Hiord$^\sharp$

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]

[1]Universidad Politécnica de Madrid (UPM)

[2]IMDEA Software Institute

[3]Spanish Council for Scientific Research (CSIC)

PROLE'26 – June 16, 2026

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 $\mathscr{A}$ for a predicate *Pred*:

```
:- pred Pred : Pre_1 => Post_1.
...
:- pred Pred : Pre_n => Post_n.
```

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

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

Assertion conditions

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

```
:- pred Pred : Pre_1 => Post_1.
...
:- pred Pred : Pre_n => Post_n.
```

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

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

Example

Consider the assertion before: "`:- pred qsort(Xs, Ys) : lst(Xs) => lst(Ys).`"
Its set of assertion conditions is:

$$\mathsf{calls}(\texttt{qsort(Xs, Ys)},\ \texttt{lst(Xs)})$$
$$\mathsf{success}(\texttt{qsort(Xs, Ys)},\ \texttt{lst(Xs)},\ \texttt{lst(Ys)})$$

Semantics with assertions

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

Semantics with assertions

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

Semantics with assertions

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

Semantics with assertions

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

Semantics with assertions

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

CONSTR$_{\mathscr{A}}$ *L is a constraint*

$$\langle L :: G \mid \theta \mid \varnothing \rangle \rightsquigarrow_{\mathscr{A}} \langle G \mid \theta \wedge L \mid \varnothing \rangle \qquad \text{if } \theta \wedge L \not\models \mathbf{ff}$$

Semantics with assertions

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

CONSTR$_{\mathscr{A}}$ *L is a constraint*

$$\langle L :: G \mid \theta \mid \varnothing \rangle \rightsquigarrow_{\mathscr{A}} \langle G \mid \theta \wedge L \mid \varnothing \rangle \quad \text{if } \theta \wedge L \not\models \mathbf{ff}$$

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

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

Semantics with assertions (cont'd)

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

$$\langle L :: G \mid \theta \mid \varnothing \rangle \rightsquigarrow_{\mathscr{A}} \begin{cases} \langle G \mid \theta \mid \{\ell\} \rangle & \text{if } \exists C = \mathsf{calls}(L,\ Pre) \in \mathscr{A} : \\ & \quad \mathsf{label}(C) = \ell \ \wedge \ \theta \not\Rrightarrow Pre \\ \langle B :: \textbf{PostC} :: G \mid \theta \mid \varnothing \rangle & \text{otherwise,} \end{cases}$$

where *PostC* is the sequence of **check literals** $\mathsf{check}(L, \ell_1) :: \ldots :: \mathsf{check}(L, \ell_n)$ such that each $\ell_i = \mathsf{label}(C_i)$, where $C_i = \mathsf{success}(L,\ Pre_i,\ Post_i) \in \mathscr{A}$, and $\theta \Rrightarrow Pre_i$

Semantics with assertions (cont'd)

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

$$\langle L :: G \mid \theta \mid \varnothing \rangle \rightsquigarrow_{\mathscr{A}} \begin{cases} \langle G \mid \theta \mid \{\ell\} \rangle & \text{if } \exists C = \mathsf{calls}(L,\ \mathit{Pre}) \in \mathscr{A} : \\ & \quad \mathsf{label}(C) = \ell \ \wedge \ \theta \not\Rrightarrow \mathit{Pre} \\ \langle B :: \mathbf{PostC} :: G \mid \theta \mid \varnothing \rangle & \text{otherwise,} \end{cases}$$

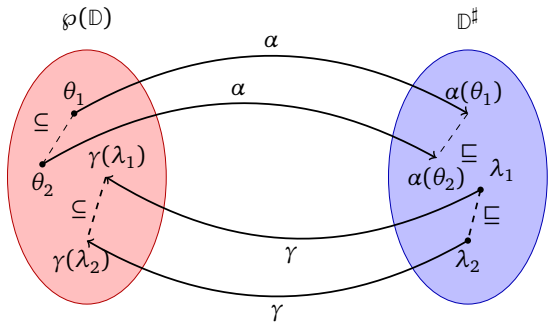
where *PostC* is the sequence of **check literals** $\mathsf{check}(L, \ell_1) :: \ldots :: \mathsf{check}(L, \ell_n)$ such that each $\ell_i = \mathsf{label}(C_i)$, where $C_i = \mathsf{success}(L,\ \mathit{Pre}_i,\ \mathit{Post}_i) \in \mathscr{A}$, and $\theta \Rrightarrow \mathit{Pre}_i$

CHECK$_{\mathscr{A}}$ $\quad$ L *is a check literal* $\mathsf{check}(L', \ell)$

$$\langle L :: G \mid \theta \mid \varnothing \rangle \rightsquigarrow_{\mathscr{A}} \begin{cases} \langle G \mid \theta \mid \{\ell\} \rangle & \text{if } \exists C = \mathsf{success}(L',\ \_,\ \mathit{Post}) \in \mathscr{A} : \\ & \quad \mathsf{label}(C) = \ell \ \wedge \ \theta \not\Rrightarrow \mathit{Post} \\ \langle G \mid \theta \mid \varnothing \rangle & \text{otherwise} \end{cases}$$

Abstract interpretation [Cousot '77]

Theory for designing (*sound by construction*) static analyzers for **proving** (concrete) **semantic properties** of a program **P**, by interpreting it over a (simpler) **abstract domain** that approximates the concrete semantics of **P**.



concrete domain $\langle \wp(\mathbb{D}), \subseteq, \cup, \cap, \mathbb{D}, \varnothing \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{D}^\sharp, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$ **abstract domain**

Abstract interpretation of (C)LP programs

- We perform **top-down** abstract interpretation, where the **concrete domain** $\wp(\mathbb{D})$ is a complete lattice that 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** $\wp(\mathbb{D})$ is a complete lattice that 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:

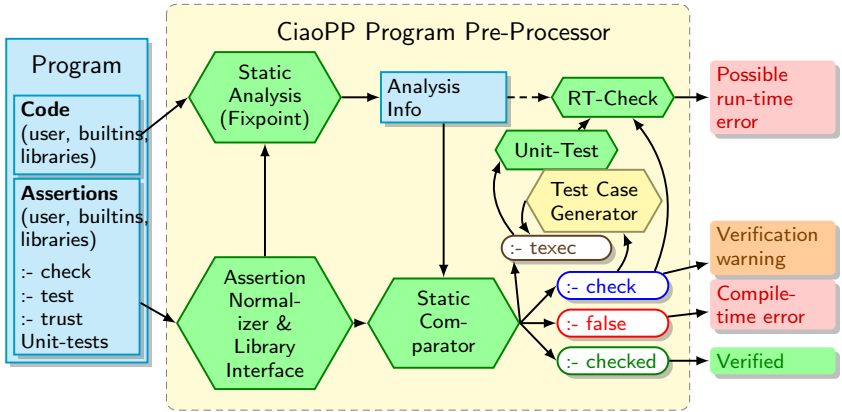
$$[\![\mathbf{P}]\!]^\sharp_{\mathcal{Q}^\sharp} = \{\langle Pred_1, \lambda_1^c, \lambda_1^s \rangle, \ldots, \langle Pred_n, \lambda_n^c, \lambda_n^s \rangle\}$$

  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) $\mathbb{D}^\sharp$.
- Statically **checks** assertions $\mathscr{A}$: **checked**/**false**/**check** for each $A \in \mathscr{A}$.



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:

```
:- predprop Π := {
     :- pred _(v̄) : Pre_1 => Post_1.
     ...
     :- pred _(v̄) : Pre_n => Post_n.
}.
```

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:

```
:- predprop Π := {
     :- pred _(v̄) : Pre_1 => Post_1.
     ...
     :- pred _(v̄) : Pre_n => Post_n.
}.
```

Example

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

```
:- predprop int_op := {
    :- pred _(X, Y, Z) : (int(X), int(Y)) => 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:

```
:- predprop Π := {
     :- pred _(v̄) : Pre_1 => Post_1.
     ...
     :- pred _(v̄) : Pre_n => Post_n.
}.
```

Example

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

```
:- predprop int_op := {
    :- pred _(X, Y, Z) : (int(X), int(Y)) => int(Z).
}.
```

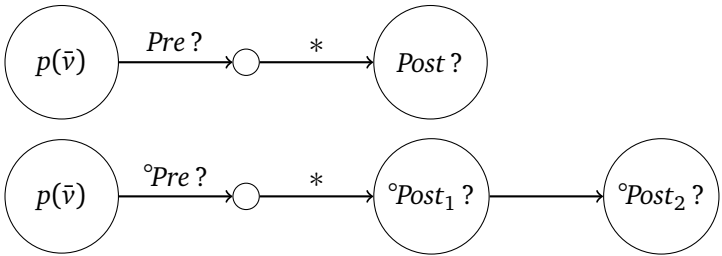
Which we then use in the following (partial) specification of an evaluation predicate:

```
:- pred eval(A, B, P, R) : (int(A), int(B), int_op(P)) => 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 $\mathscr{A}$:

$$\mathscr{A} = \{\mathsf{calls}(p(\bar{v}),\ Pre),\ \mathsf{success}(p(\bar{v}),\ Pre,\ Post)\}$$
$$\Pi = \{\mathsf{calls}(\_(\bar{v}),\ {}^{\circ}Pre),\ \mathsf{success}(\_(\bar{v}),\ {}^{\circ}Pre,\ {}^{\circ}Post_1),\ \mathsf{success}(\_(\bar{v}),\ {}^{\circ}Pre,\ {}^{\circ}Post_2)\}$$

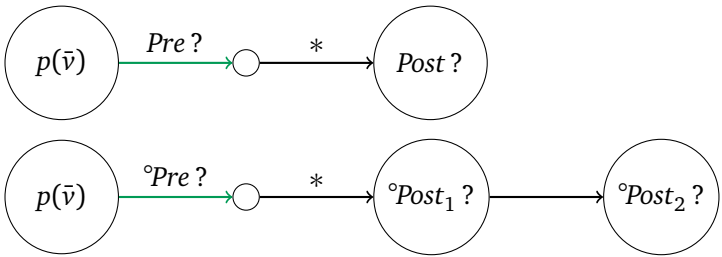


- Anonymous assertions in Π should not add new run-time check errors.

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 $\mathscr{A}$:

$$\mathscr{A} = \{\mathsf{calls}(p(\bar{v}),\ Pre),\ \mathsf{success}(p(\bar{v}),\ Pre,\ Post)\}$$
$$\Pi = \{\mathsf{calls}(\_(\bar{v}),\ {}^{\circ}Pre),\ \mathsf{success}(\_(\bar{v}),\ {}^{\circ}Pre,\ {}^{\circ}Post_1),\ \mathsf{success}(\_(\bar{v}),\ {}^{\circ}Pre,\ {}^{\circ}Post_2)\}$$

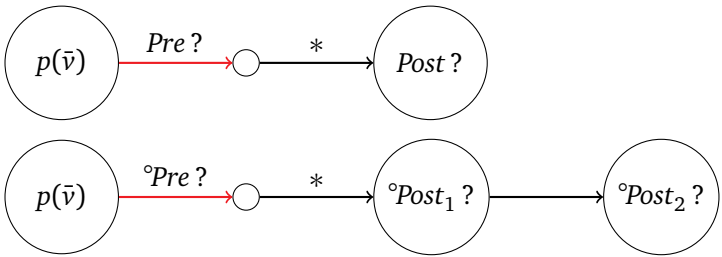


- If Pre **succeeds** to be checked, then ${}^{\circ}Pre$ should **succeed** to be checked.

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 $\mathscr{A}$:

$$\mathscr{A} = \{\mathsf{calls}(p(\bar{v}),\ \mathit{Pre}),\ \mathsf{success}(p(\bar{v}),\ \mathit{Pre},\ \mathit{Post})\}$$
$$\Pi = \{\mathsf{calls}(\_(\bar{v}),\ {}^{\circ}\mathit{Pre}),\ \mathsf{success}(\_(\bar{v}),\ {}^{\circ}\mathit{Pre},\ {}^{\circ}\mathit{Post}_1),\ \mathsf{success}(\_(\bar{v}),\ {}^{\circ}\mathit{Pre},\ {}^{\circ}\mathit{Post}_2)\}$$

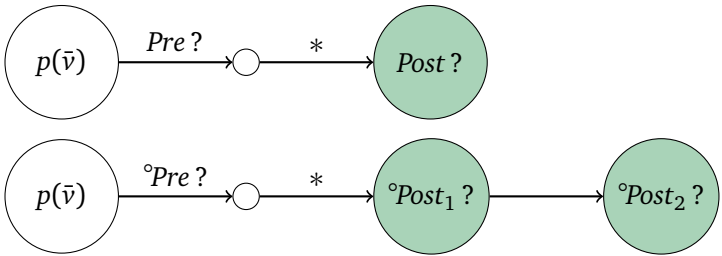


- If Pre **fails** to be checked, then ${}^{\circ}\mathit{Pre}$ should **fail** to be checked.

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 $\mathscr{A}$:

$$\mathscr{A} = \{\mathsf{calls}(p(\bar{v}),\ Pre),\ \mathsf{success}(p(\bar{v}),\ Pre,\ Post)\}$$
$$\Pi = \{\mathsf{calls}(\_(\bar{v}),\ {}^{\circ}Pre),\ \mathsf{success}(\_(\bar{v}),\ {}^{\circ}Pre,\ {}^{\circ}Post_1),\ \mathsf{success}(\_(\bar{v}),\ {}^{\circ}Pre,\ {}^{\circ}Post_2)\}$$

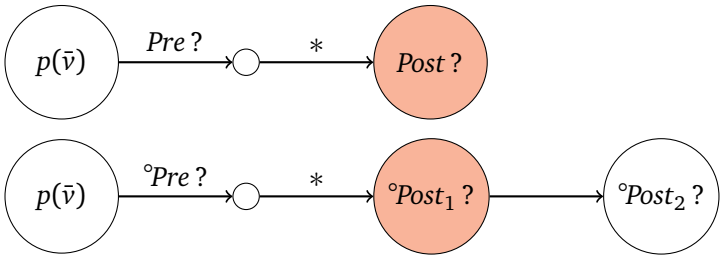


- If $Post$ **succeeds** to be checked, then ${}^{\circ}Post_1$ and ${}^{\circ}Post_2$ should **succeed** to be checked.

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 $\mathscr{A}$:

$$\mathscr{A} = \{\mathsf{calls}(p(\bar{v}),\ Pre),\ \mathsf{success}(p(\bar{v}),\ Pre,\ Post)\}$$
$$\Pi = \{\mathsf{calls}(\_(\bar{v}),\ {}^{\circ}Pre),\ \mathsf{success}(\_(\bar{v}),\ {}^{\circ}Pre,\ {}^{\circ}Post_1),\ \mathsf{success}(\_(\bar{v}),\ {}^{\circ}Pre,\ {}^{\circ}Post_2)\}$$

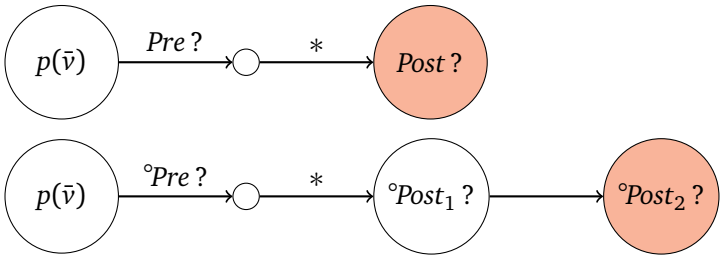


- If $Post$ **fails** to be checked, then either ${}^{\circ}Post_1$ or ${}^{\circ}Post_2$ should **fail** to be checked.

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 $\mathscr{A}$:

$$\mathscr{A} = \{\mathsf{calls}(p(\bar{v}),\ Pre),\ \mathsf{success}(p(\bar{v}),\ Pre,\ Post)\}$$
$$\Pi = \{\mathsf{calls}(\_(\bar{v}),\ {}^{\circ}Pre),\ \mathsf{success}(\_(\bar{v}),\ {}^{\circ}Pre,\ {}^{\circ}Post_1),\ \mathsf{success}(\_(\bar{v}),\ {}^{\circ}Pre,\ {}^{\circ}Post_2)\}$$



- If $Post$ **fails** to be checked, then either ${}^{\circ}Post_1$ or ${}^{\circ}Post_2$ should **fail** to be checked.

Conformance (formally)

Definition (Conformance)

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

$$p \prec \Pi \iff \forall Q_p \in \mathscr{Q}_p : \ \Pi \textit{ is redundant for } p$$

$$p \not\prec \Pi \iff \exists Q_p \in \mathscr{Q}_p : \ \Pi \textit{ is not redundant for } p$$

Definition (Redundance)

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

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

and

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

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

Abstract conformance

- Conformance is **not computable** in general, since the set $\mathscr{Q}_p$ 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 $\mathscr{Q}_p$ 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 $\mathscr{Q}_p$ 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 $\mathscr{A}$ 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)

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

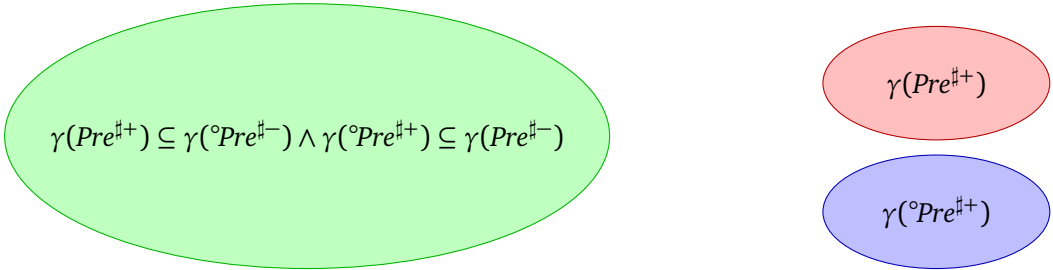


Figure: Illustrations of abstract conformance on “calls” conditions.

Theorem. Abstract conformance on calls ⇒ conformance on calls.

Abstract conformance on “calls” (formally)

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

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

Theorem. Abstract conformance on calls ⇒ conformance on calls.

Abstract conformance on “calls” (formally)

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

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

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

$$Pre^{\sharp +} \sqcap \complement Pre^{\sharp +} = \bot$$

Theorem. Abstract conformance on calls ⇒ conformance on calls.

Abstract conformance on “success” (informally)

(●) p : success($p(\bar{v})$, Pre_i, $Post_i$) · $\gamma(F^{\sharp -}) \subseteq F^{\sharp} \subseteq \gamma(F^{\sharp +})$ · (●) Π : success(_($\bar{v}$), $^\circ Pre$, $^\circ Post$)

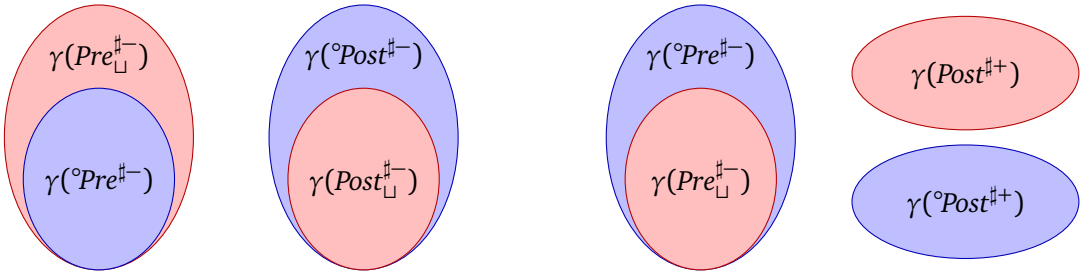


Figure: Illustration of abstract conformance on “success” conditions.

Theorem. Abstract conformance on success ⇒ conformance on success.

Abstract conformance on “success” (formally)

Definition

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

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

where

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

Theorem. Abstract conformance on success ⇒ conformance on success.

Abstract conformance on “success” (formally)

Definition

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

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

where

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

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

$$\exists\, \mathsf{success}(p(\bar{v}),\ Pre,\ Post) \in \mathscr{A} : (Pre^{\sharp+} \sqsubseteq {}^{\circ}Pre^{\sharp-}) \wedge (Post^{\sharp+} \sqcap {}^{\circ}Post^{\sharp+} = \bot) \wedge$$
$$\wedge\, \exists \theta \in Pre^{\natural} : \mathscr{S}_{\mathscr{A}}(p(\bar{v}), \theta, \mathbf{P}, \gamma(\mathscr{Q}^{\sharp}_{p})) \neq \varnothing$$

Theorem. Abstract conformance on success ⇒ conformance on success.

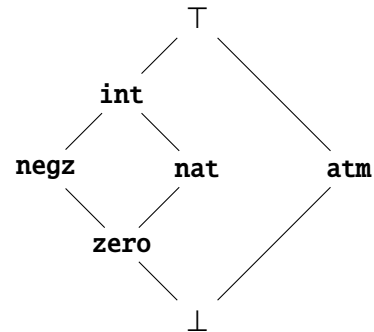
Abstract conformance example (I)

```
:- predprop 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)

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

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

Abstract conformance example (III)

Pred	Pre_i	*Relation with* $^\circ Pre$	$Post_i$	*Relation with* $^\circ Post$	*Abstract Conformance*
`n2n(X,Y)`	`nat(X)`	$\texttt{nat(X)} = \texttt{nat(X)}$	`nat(Y)`	$\texttt{nat(Y)} = \texttt{nat(Y)}$	yes
`a2n(X,Y)`	`atm(X)`	$\texttt{atm(X)} \sqcap \texttt{nat(X)} = \bot$	`nat(Y)`	$\texttt{nat(Y)} = \texttt{nat(Y)}$	maybe
`i2n(X,Y)`	`int(X)`	$\texttt{int(X)} \sqsupseteq \texttt{nat(X)}$	`zero(Y)`	$\texttt{zero(Y)} \sqsubseteq \texttt{nat(Y)}$	yes
`z2i(X,Y)`	`zero(X)`	$\texttt{zero(X)} \sqsubseteq \texttt{nat(X)}$	`int(Y)`	$\texttt{int(Y)} \sqsupseteq \texttt{nat(Y)}$	maybe
`nz2n(X,Y)`	`negz(X)`	$\texttt{negz(X)} \sqcap \texttt{nat(X)} \neq \bot \wedge \texttt{negz(X)} \not\sqsubseteq \texttt{nat(X)} \wedge \texttt{negz(X)} \not\sqsupseteq \texttt{nat(X)}$	`nat(Y)`	$\texttt{nat(Y)} = \texttt{nat(Y)}$	maybe

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

Wrappers

- **Abstract conformance on "calls"** requires the pre-conditions *Pre* and °*Pre* to be **equivalent**, which is very **restrictive** in practice.

Wrappers

- **Abstract conformance on “calls”** requires the pre-conditions *Pre* and °*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 °*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

```
:- predprop 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 °*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

```
:- predprop 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

- 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

- 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, *e.g.*, the **`eterms`** abstract domain [Vaucheret and Bueno, '02].

First-order representation of predicate properties

- 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, *e.g.*, the **`eterms`** abstract domain [Vaucheret and Bueno, '02].
- 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

- 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, *e.g.*, the **`eterms`** abstract domain [Vaucheret and Bueno, '02].
- 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.
:- regtype pp⁺/1. pp⁺ := p | q | r.
```

- `p` and `q` **for sure** conform to **`pp`**.

First-order representation of predicate properties

- 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, *e.g.*, the **`eterms`** abstract domain [Vaucheret and Bueno, '02].
- 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.
:- regtype pp⁺/1. pp⁺ := p | q | r.
```

- `p` and `q` **for sure** conform to **`pp`**.
- ..., `r` **may** conform to **`pp`**.

Fixpoint computation of abstract conformance

Since predicate properties can be defined in terms of other predicate properties, we compute conformance by computing a **fixpoint**:

Input : Program: $\mathbf{P}$, Assertions: $\mathscr{A}$, Abstract Queries: $\mathscr{Q}^\sharp$
Output: Verified Status (**checked**/**false**/**check**) for the Assertions $\mathscr{A}$ of $\mathbf{P}$: $\mathscr{V}$

$R \leftarrow \emptyset$
repeat
  $R' \leftarrow R$
  foreach *Predicate Property* $\Pi \in \boldsymbol{P}$ **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 $R = R'$
$\mathscr{V} \leftarrow \mathsf{acheck}(\mathscr{A}, [\![\mathbf{P} \cup R]\!]^\sharp_{\mathscr{Q}^\sharp})$

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

Fixpoint computation of abstract conformance

Since predicate properties can be defined in terms of other predicate properties, we compute conformance by computing a **fixpoint**:

Input : Program: $\mathbf{P}$, Assertions: $\mathscr{A}$, Abstract Queries: $\mathscr{Q}^\sharp$
Output: Verified Status (**checked**/**false**/**check**) for the Assertions $\mathscr{A}$ of $\mathbf{P}$: $\mathscr{V}$

$R \leftarrow \emptyset$
repeat
  $R' \leftarrow R$
  foreach *Predicate Property* $\Pi \in \boldsymbol{P}$ **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 $R = R'$
$\mathscr{V} \leftarrow \mathsf{acheck}(\mathscr{A}, [\![\mathbf{P} \cup R]\!]^\sharp_{\mathscr{Q}^\sharp})$

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

Fixpoint computation of abstract conformance

Since predicate properties can be defined in terms of other predicate properties, we compute conformance by computing a **fixpoint**:

Input : Program: $\mathbf{P}$, Assertions: $\mathscr{A}$, Abstract Queries: $\mathscr{Q}^\sharp$
Output: Verified Status (**checked**/**false**/**check**) for the Assertions $\mathscr{A}$ of $\mathbf{P}$: $\mathscr{V}$

$R \leftarrow \emptyset$
repeat
  $R' \leftarrow R$
  foreach *Predicate Property* $\Pi \in \boldsymbol{P}$ **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 $R = R'$
$\mathscr{V} \leftarrow \mathsf{acheck}(\mathscr{A}, [\![\mathbf{P} \cup R]\!]^\sharp_{\mathscr{Q}^\sharp})$

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

Fixpoint computation of abstract conformance

Since predicate properties can be defined in terms of other predicate properties, we compute conformance by computing a **fixpoint**:

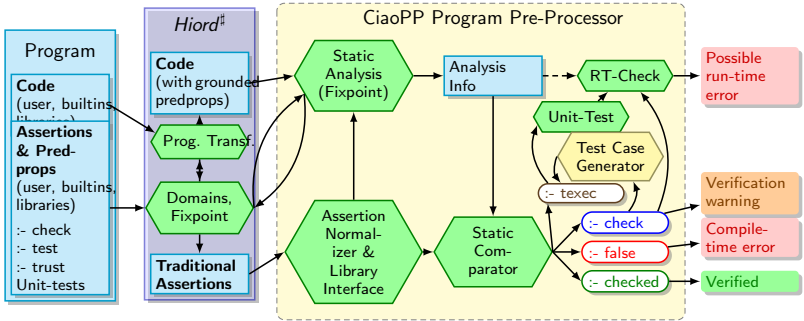
Input : Program: $\mathbf{P}$, Assertions: $\mathscr{A}$, Abstract Queries: $\mathscr{Q}^\sharp$
Output: Verified Status (**checked**/**false**/**check**) for the Assertions $\mathscr{A}$ of $\mathbf{P}$: $\mathscr{V}$

$R \leftarrow \emptyset$
repeat
  $R' \leftarrow R$
  foreach *Predicate Property* $\Pi \in \boldsymbol{P}$ **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 $R = R'$
$\mathscr{V} \leftarrow \mathsf{acheck}(\mathscr{A}, [\![\mathbf{P} \cup R]\!]^\sharp_{\mathscr{Q}^\sharp})$

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

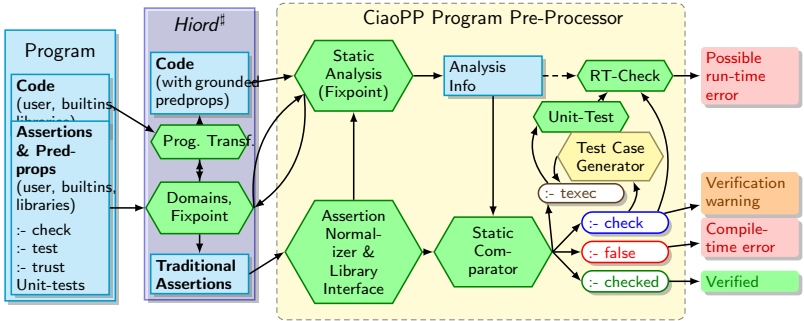
Implementation

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



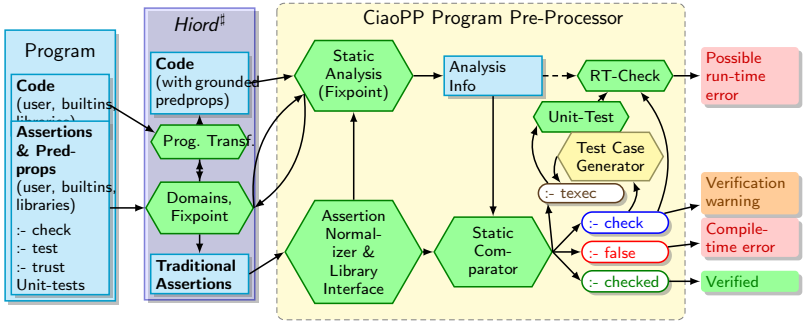
Implementation

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



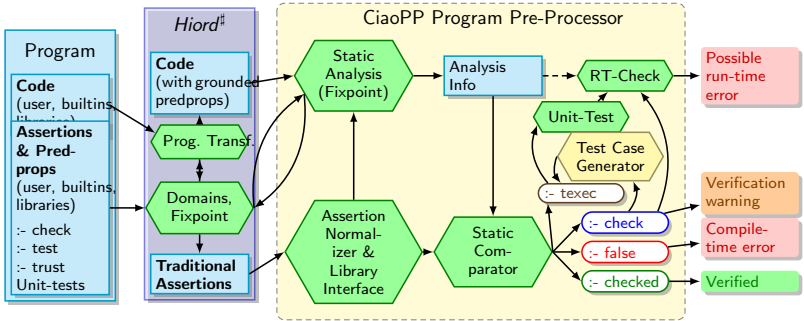
Implementation

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**.



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**.
4. Assert the **regular type representations** of each predicate property to program.



Experiments

```
:- predprop 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

```
:- predprop 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

```
:- predprop 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)

```
:- predprop 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)

```
:- predprop 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)

```
:- predprop 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** (higher-order) 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** (higher-order) 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 fixpoint computation of conformance for bigger modules.
- Extend the approach to support other aspects of higher-order in Ciao such as **anonymous predicates** and **partial applications**.

Hiord$^\sharp$

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]

[1]Universidad Politécnica de Madrid (UPM)

[2]IMDEA Software Institute

[3]Spanish Council for Scientific Research (CSIC)

PROLE'26 – June 16, 2026

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 (POPL'77)*. 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.

C. Vaucheret and F. Bueno, “More Precise yet Efficient Type Inference for Logic Programs,” in *9th International Static Analysis Symposium (SAS'02)*, ser. Lecture Notes in Computer Science, vol. 2477. Springer-Verlag, September 2002, pp. 102–116.