

Hiord[#]

An Approach to the Specification and Verification of Higher-Order (C)LP Programs

Marco Ciccalè^{1,2} Daniel Jurjo-Rivas^{1,2} José F. Morales^{1,2}
Pedro López-García^{2,3} Manuel V. Hermenegildo^{1,2}

¹Universidad Politécnica de Madrid (UPM)

²IMDEA Software Institute

³Spanish Council for Scientific Research (CSIC)



ICLP'25 – September 18, 2025

Motivation

How to **express**, **verify**, and **infer higher-order** properties using a **first-order** assertion language, and **static analyzer/verifier**?

Motivation

How to **express**, **verify**, and **infer higher-order** properties using a **first-order** assertion language, and **static analyzer/verifier**?

(in the context of (C)LP)

Traditional assertions

- **Partial specifications** for predicates: types/shapes, sharing, modes, ...
- Express **conditions** on the **state** during execution.
- Built from **property formulas** with native/user-defined **properties**.

Traditional assertions

- **Partial specifications** for predicates: types/shapes, sharing, modes, ...
- Express **conditions** on the **state** during execution.
- Built from **property formulas** with native/user-defined **properties**.

`:- pred Pred : Pre => Post .`

Traditional assertions

- **Partial specifications** for predicates: types/shapes, sharing, modes, ...
- Express **conditions** on the **state** during execution.
- Built from **property formulas** with native/user-defined **properties**.

`:- pred Pred : Pre => Post .`

Example

The user can define properties as regular predicates (but marked accordingly):

```
:- prop lst/1.  
lst([]).  
lst([_ | Xs]) :- lst(Xs).
```

Traditional assertions

- **Partial specifications** for predicates: types/shapes, sharing, modes, ...
- Express **conditions** on the **state** during execution.
- Built from **property formulas** with native/user-defined **properties**.

```
:- pred Pred : Pre => Post.
```

Example

The user can define properties as regular predicates (but marked accordingly):

```
:- prop lst/1.  
lst([]).  
lst([_|Xs]) :- lst(Xs).
```

Which can then be used in the assertion of another predicate:

```
:- pred qsort(Xs, Ys) : lst(Xs) => lst(Ys).
```

Assertion conditions

Given a set of assertions \mathcal{A} for a predicate $Pred$:

:- pred $Pred$: $Pre_1 \Rightarrow Post_1$.

...

:- pred $Pred$: $Pre_n \Rightarrow Post_n$.

We **normalize** them to a set of **assertion conditions** $\{C_0, C_1, \dots, C_n\}$, with

$$C_i = \begin{cases} \text{calls}(Pred, \bigvee_{j=1}^n Pre_j) & i = 0 \\ \text{success}(Pred, Pre_i, Post_i) & i \in 1..n \end{cases}$$

Assertion conditions

Given a set of assertions \mathcal{A} for a predicate $Pred$:

:- pred $Pred$: $Pre_1 \Rightarrow Post_1$.

...

:- pred $Pred$: $Pre_n \Rightarrow Post_n$.

We **normalize** them to a set of **assertion conditions** $\{C_0, C_1, \dots, C_n\}$, with

$$C_i = \begin{cases} \text{calls}(Pred, \bigvee_{j=1}^n Pre_j) & i = 0 \\ \text{success}(Pred, Pre_i, Post_i) & i \in 1..n \end{cases}$$

Example

Consider the assertion before: “**:- pred qsort**(Xs, Ys) : **lst**(Xs) => **lst**(Ys).”

Its set of assertion conditions is:

calls(**qsort**(Xs, Ys), **lst**(Xs))
success(**qsort**(Xs, Ys), **lst**(Xs), **lst**(Ys))

Semantics with assertions

A **state** $\langle L :: G \mid \theta \mid \mathcal{E} \rangle$ where: L is a **literal**,

Semantics with assertions

A **state** $\langle L :: G \mid \theta \mid \mathcal{E} \rangle$ where: L is a **literal**, G is a **goal**,

Semantics with assertions

A **state** $\langle L :: G \mid \theta \mid \mathcal{E} \rangle$ where: L is a **literal**, G is a **goal**, θ is a **constraint store**,

Semantics with assertions

A **state** $\langle L :: G \mid \theta \mid \mathcal{E} \rangle$ where: L is a **literal**, G is a **goal**, θ is a **constraint store**, and \mathcal{E} is a set of **false assertion conditions**; can be reduced using the following **rules**:

Semantics with assertions

A state $\langle L :: G \mid \theta \mid \mathcal{E} \rangle$ where: L is a **literal**, G is a **goal**, θ is a **constraint store**, and \mathcal{E} is a set of **false assertion conditions**; can be reduced using the following **rules**:

$$\text{CONSTR}_{\mathcal{A}} \quad L \text{ is a constraint}$$

$$\langle L :: G \mid \theta \mid \emptyset \rangle \rightsquigarrow_{\mathcal{A}} \langle G \mid \theta \wedge L \mid \emptyset \rangle \quad \text{if } \theta \wedge L \neq \text{ff}$$

Semantics with assertions

A state $\langle L :: G \mid \theta \mid \mathcal{E} \rangle$ where: L is a **literal**, G is a **goal**, θ is a **constraint store**, and \mathcal{E} is a set of **false assertion conditions**; can be reduced using the following **rules**:

CONSTR _{\mathcal{A}} L is a constraint

$\langle L :: G \mid \theta \mid \emptyset \rangle \rightsquigarrow_{\mathcal{A}} \langle G \mid \theta \wedge L \mid \emptyset \rangle$ if $\theta \wedge L \not\models \text{ff}$

HO-LIT _{\mathcal{A}} L is a higher-order literal of the form $X(\bar{t})$

$\langle L :: G \mid \theta \mid \emptyset \rangle \rightsquigarrow_{\mathcal{A}} \langle p(\bar{t}) :: G \mid \theta \mid \emptyset \rangle$ if $\exists p \in \mathbf{P} : \theta \models (X = p) \wedge \text{ar}(p) = |\bar{t}|$

Semantics with assertions (cont'd)

ATOM _{\mathcal{A}} L is an atom of the form $p(\bar{t})$, and $\exists(L \leftarrow B) \in \text{defn}(L)$

$$\langle L :: G \mid \theta \mid \emptyset \rangle \rightsquigarrow_{\mathcal{A}} \begin{cases} \langle G \mid \theta \mid \{\ell\} \rangle & \text{if } \exists C = \text{calls}(L, \text{Pre}) \in \mathcal{A} : \\ & \text{label}(C) = \ell \wedge \theta \not\Rightarrow \text{Pre} \\ \langle B :: \text{PostC} :: G \mid \theta \mid \emptyset \rangle & \text{otherwise,} \end{cases}$$

where PostC is the sequence of **check literals** $\text{check}(L, \ell_1) :: \dots :: \text{check}(L, \ell_n)$ such that each $\ell_i = \text{label}(C_i)$, where $C_i = \text{success}(L, \text{Pre}_i, \text{Post}_i) \in \mathcal{A}$, and $\theta \Rightarrow \text{Pre}_i$

Semantics with assertions (cont'd)

ATOM _{\mathcal{A}} L is an atom of the form $p(\bar{t})$, and $\exists(L \leftarrow B) \in \text{defn}(L)$

$$\langle L :: G | \theta | \emptyset \rangle \rightsquigarrow_{\mathcal{A}} \begin{cases} \langle G | \theta | \{\ell\} \rangle & \text{if } \exists C = \text{calls}(L, \text{Pre}) \in \mathcal{A} : \\ & \text{label}(C) = \ell \wedge \theta \not\Rightarrow \text{Pre} \\ \langle B :: \text{Post} :: G | \theta | \emptyset \rangle & \text{otherwise,} \end{cases}$$

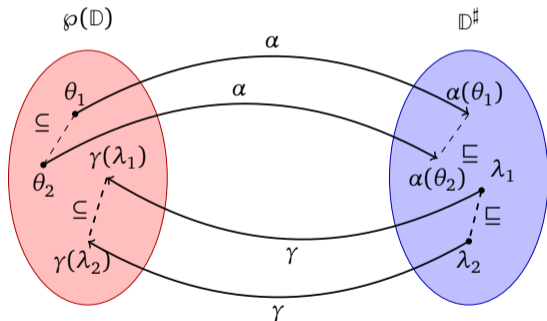
where Post is the sequence of **check literals** $\text{check}(L, \ell_1) :: \dots :: \text{check}(L, \ell_n)$ such that each $\ell_i = \text{label}(C_i)$, where $C_i = \text{success}(L, \text{Pre}_i, \text{Post}_i) \in \mathcal{A}$, and $\theta \Rightarrow \text{Pre}_i$

CHECK _{\mathcal{A}} L is a check literal $\text{check}(L', \ell)$

$$\langle L :: G | \theta | \emptyset \rangle \rightsquigarrow_{\mathcal{A}} \begin{cases} \langle G | \theta | \{\ell\} \rangle & \text{if } \exists C = \text{success}(L', _, \text{Post}) \in \mathcal{A} : \\ & \text{label}(C) = \ell \wedge \theta \not\Rightarrow \text{Post} \\ \langle G | \theta | \emptyset \rangle & \text{otherwise} \end{cases}$$

Abstract interpretation [Cousot '77]

Theory for constructing (*sound by construction*) static analysis tools to **prove** (concrete) **semantic properties** of a program **P**, by interpreting it over a (simpler) **abstract domain** that approximates the concrete semantics of **P**.



concrete domain $\langle \wp(\mathbb{D}), \sqsubseteq, \cup, \cap, \mathbb{D}, \emptyset \rangle \xrightleftharpoons[\alpha]{\gamma} \langle \mathbb{D}^\sharp, \sqsubseteq, \cup, \cap, \top, \perp \rangle$ abstract domain

Abstract interpretation of (C)LP programs

- We perform **top-down** abstract interpretation, where the **concrete domain** $\wp(\mathbb{D})$ represents **sets of variable substitutions/constraints** – *i.e.*, the characterization of any possible semantic property P of the program ($P \in \wp(\mathbb{D})$).

Abstract interpretation of (C)LP programs

- We perform **top-down** abstract interpretation, where the **concrete domain** $\wp(\mathbb{D})$ represents **sets of variable substitutions/constraints** – *i.e.*, the characterization of any possible semantic property P of the program ($P \in \wp(\mathbb{D})$).
- Given an abstract domain \mathbb{D}^\sharp , and a set of **abstract queries** \mathcal{Q}^\sharp s.t. $\gamma(\mathcal{Q}^\sharp) \supseteq \mathcal{Q}$, we denote the **abstract interpretation** of \mathbf{P} under \mathbb{D}^\sharp from \mathcal{Q}^\sharp by:

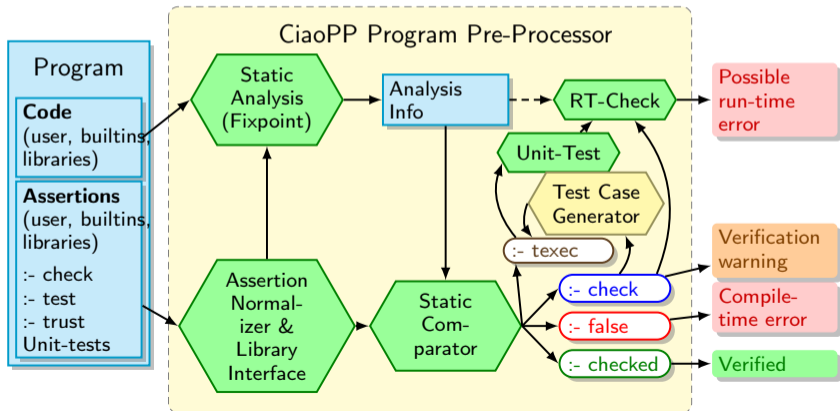
$$\llbracket \mathbf{P} \rrbracket_{\mathcal{Q}^\sharp}^\sharp = \{ \langle Pred_1, \lambda_1^c, \lambda_1^s \rangle, \dots, \langle Pred_n, \lambda_n^c, \lambda_n^s \rangle \}$$

where:

- $Pred_i$ represents a predicate, and
- λ_i^c (resp., λ_i^s) $\in \mathbb{D}^\sharp$ is an *abstraction* which **over-approximates** all **call** (resp., **success**) states of $Pred_i$ from the set of (concrete) queries $\gamma(\mathcal{Q}^\sharp)$.

CiaoPP program pre-processor [Herme. et al. '05]

- Performs **top-down abstract interpretation** (using PLAI [Muthu. et al. '90]).
- Compute the **abstract semantics** of a program **P** under abstract domain(s) $\mathbb{D}^\#$.
- Statically **checks** assertions \mathcal{A} : **checked/false/check** for each $A \in \mathcal{A}$.



Predicate properties

New kind of **property** to express conditions on **predicates passed as arguments to other predicates**, *i.e.*, variables bound to predicate symbols; defined as a set of assertions:

$$\begin{aligned} \Pi & := \{ \\ & \quad :- \text{pred } _(\bar{v}) : Pre_1 \Rightarrow Post_1 . \\ & \quad \dots \\ & \quad :- \text{pred } _(\bar{v}) : Pre_n \Rightarrow Post_n . \\ & \} . \end{aligned}$$

Predicate properties

New kind of **property** to express conditions on **predicates passed as arguments to other predicates**, *i.e.*, variables bound to predicate symbols; defined as a set of assertions:

$$\begin{aligned} \Pi := \{ & \\ & :- \text{pred } _(\bar{v}) : Pre_1 \Rightarrow Post_1 . \\ & \dots \\ & :- \text{pred } _(\bar{v}) : Pre_n \Rightarrow Post_n . \\ & \} . \end{aligned}$$

Example

We define a predicate property representing (non-det) integer binary operators:

$$\text{int_op} := \{ :- \text{pred } _(X, Y, Z) : (\text{int}(X), \text{int}(Y)) \Rightarrow \text{int}(Z) . \} .$$

Predicate properties

New kind of **property** to express conditions on **predicates passed as arguments to other predicates**, *i.e.*, variables bound to predicate symbols; defined as a set of assertions:

$$\begin{aligned} \Pi & := \{ \\ & \quad :- \text{pred } _(\bar{v}) : Pre_1 \Rightarrow Post_1 . \\ & \quad \dots \\ & \quad :- \text{pred } _(\bar{v}) : Pre_n \Rightarrow Post_n . \\ & \} . \end{aligned}$$

Example

We define a predicate property representing (non-det) integer binary operators:

$$\text{int_op} := \{ :- \text{pred } _(X, Y, Z) : (\text{int}(X), \text{int}(Y)) \Rightarrow \text{int}(Z) . \} .$$

Which we then use in the following (partial) specification of an evaluation predicate:

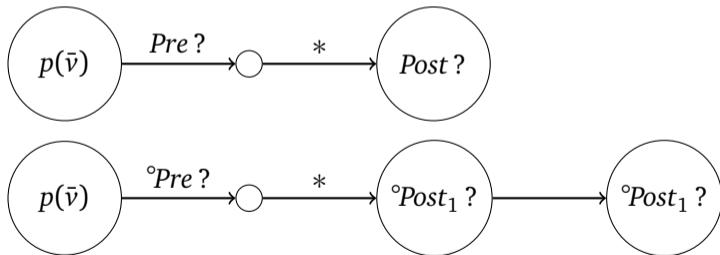
$$:- \text{pred eval}(A, B, P, R) : (\text{int}(A), \text{int}(B), \text{int_op}(P)) \Rightarrow \text{int}(R) .$$

Conformance (informally)

To check if a predicate p **conforms** to a predicate property Π , consider deriving a query Q_p to p with the set of assertions \mathcal{A} :

$$\mathcal{A} = \{\text{calls}(p(\bar{v}), \text{Pre}), \text{success}(p(\bar{v}), \text{Pre}, \text{Post})\}$$

$$\Pi = \{\text{calls}(_(\bar{v}), \text{Pre}), \text{success}(_(\bar{v}), \text{Pre}, \text{Post}_1), \text{success}(_(\bar{v}), \text{Pre}, \text{Post}_2)\}$$



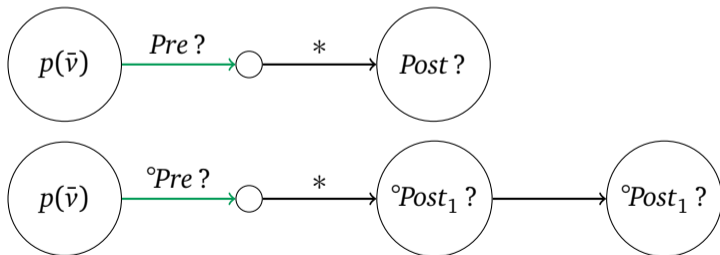
- Anonymous assertions in Π should not add new run-time check errors.

Conformance (informally)

To check if a predicate p **conforms** to a predicate property Π , consider deriving a query Q_p to p with the set of assertions \mathcal{A} :

$$\mathcal{A} = \{\text{calls}(p(\bar{v}), \text{Pre}), \text{success}(p(\bar{v}), \text{Pre}, \text{Post})\}$$

$$\Pi = \{\text{calls}(_(\bar{v}), \text{Pre}), \text{success}(_(\bar{v}), \text{Pre}, \text{Post}_1), \text{success}(_(\bar{v}), \text{Pre}, \text{Post}_2)\}$$



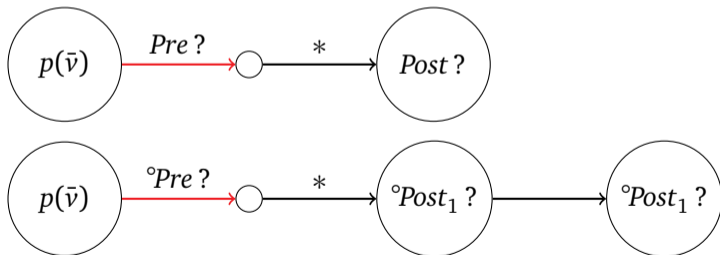
- If Pre **succeeds** to be checked, then Pre should **succeed** to be checked.

Conformance (informally)

To check if a predicate p **conforms** to a predicate property Π , consider deriving a query Q_p to p with the set of assertions \mathcal{A} :

$$\mathcal{A} = \{\text{calls}(p(\bar{v}), \text{Pre}), \text{success}(p(\bar{v}), \text{Pre}, \text{Post})\}$$

$$\Pi = \{\text{calls}(_(\bar{v}), \text{°Pre}), \text{success}(_(\bar{v}), \text{°Pre}, \text{°Post}_1), \text{success}(_(\bar{v}), \text{°Pre}, \text{°Post}_2)\}$$



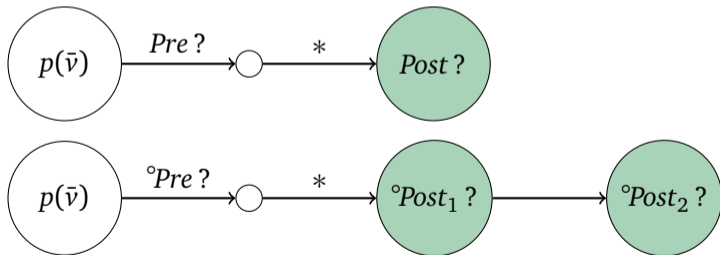
- If Pre **fails** to be checked, then $°Pre$ should **fail** to be checked.

Conformance (informally)

To check if a predicate p **conforms** to a predicate property Π , consider deriving a query Q_p to p with the set of assertions \mathcal{A} :

$$\mathcal{A} = \{\text{calls}(p(\bar{v}), \text{Pre}), \text{success}(p(\bar{v}), \text{Pre}, \text{Post})\}$$

$$\Pi = \{\text{calls}(_(\bar{v}), \text{Pre}), \text{success}(_(\bar{v}), \text{Pre}, \text{Post}_1), \text{success}(_(\bar{v}), \text{Pre}, \text{Post}_2)\}$$



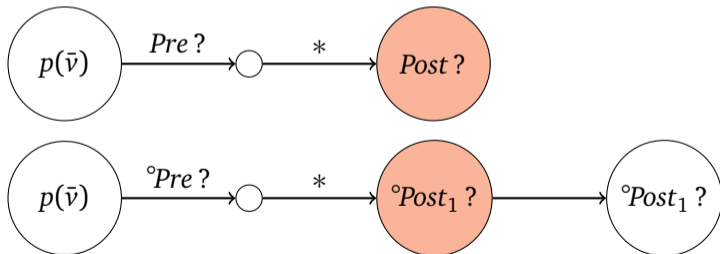
- If $Post$ **succeeds** to be checked, then $Post_1$ and $Post_2$ should **succeed** to be checked.

Conformance (informally)

To check if a predicate p **conforms** to a predicate property Π , consider deriving a query Q_p to p with the set of assertions \mathcal{A} :

$$\mathcal{A} = \{\text{calls}(p(\bar{v}), \text{Pre}), \text{success}(p(\bar{v}), \text{Pre}, \text{Post})\}$$

$$\Pi = \{\text{calls}(_(\bar{v}), \text{Pre}), \text{success}(_(\bar{v}), \text{Pre}, \text{Post}_1), \text{success}(_(\bar{v}), \text{Pre}, \text{Post}_2)\}$$



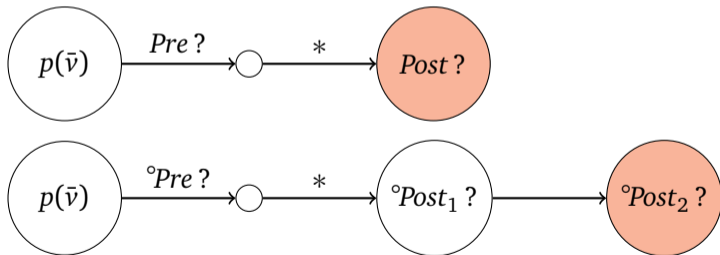
- If $Post$ **fails** to be checked, then either $Post_1$ or $Post_2$ should **fail** to be checked.

Conformance (informally)

To check if a predicate p **conforms** to a predicate property Π , consider deriving a query Q_p to p with the set of assertions \mathcal{A} :

$$\mathcal{A} = \{\text{calls}(p(\bar{v}), \text{Pre}), \text{success}(p(\bar{v}), \text{Pre}, \text{Post})\}$$

$$\Pi = \{\text{calls}(_(\bar{v}), \text{Pre}), \text{success}(_(\bar{v}), \text{Pre}, \text{Post}_1), \text{success}(_(\bar{v}), \text{Pre}, \text{Post}_2)\}$$



- If $Post$ **fails** to be checked, then either Post_1 or Post_2 should **fail** to be checked.

Conformance (formally)

Definition (Conformance)

Given the set \mathcal{Q}_p of **all possible queries** to p ,

$$p \prec \Pi \Leftrightarrow \forall Q_p \in \mathcal{Q}_p : \Pi \text{ is redundant for } p$$

$$p \not\prec \Pi \Leftrightarrow \exists Q_p \in \mathcal{Q}_p : \Pi \text{ is not redundant for } p$$

Definition (Redundance)

Assuming p is **covered** with Π , and $\mathcal{A}' \triangleq \{\text{calls}(p(\bar{v}), \text{Pre} \wedge \circ\text{Pre})\} \cup (\mathcal{A} \setminus \{C\}) \cup (\Pi \setminus \{C\})|_p$, we say that Π is **redundant** for p under Q_p iff

$$\forall D' \in \text{derivs}_{\mathcal{A}'}(\mathbf{P}, Q_p) : D'_{[-1]} = \langle G' \mid \theta \mid \{c'\} \rangle,$$

and

$$\forall D \in \text{derivs}_{\mathcal{A}}(\mathbf{P}, Q_p) : D \equiv D',$$

it holds that $D_{[-1]} \rightsquigarrow_{\mathcal{A}}^* \langle G \mid \theta \mid \{c\} \rangle$ through a derivation that reduces only check literals.

Abstract conformance

- Conformance is **not computable** in general, since the set \mathcal{Q}_p of all possible queries to a predicate p may be infinite (e.g., the **append**/3 predicate).

Abstract conformance

- Conformance is **not computable** in general, since the set \mathcal{Q}_p of all possible queries to a predicate p may be infinite (*e.g.*, the **append**/3 predicate).
- Thus, we propose **abstract conformance** as a **compile-time** alternative criterion which safely approximates the notion of conformance.

Abstract conformance

- Conformance is **not computable** in general, since the set \mathcal{Q}_p of all possible queries to a predicate p may be infinite (e.g., the **append**/3 predicate).
- Thus, we propose **abstract conformance** as a **compile-time** alternative criterion which safely approximates the notion of conformance.
- It does so by comparing the assertions \mathcal{A} of p with those of Π over the **order relation** of the **abstract domain**, safely approximating their relation in the **concrete domain**.

Abstract conformance on “calls” (informally)

(•) $p : \text{calls}(p(\bar{v}), \text{Pre})$ · $\gamma(F^{\sharp-}) \subseteq F^{\natural} \subseteq \gamma(F^{\sharp+})$ · (•) $\Pi : \text{calls}(_(\bar{v}), {}^\circ\text{Pre})$

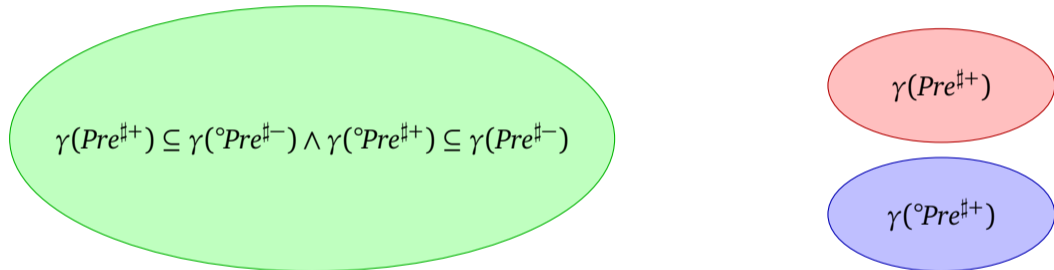


Figure: Illustrations of abstract conformance on “calls” conditions.

Theorem. Abstract conformance on calls \Rightarrow conformance on calls.

Abstract conformance on “calls” (formally)

Definition

A predicate p **abstractly conforms on calls** to the anonymous assertion condition ${}^{\circ}C$, denoted $p \prec^{\sharp-} {}^{\circ}C$, iff

$$(Pre^{\sharp+} \sqsubseteq {}^{\circ}Pre^{\sharp-}) \wedge (Pre^{\sharp-} \sqsupseteq {}^{\circ}Pre^{\sharp+})$$

Conversely, p **does not abstractly conform on calls** to ${}^{\circ}C$, denoted $p \not\prec^{\sharp+} {}^{\circ}C$, iff

$$Pre^{\sharp+} \sqcap {}^{\circ}Pre^{\sharp+} = \perp$$

Theorem. Abstract conformance on calls \Rightarrow conformance on calls.

Abstract conformance on “success” (informally)

(•) $p : \text{success}(p(\bar{v}), \text{Pre}_i, \text{Post}_i) \cdot \gamma(F^{\sharp-}) \subseteq F^{\sharp} \subseteq \gamma(F^{\sharp+}) \cdot (\bullet) \Pi : \text{success}(_(\bar{v}), \text{°Pre}, \text{°Post})$

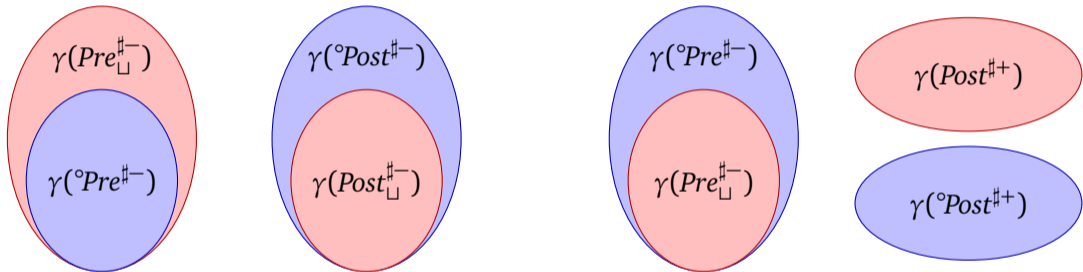


Figure: Illustration of abstract conformance on “success” conditions.

Theorem. Abstract conformance on success \Rightarrow conformance on success.

Abstract conformance on “success” (formally)

Definition

The predicate p **abstractly conforms on success** to ${}^\circ C$, denoted $p \prec^{\sharp-} {}^\circ C$, iff

$$\exists S \subset \mathcal{A} : (Pre_{\sqcup}^{\sharp-} \sqsupseteq {}^\circ Pre^{\sharp+}) \wedge (Post_{\sqcup}^{\sharp+} \sqsubseteq {}^\circ Post^{\sharp-})$$

where

$$Pre_{\sqcup}^{\sharp-} = \sqcup \{ Pre^{\sharp-} \mid \text{success}(p(\bar{v}), Pre, _) \in S \}$$

$$Post_{\sqcup}^{\sharp+} = \sqcup \{ Post^{\sharp+} \mid \text{success}(p(\bar{v}), _, Post) \in S \}$$

Conversely, p **does not abstractly conform on success** to ${}^\circ C$, denoted $p \not\prec^{\sharp+} {}^\circ C$, iff

$$\begin{aligned} \exists \text{success}(p(\bar{v}), Pre, Post) \in \mathcal{A} : & (Pre^{\sharp+} \sqsubseteq {}^\circ Pre^{\sharp-}) \wedge (Post^{\sharp+} \sqcap {}^\circ Post^{\sharp+} = \perp) \wedge \\ & \wedge \exists \theta \in Pre^{\sharp} : \mathcal{S}_{\mathcal{A}}(p(\bar{v}), \theta, \mathbf{P}, \gamma(\mathcal{Q}_p^{\sharp})) \neq \emptyset \end{aligned}$$

Theorem. Abstract conformance on success \Rightarrow conformance on success.

Abstract conformance example (I)

```
p_nat_nat := { :- pred _(X, Y) : nat(X) => nat(Y). }.
```

(a) Predicate property.

```
:- pred n2n(X, Y) : nat(X) => nat(Y).
```

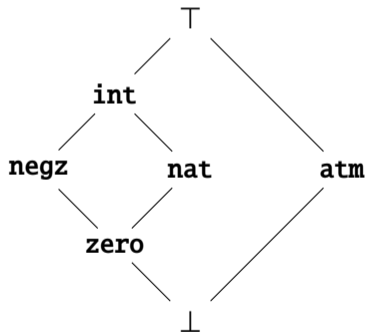
```
:- pred a2n(X, Y) : atm(X) => nat(Y).
```

```
:- pred i2z(X, Y) : int(X) => zero(Y).
```

```
:- pred z2i(X, Y) : zero(X) => int(Y).
```

```
:- pred nz2n(X, Y) : negz(X) => nat(Y).
```

(b) Assertions.



(c) Abstract domain lattice.

Abstract conformance example (II)

<i>Pred</i>	<i>Pre_i</i>	<i>Relation with ${}^{\circ}Pre$</i>	<i>Abstract Conformance</i>
n2n (X, Y)	nat (X)	nat (X) = nat (X)	yes
a2n (X, Y)	atm (X)	atm (X) \sqcap nat (X) = \perp	no
i2z (X, Y)	int (X)	int (X) \sqsupseteq nat (X)	maybe
z2i (X, Y)	zero (X)	zero (X) \sqsubseteq nat (X)	maybe
nz2n (X, Y)	negz (X)	negz (X) \sqcap nat (X) $\neq \perp$ \wedge negz (X) $\not\sqsubseteq$ nat (X) \wedge negz (X) $\not\sqsupseteq$ nat (X)	maybe

Table: Abstract conformance on “calls” example with ${}^{\circ}Pre = \mathbf{nat}(X)$.

Abstract conformance example (III)

<i>Pred</i>	<i>Pre_i</i>	<i>Relation with °Pre</i>	<i>Post_i</i>	<i>Relation with °Post</i>	<i>Abstract Conformance</i>
n2n (X, Y)	nat (X)	nat (X) = nat (X)	nat (Y)	nat (Y) = nat (Y)	yes
a2n (X, Y)	atm (X)	atm (X) \sqcap nat (X) = \perp	nat (Y)	nat (Y) = nat (Y)	maybe
i2n (X, Y)	int (X)	int (X) \sqsupseteq nat (X)	zero (Y)	zero (Y) \sqsubseteq nat (Y)	yes
z2i (X, Y)	zero (X)	zero (X) \sqsubseteq nat (X)	int (Y)	int (Y) \sqsupseteq nat (Y)	maybe
nz2n (X, Y)	negz (X)	negz (X) \sqcap nat (X) $\neq \perp$ \wedge negz (X) $\not\sqsubseteq$ nat (X) \wedge negz (X) $\not\sqsupseteq$ nat (X)	nat (Y)	nat (Y) = nat (Y)	maybe

Table: Abstract conformance on “success” example with ${}^\circ\text{Pre} = \mathbf{nat}(X)$, ${}^\circ\text{Post} = \mathbf{nat}(Y)$.

Wrappers

- **Abstract conformance on “calls”** requires the pre-conditions Pre and ${}^{\circ}Pre$ to be **equivalent**, which is very **restrictive** in practice.

Wrappers

- **Abstract conformance on “calls”** requires the pre-conditions Pre and ${}^{\circ}Pre$ to be **equivalent**, which is very **restrictive** in practice.
- We could have relaxed the definition by proposing an alternative **higher-order semantics** which **taints** predicate usage when **passing through** a predicate property.

Wrappers

- **Abstract conformance on “calls”** requires the pre-conditions Pre and ${}^{\circ}Pre$ to be **equivalent**, which is very **restrictive** in practice.
- We could have relaxed the definition by proposing an alternative **higher-order semantics** which **taints** predicate usage when **passing through** a predicate property.
- Instead, we propose using **wrappers** for creating a Π -**tailored version** of p in an *analysis-friendly manner* (no semantics changes).

Wrappers

- **Abstract conformance on “calls”** requires the pre-conditions Pre and ${}^{\circ}Pre$ to be **equivalent**, which is very **restrictive** in practice.
- We could have relaxed the definition by proposing an alternative **higher-order semantics** which **taints** predicate usage when **passing through** a predicate property.
- Instead, we propose using **wrappers** for creating a Π -**tailored version** of p in an *analysis-friendly manner* (no semantics changes).

Example

```
p_nat := { :- pred _(N) : nat(N). }.
```

```
:- pred even(N) : int(N).
```

```
even(N) :- integer(N), 0 is N mod 2.
```

```
:- pred even_nat(N) : nat(N).
```

```
even_nat(N) :- even(N).
```

- **even/1** will not raise a run-time check error when called with negatives.

Wrappers

- **Abstract conformance on “calls”** requires the pre-conditions Pre and ${}^{\circ}Pre$ to be **equivalent**, which is very **restrictive** in practice.
- We could have relaxed the definition by proposing an alternative **higher-order semantics** which **taints** predicate usage when **passing through** a predicate property.
- Instead, we propose using **wrappers** for creating a Π -**tailored version** of p in an *analysis-friendly manner* (no semantics changes).

Example

```
p_nat := { :- pred _(N) : nat(N). }.
```

```
:- pred even(N) : int(N).  
even(N) :- integer(N), 0 is N mod 2.
```

```
:- pred even_nat(N) : nat(N).  
even_nat(N) :- even(N).
```

- **even/1** will not raise a run-time check error when called with negatives.
- We define the **even_nat/1** wrapper which **trivially** conforms to **p_nat**.

First-order representation of predicate properties

- In order for a first-order analyzer (such as CiaoPP) to deal with **predicate properties** (**higher-order** properties), we propose **reducing** them to **first-order** properties.

First-order representation of predicate properties

- In order for a first-order analyzer (such as CiaoPP) to deal with **predicate properties** (**higher-order** properties), we propose **reducing** them to **first-order** properties.
- We use **regular types** [Dart et al. '92], which are **native** to CiaoPP – *i.e.*, they can be **verified** and **inferred** by CiaoPP using the **eterms** abstract domain.

First-order representation of predicate properties

- In order for a first-order analyzer (such as CiaoPP) to deal with **predicate properties** (**higher-order** properties), we propose **reducing** them to **first-order** properties.
- We use **regular types** [Dart et al. '92], which are **native** to CiaoPP – *i.e.*, they can be **verified** and **inferred** by CiaoPP using the **eterms** abstract domain.
- Given a predicate property Π , we define two associated regular types π^- and π^+ to represent an **under-** and **over-approximation** of the predicates that conform to Π .

First-order representation of predicate properties

- In order for a first-order analyzer (such as CiaoPP) to deal with **predicate properties** (**higher-order** properties), we propose **reducing** them to **first-order** properties.
- We use **regular types** [Dart et al. '92], which are **native** to CiaoPP – *i.e.*, they can be **verified** and **inferred** by CiaoPP using the **eterms** abstract domain.
- Given a predicate property Π , we define two associated regular types π^- and π^+ to represent an **under-** and **over-approximation** of the predicates that conform to Π .

Example

```
:- regtype pp-/1. pp- := p | q.           ■ p and q for sure conform to pp.  
:- regtype pp+/1. pp+ := p | q | r.
```

First-order representation of predicate properties

- In order for a first-order analyzer (such as CiaoPP) to deal with **predicate properties** (**higher-order** properties), we propose **reducing** them to **first-order** properties.
- We use **regular types** [Dart et al. '92], which are **native** to CiaoPP – *i.e.*, they can be **verified** and **inferred** by CiaoPP using the **eterms** abstract domain.
- Given a predicate property Π , we define two associated regular types π^- and π^+ to represent an **under-** and **over-approximation** of the predicates that conform to Π .

Example

```
:- regtype pp-/1. pp- := p | q.           ■ p and q for sure conform to pp.  
:- regtype pp+/1. pp+ := p | q | r.       ■ ..., r may conform to pp.
```

Algorithm

Input : Program: \mathbf{P} , Assertions: \mathcal{A} , Abstract Queries: \mathcal{Q}^\sharp

Output: Verified Status (**checked**/**false**/**check**) for the Assertions \mathcal{A} of \mathbf{P} : \mathcal{V}

$R \leftarrow \emptyset$

repeat

$R' \leftarrow R$

foreach *Predicate Property* $\Pi \in \mathbf{P}$ **do**

$R \leftarrow R \cup \{\pi^-(p) \mid p \in \mathbf{P} \wedge p \prec^{\sharp-} \Pi\} \cup \{\pi^+(p) \mid p \in \mathbf{P} \wedge p \prec^{\sharp+} \Pi\}$

until $R = R'$

$\mathcal{V} \leftarrow \text{acheck}(\mathcal{A}, \llbracket \mathbf{P} \cup R \rrbracket_{\mathcal{Q}^\sharp}^\sharp)$

- Take a predicate property $\Pi \in \mathbf{P}$.

Algorithm

Input : Program: \mathbf{P} , Assertions: \mathcal{A} , Abstract Queries: \mathcal{Q}^\sharp

Output: Verified Status (**checked**/**false**/**check**) for the Assertions \mathcal{A} of \mathbf{P} : \mathcal{V}

$R \leftarrow \emptyset$

repeat

$R' \leftarrow R$

foreach Predicate Property $\Pi \in \mathbf{P}$ **do**

$R \leftarrow R \cup \{\pi^-(p) \mid p \in \mathbf{P} \wedge p \prec^{\sharp-} \Pi\} \cup \{\pi^+(p) \mid p \in \mathbf{P} \wedge p \prec^{\sharp+} \Pi\}$

until $R = R'$

$\mathcal{V} \leftarrow \text{acheck}(\mathcal{A}, \llbracket \mathbf{P} \cup R \rrbracket_{\mathcal{Q}^\sharp}^\sharp)$

- Compute the **under-approximation** of predicates that conform to Π using the **abstract conformance** definitions presented before.

Algorithm

Input : Program: \mathbf{P} , Assertions: \mathcal{A} , Abstract Queries: \mathcal{Q}^\sharp

Output: Verified Status (**checked**/**false**/**check**) for the Assertions \mathcal{A} of \mathbf{P} : \mathcal{V}

$R \leftarrow \emptyset$

repeat

$R' \leftarrow R$

foreach Predicate Property $\Pi \in \mathbf{P}$ **do**

$R \leftarrow R \cup \{\pi^-(p) \mid p \in \mathbf{P} \wedge p \prec^{\sharp-} \Pi\} \cup \{\pi^+(p) \mid p \in \mathbf{P} \wedge p \prec^{\sharp+} \Pi\}$

until $R = R'$

$\mathcal{V} \leftarrow \text{acheck}(\mathcal{A}, \llbracket \mathbf{P} \cup R \rrbracket_{\mathcal{Q}^\sharp}^\sharp)$

- Compute the **over-approximation** of predicates that conform to Π using **abstract conformance** definitions presented before.

Algorithm

Input : Program: \mathbf{P} , Assertions: \mathcal{A} , Abstract Queries: \mathcal{Q}^\sharp

Output: Verified Status (**checked/false/check**) for the Assertions \mathcal{A} of \mathbf{P} : \mathcal{V}

$R \leftarrow \emptyset$

repeat

$R' \leftarrow R$

foreach Predicate Property $\Pi \in \mathbf{P}$ **do**

$R \leftarrow R \cup \{\pi^-(p) \mid p \in \mathbf{P} \wedge p \prec^{\sharp-} \Pi\} \cup \{\pi^+(p) \mid p \in \mathbf{P} \wedge p \prec^{\sharp+} \Pi\}$

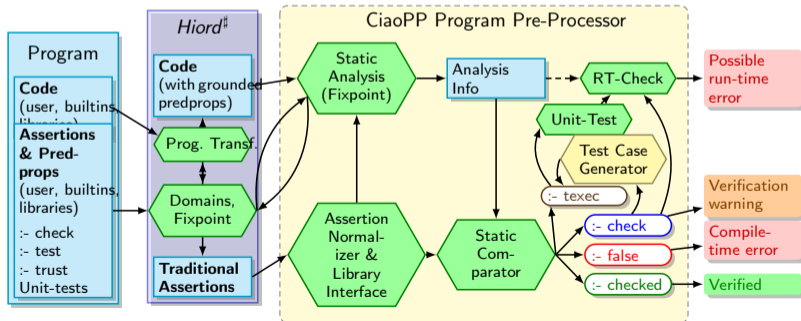
until $R = R'$

$\mathcal{V} \leftarrow \text{acheck}(\mathcal{A}, \llbracket \mathbf{P} \cup R \rrbracket_{\mathcal{Q}^\sharp}^\sharp)$

- Iterate until a **fixpoint** is reached, and statically check the assertions \mathcal{A} w.r.t. the abstract semantics $\llbracket \mathbf{P} \cup R \rrbracket_{\mathcal{Q}^\sharp}^\sharp$.

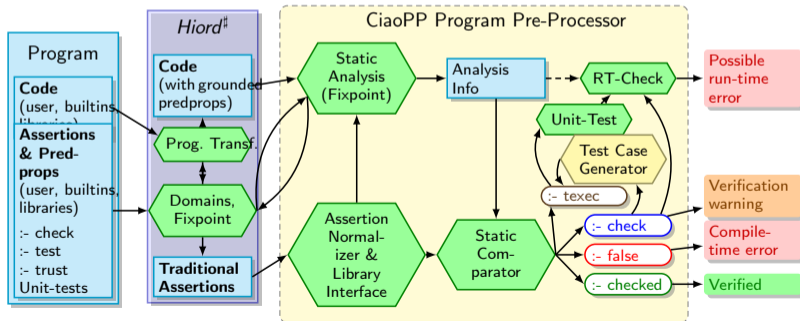
Implementation

1. **Transform predicate properties to an internal representation.**



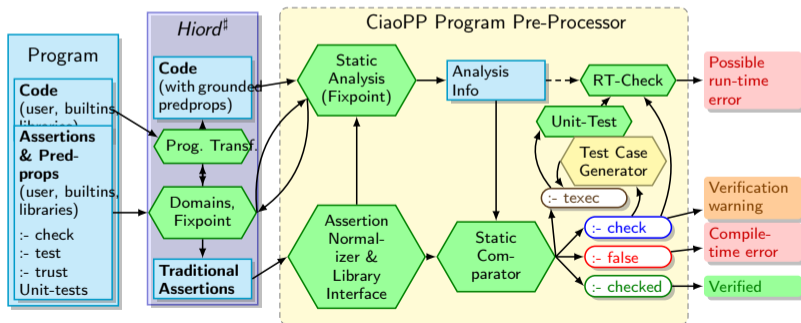
Implementation

1. **Transform predicate properties** to an **internal** representation.
2. **Analyze the user-defined properties** involved in each of the **predicate properties**.



Implementation

1. **Transform predicate properties** to an **internal** representation.
2. **Analyze the user-defined properties** involved in each of the **predicate properties**.
3. Compute **abstract conformance** using the **abstract domain operations**.



Experiments

```
t_cmp := { :- pred _(X, Y) : (t(X), t(Y)). }.
```

```
:- pred lex(X, Y) : (term(X), term(Y)).  
lex(X, Y) :- X @< Y.
```

```
:- pred lex_t(X, Y) : (t(X), t(Y)).  
lex_t(X, Y) :- lex(X, Y).
```

```
:- pred qsort(Xs, P, Ys)  
   : (list(t, Xs), t_cmp(P)) => list(t, Ys).
```

```
...
```

```
partition([X|L], Y, P, [X|L1], L2) :-  
  P(X, Y), !,  
  partition(L, Y, P, L1, L2).
```

```
...
```

- Verify **generic** *qsort* implementation.
- **t_cmp** represents **t/1**-comparator predicates.
- **lex_t/2** wraps **lex/2** with **t_cmp**.

Experiments

```
t_cmp := { :- pred _(X, Y) : (t(X), t(Y)). }.
```

```
:- pred lex(X, Y) : (term(X), term(Y)).  
lex(X, Y) :- X @< Y.
```

```
:- pred lex_t(X, Y) : (t(X), t(Y)).  
lex_t(X, Y) :- lex(X, Y).
```

```
:- pred qsort(Xs, P, Ys)  
    : (list(t, Xs), t_cmp(P)) => list(t, Ys).
```

...

```
partition([X|L], Y, P, [X|L1], L2) :-  
    P(X, Y), !,  
    partition(L, Y, P, L1, L2).
```

...

- Verify **generic** *qsort* implementation.
- **t_cmp** represents **t/1-comparator** predicates.
- **lex_t/2** wraps **lex/2** with **t_cmp**.
- We use **t_cmp** to specify **qsort/3**.

Experiments

```
t_cmp := { :- pred _(X, Y) : (t(X), t(Y)). }.
```

```
:- pred lex(X, Y) : (term(X), term(Y)).  
lex(X, Y) :- X @< Y.
```

```
:- pred lex_t(X, Y) : (t(X), t(Y)).  
lex_t(X, Y) :- lex(X, Y).
```

```
:- pred qsort(Xs, P, Ys)  
    : (list(t, Xs), t_cmp(P)) => list(t, Ys).
```

...

```
partition([X|L], Y, P, [X|L1], L2) :-  
    P(X, Y), !,  
    partition(L, Y, P, L1, L2).
```

...

- Verify **generic** *qsort* implementation.
- **t_cmp** represents **t/1-comparator** predicates.
- **lex_t/2** wraps **lex/2** with **t_cmp**.
- We use **t_cmp** to specify **qsort/3**.
- Analysis **propagates** **t_cmp** to the variable **P** in the higher-order call.

Experiments (cont'd)

```
handler := {  
  :- pred _(R1, R2) : req(R1) => res(R2).  
}.
```

```
:- regtype req/1.   :- regtype res/1.  
req := 'DELETE'    res := 'OK'  
    | 'GET'         | 'CREATED'  
    | 'POST'        | 'BAD_REQUEST'  
    | 'PUT'         | 'NOT_FOUND'.
```

```
h('DELETE', 'OK') :- ...  
h('POST', 'CREATED') :- ...  
h('PUT', 'BAD_REQ') :- ...  
h('GET', 'NOT_FOUND') :- ...
```

```
:- pred server(Hnd, R1, R2) : (handler(Hnd), req(R1)) => res(R2).
```

- Verify (schematic) **generic HTTP server**.
- **handler** represents request handler predicates.

Experiments (cont'd)

```
handler := {  
  :- pred _(R1, R2) : req(R1) => res(R2).  
}.
```

```
:- regtype req/1.   :- regtype res/1.  
req := 'DELETE'    res := 'OK'  
    | 'GET'        | 'CREATED'  
    | 'POST'       | 'BAD_REQUEST'  
    | 'PUT'        | 'NOT_FOUND'.
```

```
h('DELETE', 'OK') :- ...  
h('POST', 'CREATED') :- ...  
h('PUT', 'BAD_REQ') :- ...  
h('GET', 'NOT_FOUND') :- ...
```

```
:- pred server(Hnd, R1, R2) : (handler(Hnd), req(R1)) => res(R2).
```

- Verify (schematic) **generic HTTP server**.
- **handler** represents request handler predicates.
- Since we have a **bug**, **h/2** may not conform to **handler**.

Experiments (cont'd)

```
handler := {  
  :- pred _(R1, R2) : req(R1) => res(R2).  
}.
```

```
:- regtype req/1.   :- regtype res/1.  
req := 'DELETE'    res := 'OK'  
     | 'GET'        | 'CREATED'  
     | 'POST'       | 'BAD_REQUEST'  
     | 'PUT'        | 'NOT_FOUND'.
```

```
h('DELETE', 'OK') :- ...  
h('POST', 'CREATED') :- ...  
h('PUT', 'BAD_REQ') :- ...  
h('GET', 'NOT_FOUND') :- ...
```

```
:- pred server(Hnd, R1, R2) : (handler(Hnd), req(R1)) => res(R2).
```

- Verify (schematic) **generic HTTP server**.
- **handler** represents request handler predicates.
- Since we have a **bug**, **h/2** may not conform to **handler**.
- Thus, we would get a **warning** when calling **server/3** with **h/2**.

Conclusions

- Improved the support of **higher-order properties** in the assertion language.
- Formalized a **compile-time** criteria for the higher-order verification problem at hand.
- **Verified** programs that were **not possible** to verify until this point.
- All this using a **first-order verifier**.

Conclusions

- Improved the support of **higher-order properties** in the assertion language.
- Formalized a **compile-time** criteria for the higher-order verification problem at hand.
- **Verified** programs that were **not possible** to verify until this point.
- All this using a **first-order verifier**.

Future work

- Polish the prototype implementation in the Ciao system and integrate with the **modular/incremental** static analysis.
- Optimize the computation of conformance for each candidate.
- Extend the approach to support other aspects of higher-order in Ciao such as **anonymous predicates** and **partial applications**.

Hiord[#]

An Approach to the Specification and Verification of Higher-Order (C)LP Programs

Marco Ciccalè^{1,2} Daniel Jurjo-Rivas^{1,2} José F. Morales^{1,2}
Pedro López-García^{2,3} Manuel V. Hermenegildo^{1,2}

¹Universidad Politécnica de Madrid (UPM)





²IMDEA Software Institute

³Spanish Council for Scientific Research (CSIC)



ICLP'25 – September 18, 2025

References I

-  P. Cousot and R. Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints,” in *ACM Symposium on Principles of Programming Languages (POPL77)*. ACM Press, 1977, pp. 238–252.
-  K. Muthukumar and M. Hermenegildo, “Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs,” Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, Technical Report ACT-DC-153-90, April 1990. [Online]. Available: <https://cliplab.org/papers/mcctr-fixpt.pdf>
-  M. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia, “Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor),” *Science of Computer Programming*, vol. 58, no. 1–2, pp. 115–140, October 2005.
-  P. Dart and J. Zobel, “A Regular Type Language for Logic Programs,” in *Types in Logic Programming*. MIT Press, 1992, pp. 157–187.