Reanalyze:
   - Annotate and remove potentially
   - (Re-)Analyze (only the parts needed) module by module until an intermodular fixpoint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

Incremental Analysis/Verification: Basic Idea



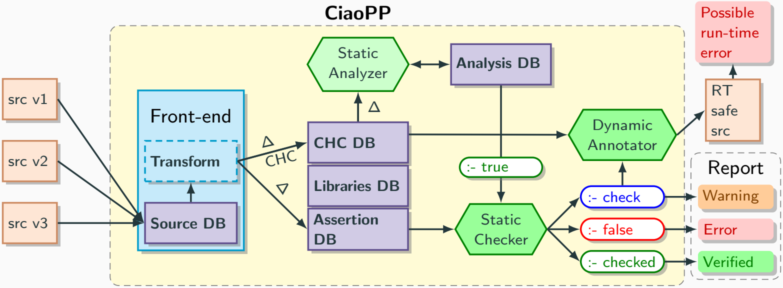
1. Take "snapshots" of the program sources
   (e.g., at each editor save/pause while developing, each commit, ...).
2. Detect the changes w.r.t. the previous snapshot.
3. Reanalyze:
   - Annotate and remove potentially outdated information.
   - (Re-)Analyze incrementally (only the parts needed) module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

Incremental Analysis/Verification: Basic Idea



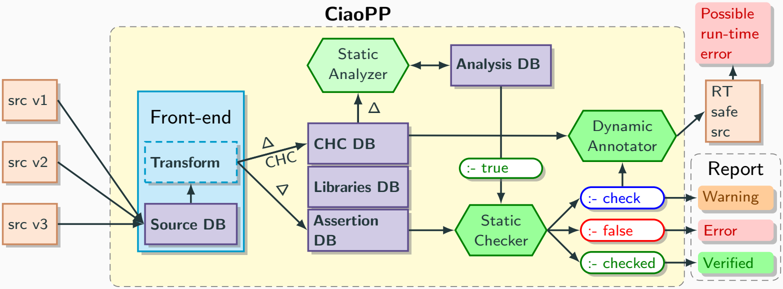
1. Take "snapshots" of the program sources
   (e.g., at each editor save/pause while developing, each commit, ...).
2. Detect the changes w.r.t. the previous snapshot.
3. Reanalyze:
   - Annotate and remove potentially outdated information.
   - (Re-)Analyze incrementally (only the parts needed) module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

Incremental Analysis/Verification: Basic Idea



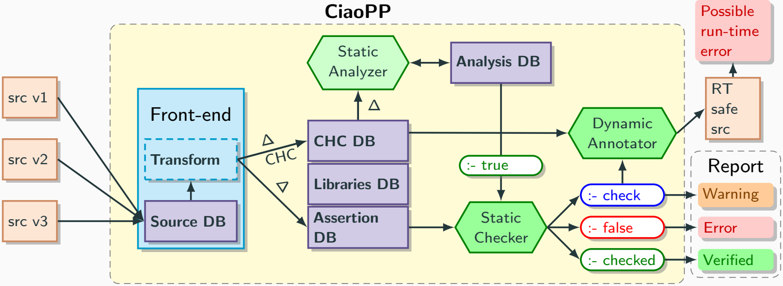
1. Take “snapshots” of the program sources (e.g., at each editor save/pause while developing, each commit, ...).
2. Detect the changes w.r.t. the previous snapshot.
3. Reanalyze:
   - Annotate and remove potentially outdated information.
   - (Re-)Analyze incrementally (only the parts needed) module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

Incremental Analysis/Verification: Basic Idea



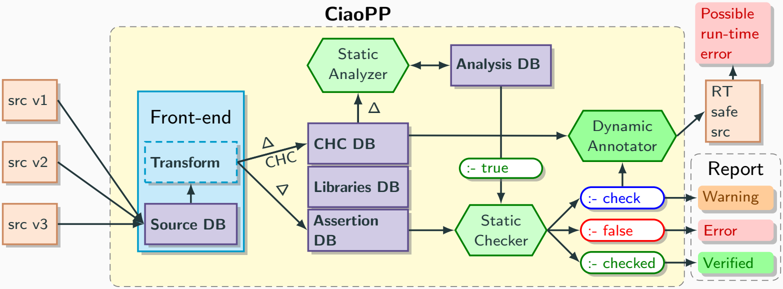
1. Take "snapshots" of the program sources
   (e.g., at each editor save/pause while developing, each commit, ...).
2. Detect the changes w.r.t. the previous snapshot.
3. Reanalyze:
   - Annotate and remove potentially outdated information.
   - (Re-)Analyze incrementally (only the parts needed) module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

Incremental Analysis/Verification: Basic Idea



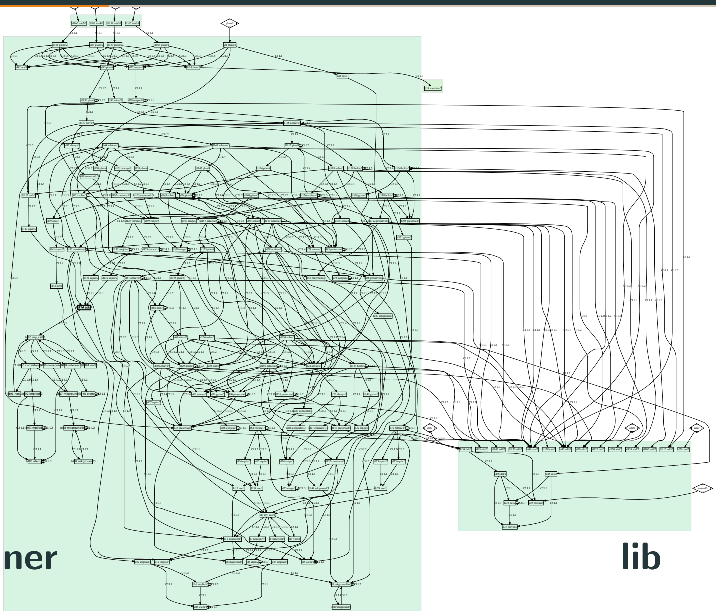
1. Take "snapshots" of the program sources
   (e.g., at each editor save/pause while developing, each commit, ...).
2. Detect the changes w.r.t. the previous snapshot.
3. Reanalyze:
   - Annotate and remove potentially outdated information.
   - (Re-)Analyze incrementally (only the parts needed) module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

Incremental Analysis/Verification: Basic Idea



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

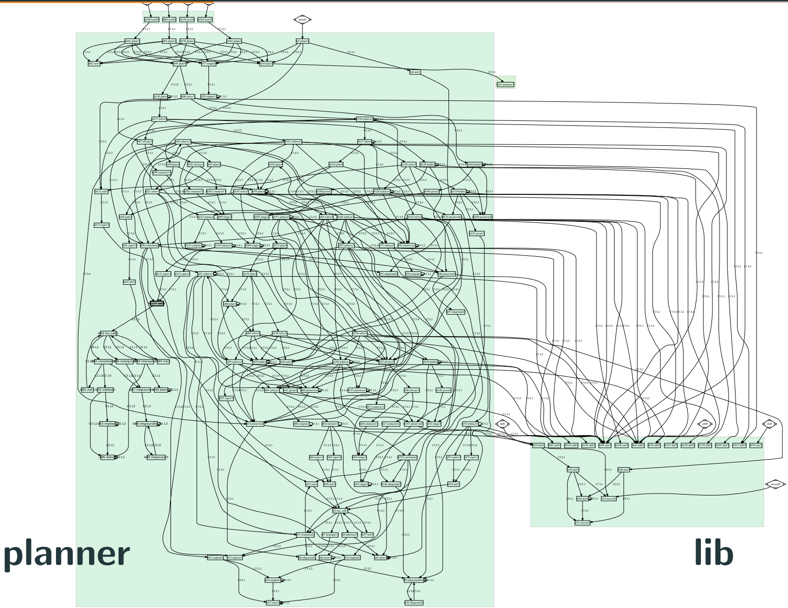
Modular and Incremental Analysis: Initial Analysis



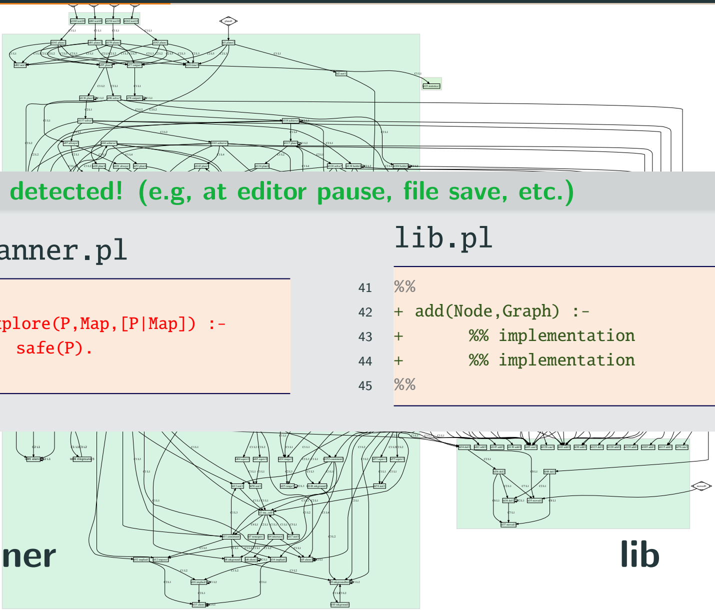
planner

lib

Modular and Incremental Analysis: Initial Analysis



Modular and Incremental: Changes Detected



Changes detected! (e.g, at editor pause, file save, etc.)

planner.pl

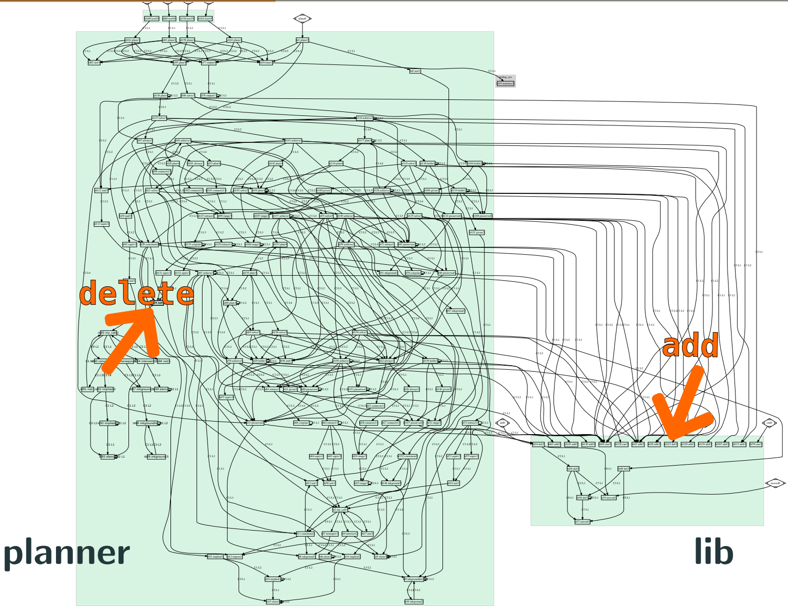
```
%%
- explore(P,Map,[P|Map]) :-
-     safe(P).
%%
```

lib.pl

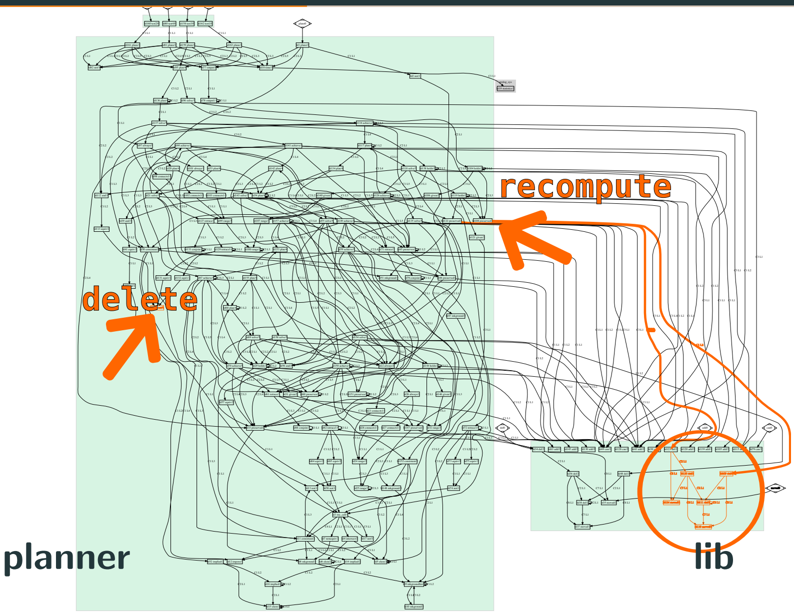
```
%%
+ add(Node,Graph) :-
+       %% implementation
+       %% implementation
%%
```

planner **lib**

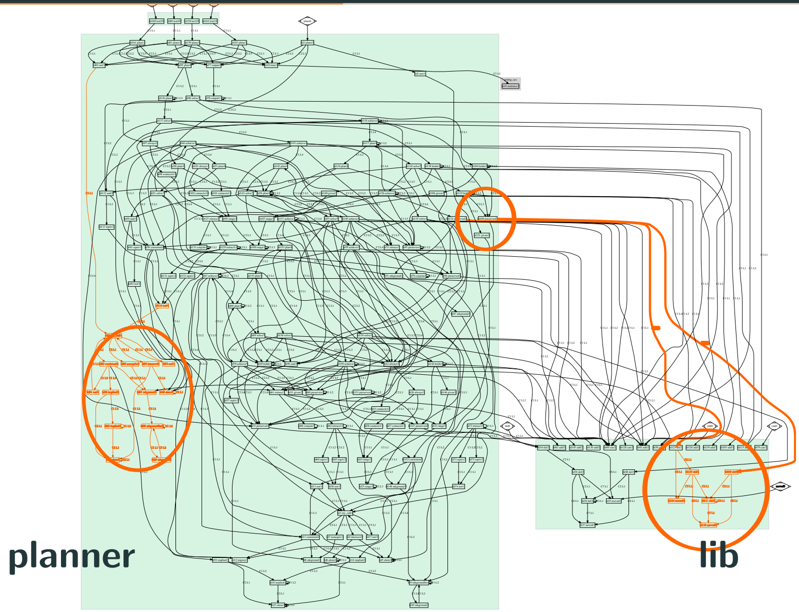
Modular and Incremental: Changes Detected



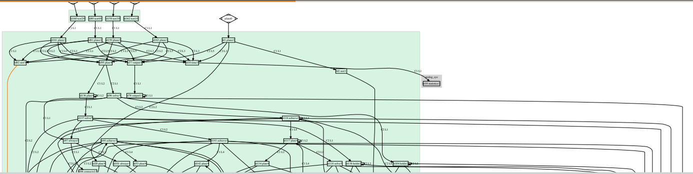
Modular and Incremental: Annotate/Remove Outdated Parts



Re-Analyze Only Parts Needed (Following Dependencies)

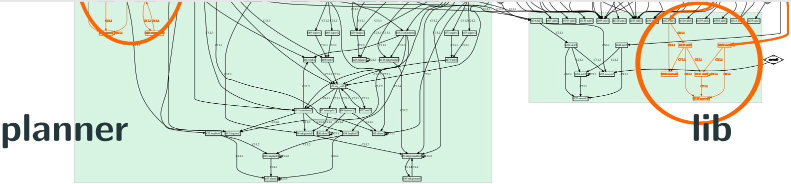


Modular and Incremental: Characteristics



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

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

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

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

Lemma 1 (Correctness of IncAnalyze modulo imported predicates)**.** *Let M be a module of program P, E a set of abstract queries. Let $\mathscr{L}_0$ 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 2 (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 $\forall \langle A, \lambda^c \rangle \in \mathscr{L}_0 . \mathsf{mod}(A) \in \mathsf{imports}(M)$ if $\mathscr{L}_0$ contains* the least fixed point *as defined in Theorem 6. The analysis result*

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

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

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

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

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_α a set of abstract queries. Then, if:*

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

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

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

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

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_α a set of abstract queries. The analysis result*

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

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

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

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

Fundamental results (very summarized)

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

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

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

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

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_α a set of abstract queries.*

Guarantees

- Correct over-approximation of the semantics (also with widening).
- But for most accurate (lfp): no widening, or conditions on the widening.

- IncAnalyze$(P, Q_\alpha, \Delta, \mathscr{A}_0)$ = IncAnalyze$(P, Q_\alpha, \emptyset, \emptyset)$.

be a previous state of the global analysis graph, if $\mathscr{L}_M$ = LocIncAnalyze$(M, E, \mathcal{G}, \emptyset, \emptyset)$. *If $\mathcal{G}$ changes to $\mathcal{G}'$ the analy-*

Additionally

- New results for reanalyzing starting from a partial analysis.

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

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

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

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

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

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

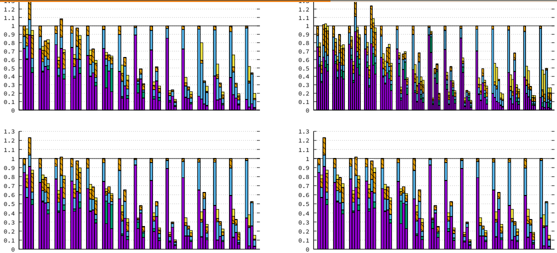
$$\mathscr{A} = \text{ModIncAnalyze}(P, Q_\alpha, \emptyset, \emptyset) = \text{ModAnalyze}(P, Q_\alpha)$$

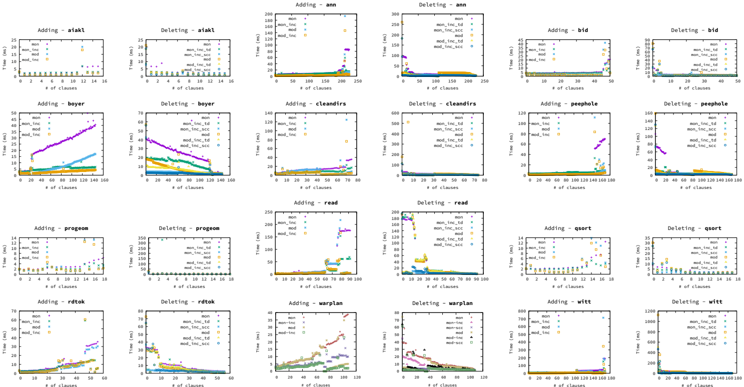
such that $\mathscr{A} = \{\mathcal{G}, \{\mathscr{L}_{M_i}\}\}$, then $\mathcal{G} = \mathcal{G}'$.

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

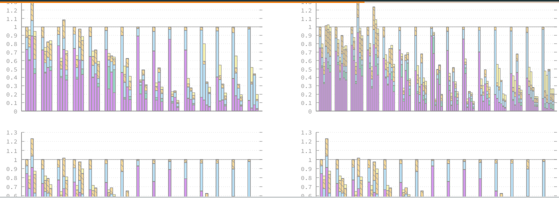
$$\text{ModIncAnalyze}(P', Q_\alpha, \emptyset, (\emptyset, \emptyset)) = \text{ModIncAnalyze}(P', Q_\alpha, \mathscr{A}, \Delta).$$

Modular and Incremental: Experimental Results



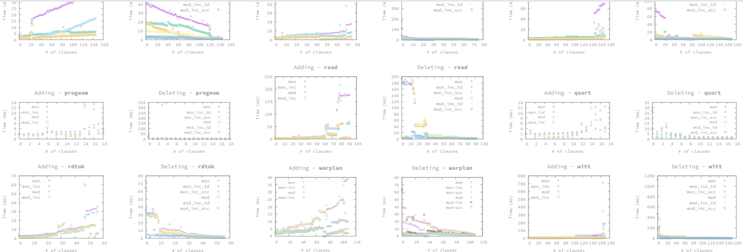


Modular and Incremental: Experimental Results

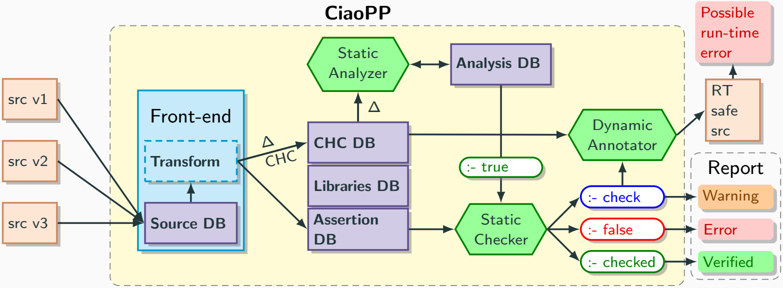


To take home:

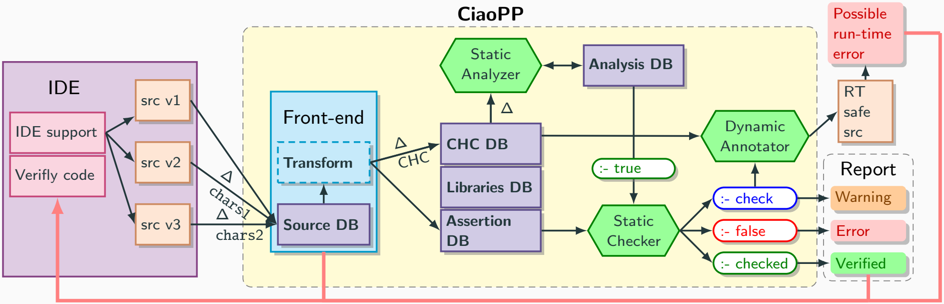
- Speedup due to incrementality in benchmarks often an order of magnitude w.r.t. non-incremental algorithm (really, unbounded).
- Modular-incremental typically $2\times$ speedup w.r.t. incremental (plus memory).
- Modular analysis from scratch also typically improved (up to $9\times$).



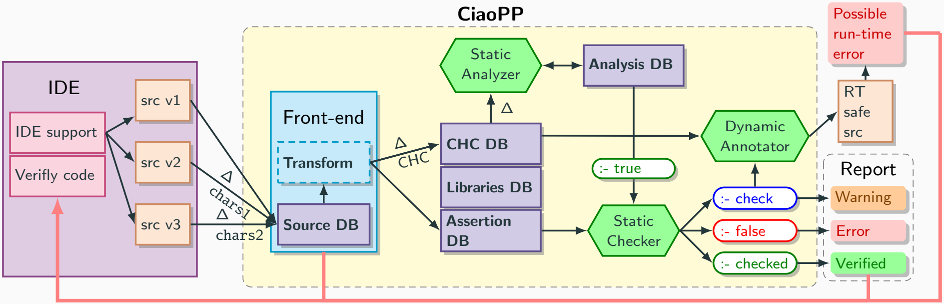
Incremental Verification



VeriFly: On-the-fly Verification/IDE Integration



VeriFly: On-the-fly Verification/IDE Integration



Tool interface components

The tool interface is implemented within the IDE, e.g.:

- In the case of Emacs, we extend `flycheck`.
- Browser version: everything runs in the browser:
  - CiaoPP runs via `ciao_wasm`
  - Verifly code and IDE are in JS (Monaco + extra code)

(Approach and results equally valid for other IDEs.)

The Assertion Language

Assertions: :- **pred** *Head* [: *Pre*] [=> *Post*] [+ *Comp*] .

```
:- pred quicksort(X,Y) : list(int) * var => sorted(Y) + (is_det, not_fails).
:- pred quicksort(X,Y) : var * list(int) => ground(X) + non_det.
```

Properties (normal predicates, but: termination, steadfastness, ...):

```
color(green).  color(blue).  color(red).          color := green | blue | red.
list([]).   list([H|T]) :- list(T).                     list := [] | [_|list].
list(_,[]). list(P,[H|T]) :- X(H),list(P,T).         list(X) := [] | [X|list].
sorted := [] | [_].        sorted([X,Y|Z]) :- X=<Y, sorted([Y|Z]).
```

Modes (are essentially "assertion macros"):

```
:- pred qs(+,-).                ↦   :- pred qs(X,Y) : (nonvar(X), var(Y)).
:- pred qs(+list,-list).}       ↦   :- pred qs(X,Y) : (list(X), var(Y)) =>
     list(Y).
```

Defined as follows:

```
:- modedef +(A) : nonvar(A)          :- modedef +(A,X) : X(A).
:- modedef -(A) : var(A).            :- modedef -(A,X) : var(A) => X(A).
```

Program-point Assertions: ..., check((int(X), X>0)), ...

Also tests, documentation, ...

The Assertion Language

Assertions: :- **pred** *Head* [: *Pre*] [=> *Post*] [+ *Comp*] .

```
:- pred quicksort(X,Y) : list(int) * var => sorted(Y) + (is_det, not_fails).
:- pred quicksort(X,Y) : var * list(int) => ground(X) + non_det.
```

Properties (normal predicates, but: termination, steadfastness, ...):

```
color(green).  color(blue).  color(red).          color := green | blue | red.
list([]).   list([H|T]) :- list(T).                   list := [] | [_|list].
list(_,[]). list(P,[H|T]) :- X(H),list(P,T).        list(X) := [] | [X|list].
sorted := [] | [_].        sorted([X,Y|Z]) :- X=<Y, sorted([Y|Z]).
```

Modes (are essentially "assertion macros"):

```
:- pred qs(+,-).                ↦    :- pred qs(X,Y) : (nonvar(X), var(Y)).
:- pred qs(+list,-list).}       ↦    :- pred qs(X,Y) : (list(X), var(Y)) =>
     list(Y).
```

Defined as follows:

```
:- modedef +(A) : nonvar(A)              :- modedef +(A,X) : X(A).
:- modedef -(A) : var(A).                :- modedef -(A,X) : var(A) => X(A).
```

Program-point Assertions: ..., check((int(X), X>0)), ...

Also tests, documentation, ...

The Assertion Language

Assertions: :- **pred** *Head* [: *Pre*] [=> *Post*] [+ *Comp*] .

```
:- pred quicksort(X,Y) : list(int) * var => sorted(Y) + (is_det, not_fails).
:- pred quicksort(X,Y) : var * list(int) => ground(X) + non_det.
```

Properties (normal predicates, but: termination, steadfastness, ...):

```
color(green).  color(blue).  color(red).            color := green | blue | red.
list([]).   list([H|T]) :- list(T).                     list := [] | [_|list].
list(_,[]). list(P,[H|T]) :- X(H),list(P,T).          list(X) := [] | [X|list].
sorted := [] | [_].        sorted([X,Y|Z]) :- X=<Y, sorted([Y|Z]).
```

Modes (are essentially "assertion macros"):

```
:- pred qs(+,-).                 ↦    :- pred qs(X,Y) : (nonvar(X), var(Y)).
:- pred qs(+list,-list).}        ↦    :- pred qs(X,Y) : (list(X), var(Y)) =>
     list(Y).
```

Defined as follows:

```
:- modedef +(A) : nonvar(A)                :- modedef +(A,X) : X(A).
:- modedef -(A) : var(A).                  :- modedef -(A,X) : var(A) => X(A).
```

Program-point Assertions: `..., check(( int(X), X>0 )), ...`

Also tests, documentation, ...

The Assertion Language

Assertions: :- **pred** *Head* [: *Pre*] [=> *Post*] [+ *Comp*] .

```
:- pred quicksort(X,Y) : list(int) * var => sorted(Y) + (is_det, not_fails).
:- pred quicksort(X,Y) : var * list(int) => ground(X) + non_det.
```

Properties (normal predicates, but: termination, steadfastness, ...):

```
color(green).  color(blue).  color(red).          color := green | blue | red.
list([]).   list([H|T]) :- list(T).                   list := [] | [_|list].
list(_,[]). list(P,[H|T]) :- X(H),list(P,T).        list(X) := [] | [X|list].
sorted := [] | [_].        sorted([X,Y|Z]) :- X=<Y, sorted([Y|Z]).
```

Modes (are essentially "assertion macros"):

```
:- pred qs(+,-).                  ↦    :- pred qs(X,Y) : (nonvar(X), var(Y)).
:- pred qs(+list,-list).}         ↦    :- pred qs(X,Y) : (list(X), var(Y)) =>
     list(Y).
```

Defined as follows:

```
:- modedef +(A) : nonvar(A)              :- modedef +(A,X) : X(A).
:- modedef -(A) : var(A).                :- modedef -(A,X) : var(A) => X(A).
```

Program-point Assertions: `..., check(( int(X), X>0 )), ...`

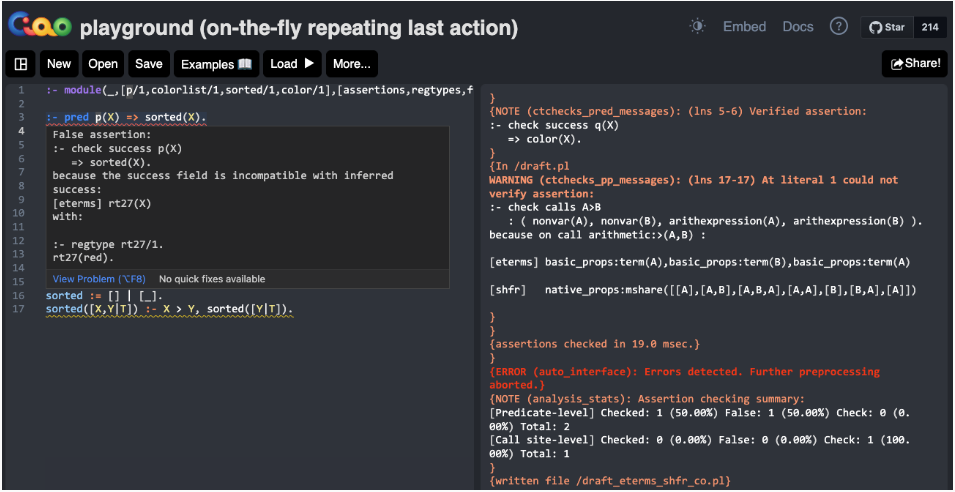
Also tests, documentation, ...

Motivation - (Incremental) Static On-the-fly Verification

```
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))
   +  not_fails.

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
    (       var(Code),!,
            Phrase = PCode
    ;       Vars = [],!,
            Phrase = Code
    ;       Phrase = (PCode,Code)
    ).


collect_info( (A;B),([],sequential,(A;B)),Cin,Cout,_X) :- !,
    collect_info(A,_Y,Cin,C,_Z),
    collect_info(B,_N,C,Cout,_M).
```

```
➤ Verified assertion:
:- check calls add_annotations(Info,Phrase,Ind,Gnd)
   : ( var(Phrase), indep(Info,Phrase) ).
➤ Verified assertion:
:- check comp add_annotations(Info,Phrase,Ind,Gnd)
   : ( var(Phrase), indep(Info,Phrase) )
   + not_fails.
➤ Verified assertion:
:- check success add_annotations(Info,Phrase,Ind,Gnd)
   : ( var(Phrase), indep(Info,Phrase) )
   => ( ground(Ind), ground(Gnd) ).
```

Interactive Verification in the Browser (Static Error Flagged)



Interactive Verification in the Browser (All Assertions Verified)

playground (on-the-fly repeating last action)

Embed Docs ? Star 214

New Open Save Examples Load ▶ More... Share!

```
1  :- module(_,[qsort/2],[assertions,nativeprops]).
2
3  %% Quick-sort with difference lists (constant time append)
4  %% Verifying various assertions
5
6  :- pred qsort(X,Y) : (ground(X),list(X),var(Y)) => ground(Y).
7  qsort(X,Y) :- qsort_(X,Y,T), T=[].
8
9  :- pred qsort_(X,Y,Z) : (list(X),var(Y),var(Z),indep(Y,Z)) => ground(X).
10 qsort_([], E, E).
11 qsort_([First|Rest],SmB,LgE) :-
12     partition(Rest,First,Sm,Lg),
13     qsort_(Sm,SmB,SmE),
14     SmE=[First|LgB],
15     qsort_(Lg,LgB,LgE).
16
17 :- pred partition(L,P,Lg,Sm)
18    => (list(Lg), list(Sm), ground(Lg), ground(Sm)).
19 partition([],_,[],[]).
20 partition([X|Y],F,[X|Y1],Y2) :-
21     X @=< F,
22     partition(Y,F,Y1,Y2).
23 partition([X|Y],F,Y1,[X|Y2]) :-
24     X @> F,
25     partition(Y,F,Y1,Y2).
26
```

```
{NOTE (ctchecks_pred_messages): (lns 9-9) Verified assertion:
:- check success qsort_(X,Y,Z)
   : ( list(X), var(Y), var(Z), indep(Y,Z) )
   => ground(X).
}
{NOTE (ctchecks_pred_messages): (lns 13-15) Verified assertion:
:- check success partition(L,P,Lg,Sm)
   => ( list(Lg), list(Sm), ground(Lg), ground(Sm) ).
}
{In /draft.pl
NOTE (ctchecks_pp_messages): (lns 17-19) At literal 1 successfully
checked assertion:
:- check calls B@=<A.

}
{In /draft.pl
NOTE (ctchecks_pp_messages): (lns 20-22) At literal 1 successfully
checked assertion:
:- check calls B@>A.

}
}
{assertions checked in 32.0 msec.}
}
{NOTE (analysis_stats): Assertion checking summary:
[Predicate-level] Checked: 4 (100.00%) False: 0 (0.00%) Check: 0 (0.
00%) Total: 4
[Call site-level] Checked: 0 (--) False: 0 (--) Check: 0 (--)
Total: 0
}

yes
?-
```

Embedding the Analyzer for Teaching Abstract Interpretation

Exercise 8 (Making predicates deterministic). *Modify the predicate to make it deterministic:*

```
:- pred sorted_insert(A,B,C) : (list_pair(A), num_pair(B), var(C)) => list_pair1(C).

sorted_insert([], X, [X]).
sorted_insert([(X1,F1)|L1], (X,F), [(X,F), (X1,F1)|L1]) :- X =< X1.
sorted_insert([P|L1], X, [P|L]) :- sorted_insert(L1, X, L).
```

the output includes the following assertions:

```
%% %% :- check pred sorted_insert(A,B,C)
%% %%    : ( list_pair(A), num_pair(B), var(C) )
%% %%    => list_pair1(C).

:- checked calls sorted_insert(A,B,C)
   : ( list_pair(A), num_pair(B), var(C) ).

:- checked success sorted_insert(A,B,C)
   : ( list_pair(A), num_pair(B), var(C) )
   => list_pair1(C).
```

Thus, we can see that the analyzer does verify the assertion that we had included. However, we can also see these other assertions:

```
:- true pred sorted_insert(A,B,C)
   : ( mshare([[C]]),
       var(C), ground([A,B]), list_pair(A), num_pair(B), term(C) )
   => ( ground([A,B,C]), list_pair(A), num_pair(B), list_pair1(C) )
   + ( multi, covered, possibly_not_mut_exclusive ).

:- true pred sorted_insert(A,B,C)
   : ( mshare([[C]]),
       var(C), ground([A,B]), list_pair(A), num_pair(B), term(C) )
   => ( ground([A,B,C]), list_pair(A), num_pair(B), list_pair1(C) )
   + ( multi, covered, possibly_not_mut_exclusive ).
```

Summary

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI) [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure). [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure. [ICLP'18, LOPSTR'19, TPLP'21b]
- IDE integration → our `VeriFly` "on-the-fly" verification tool. [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP). [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLDI'90] ... [SAS'20]

Plus of course making Ciao Prolog even better!

Summary

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI) [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure). [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure. [ICLP'18, LOPSTR'19, TPLP'21a]
- IDE integration → our `VeriFly` "on-the-fly" verification tool. [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP). [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLDI'90] ... [SAS'20]

Plus of course making Ciao Prolog even better!

Summary

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI) [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure). [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure. [ICLP'18, LOPSTR'19, TPLP'21c]
- IDE integration → our `VeriFly` "on-the-fly" verification tool. [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP). [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLDI'90] ... [SAS'20]

Plus of course making Ciao Prolog even better!

Summary

- **Objective:** Analyze/Verify software projects interactively, during development:
    - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI)
  [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure).
  [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure.
  [ICLP'18, LOPSTR'19, TPLP'21c]
- IDE integration → our `VeriFly` "on-the-fly" verification tool. [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP).
  [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLDI'90] ... [SAS'20]

Plus of course making Ciao Prolog even better!

Summary

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI) [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure). [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure. [ICLP'18, LOPSTR'19, TPLP'21c]
- IDE integration → our `VeriFly` "on-the-fly" verification tool. [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP). [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLDI'90] ... [SAS'20]

Plus of course making Ciao Prolog even better!

Summary

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI) [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure). [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure. [ICLP'18, LOPSTR'19, TPLP'21c]
- IDE integration → our `VeriFly` "on-the-fly" verification tool. [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP). [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLDI'90] ... [SAS'20]

Plus of course making Ciao Prolog even better!

Summary

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the “top-down algorithm,” PLAI) [NACLP’89, MCC’90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure). [SAS’96, TOPLAS’00]
- Extending incremental analysis to exploit much better modular structure. [ICLP’18, LOPSTR’19, TPLP’21c]
- IDE integration → our `VeriFly` “on-the-fly” verification tool. [NASA-FIDE21, TPLP’21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP). [LOPSTR’07, TPLP’18, VPT’20, TPLP’21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLDI’90] ... [SAS’20]

Plus of course making Ciao Prolog even better!

Summary

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI) [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure). [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure. [ICLP'18, LOPSTR'19, TPLP'21c]
- IDE integration → our `VeriFly` "on-the-fly" verification tool. [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP). [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLDI'90] ... [SAS'20]

Plus of course making Ciao Prolog even better!

Summary

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI) [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure). [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure. [ICLP'18, LOPSTR'19, TPLP'21c]
- IDE integration → our `VeriFly` "on-the-fly" verification tool. [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP). [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLDI'90] ... [SAS'20]

Plus of course making Ciao Prolog even better!

Summary

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI) [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure). [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure. [ICLP'18, LOPSTR'19, TPLP'21c]
- IDE integration → our `VeriFly` "on-the-fly" verification tool. [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP). [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLDI'90] ... [SAS'20]

Plus of course making Ciao Prolog even better!

Summary

- **Objective:** Analyze/Verify software projects interactively, during development:
  - Detect bugs, verify assertions on-the-fly, in the editor (also at commit, etc.).
- **Problem:** Precision (e.g., context-sensitivity, complex domains, ...) can be expensive.

In our tool (CiaoPP) we address this challenge through:

- Efficient, context/path-sensitive fixpoint (the "top-down algorithm," PLAI) [NACLP'89, MCC'90]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure). [SAS'96, TOPLAS'00]
- Extending incremental analysis to exploit much better modular structure. [ICLP'18, LOPSTR'19, TPLP'21c]
- IDE integration → our `VeriFly` "on-the-fly" verification tool. [NASA-FIDE21, TPLP'21b]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP). [LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...) [PLDI'90] ... [SAS'20]

Plus of course making Ciao Prolog even better!

CiaoPP Team


Pedro López-García


José-F. Morales


Manuel Carro


Manuel Hermenegildo


Louis Rustenholz


Daniel Jurjo


Daniela Ferreiro

IMDEA Software Institute, T.U. Madrid (UPM).
And previously at: U.T. Austin, MCC, U. of New Mexico.

- Other main contributors to Ciao and CiaoPP (incomplete):

Isabel García-Contreras	Maximiliano Klemen	Víctor Pérez	Ignacio de Casso
Jorge Navas	John Gallagher	Nai-wei Lin	Alejandro Serrano
Mario Méndez-Lojo	Germán Puebla	Francisco Bueno	María G. de la Banda
Claudio Vaucheret	Saumya Debray	Jesús Correas	Elvira Albert
Edison Mera	Pawel Pietrzak	Claudio Ochoa	Peter Stuckey
Umer Liqat	Nataliia Stulova	José M. Gómez	Kalyan Muthukumar
Amadeo Casas	Daniel Cabeza	Pablo Chico	Samir Genaim
Remy Haemmerlé	David Trallero	Gopal Gupta	Michael Codish
Christian Holzbaur	Kim Marriott	Enrico Pontelli	Ángel Pineda

- Playground at: `http://ciao-lang.org/playground`

Milestones / tools

'88	**MA3 analyzer**: memo tables (cf. OLDT resolution), practicality. **Abstract compilation**.
'89	**PLAI analyzer**: **the "top-down" algorithm**, "summaries", abstract domains as plugins. Sharing(aliasing)/independence → auto-parallelization / real speedups (on shmem).
90's	**Incremental analysis**, concurrency, automatic domain combinations, scalability.
'90-'93	**Automatic cost analysis** (upper bounds), GraCoS (**Granularity Control** System).
'96	**The Ciao/CiaoPP model for Assertion Verification by Abstract Interpretation**
'91-'06	Combined **abstract interpretation and (abstract) partial evaluation**.
'96-'00	**Lower bounds** cost analysis, **divide-and-conquer**. No-fail (no exceptions), determinacy.
'01-05	**Modular analysis** for scalability. New (imperative) shape/type domains, widenings.
'03	**Abstraction carrying code**.
'00-'05	**Multi-language support via CLP (a.k.a. CHCs) as IR**: Java, C# (shapes, resources, ...).
'04-'07	Verification/debugging/optimization of **user-defined resources**.
'06-'08	Verification of **execution time**, **energy** for Java, heap models, ... Probabilistic cost.
'12-18	(X)C **binary program energy analysis/verification**, ISA-level energy models.
'13-18	**Cost analysis as Abstract Interpretation** Sized shapes. LLVM. **Static Profiling**.
'16-18	**Sematic code search** via abstract interpretation. **Abstract distances.**
'17-19	**Combined modular/incremental analysis (scalability)**, fixpoint guidance.
'19-'23	**Verify: Interactive Verification via Abstract Interpretation**.
'19-'23	**Smart contracts**: verification of resources (gas, storage, commands, ...).

Milestones / tools

'88	**MA3 analyzer**: memo tables (cf. OLDT resolution), practicality. **Abstract compilation**.
'89	**PLAI analyzer**: **the "top-down" algorithm**, "summaries", abstract domains as plugins. Sharing(aliasing)/independence → auto-parallelization / real speedups (on shmem).
90's	**Incremental analysis**, concurrency, automatic domain combinations, scalability.
'90-'93	**Automatic cost analysis** (upper bounds), GraCoS (**Granularity Control** System).
'96	**The Ciao/CiaoPP model for Assertion Verification by Abstract Interpretation**
'91-'06	Combined **abstract interpretation and (abstract) partial evaluation**.
'96-'00	**Lower bounds** cost analysis, **divide-and-conquer**. No-fail (no exceptions), determinacy.
'01-05	**Modular analysis** for scalability. New (imperative) shape/type domains, widenings.
'03	**Abstraction carrying code**.
'00-'05	**Multi-language support via CLP (a.k.a. CHCs) as IR**: Java, C# (shapes, resources, ...).
'04-'07	Verification/debugging/optimization of **user-defined resources**.
'06-'08	Verification of **execution time**, **energy** for Java, heap models, ... Probabilistic cost.
'12-18	(X)C **binary program energy analysis/verification**, ISA-level energy models.
'13-18	**Cost analysis as Abstract Interpretation** Sized shapes. LLVM. **Static Profiling**.
'16-18	**Sematic code search** via abstract interpretation. **Abstract distances.**
'17-19	**Combined modular/incremental analysis (scalability)**, fixpoint guidance.
'19-'23	**Verify: Interactive Verification via Abstract Interpretation**.
'19-'23	**Smart contracts**: verification of resources (gas, storage, commands, ...).

Milestones / tools

'88	**MA3 analyzer**: memo tables (cf. OLDT resolution), practicality. **Abstract compilation**.
'89	**PLAI analyzer**: **the "top-down" algorithm**, "summaries", abstract domains as plugins. Sharing(aliasing)/independence → auto-parallelization / real speedups (on shmem).
90's	**Incremental analysis**, concurrency, automatic domain combinations, scalability.
'90-'93	**Automatic cost analysis** (upper bounds), GraCoS (**Granularity Control** System).
'96	**The Ciao/CiaoPP model for Assertion Verification by Abstract Interpretation**
'91-'06	Combined **abstract interpretation and (abstract) partial evaluation**.
'96-'00	**Lower bounds** cost analysis, **divide-and-conquer**. No-fail (no exceptions), determinacy.
'01-05	**Modular analysis** for scalability. New (imperative) shape/type domains, widenings.
'03	**Abstraction carrying code**.
'00-'05	**Multi-language support via CLP (a.k.a. CHCs) as IR**: Java, C# (shapes, resources, ...).
'04-'07	Verification/debugging/optimization of **user-defined resources**.
'06-'08	Verification of **execution time**, **energy** for Java, heap models, ... Probabilistic cost.
'12-18	(X)C **binary program energy analysis/verification**, ISA-level energy models.
'13-18	**Cost analysis as Abstract Interpretation** Sized shapes. LLVM. **Static Profiling**.
'16-18	**Sematic code search** via abstract interpretation. **Abstract distances.**
'17-19	**Combined modular/incremental analysis (scalability)**, fixpoint guidance.
'19-'23	**Verify: Interactive Verification via Abstract Interpretation**.
'19-'23	**Smart contracts**: verification of resources (gas, storage, commands, ...).

Milestones / tools

'88	**MA3 analyzer**: memo tables (cf. OLDT resolution), practicality. **Abstract compilation**.
'89	**PLAI analyzer**: **the "top-down" algorithm**, "summaries", abstract domains as plugins. Sharing(aliasing)/independence → auto-parallelization / real speedups (on shmem).
90's	**Incremental analysis**, concurrency, automatic domain combinations, scalability.
'90-'93	**Automatic cost analysis** (upper bounds), GraCoS (**Granularity Control** System).
'96	**The Ciao/CiaoPP model for Assertion Verification by Abstract Interpretation**
'91-'06	Combined **abstract interpretation and (abstract) partial evaluation**.
'96-'00	**Lower bounds** cost analysis, **divide-and-conquer**. No-fail (no exceptions), determinacy.
'01-05	**Modular analysis** for scalability. New (imperative) shape/type domains, widenings.
'03	**Abstraction carrying code**.
'00-'05	**Multi-language support via CLP (a.k.a. CHCs) as IR**: Java, C# (shapes, resources, ...).
'04-'07	Verification/debugging/optimization of **user-defined resources**.
'06-'08	Verification of **execution time**, **energy** for Java, heap models, ... Probabilistic cost.
'12-18	(X)C **binary program energy analysis/verification**, ISA-level energy models.
'13-18	**Cost analysis as Abstract Interpretation** Sized shapes. LLVM. **Static Profiling**.
'16-18	**Sematic code search** via abstract interpretation. **Abstract distances.**
'17-19	**Combined modular/incremental analysis (scalability)**, fixpoint guidance.
'19-'23	**Verify: Interactive Verification via Abstract Interpretation**.
'19-'23	**Smart contracts**: verification of resources (gas, storage, commands, ...).

Milestones / tools

'88	**MA3 analyzer**: memo tables (cf. OLDT resolution), practicality. **Abstract compilation**.
'89	**PLAI analyzer**: **the "top-down" algorithm**, "summaries", abstract domains as plugins. Sharing(aliasing)/independence → auto-parallelization / real speedups (on shmem).
90's	**Incremental analysis**, concurrency, automatic domain combinations, scalability.
'90-'93	**Automatic cost analysis** (upper bounds), GraCoS (**Granularity Control** System).
'96	**The Ciao/CiaoPP model for Assertion Verification by Abstract Interpretation**
'91-'06	Combined **abstract interpretation and (abstract) partial evaluation**.
'96-'00	**Lower bounds** cost analysis, **divide-and-conquer**. No-fail (no exceptions), determinacy.
'01-05	**Modular analysis** for scalability. New (imperative) shape/type domains, widenings.
'03	**Abstraction carrying code**.
'00-'05	**Multi-language support via CLP (a.k.a. CHCs) as IR**: Java, C# (shapes, resources, ...).
'04-'07	Verification/debugging/optimization of **user-defined resources**.
'06-'08	Verification of **execution time**, **energy** for Java, heap models, ... Probabilistic cost.
'12-18	(X)C **binary program energy analysis/verification**, ISA-level energy models.
'13-18	**Cost analysis as Abstract Interpretation** Sized shapes. LLVM. **Static Profiling**.
'16-18	**Sematic code search** via abstract interpretation. **Abstract distances.**
'17-19	**Combined modular/incremental analysis (scalability)**, fixpoint guidance.
'19-'23	**Verify: Interactive Verification via Abstract Interpretation**.
'19-'23	**Smart contracts**: verification of resources (gas, storage, commands, ...).

Milestones / tools

'88	**MA3 analyzer**: memo tables (cf. OLDT resolution), practicality. **Abstract compilation**.
'89	**PLAI analyzer**: **the "top-down" algorithm**, "summaries", abstract domains as plugins. Sharing(aliasing)/independence → auto-parallelization / real speedups (on shmem).
90's	**Incremental analysis**, concurrency, automatic domain combinations, scalability.
'90-'93	**Automatic cost analysis** (upper bounds), GraCoS (**Granularity Control** System).
'96	**The Ciao/CiaoPP model for Assertion Verification by Abstract Interpretation**
'91-'06	Combined **abstract interpretation and (abstract) partial evaluation**.
'96-'00	**Lower bounds** cost analysis, **divide-and-conquer**. No-fail (no exceptions), determinacy.
'01-05	**Modular analysis** for scalability. New (imperative) shape/type domains, widenings.
'03	**Abstraction carrying code**.
'00-'05	**Multi-language support via CLP (a.k.a. CHCs) as IR**: Java, C# (shapes, resources, ...).
'04-'07	Verification/debugging/optimization of **user-defined resources**.
'06-'08	Verification of **execution time**, **energy** for Java, heap models, ... Probabilistic cost.
'12-18	(X)C **binary program energy analysis/verification**, ISA-level energy models.
'13-18	**Cost analysis as Abstract Interpretation** Sized shapes. LLVM. **Static Profiling**.
'16-18	**Sematic code search** via abstract interpretation. **Abstract distances.**
'17-19	**Combined modular/incremental analysis (scalability)**, fixpoint guidance.
'19-'23	**Verify: Interactive Verification via Abstract Interpretation**.
'19-'23	**Smart contracts**: verification of resources (gas, storage, commands, ...).

Milestones / tools

'88	**MA3 analyzer**: memo tables (cf. OLDT resolution), practicality. **Abstract compilation**.
'89	**PLAI analyzer**: **the "top-down" algorithm**, "summaries", abstract domains as plugins. Sharing(aliasing)/independence → auto-parallelization / real speedups (on shmem).
90's	**Incremental analysis**, concurrency, automatic domain combinations, scalability.
'90-'93	**Automatic cost analysis** (upper bounds), GraCoS (**Granularity Control** System).
'96	**The Ciao/CiaoPP model for Assertion Verification by Abstract Interpretation**
'91-'06	Combined **abstract interpretation and (abstract) partial evaluation**.
'96-'00	**Lower bounds** cost analysis, **divide-and-conquer**. No-fail (no exceptions), determinacy.
'01-05	**Modular analysis** for scalability. New (imperative) shape/type domains, widenings.
'03	**Abstraction carrying code**.
'00-'05	**Multi-language support via CLP (a.k.a. CHCs) as IR**: Java, C# (shapes, resources, ...).
'04-'07	Verification/debugging/optimization of **user-defined resources**.
'06-'08	Verification of **execution time**, **energy** for Java, heap models, ... Probabilistic cost.
'12-18	(X)C **binary program energy analysis/verification**, ISA-level energy models.
'13-18	**Cost analysis as Abstract Interpretation** Sized shapes. LLVM. **Static Profiling**.
'16-18	**Sematic code search** via abstract interpretation. **Abstract distances.**
'17-19	**Combined modular/incremental analysis (scalability)**, fixpoint guidance.
'19-'23	**Verify: Interactive Verification via Abstract Interpretation**.
'19-'23	**Smart contracts**: verification of resources (gas, storage, commands, ...).

Milestones / tools

'88	**MA3 analyzer**: memo tables (cf. OLDT resolution), practicality. **Abstract compilation**.
'89	**PLAI analyzer**: **the "top-down" algorithm**, "summaries", abstract domains as plugins. Sharing(aliasing)/independence → auto-parallelization / real speedups (on shmem).
90's	**Incremental analysis**, concurrency, automatic domain combinations, scalability.
'90-'93	**Automatic cost analysis** (upper bounds), GraCoS (**Granularity Control** System).
'96	**The Ciao/CiaoPP model for Assertion Verification by Abstract Interpretation**
'91-'06	Combined **abstract interpretation and (abstract) partial evaluation**.
'96-'00	**Lower bounds** cost analysis, **divide-and-conquer**. No-fail (no exceptions), determinacy.
'01-05	**Modular analysis** for scalability. New (imperative) shape/type domains, widenings.
'03	**Abstraction carrying code**.
'00-'05	**Multi-language support via CLP (a.k.a. CHCs) as IR**: Java, C# (shapes, resources, ...).
'04-'07	Verification/debugging/optimization of **user-defined resources**.
'06-'08	Verification of **execution time**, **energy** for Java, heap models, ... Probabilistic cost.
'12-18	(X)C **binary program energy analysis/verification**, ISA-level energy models.
'13-18	**Cost analysis as Abstract Interpretation** Sized shapes. LLVM. **Static Profiling**.
'16-18	**Sematic code search** via abstract interpretation. **Abstract distances.**
'17-19	**Combined modular/incremental analysis (scalability)**, fixpoint guidance.
'19-'23	**Verify: Interactive Verification via Abstract Interpretation**.
'19-'23	**Smart contracts**: verification of resources (gas, storage, commands, ...).

Milestones / tools

'88	**MA3 analyzer**: memo tables (cf. OLDT resolution), practicality. **Abstract compilation**.
'89	**PLAI analyzer**: **the "top-down" algorithm**, "summaries", abstract domains as plugins. Sharing(aliasing)/independence → auto-parallelization / real speedups (on shmem).
90's	**Incremental analysis**, concurrency, automatic domain combinations, scalability.
'90-'93	**Automatic cost analysis** (upper bounds), GraCoS (**Granularity Control** System).
'96	**The Ciao/CiaoPP model for Assertion Verification by Abstract Interpretation**
'91-'06	Combined **abstract interpretation and (abstract) partial evaluation**.
'96-'00	**Lower bounds** cost analysis, **divide-and-conquer**. No-fail (no exceptions), determinacy.
'01-05	**Modular analysis** for scalability. New (imperative) shape/type domains, widenings.
'03	**Abstraction carrying code**.
'00-'05	**Multi-language support via CLP (a.k.a. CHCs) as IR**: Java, C# (shapes, resources, ...).
'04-'07	Verification/debugging/optimization of **user-defined resources**.
'06-'08	Verification of **execution time**, **energy** for Java, heap models, ... Probabilistic cost.
'12-18	(X)C **binary program energy analysis/verification**, ISA-level energy models.
'13-18	**Cost analysis as Abstract Interpretation** Sized shapes. LLVM. **Static Profiling**.
'16-18	**Sematic code search** via abstract interpretation. **Abstract distances.**
'17-19	**Combined modular/incremental analysis (scalability)**, fixpoint guidance.
'19-'23	**Verify: Interactive Verification via Abstract Interpretation**.
'19-'23	**Smart contracts**: verification of resources (gas, storage, commands, ...).

Milestones / tools

'88	**MA3 analyzer**: memo tables (cf. OLDT resolution), practicality. **Abstract compilation**.
'89	**PLAI analyzer**: **the "top-down" algorithm**, "summaries", abstract domains as plugins. Sharing(aliasing)/independence → auto-parallelization / real speedups (on shmem).
90's	**Incremental analysis**, concurrency, automatic domain combinations, scalability.
'90-'93	**Automatic cost analysis** (upper bounds), GraCoS (**Granularity Control** System).
'96	**The Ciao/CiaoPP model for Assertion Verification by Abstract Interpretation**
'91-'06	Combined **abstract interpretation and (abstract) partial evaluation**.
'96-'00	**Lower bounds** cost analysis, **divide-and-conquer**. No-fail (no exceptions), determinacy.
'01-05	**Modular analysis** for scalability. New (imperative) shape/type domains, widenings.
'03	**Abstraction carrying code**.
'00-'05	**Multi-language support via CLP (a.k.a. CHCs) as IR**: Java, C# (shapes, resources, ...).
'04-'07	Verification/debugging/optimization of **user-defined resources**.
'06-'08	Verification of **execution time**, **energy** for Java, heap models, ... Probabilistic cost.
'12-18	(X)C **binary program energy analysis/verification**, ISA-level energy models.
'13-18	**Cost analysis as Abstract Interpretation** Sized shapes. LLVM. **Static Profiling**.
'16-18	**Sematic code search** via abstract interpretation. **Abstract distances.**
'17-19	**Combined modular/incremental analysis (scalability)**, fixpoint guidance.
'19-'23	**Verify: Interactive Verification via Abstract Interpretation**.
'19-'23	**Smart contracts**: verification of resources (gas, storage, commands, ...).

Milestones / tools

'88	**MA3 analyzer**: memo tables (cf. OLDT resolution), practicality. **Abstract compilation**.
'89	**PLAI analyzer**: **the "top-down" algorithm**, "summaries", abstract domains as plugins. Sharing(aliasing)/independence → auto-parallelization / real speedups (on shmem).
90's	**Incremental analysis**, concurrency, automatic domain combinations, scalability.
'90-'93	**Automatic cost analysis** (upper bounds), GraCoS (**Granularity Control** System).
'96	**The Ciao/CiaoPP model for Assertion Verification by Abstract Interpretation**
'91-'06	Combined **abstract interpretation and (abstract) partial evaluation**.
'96-'00	**Lower bounds** cost analysis, **divide-and-conquer**. No-fail (no exceptions), determinacy.
'01-05	**Modular analysis** for scalability. New (imperative) shape/type domains, widenings.
'03	**Abstraction carrying code**.
'00-'05	**Multi-language support via CLP (a.k.a. CHCs) as IR**: Java, C# (shapes, resources, ...).
'04-'07	Verification/debugging/optimization of **user-defined resources**.
'06-'08	Verification of **execution time**, **energy** for Java, heap models, ... Probabilistic cost.
'12-18	(X)C **binary program energy analysis/verification**, ISA-level energy models.
'13-18	**Cost analysis as Abstract Interpretation** Sized shapes. LLVM. **Static Profiling**.
'16-18	**Sematic code search** via abstract interpretation. **Abstract distances.**
'17-19	**Combined modular/incremental analysis (scalability)**, fixpoint guidance.
'19-'23	**Verify: Interactive Verification via Abstract Interpretation**.
'19-'23	**Smart contracts**: verification of resources (gas, storage, commands, ...).

Milestones / tools

'88	**MA3 analyzer**: memo tables (cf. OLDT resolution), practicality. **Abstract compilation**.
'89	**PLAI analyzer**: **the "top-down" algorithm**, "summaries", abstract domains as plugins. Sharing(aliasing)/independence → auto-parallelization / real speedups (on shmem).
90's	**Incremental analysis**, concurrency, automatic domain combinations, scalability.
'90-'93	**Automatic cost analysis** (upper bounds), GraCoS (**Granularity Control** System).
'96	**The Ciao/CiaoPP model for Assertion Verification by Abstract Interpretation**
'91-'06	Combined **abstract interpretation and (abstract) partial evaluation**.
'96-'00	**Lower bounds** cost analysis, **divide-and-conquer**. No-fail (no exceptions), determinacy.
'01-05	**Modular analysis** for scalability. New (imperative) shape/type domains, widenings.
'03	**Abstraction carrying code**.
'00-'05	**Multi-language support via CLP (a.k.a. CHCs) as IR**: Java, C# (shapes, resources, ...).
'04-'07	Verification/debugging/optimization of **user-defined resources**.
'06-'08	Verification of **execution time**, **energy** for Java, heap models, ... Probabilistic cost.
'12-18	(X)C **binary program energy analysis/verification**, ISA-level energy models.
'13-18	**Cost analysis as Abstract Interpretation** Sized shapes. LLVM. **Static Profiling**.
'16-18	**Sematic code search** via abstract interpretation. **Abstract distances.**
'17-19	**Combined modular/incremental analysis (scalability)**, fixpoint guidance.
'19-'23	**Verify: Interactive Verification via Abstract Interpretation**.
'19-'23	**Smart contracts**: verification of resources (gas, storage, commands, ...).

Milestones / tools

'88	**MA3 analyzer**: memo tables (cf. OLDT resolution), practicality. **Abstract compilation**.
'89	**PLAI analyzer**: **the "top-down" algorithm**, "summaries", abstract domains as plugins. Sharing(aliasing)/independence → auto-parallelization / real speedups (on shmem).
90's	**Incremental analysis**, concurrency, automatic domain combinations, scalability.
'90-'93	**Automatic cost analysis** (upper bounds), GraCoS (**Granularity Control** System).
'96	**The Ciao/CiaoPP model for Assertion Verification by Abstract Interpretation**
'91-'06	Combined **abstract interpretation and (abstract) partial evaluation**.
'96-'00	**Lower bounds** cost analysis, **divide-and-conquer**. No-fail (no exceptions), determinacy.
'01-05	**Modular analysis** for scalability. New (imperative) shape/type domains, widenings.
'03	**Abstraction carrying code**.
'00-'05	**Multi-language support via CLP (a.k.a. CHCs) as IR**: Java, C# (shapes, resources, ...).
'04-'07	Verification/debugging/optimization of **user-defined resources**.
'06-'08	Verification of **execution time**, **energy** for Java, heap models, ... Probabilistic cost.
'12-18	(X)C **binary program energy analysis/verification**, ISA-level energy models.
'13-18	**Cost analysis as Abstract Interpretation** Sized shapes. LLVM. **Static Profiling**.
'16-18	**Sematic code search** via abstract interpretation. **Abstract distances.**
'17-19	**Combined modular/incremental analysis (scalability)**, fixpoint guidance.
'19-'23	**Verify: Interactive Verification via Abstract Interpretation**.
'19-'23	**Smart contracts**: verification of resources (gas, storage, commands, ...).

Thanks!

Ciao/CiaoPP

Site:	`https://ciao-lang.org`
Playground:	`https://ciao-lang.org/playground`
Source:	`https://github.com/ciao-lang`

Energy Usage Verification

Example: XC Program (FIR Filter), w/Energy Specification [HIP3ES'15, TPLP'18]

```
#pragma check fir(xn, coeffs, state, ELEMENTS) :
          (1 <= ELEMENTS && energy <= 416.0)
#pragma true fir(xn, coeffs, state, ELEMENTS) :
          ( energy >= 3.35*ELEMENTS + 13.96 &&
            energy <= 3.35*ELEMENTS + 14.4 )
#pragma checked fir(xn, coeffs, state, ELEMENTS) :
          (1 <= ELEMENTS && ELEMENTS <= 120 && energy <= 416.1)
#pragma false fir(xn, coeffs, state, ELEMENTS) :
          (121 <= ELEMENTS && energy <= 416.1)

int fir(int xn, int coeffs[], int state[], int ELEMENTS)
{ unsigned int ynl; int ynh;
  ynl = (1<<23); ynh = 0;
  for(int j=ELEMENTS-1; j!=0; j--) {
      state[j] = state[j-1];
      {ynh, ynl} = macs(coeffs[j], state[j], ynh, ynl); }
  state[0] = xn;
  {ynh, ynl} = macs(coeffs[0], xn, ynh, ynl);
  if (sext(ynh,24) == ynh) {
      ynh = (ynh << 8) | (((unsigned) ynl) >> 24);}
  else if (ynh < 0) { ynh = 0x80000000; }
  else { ynh = 0x7fffffff; }
  return ynh; }
```

Energy Usage Verification

Example: XC Program (FIR Filter), w/Energy Specification [HIP3ES'15, TPLP'18]

```
#pragma check fir(xn, coeffs, state, ELEMENTS) :
          (1 <= ELEMENTS && energy <= 416.0)
#pragma true fir(xn, coeffs, state, ELEMENTS) :
          ( energy >= 3.35*ELEMENTS + 13.96 &&
            energy <= 3.35*ELEMENTS + 14.4 )
#pragma checked fir(xn, coeffs, state, ELEMENTS) :
          (1 <= ELEMENTS && ELEMENTS <= 120 && energy <= 416.1)
#pragma false fir(xn, coeffs, state, ELEMENTS) :
          (121 <= ELEMENTS && energy <= 416.1)

int fir(int xn, int coeffs[], int state[], int ELEMENTS)
{ unsigned int ynl; int ynh;
  ynl = (1<<23); ynh = 0;
  for(int j=ELEMENTS-1; j!=0; j--) {
      state[j] = state[j-1];
      {ynh, ynl} = macs(coeffs[j], state[j], ynh, ynl); }
  state[0] = xn;
  {ynh, ynl} = macs(coeffs[0], xn, ynh, ynl);
  if (sext(ynh,24) == ynh) {
      ynh = (ynh << 8) | (((unsigned) ynl) >> 24);}
  else if (ynh < 0) { ynh = 0x80000000; }
  else { ynh = 0x7fffffff; }
  return ynh; }
```

Energy Usage Verification

Example: XC Program (FIR Filter), w/Energy Specification [HIP3ES'15, TPLP'18]

```
#pragma check fir(xn, coeffs, state, ELEMENTS) :
          (1 <= ELEMENTS && energy <= 416.0)
#pragma true fir(xn, coeffs, state, ELEMENTS) :
          ( energy >= 3.35*ELEMENTS + 13.96 &&
            energy <= 3.35*ELEMENTS + 14.4 )
#pragma checked fir(xn, coeffs, state, ELEMENTS) :
          (1 <= ELEMENTS && ELEMENTS <= 120 && energy <= 416.1)
#pragma false fir(xn, coeffs, state, ELEMENTS) :
          (121 <= ELEMENTS && energy <= 416.1)

int fir(int xn, int coeffs[], int state[], int ELEMENTS)
{ unsigned int ynl; int ynh;
  ynl = (1<<23); ynh = 0;
  for(int j=ELEMENTS-1; j!=0; j--) {
      state[j] = state[j-1];
      {ynh, ynl} = macs(coeffs[j], state[j], ynh, ynl); }
  state[0] = xn;
  {ynh, ynl} = macs(coeffs[0], xn, ynh, ynl);
  if (sext(ynh,24) == ynh) {
      ynh = (ynh << 8) | (((unsigned) ynl) >> 24);}
  else if (ynh < 0) { ynh = 0x80000000; }
  else { ynh = 0x7fffffff; }
  return ynh; }
```

Results for some sample contracts (I)

Contract	Metrics		Resource A.	Time
	Parameter (α)	Storage (β)	gas	(ms)
`reverse`	*length*	*length*	α	216
`addition`	*value*	*value*	$\log \alpha$	147
`michelson_arith`	*value*	*value*	$\log(\alpha^2 + 2 * \beta)$	208
`bytes`	*value*	*length*	β	229
`list_inc`	*value*	*length*	β	273
`lambda`	*value*	*value*	$\log \alpha$	99
`lambda_apply`	(*value*, *size*)	*size*	k	114
`inline`	*size*	*value*	$\log \beta$	870
`cross_product`	(*length*, *length*)	*value*	$\alpha_1 + \alpha_2$	424
`lineal`	*value*	*value*	α	244
`assertion_map`	(*value*, *size*)	*length*	$\log \beta * \log \alpha_1$	393

Results for some sample contracts (II)

Contract	Metrics		Resource A.	Time
	Parameter (α)	Storage (β)	gas	(ms)
`quadratic`	*length*	*length*	$\alpha * \beta$	520
`queue`	*size*	(*value*, *size*, *length*)	$\log \beta_1 * \log \beta_3$	831
`king_of_tez`	*size*	(*value*, *value*, *size*)	k	635
`set_management`	*length*	*length*	$\alpha * \log \beta$	357
`lock`	*size*	(*value*, *value*, *size*)	k	421
`max_list`	*length*	*size*	α	473
`zipper`	*length*	(*length*, *length*, *length*)	k	989
`auction`	*size*	(*value*, *value*, *size*)	k	573
`union`	(*length*, *length*)	*length*	$\alpha_1 * \log \alpha_2$	486
`append`	(*length*, *length*)	*length*	α_1	371
`subset`	(*length*, *length*)	*size*	$\alpha_1 * \log \alpha_2$	389

Some related CiaoPP references

Interactive verification

[TPLP'21b] M.A. Sanchez-Ordaz, I. Garcia-Contreras, V. Perez-Carrasco, J. F. Morales, P. Lopez-Garcia, M. V. Hermenegildo.
VeriFly: On-the-fly Assertion Checking via Incrementality.
In *Theory and Practice of Logic Programming*, Vol. 21, Num. 6, pages 768-784, Cambridge U. Press, September 2021. Special Issue on ICLP'21.

[NASA-FIDE21] M. A. Sanchez-Ordaz, I. Garcia-Contreras, V. Perez-Carrasco, J. F. Morales, P. Lopez-Garcia, M.V. Hermenegildo.
VeriFly: On-the-fly Assertion Checking with CiaoPP.
In *6th Workshop on Formal Integrated Development Environment (F-IDE 2021, part of NASA NFM'21)*, Electronic Proceedings in Theoretical Computer Science (EPTCS), pages 1-5, Open Publishing Association (OPA), May 2021. Co-located with ETAPS 2021.

[LOPSTR'18] I. Garcia-Contreras, J.F. Morales, M. V. Hermenegildo.
Multivariant Assertion-based Guidance in Abstract Interpretation.
In *Post-Proceedings of the 28th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'18)*, LNCS, Num. 11408, pages 184-201, Springer-Verlag, January 2019.

Horn Clauses as Intermediate Representation / Multi-Language Support

[LOPSTR'07] M. Méndez-Lojo, J. Navas, and M. Hermenegildo.
A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs.
In *17th Intl. Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*, number 4915 in LNCS, pages 154–168. Springer-Verlag, August 2007.

[VPT'20] J. Gallagher, M. V. Hermenegildo, B. Kafle, M. Klemen, P. Lopez-Garcia, J.F. Morales.
From big-step to small-step semantics and back with interpreter specialization (invited paper).
In *Proceedings of the Eighth International Workshop on Verification and Program Transformation (VPT 2020)*, Electronic Proceedings in Theoretical Computer Science (EPTCS), pages 50-65, Open Publishing Association (OPA), 2020. Co-located with ETAPS 2020.

[TPLP'21a] Emanuele De Angelis, Fabio Fioravanti, John P. Gallagher, Manuel V. Hermenegildo, Alberto Pettorossi, Maurizio Proietti.
Analysis and Transformation of Constrained Horn Clauses for Program Verification.
In *Theory and Practice of Logic Programming*, Vol. FirstView, pages 1-69, Cambridge U. Press, November 2021.

Scalability/Modularity/Incrementality in Analysis/Specialization/Verification

[TPLP'21c] I. Garcia-Contreras, J. F. Morales, M. V. Hermenegildo.
Incremental and Modular Context-sensitive Analysis.
In *Theory and Practice of Logic Programming*, Vol. 21, Num. 2, pages 196-243, Cambridge U. Press, January 2021.

[ICLP'18] I. Garcia-Contreras, J. F. Morales, and M. V. Hermenegildo.
Towards Incremental and Modular Context-sensitive Analysis.
In *Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018)*, OpenAccess Series in Informatics (OASIcs), July 2018.

[LPAR'06] P. Pietrzak, J. Correas, G. Puebla, and M. Hermenegildo.
Context-Sensitive Multivariant Assertion Checking in Modular Programs.
In *LPAR'06*, number 4246 in LNCS, pages 392–406. Springer-Verlag, November 2006.

[LOPSTR'01] F. Bueno, M. García de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey.
A Model for Inter-module Analysis and Optimizing Compilation.
In *Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, March 2001.

[TOPLAS'00] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey.
Incremental Analysis of Constraint Logic Programs.
ACM Transactions on Programming Languages and Systems, 22(2):187–223, March 2000.

[ENTCS'00] G. Puebla and M. Hermenegildo.
Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs.
In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[SAS'96] G. Puebla and M. Hermenegildo.
Optimized Algorithms for the Incremental Analysis of Logic Programs.
In *Intl. Static Analysis Symposium (SAS 1996)*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[ICLP'95] M. V. Hermenegildo, G. Puebla, K. Marriott, P. Stuckey.
Incremental Analysis of Logic Programs.
In *International Conference on Logic Programming*, pages 797-811, MIT Press, June 1995.

The Basic Analysis Framework (Abstract Interpreter, Fixpoint)

[LOPSTR'19] M. Klemen, P. Lopez-Garcia, J. Gallagher, J.F. Morales, M. V. Hermenegildo.
A General Framework for Static Cost Analysis of Parallel Logic Programs.
In *Post-Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19)*, LNCS, Vol. 12042, pages 19-35, Springer-Verlag, April 2020.

[Bytecode'07] M. Méndez-Lojo, J. Navas, and M. Hermenegildo.
An Efficient, Parametric Fixpoint Algorithm for Analysis of Java Bytecode.
In *ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, March 2007.

[JLP'97] M. Codish, M. Bruynooghe, M. García de la Banda, and M. V. Hermenegildo.
Exploiting Goal Independence in the Analysis of Logic Programs.
Journal of Logic Programming, 32(3):247–261, September 1997.

[TOPLAS'95] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo.
Improving Abstract Interpretations by Combining Domains.
ACM Transactions on Programming Languages and Systems, 17(1):28–44, January 1995.

[TOPLAS'96] M. García de la Banda, M. V. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens.
Global Analysis of Constraint Logic Programs.
ACM Trans. on Programming Languages and Systems, 18(5):564–615, 1996.

[POPL'94] K. Marriott, M. García de la Banda, and M. Hermenegildo.
Analyzing Logic Programs with Dynamic Scheduling.
In *20th. Annual ACM Conf. on Principles of Programming Languages*, pages 240–254. ACM, January 1994.

[JLP'92] K. Muthukumar and M. Hermenegildo.
Compile-time Derivation of Variable Dependency Using Abstract Interpretation.
Journal of Logic Programming, 13(2/3):315–347, July 1992.

[MCC'90] K. Muthukumar, M. Hermenegildo.
Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs.
TR Num. ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.

[NACLP'89] K. Muthukumar and M. Hermenegildo.
Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation.
In *1989 N. American Conf. on Logic Programming*, pages 166–189. MIT Press, October 1989.

[ICLP'88] R. Warren, M. Hermenegildo, and S. K. Debray
On the Practicality of Global Flow Analysis of Logic Programs
In *5th Intl. Conf. and Symp. on Logic Programming*, pp. 684–699, MIT Press, August 1988.

Semantic Code Search

[ICLP'16] I. Garcia-Contreras, J. F. Morales, and M. V. Hermenegildo.
Semantic Code Browsing.
Theory and Practice of Logic Programming, 32nd Int'l. Conference on Logic Programming (ICLP'16) Special Issue, 16(5-6):721–737, October 2016.

Abstraction-Carrying Code

[TPLP'12] E. Albert, P. Arenas, G. Puebla, and M. Hermenegildo.
Certificate Size Reduction in Abstraction-Carrying Code.
Theory and Practice of Logic Programming, Cambridge U. Press, 2012.

[PPDP'05] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla.
Abstraction Carrying Code and Resource-Awareness.
In *PPDP*. ACM Press, 2005.

[LPAR'04] E. Albert, G. Puebla, and M. Hermenegildo.
Abstraction-Carrying Code.
In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.

[COCV'04] E. Albert, G. Puebla, and M. Hermenegildo.
An Abstract Interpretation-based Approach to Mobile Code Safety.
In *Proc. of Compiler Optimization meets Compiler Verification (COCV'04)*, Electronic Notes in Theoretical Computer Science 132(1), pages 113–129. Elsevier - North Holland, April 2004.

Other Abstract Interpretation-Related Techniques

[LOPSTR'20] Ignacio Casso, José F. Morales, Pedro López-García, Manuel V. Hermenegildo.
Testing Your (Static Analysis) Truths.
In *Logic-Based Program Synthesis and Transformation - 30th International Symposium, Post-Proceedings*, Lecture Notes in Computer Science, Vol. 12561, pages 271-292, Springer, 2021.

[LOPSTR'19] I. Casso, J. F. Morales, P. Lopez-Garcia, R. Giacobazzi, M. V. Hermenegildo.
Computing Abstract Distances in Logic Programs.
In *Post-Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19)*, LNCS, Vol. 12042, Springer-Verlag, April 2020

Abstract Domains: Sharing/Aliasing

[ISMM'09] M. Marron, D. Kapur, and M. Hermenegildo.
Identification of Logically Related Heap Regions.
In *ISMM'09: Proceedings of the 8th international symposium on Memory management*, New York, NY, USA, June 2009. ACM Press.

[LCPC'08] M. Méndez-Lojo, O. Lhoták, and M. Hermenegildo.
Efficient Set Sharing using ZBDDs.
In *21st Int'l. WS on Languages and Compilers for Parallel Computing (LCPC'08)*, LNCS. Springer-Verlag, August 2008.

[ICLP'08] E. Trias, J. Navas, E. S. Ackley, S. Forrest, and M. V. Hermenegildo.
Negative Ternary Set-Sharing.
In *Int'l. Conference on Logic Programming, ICLP*, LNCS 5366, pages 301–316, Springer-Verlag, December 2008.

[LCPC'08] M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo.
Identification of Heap-Carried Data Dependence Via Explicit Store Heap Models.
In *21st Int'l. WS on Languages and Compilers for Parallel Computing (LCPC'08)*, LNCS. Springer-Verlag, August 2008.

[PASTE'08] M. Marron, M. Méndez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur.
Sharing Analysis of Arrays, Collections, and Recursive Structures.
In *ACM WS on Program Analysis for SW Tools and Engineering (PASTE'08)*. ACM, November 2008.

[VMCAI'08] M. Méndez-Lojo and M. Hermenegildo.
Precise Set Sharing Analysis for Java-style Programs.
In *9th Intl. Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*, number 4905 in LNCS, pages 172–187. Springer-Verlag, January 2008.

[PADL'06] J. Navas, F. Bueno, and M. Hermenegildo.
Efficient top-down set-sharing analysis using cliques.
In *Eight Intl. Symposium on Practical Aspects of Declarative Languages*, number 2819 in LNCS, pages 183–198. Springer-Verlag, January 2006.

[TOPLAS'99] F. Bueno, M. García de la Banda, and M. Hermenegildo.
Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming.
ACM Transactions on Programming Languages and Systems, 21(2):189–238, March 1999.

[NACLP'89] K. Muthukumar and M. Hermenegildo.
Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation.
In *1989 N. American Conf. on Logic Programming*, pages 166–189. MIT Press, October 1989.

Abstract Domains: Shape/Type Analysis

[ICLP'13] A. Serrano, P. López-Garcia, F. Bueno, and M. Hermenegildo.
Sized Type Analysis of Logic Programs (Technical Communication).
In *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement,* pages 1–14, Cambridge U. Press, August 2013.

[CC'08] M. Marron, M. Hermenegildo, D. Kapur, and D. Stefanovic.
Efficient context-sensitive shape analysis with graph-based heap models.
In Laurie Hendren, editor, *Intl. Conference on Compiler Construction (CC 2008)*, Lecture Notes in Computer Science. Springer, April 2008.

[PASTE'07] M. Marron, D. Stefanovic, M. Hermenegildo, and D. Kapur.
Heap Analysis in the Presence of Collection Libraries.
In *ACM WS on Program Analysis for SW Tools and Engineering (PASTE'07)*. ACM, June 2007.

[SAS'02] C. Vaucheret and F. Bueno.
More Precise yet Efficient Type Inference for Logic Programs.
In *Intl. Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.

Abstract Domains: Non-failure, Determinacy

[NGC'10] P. López-García, F. Bueno, and M. Hermenegildo.
Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information.
New Generation Computing, 28(2):117–206, 2010.

[LOPSTR'04] P. López-García, F. Bueno, and M. Hermenegildo.
Determinacy Analysis for Logic Programs Using Mode and Type Information.
In *Proceedings of the 14th Intl. Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 19–35. Springer-Verlag, August 2005.

[FLOPS'04] F. Bueno, P. López-García, and M. Hermenegildo.
Multivariant Non-Failure Analysis via Standard Abstract Interpretation.
In *7th Intl. Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100–116, Heidelberg, Germany, April 2004. Springer-Verlag.

[ICLP'97] S.K. Debray, P. López-García, and M. Hermenegildo.
Non-Failure Analysis for Logic Programs.
In *1997 Intl. Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

Analysis and Verification of Energy

[TPLP'18] P. Lopez-Garcia, L. Darmawan, M. Klemen, U. Liqat, F. Bueno, and M. V. Hermenegildo.
Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption.
Theory and Practice of Logic Programming, Special Issue on Computational Logic for Verification, 18:167–223, March 2018. arXiv:1803.04451.

[HIP3ES'16] U. Liqat, Z. Banković, P. Lopez-Garcia, and M. V. Hermenegildo.
Inferring Energy Bounds Statically by Evolutionary Analysis of Basic Blocks.
In *(HIP3ES'16)*, 2016. arXiv:1601.02800.

[FOPARA'15] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder.
Inferring Parametric Energy Consumption Functions at Different SW Levels: ISA vs. LLVM IR.
In *Foundational and Practical Aspects of Resource Analysis: 4th Intl. Workshop, FOPARA 2015, Revised Selected Papers*, volume 9964 LNCS, pages 81–100. Springer, 2016.

[HIP3ES'15] P. Lopez-Garcia, R. Haemmerlé, M. Klemen, U. Liqat, and M. V. Hermenegildo
Towards Energy Consumption Verification via Static Analysis.
In *HIPEAC Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES 2015)*, Amsterdam.

[LOPSTR'13] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. López-Garcia, N. Grech, M.V. Hermenegildo, and K. Eder.
Energy Consumption Analysis of Programs based on XMOS ISA-Level Models.
In *Pre-proceedings of the 23rd Intl. Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*, LNCS, Springer, September 2013.

[NASA FM'08] J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications.
In *6th NASA Langley Formal Methods Workshop (LFM 08)*, April 2008. Extended Abstract.

Analysis and Verification of Resources in General

[TPLP'16] P. Lopez-Garcia, M. Klemen, U. Liqat, and M.V. Hermenegildo.
A General Framework for Static Profiling of Parametric Resource Usage.
Theory and Practice of Logic Programming, 32nd Int'l. Conference on Logic Programming (ICLP'16) Special Issue, 16(5-6):849–865, October 2016.

[FLOPS'16] R. Haemmerlé, P. Lopez-Garcia, U. Liqat, M. Klemen, J. P. Gallagher, and M. V. Hermenegildo.
A Transformational Approach to Parametric Accumulated-cost Static Profiling.
In *FLOPS'16*, volume 9613 of *LNCS*, pages 163–180. Springer, 2016.

[TPLP'14] A. Serrano, P. López-Garcia, and M. Hermenegildo.
Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types.
In *Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue*, Vol. 14, Num. 4-5, pages 739-754, Cambridge U. Press, 2014.

[ICLP'13] A. Serrano, P. López-Garcia, F. Bueno, and M. Hermenegildo.
Sized Type Analysis of Logic Programs (Technical Communication).
In *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement,* pages 1–14, Cambridge U. Press, August 2013.

[FOPARA'12] P. Lopez-Garcia, L. Darmawan, F. Bueno, and M. Hermenegildo
Interval-Based Resource Usage Verification: Formalization and Prototype
In *Foundational and Practical Aspects of Resource Analysis. Second Iternational Workshop FOPARA 2011, Revised Selected Papers*. Lecture Notes in Computer Science, 2012, 7177, 54–71, Springer.

[ICLP'10] P. López-García, L. Darmawan, and F. Bueno.
A Framework for Verification and Debugging of Resource Usage Properties.
In *Technical Communications of the 26th ICLP. Leibniz Int'l. Proc. in Informatics (LIPIcs)*, Vol. 7, pages 104–113, Dagstuhl, Germany, July 2010.

[Bytecode'09] J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
User-Definable Resource Usage Bounds Analysis for Java Bytecode.
In *Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253 of *ENTCS*, pages 6–86. Elsevier, March 2009.

[PPDP'08] E. Mera, P. López-García, M. Carro, and M. Hermenegildo.
Towards Execution Time Estimation in Abstract Machine-Based Languages.
In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 174–184. ACM Press, July 2008.

[ICLP'07] J. Navas, E. Mera, P. López-García, and M. Hermenegildo.
User-Definable Resource Bounds Analysis for Logic Programs.
In *23rd International Conference on Logic Programming (ICLP'07)*, LNCS Vol. 4670. Springer, 2007. **ICLP 2017 10-year Test of Time Award.**

[CLEI'06] H. Soza, M. Carro, and P. López-García.
Probabilistic Cost Analysis of Logic Programs: A First Case Study.
In *XXXII Latin-American Conference on Informatics* (CLEI 2006), August 2006.

[ILPS'97] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin.
Lower Bound Cost Estimation for Logic Programs.
In *1997 Intl. Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[JSC'96] P. López-García, M. Hermenegildo, and S. K. Debray.
A Methodology for Granularity Based Control of Parallelism in Logic Programs.
Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation, 21(4–6):715–734, 1996.

[ICLP'95] P. López-García and M. Hermenegildo.
Efficient Term Size Computation for Granularity Control.
In *International Conference on Logic Programming*, pages 647–661, Cambridge, MA, June 1995. MIT Press, Cambridge, MA.

[SAS'94] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin.
Estimating the Computational Cost of Logic Programs.
In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Sep 1994. Springer.

[PASCO'94] P. López-García, M. Hermenegildo, and S.K. Debray.
Towards Granularity Based Control of Parallelism in Logic Programs.
In Hoon Hong, editor, *Proc. of First Intl. Symposium on Parallel Symbolic Computation, PASCO'94*, pages 133–144. World Scientific, September 1994.

[PLDI'90] S. K. Debray, N.-W. Lin, and M. Hermenegildo.
Task Granularity Analysis in Logic Programs.
In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 174–188. ACM Press, June 1990.

Application of the CiaoPP framework to Smart Contracts

[SAS'20] V. Perez-Carrasco, M. Klemen, P. Lopez-Garcia, J.F. Morales, M. V. Hermenegildo.
Cost Analysis of Smart Contracts via Parametric Resource Analysis.
In *Proceedings of the 27th Static Analysis Symposium (SAS 2020)*, LNCS, Vol. 12389, pages 7-31, Springer, November 2020.

Some applications of the CiaoPP framework to Java bytecode

[VMCAI'08] M. Méndez-Lojo, M. Hermenegildo.
Precise Set Sharing Analysis for Java-style Programs.
9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08), LNCS, Num. 4905, pages 172-187, Springer-Verlag, January 2008.

[FTfJP'07] J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
An Efficient, Context and Path Sensitive Analysis Framework for Java Programs.
In *9th Workshop on Formal Techniques for Java-like Programs FTfJP 2007*, July 2007.

[Bytecode'07] M. Méndez-Lojo, J. Navas, and M. Hermenegildo.
An Efficient, Parametric Fixpoint Algorithm for Analysis of Java Bytecode.
In *ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, March 2007.

[Bytecode'09] J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
User-Definable Resource Usage Bounds Analysis for Java Bytecode.
In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 6–86. Elsevier - North Holland, March 2009.