Exploiting Multiple Abstract Call Patterns for Optimizing Run-Time Checks

Daniela Ferreiro[1,2], Daniel Jurjo-Rivas[1,2], Marco Ciccalè[1,2],
José F. Morales[1,2], Pedro López-García[1,3] and Manuel V. Hermenegildo[1,2]

[1] IMDEA Software Institute
[2] Universidad Politécnica de Madrid
[3] Spanish Council for Scientific Research

ICLP'26, July 22, 2026

Introduction — checking types, modes, determinacy, non-failure, ...

Static languages: (e.g., Mercury) checked at **compile time**

- ▲ No **run-time overhead**
- ▼ Correct programs may be **rejected**

Dynamic languages: (e.g., traditional Prolog) checked at **run time**

- ▼ **Run-time overhead**
- ▲ No correct programs are **rejected**

Do we need to choose? **No!** We have the **Ciao Prolog** model!

Introduction — the Ciao assertion-based approach

- **Assertions** are **optional**, partial specifications for procedures:

```
:- pred member(X, T) : tree(num, T) => num(X).
```

  Pre/Post: **native** or **user-defined** properties (types, modes, determinacy, non-failure, ...):

```
:- prop tree/2.
tree(_, void).
tree(P, tree(L, X, R)) :- P(X), tree(P, L), tree(P, R).
```

- **Static analysis** for **discharging** as many properties as possible during **compile time**
- *Optional* **run-time checks** for the remaining checks, still **expensive**...

This paper:

- Integrate the **run-time checking** semantics of assertions into the static analyzer
- Exploit **multivariance** to check different **calling patterns** independently

⇒ **More** checks discharged at **compile-time**, **soundly**

Framework — operational semantics

Standard SLD semantics: unfold atoms, propagate constraints

$$\langle H :: G \mid \theta \rangle \rightsquigarrow \langle B :: G \mid \theta \rangle \qquad \text{(Atom)}$$
$$\langle \theta' :: G \mid \theta \rangle \rightsquigarrow \langle G' \mid \theta \wedge \theta' \rangle \quad \text{if } (\theta \wedge \theta') \not\models \mathit{false} \qquad \text{(Constr)}$$

Run-time checking semantics: insert pre/postcondition checks

$$\langle H :: G \mid \theta \rangle \rightsquigarrow_{\mathcal{A}} \langle \mathrm{c}(\mathit{Pre}) :: B :: \mathrm{c}(\mathit{Post}) :: G \mid \theta \rangle \qquad (\text{Atom}_{\mathcal{A}})$$
$$\langle \mathrm{c}(F) :: G \mid \theta \rangle \rightsquigarrow_{\mathcal{A}} \langle G \mid \theta \rangle \quad \text{if } \mathrm{check}(F, \theta) \qquad (\text{OkCheck}_{\mathcal{A}})$$
$$\langle \mathrm{c}(F) :: G \mid \theta \rangle \rightsquigarrow_{\mathcal{A}} \langle \mathrm{e}(F) \mid \theta \rangle \quad \text{if } \neg\mathrm{check}(F, \theta) \qquad (\text{ErrCheck}_{\mathcal{A}})$$

Observation

A program executed with run-time checking semantics has **fewer** reachable states.
$\Rightarrow$ standard semantics is an **overapproximation** of run-time checking semantics.

Running example — number list and tree `member/2` with assertions

```
:- pred member(X, L) : list(num, L) => num(X).
:- pred member(X, L) : num(X) => list(num, L).

member(X, [X|L]) :- list(num, L).
member(X, [_|L]) :- member(X, L).
```

← **Lists**

```
:- pred member(X, T) : tree(num, T) => num(X).
:- pred member(X, T) : num(X) => tree(num, T).

member(X, tree(X, L, R)) :- tree(num, L), tree(num, R).
member(X, tree(_, L, _)) :- member(X, L).
member(X, tree(_, _, R)) :- member(X, R).
```

← **Trees**

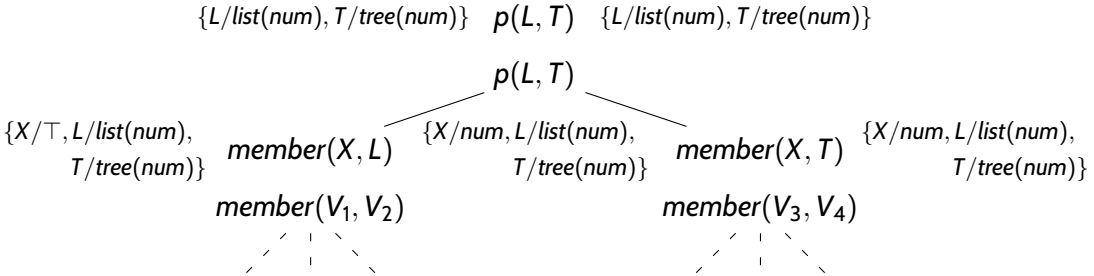
- **Checking** assertions at **every** recursive call: $\mathcal{O}(n) \implies \mathcal{O}(n^2)$
- There is much **work** on the **optimization** of run-time checks...
- Best (obviously) to **discharge** checks **statically**

$\Rightarrow$ **This work: trying to improve this further**

Framework (cont'd) — the CiaoPP analyzer (PLAI)

Abstract interpretation-based **top-down**, **multivariant** static analyzer for our target language.

- **Top-down:** builds an **analysis graph** starting from a set of initial **program entry points**
- **Multivariant:** different **call patterns** for the same **program point** are explored separately



- For the invokation of a procedure *Pred* with **call pattern** λ^c, CiaoPP analyzes the corresponding procedure and computes a **success pattern** λ^s
- **Call-success pattern** $\langle \lambda^c, Pred, \lambda^s \rangle$; in the following we use assertion notation $\rightarrow$

Analysis of the running example — no run-time checking semantics

Since the analysis is **multivariant**, we get **multiple** call-success patterns:

```
:- true pred member(X, S) : (term(X), tree(num, S)) => (num(X), tree(num,S)).
:- true pred member(X, S) : (term(X), list(num, S)) => (num(X), list(num,S)).
:- true pred member(X, S) : ( num(X),      term(S)) => (num(X),     rt42(S)).
   ...
:- regtype rt42/1.
rt42([]).
rt42([A|B]) :- term(A), rt42(B).
rt42(tree(A,B,C)) :- term(A), term(B), term(C).
```

Collapsing all patterns: $=^{\sqcup}$ $=^{\sqcup}$ $=^{\sqcup}$ $=^{\sqcup}$

```
:- true pred member(X, S) : (term(X),      term(S)) => (num(X),     rt42(S)).
```

yields results that are **not enough** for discharging many run-time checks:

```
:- checked pred member(X, T) : tree(num, T) => num(X).
:- checked pred member(X, L) : list(num, L) => num(X).
:- check   pred member(X, T) : num(X) => tree(num, T).
:- check   pred member(X, L) : num(X) => list(num, L).
```

Checked at compile time! :)
Checked at compile time! :)
Needs to be checked at run time. :(
Needs to be checked at run time. :(

Improvement 1 — run-time checking semantics in CiaoPP

Approach

Given **call** and **success** patterns λ^c and λ^s for procedure *Pred*, we let

$$\langle \lambda^c_{\mathcal{A}}, \mathit{Pred}, \lambda^s_{\mathcal{A}} \rangle \text{ where } \begin{cases} \lambda^c_{\mathcal{A}} & = \lambda^c \sqcap \alpha(\mathit{Pre}) \\ \lambda^s_{\mathcal{A}} & = \lambda^s \sqcap \alpha(\mathit{Post}) \end{cases}$$

to be new, **more** (or equally) **precise** call and success patterns for *Pred* based on the **assertion**:

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

Key insight

If we assume that the program **executes** using **run-time checking semantics**, we can **enrich** our call-success patterns with **assertions** since, if **execution reaches** the corresponding program point, then the assertion must be **true**

Improvement 1 — a small caveat

By **incorporating** the **run-time checking semantics** in CiaoPP we are **more precise**. However, **assertion checking** becomes **unsound**

Key insight

Once $\alpha(\mathit{Prop})$ is included, the enriched abstraction **trivially** satisfies *Prop*, so assertions would **always** be "verified" because we **assumed them** in the first place.

The **analysis** is **sound**. The **assertion checking step** is **not**

Solution

- **Analysis**: use the **enriched** $\lambda_{\mathcal{A}} = \lambda \sqcap \alpha(\mathit{Prop})$ for **more precision**
- **Assertion checking**: keep the **un-enriched** λ **stored** for the assertion checking process

Analysis of the running example — run-time checking semantics

Now we have:

```
:- true pred member(X, S) : (term(X), tree(num, S)) => (num(X),     tree(num,S)).
:- true pred member(X, S) : (term(X), list(num, S)) => (num(X),     list(num,S)).
:- true pred member(X, S) : ( num(X),      term(S)) => (num(X), list_or_tree(S)).
  ...
```

Collapsing all patterns: $=^{\sqcup}$ $=^{\sqcup}$ $=^{\sqcup}$ $=^{\sqcup}$

```
:- true pred member(X, S) : (term(X),      term(S)) => (num(X), list_or_tree(S)).
```

yields **more precise** results, but still **not enough** for discharging the previous run-time checks:

```
:- check pred member(X, T) : num(X) => tree(num, T).
:- check pred member(X, L) : num(X) => list(num, L).
```

Needs to be checked at run time. :(
Needs to be checked at run time. :(

Improvement 2 — abstract multiple specialization

Key insight

- Multiple **paths / call patterns** for a procedure *Pred* represent multiple **versions** of *Pred*
- For example, `append(+,+,-)` vs. `append(-,-,+)`

CiaoPP's abstract multiple specialization

- Each version is **specialized** for the corresponding calling pattern:
  - The original predicate and assertions are renamed apart as a **different procedure**
  - The unnecessary **clauses** and **assertions** are removed
- This way, **assertion checking** can be **specialized** for each version and we can **discharge more properties** at compile time

Analysis of the running example — run-time checking semantics + abstract multiple specialization

For one of the **specialized versions** of `member/2` **after analysis** we have:

```
:- true pred member_1(X, L) : (num(X), tree(num, L)) => (num(X), tree(num, L)).

member_1(X, tree(X, L, R)) :- tree(num, L), tree(num, R).
member_1(X, tree(_, L, _)) :- member_1(X, L).
member_1(X, tree(_, _, R)) :- member_1(X, R).
```

with which we can now discharge the remaining assertions:

```
:- checked pred member_1(X, T) : num(X) => tree(num, T).
```

Checked at compile time! :)

A similar case for lists, `member_2/2`, is analogous:

```
:- checked pred member_2(X, L) : num(X) => list(num, L).
```

Checked at compile time! :)

Experimental evaluation — overview

We have **implemented** and **evaluated** our approach in the **Ciao system**[1]

- Classic Prolog benchmarks:
  - `qsort`, `hanoi`, `primes`, `nqueens`, ...
  - Often a **single assertion** per predicate
- Prolog libraries
  - `atom_concat`, `sets`, `binary_trees`, `random_utils`, ...
  - **Several** (**complex**) assertions per predicate
  - **Multiple** different possible **call patterns**
- LPdoc
  - Documentation generator for logic programming (used in Ciao and XSB)
  - **22k loc** + **many** Prolog **libraries**

We analyze with **types** (`eterms`) and **variable sharing** (`shfr`) analyses

- Measure the **number** of **properties** reduced
- Finer metric: properties **within** assertions, not whole assertions

[1] `https://ciao-lang.org/` — or scan the QR code in my t-shirt! :)

Experimental evaluation — # properties verified

First, we measure the **number** of **properties** that can be **discharged** at **compile time**

We compare with:

- Static analysis **without** run-time checking semantics
- Static analysis **with** run-time checking semantics
- The approach of [Stulova et al. 2018]
- Type analysis `eterms` over the program versioned with the sharing analysis `shfr`
- Sharing analysis `shfr` over the program versioned with the type analysis `shfr`

And we **gain precision! :)**

	RT-off	[St'18]	RT-on	Vers-et-sh	Vers sh-et
Clas. bench.	90%	93%	93%	**97%**	**96%**
Libraries	45%	51%	60%	**75%**	73%
LPdoc.	42%	51%	63 %	68%	**71%**

Experimental evaluation — run time

- But... are we **discharging** the properties that make the programs **slow**?
- We evaluated the **speedups** of executing the programs, comparing with the execution with run-time checking semantics

	RT-off	[St'18]	**RT-on**	**Vers-sh-et**	**Vers sh-et**
Clas. bench	62×	81×	70×	71×	**156**×
Libraries	3×	5×	5×	**7**×	5×

Conclusion

We proposed an **approach** for addressing the problem of **reducing run-time checking overhead** for assertions

In particular,

- We **extended** the CiaoPP algorithm to **support run-time checking semantics** for inferring more precise properties about programs
- We **confirmed** that analysis **multivariance** does reduce the **run-time execution** of programs with run-time checks
- **Implemented** our approach in the **Ciao system**
- **Greatly reduced** number of not discharged properties and **reduced** execution times.

Exploiting Multiple Abstract Call Patterns for Optimizing Run-Time Checks

Daniela Ferreiro[1,2], Daniel Jurjo-Rivas[1,2], Marco Ciccalè[1,2],
José F. Morales[1,2], Pedro López-García[1,3] and Manuel V. Hermenegildo[1,2]

[1] IMDEA Software Institute
[2] Universidad Politécnica de Madrid
[3] Spanish Council for Scientific Research

ICLP'26, July 22, 2026