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

Manuel Hermenegildo¹,² I. García-Contreras¹,² J. Morales¹,²
P. López-García¹,³ L. Rustenholz¹,² D. Ferreiro¹,² D. Jurjo¹,²

LOPSTR’23 (w/SPLASH)
Cascais, Portugal – October 23-24, 2023

¹IMDEA Software Institute
²T.U. of Madrid (UPM)
³Spanish 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](http://example.com)

- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure).  
  
  ![SAS'96, TOPLAS'00](http://example.com)

- Extending incremental analysis to exploit much better modular structure.  
  
  ![ICLP'18, LOPSTR'19, TPLP'21c](http://example.com)

- IDE integration → our **VeriFly** “on-the-fly” verification tool.  
  
  ![NASA-FIDE21, TPLP'21b](http://example.com)

All while:

- Supporting **multiple languages** via translation to CHCs (a.k.a. Prolog/CLP).  
  
  ![LOPSTR'07, TPLP'18, VPT'20, TPLP'21a](http://example.com)

- Covering both **functional and non-functional** properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...)  
  
  ![PLDI'90] ![SAS'20](http://example.com)
**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)
  \[\text{[NACLP'89, MCC'90]}\]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure).
  \[\text{[SAS'96, TOPLAS'00]}\]
- Extending incremental analysis to exploit much better modular structure.
  \[\text{[ICLP'18, LOPSTR'19, TPLP'21c]}\]
- IDE integration → our VeriFly “on-the-fly” verification tool.
  \[\text{[NASA-FIDE21, TPLP'21b]}\]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP).
  \[\text{[LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]}\]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ...
  time, memory, energy, gas, ...)
  \[\text{[PLDI'90, ... SAS'20]}\]
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]
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]
• **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)
  \[\text{[NAACL’89, MCC’90]}\]
- **Fine-grain (clause-level) incremental analysis** (originally not exploiting module structure).
  \[\text{[SAS’96, TOPLAS’00]}\]
- **Extending incremental analysis to exploit much better modular structure.**
  \[\text{[ICLP’18, LOPSTR’19, TPLP’21c]}\]
- **IDE integration → our VeriFly “on-the-fly” verification tool.**
  \[\text{[NASA-FIDE21, TPLP’21b]}\]

All while:

- **Supporting multiple languages** via translation to CHCs (a.k.a. Prolog/CLP).
  \[\text{[LOPSTR’07, TPLP’18, VPT’20, TPLP’21a]}\]
- **Covering both functional and non-functional properties** (types, pointers, shapes, intervals, ...
  time, memory, energy, gas, ...)
  \[\text{[PLDI’90] ... [SAS’20]}\]
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]
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.
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.
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.
<table>
<thead>
<tr>
<th>90’s: mostly statically-typed languages: ML, Haskell — Gödel, Mercury</th>
</tr>
</thead>
<tbody>
<tr>
<td>~~~ Developed the Ciao Prolog language, to provide:</td>
</tr>
<tr>
<td>Of course, an excellent Prolog, but, in addition:</td>
</tr>
<tr>
<td>► the flexibility / fast prototyping of dynamic languages,</td>
</tr>
<tr>
<td>► with the guarantees of static / strongly typed languages.</td>
</tr>
<tr>
<td>• Objective: bridge the gap between dynamic and static languages.</td>
</tr>
<tr>
<td>(First?) dynamic language with safety assurances.</td>
</tr>
</tbody>
</table>
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.

Keys:

- **Assertions** rather than (traditional) types, and **optional**.
- Do not restrict the properties → accept undecidability.
- Use safe approximations ~≈ abstract interpretation-based verification.

<table>
<thead>
<tr>
<th></th>
<th>Definition</th>
<th>Sufficient condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ is prt. correct w.r.t. $\mathcal{I}_\alpha$ if</td>
<td>$\alpha([P]) \leq \mathcal{I}_\alpha$</td>
<td>$[P]<em>{\alpha^+} \leq \mathcal{I}</em>\alpha$</td>
</tr>
<tr>
<td>$P$ is complete w.r.t. $\mathcal{I}_\alpha$ if</td>
<td>$\mathcal{I}_\alpha \leq \alpha([P])$</td>
<td>$\mathcal{I}<em>\alpha \leq [P]</em>{\alpha^=}$</td>
</tr>
<tr>
<td>$P$ is incorrect w.r.t. $\mathcal{I}_\alpha$ if</td>
<td>$\alpha([P]) \not\leq \mathcal{I}_\alpha$</td>
<td>$[P]<em>{\alpha^=} \not\leq \mathcal{I}</em>\alpha$, or $[P]<em>{\alpha^+} \cap \mathcal{I}</em>\alpha = \emptyset \land [P]_{\alpha^=} \neq \emptyset$</td>
</tr>
<tr>
<td>$P$ is incomplete w.r.t. $\mathcal{I}_\alpha$ if</td>
<td>$\mathcal{I}_\alpha \not\leq \alpha([P])$</td>
<td>$\mathcal{I}<em>\alpha \not\leq [P]</em>{\alpha^+}$</td>
</tr>
</tbody>
</table>
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.

Keys:

● **Assertions** rather than (traditional) types, and optional.
● **Do not restrict** the properties → accept undecidability.
● **Use safe approximations** ⇝ abstract interpretation-based verification.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Sufficient condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ is prt. correct w.r.t. $I_\alpha$ if $\alpha([P]) \leq I_\alpha$</td>
<td>$[[P]]<em>{\alpha^+} \leq I</em>\alpha$</td>
</tr>
<tr>
<td>$P$ is complete w.r.t. $I_\alpha$ if $I_\alpha \leq \alpha([P])$</td>
<td>$I_\alpha \leq [[P]]_{\alpha^+}$</td>
</tr>
<tr>
<td>$P$ is incorrect w.r.t. $I_\alpha$ if $\alpha([P]) \not\leq I_\alpha$</td>
<td>$[[P]]<em>{\alpha^+} \not\leq I</em>\alpha$, or</td>
</tr>
<tr>
<td></td>
<td>$[[P]]<em>{\alpha^+} \cap I</em>\alpha = \emptyset \land [[P]]_{\alpha} \neq \emptyset$</td>
</tr>
<tr>
<td>$P$ is incomplete w.r.t. $I_\alpha$ if $I_\alpha \not\leq \alpha([P])$</td>
<td>$I_\alpha \not\leq [[P]]_{\alpha^+}$</td>
</tr>
</tbody>
</table>
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.

Keys:

• **Assertions** rather than (traditional) types, and **optional**.
• **Do not restrict** the properties ⇒ accept undecidability.
• **Use safe approximations** ⇝ **abstract interpretation-based verification**.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Definition</th>
<th>Sufficient condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ is pt. correct w.r.t. $\mathcal{I}_\alpha$ if</td>
<td>$\alpha([P]) \leq \mathcal{I}_\alpha$</td>
<td>$[P]<em>{\alpha^+} \leq \mathcal{I}</em>\alpha$</td>
</tr>
<tr>
<td>$P$ is complete w.r.t. $\mathcal{I}_\alpha$ if</td>
<td>$\mathcal{I}_\alpha \leq \alpha([P])$</td>
<td>$\mathcal{I}<em>\alpha \leq {P}</em>{\alpha^+}$</td>
</tr>
<tr>
<td>$P$ is incorrect w.r.t. $\mathcal{I}_\alpha$ if</td>
<td>$\alpha([P]) \not\leq \mathcal{I}_\alpha$</td>
<td>${P}<em>{\alpha^+} \not\subseteq \mathcal{I}</em>\alpha$, or ( {P}<em>{\alpha^+} \cap \mathcal{I}</em>\alpha = \emptyset \land {P}_{\alpha} \neq \emptyset )</td>
</tr>
<tr>
<td>$P$ is incomplete w.r.t. $\mathcal{I}_\alpha$ if</td>
<td>$\mathcal{I}_\alpha \not\subseteq \alpha([P])$</td>
<td>$\mathcal{I}<em>\alpha \not\subseteq {P}</em>{\alpha^+}$</td>
</tr>
</tbody>
</table>
**PLAI (CiaoPP’s Generic AI Framework)**

- **Generic framework**: given $P$ (as a set of CHCs) and abstract domain(s), computes $\lambda \text{f}(S_P^\alpha) = \llbracket P \rrbracket^\alpha$, s.t. $\llbracket P \rrbracket^\alpha$ safely approximates $\llbracket P \rrbracket$.

→ 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 $\{\text{block} : \lambda_{\text{in}} \mapsto \lambda_{\text{out}}\}$.
    - Exit states for calls to $\text{block}$ satisfying precond $\lambda_{\text{in}}$ meet postcond $\lambda_{\text{out}}$.
  - **A dependency arc table**: $\{A : \lambda_{\text{in}A} \Rightarrow B : \lambda_{\text{in}B}\}$.
    - Answers for call $A : \lambda_{\text{in}A}$ depend on the answers for $B : \lambda_{\text{in}B}$: (if exit for $B : \lambda_{\text{in}B}$ changes, exit for $A : \lambda_{\text{in}A}$ 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]
Generic framework: given $P$ (as a set of CHCs) and abstract domain(s), computes $\text{lfp} (S_P^\alpha) = [\llbracket P \rrbracket]_\alpha$, s.t. $[\llbracket P \rrbracket]_\alpha$ safely approximates $[\llbracket P \rrbracket]$.

→ Essentially efficient, incremental, abstract OLDT resolution algo. for CHC’s. It is the original “top-down” algorithm! [NACL’89]

- It maintains and computes as a result (simplified):
  - **A call-answer table:** with (multiple) entries {block : $\lambda_{\text{in}} \mapsto \lambda_{\text{out}}$}.
    - Exit states for calls to block satisfying precond $\lambda_{\text{in}}$ meet postcond $\lambda_{\text{out}}$.
  - **A dependency arc table:** {A : $\lambda_{\text{in}A} \Rightarrow B : \lambda_{\text{in}B}$}.
    - Answers for call A : $\lambda_{\text{in}A}$ depend on the answers for B : $\lambda_{\text{in}B}$:
      (if exit for B : $\lambda_{\text{in}B}$ changes, exit for A : $\lambda_{\text{in}A}$ 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).

[NACL’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 \( \text{lfp}(S^\alpha_P) = \llbracket\ P \rrbracket^\alpha \), s.t. \( \llbracket\ P \rrbracket^\alpha \) safely approximates \( \llbracket\ P \rrbracket \).

\[ \text{Essentially efficient, incremental, abstract OLDT resolution algo. for CHC’s.} \]

It is the original “top-down” algorithm! [NACL’89]

• It maintains and computes as a result (simplified):
  
  ▶ **A call-answer table:** with (multiple) entries \( \{ \text{block} : \lambda_{\text{in}} \mapsto \lambda_{\text{out}} \} \).
    
  • Exit states for calls to \( \text{block} \) satisfying precond \( \lambda_{\text{in}} \) meet postcond \( \lambda_{\text{out}} \).

  ▶ **A dependency arc table:** \( \{ A : \lambda_{\text{in}A} \Rightarrow B : \lambda_{\text{in}B} \} \).
    
  • Answers for call \( A : \lambda_{\text{in}A} \) depend on the answers for \( B : \lambda_{\text{in}B} \): (if exit for \( B : \lambda_{\text{in}B} \) changes, exit for \( A : \lambda_{\text{in}A} \) 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 (\( \text{trust} \) run-time checks, external proofs, etc.)

  ▶ **Modular** (reduced working set) and **incremental** (reuse past analyses).

[NACL’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 $\text{lfp}(S_P^\alpha) = \llbracket P \rrbracket_{\alpha}$, s.t. $\llbracket P \rrbracket_{\alpha}$ safely approximates $\llbracket P \rrbracket$.

  → 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 $\{\text{block} : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to $\text{block}$ satisfying precond $\lambda_{in}$ meet postcond $\lambda_{out}$.
  - **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]
Generic framework: given \( P \) (as a set of CHCs) and abstract domain(s), computes \( \text{lfp}(S_P^\alpha) = \llbracket P \rrbracket_\alpha \), s.t. \( \llbracket P \rrbracket_\alpha \) safely approximates \( \llbracket P \rrbracket \).

→ Essentially efficient, incremental, abstract OLDT resolution algo. for CHC’s. It is the original “top-down” algorithm! [NACL’89]

It maintains and computes as a result (simplified):

- **A call-answer table**: with (multiple) entries \{\( \text{block} : \lambda_{\text{in}} \mapsto \lambda_{\text{out}} \}\).
  - Exit states for calls to \( \text{block} \) satisfying precond \( \lambda_{\text{in}} \) meet postcond \( \lambda_{\text{out}} \).

- **A dependency arc table**: \{\( A : \lambda_{\text{in}A} \Rightarrow B : \lambda_{\text{in}B} \}\).
  - Answers for call \( A : \lambda_{\text{in}A} \) depend on the answers for \( B : \lambda_{\text{in}B} \): (if exit for \( B : \lambda_{\text{in}B} \) changes, exit for \( A : \lambda_{\text{in}A} \) 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 (\textit{trust} run-time checks, external proofs, etc.)
- **Modular** (reduced working set) and **incremental** (reuse past analyses).
PLAI (CiaoPP’s Generic AI Framework)

- **Generic framework**: given $P$ (as a set of CHCs) and abstract domain(s), computes $\text{lfp}(S_P) = [P]_\alpha$, 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 $\{\text{block} : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to $\text{block}$ satisfying precond $\lambda_{in}$ meet postcond $\lambda_{out}$.
  - **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 ($\text{trust}$ 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 $\text{lfp}(S^\alpha_P) = \llbracket P \rrbracket^\alpha$, s.t. $\llbracket P \rrbracket^\alpha$ safely approximates $\llbracket P \rrbracket$.

→ Essentially efficient, incremental, abstract OLDT resolution algo. for CHC’s. It is the original “top-down” algorithm! [NACL’89]

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{\text{block} : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to block satisfying precond $\lambda_{in}$ meet postcond $\lambda_{out}$.

  - **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 ($\text{trust}$ run-time checks, external proofs, etc.)
  - Modular (reduced working set) and incremental (reuse past analyses).

[NACL’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 \( \text{lfp}(S^\alpha_P) = [P]_\alpha \), s.t. \([P]_\alpha\) safely approximates \([P]\).

\[ \rightarrow \text{Essentially efficient, incremental, abstract OLDT resolution algo. for CHC’s. It is the original “top-down” algorithm! [NACLP’89] \]

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries \( \{ \text{block} : \lambda_{in} \mapsto \lambda_{out} \} \).
    - Exit states for calls to \( \text{block} \) satisfying precond \( \lambda_{in} \) meet postcond \( \lambda_{out} \).
  - **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 $\text{lfp}(S^\alpha_P) = [P]_\alpha$, s.t. $[P]_\alpha$ safely approximates $[P]$.

  → Essentially efficient, incremental, abstract OLDT resolution algo. for CHC’s. It is the original “top-down” algorithm! [NACL’89]

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{\text{block} : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to block satisfying precond $\lambda_{in}$ meet postcond $\lambda_{out}$.
  - **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 ($\text{trust}$ run-time checks, external proofs, etc.)
  - **Modular** (reduced working set) and **incremental** (reuse past analyses).

[NACL’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, ...
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.
**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]

```c
#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]

```c
#include <stdint.h>

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;
}
```

The code snippet above demonstrates the implementation of an FIR filter with energy constraints. The `#pragma` directives are used to specify different energy usage conditions for various stages of the program. The `fir` function calculates the output of the FIR filter based on the input `xn` and coefficients `coeffs`, with state `state` and the specified number of filter elements `ELEMENTS`. The energy consumption is evaluated based on these conditions, ensuring the program meets the specified energy specifications.
Energy Usage Verification

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

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

CiaoPP

- Front-end
  - Transform
  - Source DB

- Static Analyzer
  - CHC DB
  - Libraries DB
  - Assertion DB

- Analysis DB

- Dynamic Annotator
  - RT safe src
  - Possible run-time error

- Static Checker
  - :- true
  - :- false
  - :- checked

- Dynamic Annotator

Report
- Warning
- Error
- Verified
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, i.e., only the parts needed) module by module until an intermodular fixpoint is reached again.

4. Recheck assertions/Reoptimize.
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.
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.**
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.
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.
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.
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
Modular and Incremental Analysis: Initial Analysis

lib planner

planner

lib
Modular and Incremental: Changes Detected

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

```
planner.pl
100  %%
101  - explore(P,Map,[P|Map]) :-
102  -   safe(P).
103  %

lib.pl
41  %%
42  + add(Node,Graph) :-
43  +   %%% implementation
44  +   %%% implementation
45  %
```

planner

lib
Modular and Incremental: Changes Detected
Modular and Incremental: Annotate/Remove Outdated Parts

- Planner
- Lib
- Recompute
- Delete
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.
**Theorem 4** (Correctness of IncAnalyze starting from a partial analysis). Let \( P \) be a program, \( Q_\alpha \) a set of abstract queries, and \( \mathcal{A}_0 \) any analysis graph. Let \( \mathcal{A} = \text{IncAnalyze}(P, Q_\alpha, \emptyset, \mathcal{A}_0) \). \( \mathcal{A} \) is correct for \( P \) and \( \gamma(Q_\alpha) \) if for all concrete queries \( q \in \gamma(Q_\alpha) \) all nodes \( n \) from which there is a path in the concrete execution \( q \xrightarrow{} n \) in \( [P]_Q \), that are abstracted in the analysis \( \mathcal{A}_0 \) are included in \( Q_\alpha \), i.e.:
\[
\forall Q, n. Q \in \gamma(Q_\alpha) \land q \xrightarrow{} n \in [P]_Q,
\]
\[
\forall n_\alpha \in \mathcal{A}_0. n \in \gamma(n_\alpha) \Rightarrow n_\alpha \in Q_\alpha.
\]

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

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

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

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

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

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

**Theorem 10** (Correctness of ModIncAnalyze from scratch). Let \( P \) be a modular program, and \( Q_\alpha \) a set of abstract queries. Then, if:
\[
\{\mathcal{G}, \{\mathcal{L}_M\}\} = \text{ModIncAnalyze}(P, Q_\alpha, \emptyset, \emptyset)
\]
\( \mathcal{G} \) is correct for \( P \) and \( \gamma(Q_\alpha) \).

**Lemma 4** (Precision updating \( \mathcal{L} \) modulo \( \mathcal{G} \)). Let \( M \) be a module contained in program \( P \), \( E \) a set of entries. Let \( \mathcal{G} \) be a previous state of the global analysis graph, if \( \mathcal{L}_M = \text{LocIncAnalyze}(M, E, \mathcal{G}, \emptyset, \emptyset) \). If \( \mathcal{G} \) changes to \( \mathcal{G}' \) the analysis result:
\[
\text{LocIncAnalyze}(M, E, \mathcal{G}', \mathcal{L}_M, \emptyset) = \text{LocIncAnalyze}(M, E, \mathcal{G}', \emptyset, \emptyset)
\]
is the same as analyzing from scratch, i.e., the lfp of \( M, E \).

**Theorem 11** (Precision of ModIncAnalyze from scratch). Let \( P \) be a modular program and \( Q_\alpha \) a set of abstract queries. The analysis result
\[
\mathcal{A} = \text{ModIncAnalyze}(P, Q_\alpha, \emptyset, \emptyset) = \text{ModAnalyze}(P, Q_\alpha)
\]
such that \( \mathcal{A} = \{\mathcal{G}, \{\mathcal{L}_M\}\} \), then \( \mathcal{G} = \mathcal{G}' \).

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

**Theorem 4** (Correctness of IncAnalyze starting from a partial analysis). Let $P$ be a program, $Q_\alpha$ a set of abstract queries, and $A_0$ any analysis graph. Let $A = \text{IncAnalyze}(P, Q_\alpha, A_0)$. $A$ is correct for $P$ and $\gamma(Q_\alpha)$ if for all concrete queries $q \in \gamma(Q_\alpha)$ all nodes $n$ from which there is a path in the concrete execution $q \leadsto n$ in $\llbracket P \rrbracket_Q$, that are abstracted in the analysis $A_0$ are included in $Q_\alpha$, i.e.:

$$\forall Q, n. Q \in \gamma(Q_\alpha) \land q \leadsto n \in \llbracket P \rrbracket_Q,$$

$$\forall n_\alpha \in A. n \in \gamma(n_\alpha) \Rightarrow n_\alpha \in Q_\alpha.$$

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

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

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

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

$$L = \text{IncAnalyze}(M, E, \emptyset, L_0)$$

is correct for $M$ and $\gamma(E)$ assuming $L_0$.

**Lemma 2** (Precision of IncAnalyze modulo imported predicates). Let $M$ be a module of program $P$, $E$ a set of abstract queries. Let $L_0$ be an analysis graph such that $\forall (A, \lambda^c) \in L_0, \text{mod}(A) \in \text{imports}(M)$ if $L_0$ contains the least fixed point as defined in Theorem 6. The analysis result

$$L = \text{IncAnalyze}(M, E, \emptyset, L_0)$$

is the least program analysis graph for $M$ and $\gamma(E)$ assuming $L_0$.

**Additionally**

- **Correct over-approximation** of the semantics (also with widening).
- **But for most accurate** (lfp): no widening, or conditions on the widening.
- $\text{IncAnalyze}(P', Q_\alpha, \Delta, A_0) = \text{IncAnalyze}(P, Q_\alpha, \emptyset, \emptyset)$
- $\text{LocIncAnalyze}(M, E, G', L_M, \emptyset) = \text{LocIncAnalyze}(M, E, G, L_M, \emptyset)$

**New results for reanalyzing starting from a partial analysis.**

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

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

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

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

$$\text{ModIncAnalyze}(P', Q_\alpha, \emptyset, (\emptyset, \emptyset)) = \text{ModIncAnalyze}(P', Q_\alpha, \mathcal{A}, \Delta).$$
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× speedup w.r.t. incremental (plus memory).
- **Modular analysis from scratch** also typically improved (up to 9×).
Incremental Verification

CiaoPP

Front-end
Transform
Source DB

Static Analyzer
Analysis DB

CHC DB
Libraries DB
Assertion DB

Dynamic Annotator

:- true
:- check
:- false
:- checked

Error
Warning
Verified

RT safe src
Possible run-time error

Report

19
VeriFly: On-the-fly Verification/IDE Integration

IDE
- IDE support
- Verify code

src v1
src v2
src v3

Front-end
- Transform

Source DB

CiaoPP
- Static Analyzer
- Analysis DB
- CHC DB
- Libraries DB
- Assertion DB
- Static Checker

Dynamic Annotator
- RT safe src
- Possible run-time error

Report
- Warning
- Error
- Verified

IDE support
Verifly code

Transform

Δ chars1
 Δ chars2

Static Analyzer

Analysis DB

:- true

Dynamic Annotator

:- check

Static Checker

:- false

:- checked

Analysis DB

:- check

:- false

:- checked

Error

Verified

Possible run-time error

RT safe src

Warning

Error

Verified

Dynamic Annotator

Source DB

IDE

Front-end

Transform

Δ chars1
 Δ chars2

Static Analyzer

Analysis DB

:- true

Dynamic Annotator

:- check

Static Checker

:- false

:- checked

Analysis DB

:- check

:- false

:- checked

Error

Verified

Possible run-time error

RT safe src

Warning

Error

Verified
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).
list([]).  list([H|T]) :- list(T).
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(+list,-list).}  
```

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

Defined as follows:

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

list([]).  list([H|T]) :- list(T).
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:  \[ \text{:: pred} \text{ Head} [: \text{ Pre}] [\Rightarrow \text{ Post}] [+ \text{ Comp }] . \]

\[- \text{ pred quicksort}(X,Y) : \text{ list}(\text{int}) * \text{ var} \Rightarrow \text{ sorted}(Y) + (\text{is_det}, \text{not_fails}). \]
\[- \text{ pred quicksort}(X,Y) : \text{ var} * \text{ list}(\text{int}) \Rightarrow \text{ ground}(X) + \text{ non_det}. \]

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

color(green).  color(blue).  color(red).  \text{color} := \text{green} | \text{blue} | \text{red}.
\text{list}([]).  \text{list}([H|T]) :: \text{list}(T).
\text{list}(_,[]).  \text{list}(P,[H|T]) :: X(H),\text{list}(P,T).
\text{list}(X) := [] | [\text{X}|\text{list}].
\text{sorted} := [] | [\_].  \text{sorted}([X,Y|Z]) :: X=<Y, \text{sorted}([Y|Z]).

Modes (are essentially “assertion macros”):

\[- \text{ pred qs}(+,-). \quad \rightarrow \quad \text{ pred qs}(X,Y) : (\text{nonvar}(X), \text{var}(Y)). \]
\[- \text{ pred qs}(+\text{list},-\text{list}). \quad \rightarrow \quad \text{ pred qs}(X,Y) : (\text{list}(X), \text{var}(Y)) \Rightarrow \text{list}(Y). \]

Defined as follows:

\[- \text{ modedef} +\text{A} : \text{nonvar}(\text{A}) \quad \rightarrow \quad \text{ modedef} +\text{A,X} : \text{X}(A). \]
\[- \text{ modedef} -\text{A} : \text{var}(\text{A}) \quad \rightarrow \quad \text{ modedef} -\text{A,X} : \text{var}(\text{A}) \Rightarrow \text{X}(A). \]

Program-point Assertions:

Also tests, documentation, ...
The Assertion Language

### Assertions:

```prolog
:- pred Head [: Pre] [=> Post] [+ Comp ] .
```

```prolog
:- 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, ...):

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

```prolog
sorted := [] | [X].
```

```prolog
sorted([X,Y|Z]) :- X=<Y, sorted([Y|Z]).
```

### Modes (are essentially “assertion macros”):

```prolog
:- 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:

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

### Program-point Assertions:

...,

```prolog
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,_,Cin,C,_,Z),
  collect_info(B,_,N,C,Cout,_,M).
```
Interactive Verification in the Browser (Static Error Flagged)

playground (on-the-fly repeating last action)

```prolog
:- module(_, [p/1, colorlist/1, sorted/1, color/1], [assertions, regtypes, f]).

:- pred p(X) :- sorted(X).

False assertion:
:- check success p(X)
    => sorted(X).

because the success field is incompatible with inferred success:
[eterns] rt27(X)
with:

:- regtype rt27/1.
rt27(red).
```

{NOTE (ctchecks_pred_messages): (Lns 5-6) Verified assertion:
:- check success q(X)
    => color(X).
}

{In /draft.pl
WARNING (ctchecks_cc_messages): (Lns 17-17) At literal 1 could not verify assertion:
:- check calls A>B
    : ( nonvar(A), nonvar(B), arithexpression(A), arithexpression(B) ).
because on call arithmetic>>(A,B):

[eterns] basic_props:term(A), basic_props:term(B), basic_props:term(A)

}

{assertions checked in 19.0 msec.}

{ERROR (auto_interface): Errors detected. Further preprocessing aborted.}

{NOTE (analysis_stats): Assertion checking summary:
[Predicate-level] Checked: 1 (50.00%) False: 1 (50.00%) Check: 0 (0.00%) Total: 2
[Call site-level] Checked: 0 (0.00%) False: 0 (0.00%) Check: 1 (100.00%) Total: 1
}

{written file /draft_eterns_shfr_co.pl}
Interactive Verification in the Browser (All Assertions Verified)

```
:- module(_, [qsort/2], [assertions, nativeprops]).
%% Quick-sort with difference lists (constant time append)
%% Verifying various assertions
:- pred qsort(X,Y) : (ground(X), list(X), var(Y)) => ground(Y).
qsort(X,Y) :- qsort_(X,Y,T), T=[].

:- pred qsort_(X,Y,Z) : (list(X), var(Y), var(Z), indep(Y,Z)) => ground(X).
qsort_([], E, E).
qsort_([First|Rest], SmB, LgE) :-
  partition(Rest, First, Sm, Lg),
  qsort_(Sm, SmB, SmE),
  SmE=[First|LgB],
  qsort_(Lg, LgB, LgE).

:- pred partition(L,P,Lg,Sm)
  => (list(Lg), list(Sm), ground(Lg), ground(Sm)).
partition([], [], []).
partition([X|Y], F, [X|Y1], Y2) :-
  X <= Y, F,
  partition(Y, F, Y1, Y2).
partition([X|Y], F, Y1, [X|Y2]) :-
  X >= Y, F,
  partition(Y, F, Y1, Y2).
```

{NOTE (ctchecks_pred_messages): (Ins 13-15) Verified assertion:
  : ( list(X), var(Y), var(Z), indep(Y,Z) )
  => ground(X).}

{NOTE (ctchecks_pp_messages): (Ins 17-19) At literal 1 successfully
  checked assertion:
  : check calls B@<A.}

{NOTE (ctchecks_pp_messages): (Ins 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:

```prolog
1  :- pred sorted_insert(A,B,C) : (list_pair(A), num_pair(B), var(C)) => list_pair1(C).
2  
3  sorted_insert([], X, [X]).
4  sorted_insert([[X1,F1]|L1], (X,F), [(X,F), (X1,F1)|L1]) :- X =< X1.
5  sorted_insert([P|L1], X, [P|L]) :- sorted_insert(L1, X, L).
```

The output includes the following assertions:

```prolog
%% %% :- 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:

```prolog
:- 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'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)
  
  \[\text{[NACLP'89, MCC'90]}\]

- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure).
  
  \[\text{[SAS'96, TOPLAS'99]}\]

- Extending incremental analysis to exploit much better modular structure.
  
  \[\text{[ICLP'18, LOPSTR'19, TPLP'21c]}\]

- IDE integration → our VeriFly “on-the-fly” verification tool.
  
  \[\text{[NASA-FIDE21, TPLP'21b]}\]

All while:

- Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP).
  
  \[\text{[LOPSTR'07, TPLP'18, VPT'20, TPLP'21a]}\]

- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ...
  time, memory, energy, gas, ...)
  
  \[\text{[PLDI'90, ...]}\]

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)  
  \[\text{[NACLP'89, MCC'90]}\]
- Fine-grain (clause-level) incremental analysis (originally not exploiting module structure).  
  \[\text{[SAS'96, TOPLAS'00]}\]
- Extending incremental analysis to exploit much better modular structure.  
  \[\text{[ICLP'18, LOPSTR'19, TPLP'21c]}\]
- IDE integration $\rightarrow$ our VeriFly “on-the-fly” verification tool.  
  \[\text{[NASA-FIDE21, TPLP'21b]}\]

**All while:**

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

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) \([\text{NACLP'89, MCC'90}]\)
- **Fine-grain** (clause-level) **incremental analysis** (originally not exploiting module structure). \([\text{SAS'96, TOPLAS'00}]\)
- Extending incremental analysis to exploit much better modular structure. \([\text{ICLP'18, LOPSTR'19, TPLP'21c}]\)
- **IDE integration** → our **VeriFly** “on-the-fly” verification tool. \([\text{NASA-FIDE21, TPLP'21b}]\)

**All while:**

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

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:

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

All while:

<table>
<thead>
<tr>
<th>Item</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supporting multiple languages via translation to CHCs (a.k.a. Prolog/CLP).</td>
<td>LOPSTR’18, TPLP’18, VPT’18, TPLP’21a</td>
</tr>
<tr>
<td>Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...)</td>
<td>PLDI’90, ... SAS’20</td>
</tr>
</tbody>
</table>

Plus of course making Ciao Prolog even better!
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’18, TPLP’18a, TPLP’21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...). [LOPSTR’18, 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’18, TPLP′18, VPT′20, TPLP′21a]
- Covering both functional and non-functional properties (types, pointers, shapes, intervals, ... time, memory, energy, gas, ...). [LOPSTR’18, ... 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, ...).\[PLDI’90\]...

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] ...

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  Ángel Pineda
Christian Holzbaur  Kim Marriott  Enrico Pontelli

Playground at: http://ciao-lang.org/playground
## Milestones / tools

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>'88</td>
<td><strong>MA3 analyzer</strong>: memo tables (cf. OLDT resolution), practicality. <strong>Abstract compilation.</strong></td>
</tr>
<tr>
<td>'89</td>
<td><strong>PLAI analyzer</strong>: the <strong>“top-down” algorithm</strong>, “summaries”, abstract domains as plugins. Sharing (aliasing)/independence → auto-parallelization / real speedups (on shmem).</td>
</tr>
<tr>
<td>90’s</td>
<td>Incremental analysis, concurrency, automatic domain combinations, scalability.</td>
</tr>
<tr>
<td>'90-'93</td>
<td><strong>Automatic cost analysis</strong> (upper bounds), GraCoS (<strong>Granularity Control System</strong>).</td>
</tr>
<tr>
<td>'96</td>
<td>The Ciao/CiaoPP model for <strong>Assertion Verification by Abstract Interpretation</strong>.</td>
</tr>
<tr>
<td>'91-'06</td>
<td>Combined abstract interpretation and (abstract) partial evaluation.</td>
</tr>
<tr>
<td>'96-'00</td>
<td>Lower <strong>bounds</strong> cost analysis, <strong>divide-and-conquer</strong>. No-fail (no exceptions), determinacy.</td>
</tr>
<tr>
<td>'01-05</td>
<td>Modular analysis for scalability. New (imperative) shape/type domains, widenings.</td>
</tr>
<tr>
<td>'03</td>
<td>Abstraction carrying code.</td>
</tr>
<tr>
<td>'00-'05</td>
<td>Multi-language support via CLP (a.k.a. CHCs) as IR: Java, C# (shapes, resources, ...).</td>
</tr>
<tr>
<td>'04-'07</td>
<td>Verification/debugging/optimization of user-defined resources.</td>
</tr>
<tr>
<td>'06-'08</td>
<td>Verification of execution <strong>time</strong>, <strong>energy</strong> for Java, heap models, ... Probabilistic cost.</td>
</tr>
<tr>
<td>'12-18</td>
<td>(X)C binary program energy analysis/verification, ISA-level energy models.</td>
</tr>
<tr>
<td>'13-18</td>
<td>Cost analysis as <strong>Abstract Interpretation</strong> Sized shapes. LLVM. <strong>Static Profiling.</strong></td>
</tr>
<tr>
<td>'16-18</td>
<td>Semantic code search via abstract interpretation. <strong>Abstract distances.</strong></td>
</tr>
<tr>
<td>'17-19</td>
<td>Combined modular/incremental analysis (scalability), fixpoint guidance.</td>
</tr>
<tr>
<td>'19-'23</td>
<td><strong>Verify</strong>: Interactive Verification via Abstract Interpretation.</td>
</tr>
<tr>
<td>'19-'23</td>
<td>Smart contracts: verification of resources (gas, storage, commands, ...).</td>
</tr>
<tr>
<td>Year</td>
<td>Milestones / tools</td>
</tr>
<tr>
<td>------</td>
<td>-------------------</td>
</tr>
<tr>
<td>1988</td>
<td><strong>MA3 analyzer</strong>: memo tables (cf. OLDT resolution), practicality. <strong>Abstract compilation.</strong></td>
</tr>
<tr>
<td>1989</td>
<td><strong>PLAI analyzer</strong>: the “top-down” algorithm, “summaries”, abstract domains as plugins. Sharing(aliasing)/independence $\rightarrow$ auto-parallelization / real speedups (on shmem).</td>
</tr>
<tr>
<td>90’s</td>
<td><strong>Incremental analysis</strong>, concurrency, automatic domain combinations, scalability.</td>
</tr>
<tr>
<td>1990-93</td>
<td>Automatic cost analysis (upper bounds), GraCoS (<strong>Granularity Control System</strong>).</td>
</tr>
<tr>
<td>1996</td>
<td>The Ciao/CiaoPP model for Assertion Verification by Abstract Interpretation</td>
</tr>
<tr>
<td>1991-06</td>
<td>Combined abstract interpretation and (abstract) partial evaluation.</td>
</tr>
<tr>
<td>1996-00</td>
<td>Lower bounds cost analysis, <strong>divide-and-conquer</strong>. No-fail (no exceptions), determinacy.</td>
</tr>
<tr>
<td>2001-05</td>
<td>Modular analysis for scalability. New (imperative) shape/type domains, widenings.</td>
</tr>
<tr>
<td>2003</td>
<td>Abstraction carrying code.</td>
</tr>
<tr>
<td>2000-05</td>
<td>Multi-language support via CLP (a.k.a. CHCs) as IR: Java, C# (shapes, resources, ...).</td>
</tr>
<tr>
<td>2004-07</td>
<td>Verification/debugging/optimization of <strong>user-defined resources</strong>.</td>
</tr>
<tr>
<td>2006-08</td>
<td>Verification of <strong>execution time</strong>, energy for Java, heap models, ... Probabilistic cost.</td>
</tr>
<tr>
<td>2012-18</td>
<td>(X)C binary program energy analysis/verification, ISA-level energy models.</td>
</tr>
<tr>
<td>2013-18</td>
<td>Cost analysis as <strong>Abstract Interpretation</strong> Sized shapes. LLVM. <strong>Static Profiling.</strong></td>
</tr>
<tr>
<td>2016-18</td>
<td>Semantic code search via abstract interpretation. <strong>Abstract distances.</strong></td>
</tr>
<tr>
<td>2017-19</td>
<td>Combined modular/incremental analysis (scalability), fixpoint guidance.</td>
</tr>
<tr>
<td>2019-23</td>
<td>Verify: Interactive Verification via Abstract Interpretation.</td>
</tr>
<tr>
<td>2019-23</td>
<td><strong>Smart contracts</strong>: verification of resources (gas, storage, commands, ...).</td>
</tr>
</tbody>
</table>
## Milestones / tools

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

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

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

<table>
<thead>
<tr>
<th>Year</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'88</td>
<td>MA3 analyzer: memo tables (cf. OLDT resolution), practicality. <strong>Abstract compilation.</strong></td>
</tr>
<tr>
<td>90's</td>
<td><strong>Incremental analysis</strong>, concurrency, automatic domain combinations, scalability.</td>
</tr>
<tr>
<td>'90-'93</td>
<td><strong>Automatic cost analysis</strong> (upper bounds), GraCoS (<strong>Granularity Control System</strong>).</td>
</tr>
<tr>
<td>'96</td>
<td><strong>The Ciao/CiaoPP model for Assertion Verification by Abstract Interpretation</strong></td>
</tr>
<tr>
<td>'91-'06</td>
<td>Combined <strong>abstract interpretation and (abstract) partial evaluation</strong>.</td>
</tr>
<tr>
<td>'96-'00</td>
<td><strong>Lower bounds</strong> cost analysis, <strong>divide-and-conquer</strong>. No-fail (no exceptions), determinacy.</td>
</tr>
<tr>
<td>'01-05</td>
<td><strong>Modular analysis</strong> for scalability. New (imperative) shape/type domains, widenings.</td>
</tr>
<tr>
<td>'03</td>
<td><strong>Abstraction carrying code</strong>.</td>
</tr>
<tr>
<td>'00-'05</td>
<td><strong>Multi-language support via CLP (a.k.a. CHCs) as IR</strong>: Java, C# (shapes, resources, ...).</td>
</tr>
<tr>
<td>'04-'07</td>
<td>Verification/debugging/optimization of user-defined resources.</td>
</tr>
<tr>
<td>'06-'08</td>
<td>Verification of execution time, energy for Java, heap models, ... Probabilistic cost.</td>
</tr>
<tr>
<td>'12-18</td>
<td>(X)C binary program energy analysis/verification, ISA-level energy models.</td>
</tr>
<tr>
<td>'13-18</td>
<td>Cost analysis as Abstract Interpretation sized shapes. LLVM. <strong>Static Profiling</strong>.</td>
</tr>
<tr>
<td>'17-19</td>
<td>Combined modular/incremental analysis (scalability), fixpoint guidance.</td>
</tr>
<tr>
<td>'19-'23</td>
<td>Verify: Interactive Verification via Abstract Interpretation.</td>
</tr>
<tr>
<td>'19-'23</td>
<td>Smart contracts: verification of resources (gas, storage, commands, ...).</td>
</tr>
</tbody>
</table>
## Milestones / tools

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

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

```c
#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]

```c
#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]

```c
#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)

<table>
<thead>
<tr>
<th>Contract</th>
<th>Metrics</th>
<th>Resource A.</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Parameter ((\alpha))</td>
<td>Storage ((\beta))</td>
<td>gas</td>
</tr>
<tr>
<td>reverse</td>
<td>length</td>
<td>length</td>
<td>(\alpha)</td>
</tr>
<tr>
<td>addition</td>
<td>value</td>
<td>value</td>
<td>(\log \alpha)</td>
</tr>
<tr>
<td>michelson_arith</td>
<td>value</td>
<td>value</td>
<td>(\log (\alpha^2 + 2 \times \beta))</td>
</tr>
<tr>
<td>bytes</td>
<td>value</td>
<td>length</td>
<td>(\beta)</td>
</tr>
<tr>
<td>list_inc</td>
<td>value</td>
<td>length</td>
<td>(\beta)</td>
</tr>
<tr>
<td>lambda</td>
<td>value</td>
<td>value</td>
<td>(\log \alpha)</td>
</tr>
<tr>
<td>lambda_apply</td>
<td>(value, size)</td>
<td>size</td>
<td>(k)</td>
</tr>
<tr>
<td>inline</td>
<td>size</td>
<td>value</td>
<td>(\log \beta)</td>
</tr>
<tr>
<td>cross_product</td>
<td>(length, length)</td>
<td>value</td>
<td>(\alpha_1 + \alpha_2)</td>
</tr>
<tr>
<td>lineal</td>
<td>value</td>
<td>value</td>
<td>(\alpha)</td>
</tr>
<tr>
<td>assertion_map</td>
<td>(value, size)</td>
<td>length</td>
<td>(\log \beta \times \log \alpha_1)</td>
</tr>
<tr>
<td>Contract</td>
<td>Metrics</td>
<td>Resource A.</td>
<td>Time (ms)</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------</td>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td></td>
<td>Parameter ((\alpha))</td>
<td>Storage ((\beta))</td>
<td>gas</td>
</tr>
<tr>
<td>quadratic</td>
<td>length</td>
<td>length</td>
<td>(\alpha \times \beta)</td>
</tr>
<tr>
<td>queue</td>
<td>size</td>
<td>(value, size, length)</td>
<td>(\log \beta_1 \times \log \beta_3)</td>
</tr>
<tr>
<td>king_of_tez</td>
<td>size</td>
<td>(value, value, size)</td>
<td>(k)</td>
</tr>
<tr>
<td>set_management</td>
<td>length</td>
<td>length</td>
<td>(\alpha \times \log \beta)</td>
</tr>
<tr>
<td>lock</td>
<td>size</td>
<td>(value, value, size)</td>
<td>(k)</td>
</tr>
<tr>
<td>max_list</td>
<td>length</td>
<td>size</td>
<td>(\alpha)</td>
</tr>
<tr>
<td>zipper</td>
<td>length</td>
<td>(length, length, length)</td>
<td>(k)</td>
</tr>
<tr>
<td>auction</td>
<td>size</td>
<td>(value, value, size)</td>
<td>(k)</td>
</tr>
<tr>
<td>union</td>
<td>(length, length)</td>
<td>length</td>
<td>(\alpha_1 \times \log \alpha_2)</td>
</tr>
<tr>
<td>append</td>
<td>(length, length)</td>
<td>length</td>
<td>(\alpha_1)</td>
</tr>
<tr>
<td>subset</td>
<td>(length, length)</td>
<td>size</td>
<td>(\alpha_1 \times \log \alpha_2)</td>
</tr>
</tbody>
</table>
Some related CiaoPP references
Interactive verification


VeriFly: On-the-fly Assertion Checking via Incrementality.


VeriFly: On-the-fly Assertion Checking with CiaoPP.


Multivariant Assertion-based Guidance in Abstract Interpretation.

Horn Clauses as Intermediate Representation / Multi-Language Support


A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs.


From big-step to small-step semantics and back with interpreter specialization (invited paper).


Analysis and Transformation of Constrained Horn Clauses for Program Verification.
Scalability/Modularity/Incrementality in Analysis/Specialization/Verification


The Basic Analysis Framework (Abstract Interpreter, Fixpoint)

A General Framework for Static Cost Analysis of Parallel Logic Programs.

An Efficient, Parametric Fixpoint Algorithm for Analysis of Java Bytecode.

Exploiting Goal Independence in the Analysis of Logic Programs.

Improving Abstract Interpretations by Combining Domains.

Global Analysis of Constraint Logic Programs.
ACM Trans. on Programming Languages and Systems, 18(5):564–615, 1996.

Analyzing Logic Programs with Dynamic Scheduling.

Compile-time Derivation of Variable Dependency Using Abstract Interpretation.
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.

On the Practicality of Global Flow Analysis of Logic Programs
Semantic Code Search


Semantic Code Browsing.

Abstraction-Carrying Code


Abstraction Carrying Code and Resource-Awareness.


Abstraction-Carrying Code.


Other Abstract Interpretation-Related Techniques

[LOPSTR’20] Ignacio Casso, José F. Morales, Pedro López-García, Manuel V. Hermenegildo.

Testing Your (Static Analysis) Truths.


Computing Abstract Distances in Logic Programs.
Abstract Domains: Sharing/Aliasing

Identification of Logically Related Heap Regions.

Efficient Set Sharing using ZBDDs.
In 21st Int’l. WS on Languages and Compilers for Parallel Computing (LCPC’08), LNCS. Springer-Verlag, August 2008.

Negative Ternary Set-Sharing.

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.

Sharing Analysis of Arrays, Collections, and Recursive Structures.

Precise Set Sharing Analysis for Java-style Programs.

Efficient top-down set-sharing analysis using cliques.
Abstract Domains: Shape/Type Analysis

Sized Type Analysis of Logic Programs (Technical Communication).  

Efficient context-sensitive shape analysis with graph-based heap models.  

Heap Analysis in the Presence of Collection Libraries.  

More Precise yet Efficient Type Inference for Logic Programs.  
Abstract Domains: Non-failure, Determinacy

Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information.

Determinacy Analysis for Logic Programs Using Mode and Type Information.

Multivariant Non-Failure Analysis via Standard Abstract Interpretation.

Non-Failure Analysis for Logic Programs.
Analysis and Verification of Energy

Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption.

Inferring Energy Bounds Statically by Evolutionary Analysis of Basic Blocks.

Inferring Parametric Energy Consumption Functions at Different SW Levels: ISA vs. LLVM IR.

Towards Energy Consumption Verification via Static Analysis.

Energy Consumption Analysis of Programs based on XMOS ISA-Level Models.

Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications.
Analysis and Verification of Resources in General


A Transformational Approach to Parametric Accumulated-cost Static Profiling.

Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types.

Sized Type Analysis of Logic Programs (Technical Communication).

Interval-Based Resource Usage Verification: Formalization and Prototype
**A Framework for Verification and Debugging of Resource Usage Properties.**  

**User-Definable Resource Usage Bounds Analysis for Java Bytecode.**  

**Towards Execution Time Estimation in Abstract Machine-Based Languages.**  

**User-Definable Resource Bounds Analysis for Logic Programs.**  
**ICLP 2017 10-year Test of Time Award.**

**Probabilistic Cost Analysis of Logic Programs: A First Case Study.**  
Lower Bound Cost Estimation for Logic Programs.

A Methodology for Granularity Based Control of Parallelism in Logic Programs.

[ICLP’95] P. López-García and M. Hermenegildo.
Efficient Term Size Computation for Granularity Control.

Estimating the Computational Cost of Logic Programs.

Towards Granularity Based Control of Parallelism in Logic Programs.

Task Granularity Analysis in Logic Programs.
Application of the CiaoPP framework to Smart Contracts

Cost Analysis of Smart Contracts via Parametric Resource Analysis.

Some applications of the CiaoPP framework to Java bytecode

Precise Set Sharing Analysis for Java-style Programs.

An Efficient, Context and Path Sensitive Analysis Framework for Java Programs.

An Efficient, Parametric Fixpoint Algorithm for Analysis of Java Bytecode.

User-Definable Resource Usage Bounds Analysis for Java Bytecode.