TPLP: Page 1–8. © The Author(s), 2021. Published by Cambridge University Press 2021
doi:10.1017/xxxxx

*Exploiting Multiple Abstract Call Patterns for Optimizing Run-Time Checks**

DANIELA FERREIRO, DANIEL JURJO-RIVAS,
MARCO CICCALÈ and JOSE F. MORALES
Universidad Politécnica de Madrid (UPM), IMDEA Software Institute, Madrid, Spain
(*e-mails:* d.ferreiro@alumnos.upm.es, daniela.ferreiro@imdea.org, daniel.jurjo@alumnos.upm.es, daniel.jurjo@imdea.org, m.ciccale@alumnos.upm.es, marco.ciccale@imdea.org, josefrancisco.morales@upm.es, josef.morales@imdea.org)

PEDRO LÓPEZ-GARCÍA
Spanish Council for Scientific Research, IMDEA Software Institute, Madrid, Spain
(*e-mails:* pedro.lopez@csic.es, pedro.lopez@imdea.org)

MANUEL V. HERMENEGILDO
Universidad Politécnica de Madrid (UPM), IMDEA Software Institute, Madrid, Spain
(*e-mails:* manuel.hermenegildo@upm.es, manuel.hermenegildo@imdea.org)

submitted xx xx xxxx; revised xx xx xxxx; accepted xx xx xxxx

Abstract

In strongly-typed languages, types are verified at compile time, while dynamically typed languages, such as Prolog, perform type consistency checks entirely at run-time. Extending dynamic languages with assertions allows expressing both classical types and more general properties, providing high expressiveness, but at the cost of run-time overhead. Abstract interpretation allows safely approximating such program properties at compile time, which has been used to reduce the number of properties that require run-time checks, while still reporting unverified properties that can guide further static analyses, testing, or domain refinement. In this work, we first study how to selectively integrate the run-time semantics of assertion properties into a multivariant, top-down, goal-directed abstract interpretation algorithm. We then show how multiple inferred calling patterns can be exploited to reduce the number of properties that must be checked at run-time, thus minimizing the overhead. Finally, we report on an implementation of our approach in the Ciao system and provide performance results showing improvements over previously reported techniques.

KEYWORDS: Abstract Interpretation, Assertions, (Constraint) Logic Programming, Verification, Run-Time Checking

1 Introduction

Detecting incorrect program behaviors during the compilation phase is an important and complex part of the software development cycle. With the advent of machine-generated

* Partially funded by MICIU project CEX2024-001471-M *María de Maeztu* and by the European Union MSCA GA 101154447 NEAT. We would also like to thank the anonymous reviewers for their very useful and constructive feedback.

code, and as more non-experts are able to write complex software, ensuring that semi-automatically generated programs behave as expected has become increasingly important. To aid in this process, a number of tools have been developed to compare actual program behavior against expectations, such as code analyzers/verifiers and run-time verification frameworks. These tools rely on language-level constructs to describe expected program behavior. Approaches that fall into this category include assertion-based frameworks used in (Constraint) Logic Programming ((C)LP) (Drabent et al. 1988; Puebla et al. 1997; Bueno et al. 1997; Boye et al. 1997; Hermenegildo et al. 1999; Puebla et al. 2000b; Laï 2000; Hermenegildo et al. 2005; Mera et al. 2009; Hanus 2017), soft/gradual typing approaches in functional programming (Cartwright and Fagan 1991; Findler and Felleisen 2002; Tobin-Hochstadt and Felleisen 2008; Dimoulas and Felleisen 2011; Rastogi et al. 2015; Takikawa et al. 2015; 2016; Vazou et al. 2018) and contract-based extensions in object-oriented programming (Lamport and Paulson 1999; Leavens et al. 2007; Fähndrich and Logozzo 2011). These approaches often involve a certain degree of run-time testing, especially for non-trivial properties. However, such testing can incur significant performance overhead (Mera et al. 2009), even when performing simple type checks between typed and untyped parts of programs (Rastogi et al. 2015; Takikawa et al. 2016).

Some proposals have been made to reduce the run-time overhead of assertion checking by optimizing the run-time checking mechanisms themselves, at the expense of increased memory consumption (Koukoutos and Kuncak 2014; Stulova et al. 2015). Repeated checks on immutable recursive data structures are converted from execution-time overhead into increased memory use via caching and/or tabling techniques. Despite these advances, full run-time checking often remains impractically expensive, especially for complex properties such as deep data structure tests. This reduces the attractiveness of run-time checking to programmers, who may allow sporadic checking of very simple conditions but tend to disable run-time checking for more complex properties. Motivated by this problem, assertion-based frameworks have been proposed in which static analysis is used to reduce the number of program points where run-time checks are required, or to rule out incorrect program behaviors (Puebla et al. 1997; Bueno et al. 1997; Hermenegildo et al. 1999). A number of practical *assertion checking modes* have been studied, representing different trade-offs between code annotation depth, execution-time slowdown, and program behavior safety guarantees (Stulova et al. 2018). The latter proposed a method to modify the semantics inferred by the analyzer via *program transformations* to capture executions with active run-time checking. While this technique had limitations, it showed the potential to increase the number of checks verified at compile time.

In this work, we take a complementary approach by incorporating the run-time checking semantics directly into the analyzer, but making it optional, allowing abstract executions to more closely match their concrete counterparts. By making the presence or absence of run-time checks part of the analyzed semantics, the analyzer can reason precisely about program executions under different assertion checking configurations. We also study the effect of multivariant analysis in this setting, which makes it possible to distinguish between different calling contexts and different versions of the same predicate or procedure, depending on the properties known to hold and on whether run-time checks are required. Multivariant analysis is particularly well suited to (C)LP, where predicates are naturally invoked under different modes of use. If such

$$\langle \theta' :: G \,|\, \theta \rangle \rightsquigarrow \langle G \,|\, \theta \wedge \theta' \rangle \qquad \textit{if } (\theta \wedge \theta') \not\models \textit{false} \qquad \text{(CONSTR)}$$
$$\langle H :: G \,|\, \theta \rangle \rightsquigarrow \langle B :: G \,|\, \theta \rangle \qquad \textit{if } \exists (H \texttt{ :- } B) \in \mathcal{P} \qquad \text{(ATOM)}$$

Figure 1: Reduction rules for the operational semantics of (C)LP programs.

modes are not taken into account, the analysis is forced to merge distinct execution contexts, often leading to a loss of precision and overly conservative decisions regarding run-time checking. By explicitly representing different analysis versions corresponding to different calling modes and run-time checking configurations, we study how making these versions explicit can improve analysis precision and run-time checking efficiency. Finally, beyond measuring how many assertions are completely verified, as is common practice, we also evaluate whether individual properties within an assertion are checked. Note that assertions often consist of multiple properties of varying complexity: for example, in a formula such as ($simple\_property(X)$, $complex\_property(Y)$, $less\_complex\_property(Z)$), some properties (such as $simple\_property(X)$) may be fully verified statically, others ($complex\_property(Y)$) may never be discharged, and yet others ($less\_complex\_property(Z)$) may be verified only under certain calling contexts. We aim to capture this finer-grained information, allowing us to quantify how many properties of each assertion are verified, under which conditions, and how many remaining properties require run-time checking, with special attention to multivariance.

2 Preliminaries and notation

Variables start with a capital letter. The set of terms is inductively defined as follows: (1) variables are terms (2) if f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. An *atom* has the form $p(t_1, \ldots, t_n)$ where p is an n-ary predicate symbol, and $t_1, \ldots, t_n$ are terms. A *constraint* is a conjunction of expressions built from predefined predicates whose arguments are constructed using predefined functions and variables, *e.g.*, $X - Y > abs(Z)$. A *literal* is either an atom or a constraint. We denote the set of variables of a literal L by $\mathsf{vars}(L)$. *Negation* is encoded as finite failure, supported through a program expansion. A *goal* is a finite sequence of literals. A *rule* has the form H `:-` B where H, the *head*, is an atom and B, the *body*, is a possibly empty finite sequence of literals. A *predicate* is a set of rules with the same head. We refer to a predicate by either its function symbol and arity p/n, or its head $p(t_1, \ldots, t_n)$. A *constraint logic program*, or *program*, is a finite set of rules. For simplicity, we assume a single program $\mathcal{P}$ in which every rule has distinct variables in its head; and a class of constraints—over a given constraint domain—for which projection and a complete solver exist.

Operational semantics. The operational semantics, adapted from that of Jaffar et al. (1998), is given in terms of *derivations* from goals. Derivations are sequences of *reductions* between *states*. A state $\langle G \,|\, \theta \rangle$ consists of a goal G, and the current constraint θ. We denote sequence concatenation by (::). We use $S \rightsquigarrow S'$ to indicate that a reduction step can be applied to state S to obtain state S'. And $S \rightsquigarrow^* S'$ to indicate that there exists a sequence of reduction steps from S to S'. Figure 1 depicts the reduction rules. Intuitively,

satisfiable constraints are added to the current constraint (CONSTR), and atoms are reduced by unfolding them into the bodies of matching program rules (ATOM).

Given a literal L and reduction steps $S \rightsquigarrow^* S'$ from state $S = \langle L :: G \,|\, \theta \rangle$ to state $S' = \langle G \,|\, \theta' \rangle$, we refer to S as a *call state* for L, and to S' as a *success state* for L.

A *query* is a pair (L, θ), where L is a literal and θ is a constraint for which the (C)LP system starts a computation from state $\langle L \,|\, \theta \rangle$. A finite derivation from a query $Q = (L, \theta)$ is *finished* if the last state in the derivation cannot be reduced. Such a finished derivation is *successful* if the last state is of the form $\langle \square \,|\, \theta' \rangle$, where $\square$ denotes the empty goal sequence, and the projection of the constraint θ' onto the variables $\mathsf{vars}(L)$ is an *answer* to Q. Conversely, a derivation is *failed* if the last state is not of the form $\langle \square \,|\, \theta \rangle$. We denote by $\mathsf{answers}(Q)$ the set of answers to the query Q.

Properties & property formulas. Conditions on the current constraint are stated as *property formulas*, which are conjunctions of *properties*. Properties are predicates, typically defined in the source language, and thus runnable. However, not all predicates can be used as properties. They are generally required to be checkable at run time so that they can be effectively used as run-time checks, but not necessarily decidable at compile time, where they can be safely approximated (Hermenegildo et al. 1999; Puebla et al. 2000b).

Example 2.1 (*Properties*)
The following are the definitions of the list/2, tree/2 and list_or_tree/2 properties:

```
% Lists
:- prop list/2.
list(_,[]).
list(P,[X|L]) :-
    P(X),
    list(P,L).
```

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

```
% Lists or binary trees
:- prop list_or_tree/2.
list_or_tree(P,L) :-
    list(P,L).
list_or_tree(P,T) :-
    tree(P,T).
```

Note that all definitions are parametric *on an additional property (their first argument), which is then called using higher-order notation (*`P(X)`*).*

Assertions. Assertions are syntactic objects which allow expressing properties of programs that must be satisfied at certain points of program execution. We recall the relevant parts of the assertion schema of Puebla et al. (2000a). *Predicate* (or *pred*) assertions have the following syntax:

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

where *Pred* is an atom representing a predicate, and *Pre* and *Post* are property formulas. They express that all calls to *Pred must* satisfy the precondition *Pre*, and, if such calls succeed, the postcondition *Post must* be satisfied. If there are several *pred* assertions, the *Pre* field of at least one of them *must* be satisfied. To simplify both presentation and use, assertions are normalized into a set of corresponding *assertion conditions*.

Definition 2.1 (*Assertion conditions*)
Given a predicate Pred and its corresponding set of assertions $\{A_1, \ldots, A_n\}$ *with* A_i = `:-` **`pred`** *Pred* `:` Pre_i `=>` $Post_i$, *its set of* assertion conditions *is defined as*

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

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

Figure 2: An example Prolog program with an assertion capturing its behavior.

$$\langle \theta' :: G \,|\, \theta \rangle \rightsquigarrow_{\mathcal{A}} \langle G \,|\, \theta \wedge \theta' \rangle \quad \textit{if } (\theta \wedge \theta') \not\models \textit{false} \quad (\textsc{Constr}_{\mathcal{A}})$$

$$\langle H :: G \,|\, \theta \rangle \rightsquigarrow_{\mathcal{A}} \langle \mathsf{wrap}_{\mathcal{A}}(H, B, \theta) :: G \,|\, \theta \rangle \quad \textit{if } \exists (H \texttt{ :- } B) \in \mathcal{P} \quad (\textsc{Atom}_{\mathcal{A}})$$

$$\langle \mathsf{c}(F) :: G \,|\, \theta \rangle \rightsquigarrow_{\mathcal{A}} \langle G \,|\, \theta \rangle \quad \textit{if } \mathsf{check}(F, \theta) \quad (\textsc{OkCheck}_{\mathcal{A}})$$

$$\langle \mathsf{c}(F) :: G \,|\, \theta \rangle \rightsquigarrow_{\mathcal{A}} \langle \mathsf{e}(F) \,|\, \theta \rangle \quad \textit{if } \neg\mathsf{check}(F, \theta) \quad (\textsc{ErrCheck}_{\mathcal{A}})$$

$$\mathsf{check}(F, \theta) \triangleq \exists \theta' \in \mathsf{answers}((F, \theta)).\ \theta \models \theta'$$

$$\mathsf{wrap}_{\mathcal{A}}(H, B, \theta) \triangleq \mathsf{c}(\mathit{Pre}) :: B :: \{\mathsf{c}(\mathit{Post}_i) \mid \mathsf{success}(H, \mathit{Pre}_i, \mathit{Post}_i) \in \mathcal{A} \wedge \mathsf{check}(\mathit{Pre}_i, \theta)\},$$
$$\textit{with } \exists !\mathsf{calls}(H, \mathit{Pre}) \in \mathcal{A}$$

Figure 3: Reduction rules for the operational semantics of (C)LP programs with assertions.

$\{C_0, C_1, \ldots, C_n\}$ *with:*

$$C_i = \begin{cases} \mathsf{calls}(\mathit{Pred}, \bigvee_{j=1}^{n} \mathit{Pre}_j) & i = 0 \\ \mathsf{success}(\mathit{Pred}, \mathit{Pre}_i, \mathit{Post}_i) & i \in \{1, \ldots, n\} \end{cases}$$

C_0 encodes the checks that the calls to the predicate represented by Pred are within those admissible by the set of assertions. We refer to it as the calls assertion condition. *$C_1, \ldots, C_n$ encode the checks for compliance of the success states for particular sets of calls, and we call them the* success assertion conditions.

We denote by $\mathcal{A}$ both the set of assertions of the program and, interchangeably, its associated set of assertion conditions. And we assume that, if there are no assertions associated with a predicate *Pred*, its set of assertion conditions is defined as $\{\mathsf{calls}(\mathit{Pred}, \mathit{true}), \mathsf{success}(\mathit{Pred}, \mathit{true}, \mathit{true})\}$; that is, the most-general assertion/assertion conditions.

Example 2.2 (*Member program with assertions*)
Consider the program in Figure 2, which defines the predicate member(X, S), relating a number X with a structure S. As specified by its assertion, it succeeds when the number X occurs at some position in the structure S, where S may be a list *or a* binary tree.

Run-time assertion checking. Run-time assertion checking consists of *dynamically* checking that the conditions imposed by the assertions hold while computing the derivations from a query. To this end, we incorporate instrumental checks into the operational semantics in Figure 1.

We extend the set of literals with additional syntactic objects: *check* literals of the form $\mathsf{c}(F)$, where F is a property formula to be checked at run-time; and *error* literals of the form $\mathsf{e}(F)$, where F is a property formula that has been violated.

We use $S \rightsquigarrow_{\mathcal{A}} S'$ to indicate run-time checking reductions *w.r.t.* a set of assertions $\mathcal{A}$. Figure 3 depicts the reduction rules and some auxiliary functions. Intuitively, atoms

are still reduced by unfolding them into the bodies of matching program rules. However, the body is now preceded by a run-time check for the predicate's pre-condition, *i.e.*, its *calls* assertion condition, and followed by a sequence of run-time checks for the predicate's applicable post-conditions, *i.e.*, its set of *success* assertion conditions whose pre-condition held at the atom's call state (ATOM$_{\mathcal{A}}$). Run-time checks are computed by means of the check function, which determines whether a property formula F holds under the current constraint θ by searching for a solution for F that is entailed by θ. Since run-time checks may fail, we introduce the notion of (finite) *erroneous* derivations, whose final state is of the form $\langle \mathsf{e}(F) \,|\, \theta \rangle$ indicating that the run-time check of F under θ was *false* (ERRCHECK$_{\mathcal{A}}$).

While useful, run-time assertion checking can be prohibitively expensive in both time and memory. Consider calling the *member/2* predicate in Example 2.2 with the calling mode expressed by `:- pred` $member(X, S)$ `:` $(var(X), list\_or\_tree(num, S))$. Then, checking that S is a list at each recursive step turns the $\mathcal{O}(n)$ algorithm into $\mathcal{O}(n^2)$.

Abstract interpretation. The main idea behind *abstract interpretation* (Cousot and Cousot 1977) is to interpret the program over a special, abstract domain whose elements are finite representations of possibly infinite sets of states in the concrete domain. We denote the concrete domain as D_γ, the abstract domain as D_α, and the functions that relate sets of states with their abstractions as the *abstraction* function $\alpha : D_\gamma \to D_\alpha$ and the *concretization* function $\gamma : D_\alpha \to D_\gamma$. The concrete domain is typically a complete lattice with the set inclusion order which induces an ordering relation in the abstract domain represented by $\sqsubseteq$. Under this relation the abstract domain is usually a complete lattice and $(D_\gamma, \alpha, D_\alpha, \gamma)$ is a Galois insertion/connection (Cousot and Cousot 1977).

Top-down analyses are a family of static analyses that build an *analysis graph* starting from a series of program *entry points*. This approach was first used in analyzers such as MA3 and Ms (Warren et al. 1988), and matured in the PLAI analyzer (Muthukumar and Hermenegildo 1990; 1992) using an optimized fixpoint algorithm now also referred to as the *top-down algorithm* or *solver*. This algorithm was later applied to the analysis of CLP/CHCs (García de la Banda et al. 1996)[1] and imperative programs (De Angelis et al. 2021; Henriksen and Gallagher 2006; Méndez-Lojo et al. 2007a;b), and used in analyzers such as GAIA (Le Charlier and Van Hentenryck 1994), the CLP($\mathcal{R}$) analyzer (Kelly et al. 1998), or Goblint (Seidl and Vogler 2021; Tilscher et al. 2023). The graph constructed by the PLAI algorithm during analysis is a finite, abstract object whose concretization approximates the (possibly infinite) set of (possibly infinite) maximal AND-trees of the concrete semantics. This follows the overall abstraction scheme of Bruynooghe (1991), but implementing it with an efficient fixpoint algorithm that uses tabling, dependency-based acceleration, incrementality, etc. PLAI separates the abstraction of the structure of the concrete trees (the paths through the program) from the abstraction of the *constraints* at the nodes in those concrete trees (the program states in those paths). The first abstraction, T_α, is typically built-in, as an abstract domain of *analysis graphs*. The framework is *parametric* on a second abstract domain, D_α, whose elements appear as labels in the nodes of the analysis graph. Each node of the analysis graph is of the

[1] The cited paper showed that CLP analysis can in fact be done with the same techniques used for LP provided suitable abstractions are provided in the abstract domains for the constraint primitives.

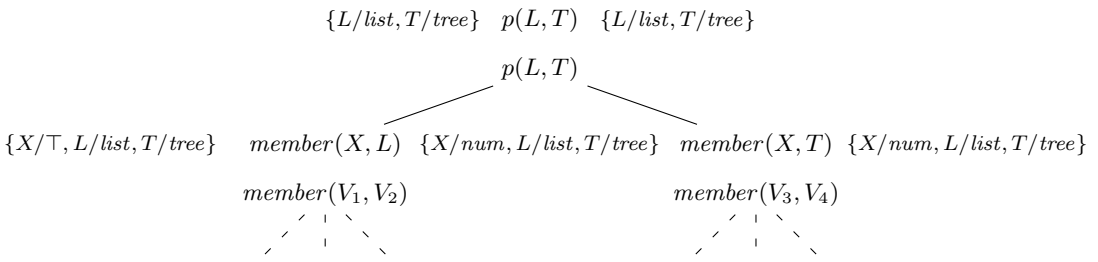

Figure 4: Analysis graph of the *p/2* predicate in Figure 5 with the abstract domain in Figure 6.

```
p(L,T) :-
    member(X,L),
    member(X,T).
```

Figure 5: Program with different calling modes.

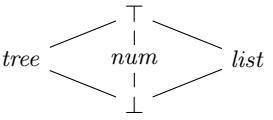

Figure 6: Abstract domain lattice.

form $\langle \lambda^c, H, \lambda^s \rangle$ with H a call pattern for a predicate of the program and λ^c and λ^s the abstractions of the call and success states.

Multivariance and path-sensitivity. The abstract semantics computed by PLAI is *multivariant*. That is, for a single program point, it computes separate path information for each of its *calling contexts* (also referred to as *variants*, *versions* or *modes*). In the analysis graph, this is represented by multiple nodes corresponding to the same program point, each abstracting a different calling mode.

Example 2.3 (*Analysis*)
Consider the simple abstract domain depicted in Figure 6. Figure 4 shows a (partial) analysis graph for the program in Figure 5. Notice that the success abstraction of $member(X, L)$ *becomes the call abstraction of* $member(X, T)$*. Notice also how two different versions for member/2 do appear, with (abstract) call patterns* $\{X/\top, L/list, T/tree\}$ *and* $\{X/num, L/list, T/tree\}$*, respectively.*

Compile-time assertion verification & simplification. Compile-time assertion verification consists of *statically* checking that the conditions imposed by the assertions hold for *all* possible derivations of a program from a set of queries. This verification can be performed using the analysis graph computed by PLAI, which is an over-approximation of the program's semantics for all such derivations. However, since PLAI is multivariant, a single program point H may correspond to multiple nodes in the analysis graph: $\langle \lambda_1^c, H, \lambda_1^s \rangle, \ldots, \langle \lambda_n^c, H, \lambda_n^s \rangle$, with each node over-approximating the semantics of that point for *some* derivations. Consequently, the over-approximation of the semantics of a program point for *all* derivations is the combination (*lub*) of all corresponding nodes in the analysis graph: $\langle \sqcup\{\lambda_1^c, \ldots, \lambda_n^c\}, H, \sqcup\{\lambda_1^s, \ldots, \lambda_n^s\} \rangle$. This can then be used for *simplifying* (as much as possible) the property formulas in the corresponding assertion conditions. That is, given a program point and its abstraction, properties *implied* by the abstraction simplify to *true*, properties *incompatible* with it simplify to *false*, while the remaining

ones remain unchanged. This approach has the advantage that, even if a property formula cannot be proved in its entirety—given the undecidable nature of (non-trivial) semantic properties—it may be possible to discharge some parts of it. The verification results are reported as changes in the status and transformations of the assertions. An assertion is `checked` if both property formulas are simplified to *true*, and `false` if some property formula is simplified to *false*. If some property formula cannot be fully proved, the assertion is `check`, and its property formulas are replaced with their simplified versions.

From this point on, we assume that the program $\mathcal{P}$, the set of assertions $\mathcal{A}$ and the analysis graph $\mathcal{G}$ are implicit variables.

3 Assertion checking with abstract run-time checking semantics

The operational semantics of (C)LP with run-time assertion checking (cf. Figure 3) differs from the default (C)LP semantics (cf. Figure 1) in that the latter can be seen as an over-approximation of the former. Intuitively, run-time assertion checking may cause certain program points to become unreachable due to failing checks, whereas they would be reachable under the default semantics. As a result, for a given program $\mathcal{P}$, the set of AND-trees generated under the default (C)LP semantics is a superset of those generated when run-time assertion checking is enabled. Thus, abstract interpretation analyses based on the default semantics are correct but may suffer from a loss of precision, since they abstract a set of AND-trees that over-approximates the actual execution space. In contrast, analyzing programs under run-time checking semantics allows us to exploit the additional information provided by assertions. Concretely, when entering a predicate definition, we may assume that the corresponding calls assertion condition holds, and thus enrich the call abstraction λ^c with this information. Similarly, upon predicate exit, we can assume that the relevant success assertion conditions hold and use them to refine the resulting success abstraction λ^s.

Stulova et al. (2018) propose a program transformation based on the introduction of additional link predicates with assertions that enriches the analysis with the run-time checking semantics. In contrast, in this work we focus on integrating run-time checking semantics directly into the top-down abstract interpretation framework without transforming the original program. However, since the information in the assertions is now included in the abstractions, those abstractions would trivially satisfy the assertions they are intended to check, turning the analysis results unsuitable for assertion checking. To address this, we generate alternative call and success abstractions at those program points where the assertions' information causes a gain in precision, and use these during the (abstract) assertion checking phase to ensure correct behavior. We now describe this technique in detail.

Assertion-based abstraction refinement. Assume a literal L in the body of some clause in the program, and a rule H `:-` B *s.t.* H unifies with L. Assume also that the constraint affecting L at the time of this call is approximated by the abstraction λ^c such that $\mathsf{vars}(L) \subseteq \mathsf{dom}(\lambda^c)$ and $\mathsf{vars}(\lambda^c) \cap (\mathsf{vars}(H) \cup \mathsf{vars}(B)) = \varnothing$. The success (exit state)

Algorithm 1 absAssrtProjs function.

```
1 function absAssrtProjs(λ^pr, L)
2     C ← {α(F_c) | calls(L, F_c) ∈ A}
3     λ^pr_rt ← ⊓(C ∪ {λ^pr})
4     if λ^pr_rt ⊑ λ^pr then
5         storeAltCall(L, λ^pr)
6     return λ^pr_rt
```

Algorithm 2 absAssrtPrimes function.

```
1 function absAssrtPrimes(λ^pr, λ^p, L)
2     S ← {α(F_s) | success(L, F_c, F_s) ∈ A ∧ λ^pr ⊑ α(F_c)}
3     λ^p_rt ← ⊓(S ∪ {λ^p})
4     if λ^p_rt ⊑ λ^p then
5         storeAltSucc(L, λ^pr, λ^p)
6     return λ^p_rt
```

of L after executing the rule above is represented by the abstraction λ^s given by:

$$\begin{aligned}
\lambda^s &= \mathsf{extend}(\lambda^c, L, \lambda^p_{rt}) \\
\lambda^p_{rt} &= \mathsf{absAssrtPrimes}(\lambda^{pr}_{rt}, \lambda^p, L) \\
\lambda^p &= \mathsf{exitToPrime}(\mathsf{project}(\mathsf{vars}(H), \lambda^{ex}), H, L) \\
\lambda^{ex} &= \mathsf{entryToExit}(\lambda^{en}, H, B) \\
\lambda^{en} &= \mathsf{augment}(\mathsf{vars}(B) \backslash \mathsf{vars}(H), \mathsf{callToEntry}(\lambda^{pr}_{rt}, L, H)) \\
\lambda^{pr}_{rt} &= \mathsf{absAssrtProjs}(\lambda^{pr}, L) \\
\lambda^{pr} &= \mathsf{project}(\mathsf{vars}(L), \lambda^c)
\end{aligned}$$

The analyzer operates as follows: (**1**) First, it projects the call abstraction (λ^c) over the variables in the literal (L). (**2**) The computed λ^{pr} assertion is then enriched by incorporating the *glb* ($\sqcap$) of the assertion pre-conditions for the predicate of L. This is achieved by invoking the absAssrtProjs (Algorithm 1). If the resulting abstraction is more precise than the initial λ^{pr}, an alternative call abstraction is stored for use during the assertion checking phase (Algorithm 1, lines 4–6). (**3**) Next, it performs abstract unification using the callToEntry function. (**4**) The resulting abstraction is then augmented with any variables in the body not present in the head. (**5**) The clause body is then traversed, and each literal is analyzed using the entryToExit function, producing an exit abstraction (λ^{ex}). (**6**) The abstract unification from step (2) is reverted via exitToPrime, yielding a prime abstraction (λ^p). (**7**) This abstraction is then enriched with the applicable assertion post-conditions by computing absAssrtPrimes (Algorithm 2). As in step (2), if the abstraction obtained is more precise than the initial one, an alternative success abstraction is stored (Algorithm 2, lines 4–6). (**8**) Finally, this abstraction is incorporated into the original call, returning the success abstraction (λ^s) through the extend function.

As some final remarks, if no predicate head can be unified with the literal under analysis, a bottom abstraction ($\bot$) is returned (representing that the exit state is unreachable). If several clauses are available, all of them are analyzed, and a collection of *prime* abstractions $\lambda^p_1, \ldots, \lambda^p_m$ is obtained, one abstraction per clause, where m is the number of clauses. Then, the success abstraction is computed as $\lambda^s = \mathsf{extend}(\lambda^c, \sqcup\{\lambda^p_1, \ldots, \lambda^p_m\})$ by means of the *lub* of the collection of abstractions (other operators, including disjunction and widenings, are possible).

4 Exploiting versions for more optimization

By default, after computing the analysis graph, PLAI produces an output that is easy for the programmer to inspect, *i.e.*, close to the source program, where the abstract semantics of each predicate is the *combination* of all its versions. That is, for a predi-

Algorithm 3 vers program transformation.

```
1  function vers
2      V ← ∅
3      for Pred in the program P do
4          V ← dump(Pred, V)
5          P ← P \ {H :- B ∈ P | H unifies with Pred}
6          A ← A \ {calls(Pred, _) ∈ A}
7          A ← A \ {success(Pred, _, _) ∈ A}
8      for H :- B ∈ P do
9          B' ← replace(B, V)
10         P ← P ∪ {H :- B'}
11
12 function dump(H,V)
13     R ← {H :- B ∈ P}
14     for ⟨λ^c, H, λ^s⟩ ∈ G do
15         H' ← newHead(H)
16         R' ← {H' :- B | H :- B ∈ R}
17         P ← P ∪ R'
18         A ← A ∪ {calls(H', F_c) | calls(H, F_c) ∈ A}
19         A ← A ∪ {success(H', F_c, F_s) | success(H, F_c, F_s) ∈ A}
20         G ← G ∪ {⟨λ^c, H', λ^s⟩}
21         V ← V ∪ {H : H'}
22     return V
23 function replace(B,V)
24     if B = □ then
25         return □
26     else B = L :: G
27         if ∃H s.t. ∃!⟨λ^c, L, λ^s⟩ ∈ G
                  s.t. ∃!⟨λ^c, H, λ^s⟩ ∈ G
                  s.t. L : H ∈ V then
28             return H :: replace(G, V)
29         else
30             return L :: replace(G, V)
```

cate *Pred*, the multiple nodes $\{\langle\lambda_1^c, Pred, \lambda_1^s\rangle, \ldots, \langle\lambda_n^c, Pred, \lambda_n^s\rangle\}$ in the analysis graph corresponding to the different calling modes of *Pred*, are *collapsed* into a single node $\langle\sqcup\{\lambda_1^c, \ldots, \lambda_n^c\}, Pred, \sqcup\{\lambda_1^s, \ldots, \lambda_n^s\}\rangle$. This approach, while convenient for the end-user, may incur a significant loss of precision, not due to the analysis itself, but rather to the over-simplification of the abstractions. To address this issue, PLAI is also capable of producing an output where multiple versions of each original predicate *Pred*, *i.e.*, multiple nodes for *Pred*, are *materialized* into different predicates which, through a program transformation, are "folded back" to the program. That is, for a predicate *Pred*, the multiple nodes: $\{\langle\lambda_1^c, Pred, \lambda_1^s\rangle, \ldots, \langle\lambda_n^c, Pred, \lambda_n^s\rangle\}$ in the analysis graph corresponding to the different calling modes of *Pred* are *materialized* into different predicates for each version: $\{\langle\lambda_1^c, Pred_1, \lambda_1^s\rangle, \ldots, \langle\lambda_n^c, Pred_n, \lambda_n^s\rangle\}$.

A high-level view of the program transformation is depicted in Algorithm 3. For each predicate in the program, we first dump its versions (line 4). That is, for each version of a predicate in the analysis graph (line 14), a copy is created by generating a new head (line 15) and attaching to it the corresponding rules (line 16), assertions (lines 18–19), and analysis graph node (line 20), updating the program, assertion set, and analysis graph accordingly. Additionally, a *versions mapping* V is created in order to record which versions correspond to the original predicates (line 21). Then, each rule in the program is rewritten by replacing calls to the original predicates, under a specific calling mode, with calls to the corresponding specialized predicates generated by dump (lines 9 and 10). Concretely, during the traversal of a rule's body, any literal corresponding to a predicate call for which a specialized version exists for the given call pattern is replaced by a call to that specialized version (line 27).

Example 4.1 (*Exploiting versions*)
Applying the vers *transformation to the program in Figure 2 (with the analysis information of the Eterms abstract domain (Vaucheret and Bueno 2002)) generates two versions of the predicate for its possible abstract call patterns (member_1/2 and member_2/2):*

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

member(X,[X|L]) :- list(num,L).
member(X,[_|L]) :- member_2(X,L).
```

For instance, for the analysis version with calling abstraction $\{X/term, L/list(num)\}$*, the member_2/2 version with success abstraction* $\{X/num, L/rt1\}$ *is created.*

```
:- true pred member_2(X,L) : (term(X), list(num,L)) => (num(X), rt1(L)).
:- prop rt1/1. rt1([A|B]) :- num(A), list(num,B). %
```

The obtained success abstraction is strictly more precise *than that obtained by analyzing the program in Figure 2 with the Eterms abstract domain. Therefore, all assertions proven to be checked in the original program remain checked.*

```
:- checked pred member_2(X,S) : (var(X), list_or_tree(num,S)) => num(X).
```

However, without versioning, recursive calls to member/2 must be checked at run time. In contrast, with versions, recursive calls are directed to the specialized predicate version, for which the assertions have already been statically verified. As a result, additional run-time checks are unnecessary.

5 Experimental evaluation

In order to assess the benefits of the proposed techniques, we evaluate two sets of benchmarks. These benchmarks include a variety of examples, ranging from a collection of classic benchmarks for Prolog to a subset of libraries of the Ciao system. While the predicates in the first set often have a single assertion per predicate, those in the second set typically present more assertions per definition, describing several possible call patterns. Our objective is to measure the effects of applying different techniques for the optimization of run-time checks. To this end, instead of measuring the number of assertions that are completely checked, *i.e.*, those that are reduced to *true*, we count the number of properties that are simplified within the assertions. We use this metric because assertions can sometimes contain one or more properties that are hard to verify, while other properties in the assertion are verified. The metric chosen allows us to measure the effects of different techniques at a finer level of granularity. Then, we also evaluate the impact of these techniques on the execution times of the programs when executed under run-time checking semantics. The experiments were run on a MacBook Air with the Apple M1 chip and 16 GB of RAM, with a per-run timeout of 3 minutes.

Table 1 presents the results of the first evaluation. Each column displays the number of properties actually verified ("checked") at compile time, the total number of properties in the corresponding module, and the percentage of properties checked. These experiments use the *Eterms* and *Sharing-Freeness* abstract domains. Column **ct** presents results when using the information obtained from the classic fixpoint with default semantics. Column **ctrt_f** shows results when executing the fixpoint with run-time checking semantics, as defined in Section 3. Column **ctrt_f_vers_etsh** (resp. **ctrt_f_vers_shet**) displays results when executing the fixpoint with run-time checking semantics, followed by the versions transformation (*i.e.*, materializing the versions contained in the multivariant information of *Eterms*), and then analyzing these using *Sharing-Freeness* (resp. with the multivariant information of *Sharing-Freeness* and analyzing using *Eterms*). A domain combination approach can be used to join versions but we wanted to observe the effects of the versions of each domain separately. The results show that applying run-time checking semantics to the analyzer increases the number of verified properties. The transformation obviously raises the total property count (since assertions are duplicated across versions),

Table 1: Number of reduced properties (%).

module	ct	ctrt	ctrt_f	ctrt_f_vers_etsh	ctrt_f_vers_shet
exp	18/21 (85.71%)	18/21 (85.71%)	18/21 (85.71%)	27/30 (90.00%)	18/21 (85.71%)
factorial	4/6 (66.67%)	4/6 (66.67%)	4/6 (66.67%)	4/6 (66.67%)	**10/12 (83.33%)**
fft	69/95 (72.63%)	92/95 (96.84%)	91/95 (95.79%)	175/179 (97.77%)	214/218 (98.17%)
fib	16/18 (88.89%)	16/18 (88.89%)	16/18 (88.89%)	**40/42 (95.24%)**	28/30 (93.33%)
guardians	51/54 (94.44%)	51/54 (94.44%)	51/54 (94.44%)	123/126 (97.62%)	69/72 (95.83%)
hamming	94/96 (97.92%)	94/96 (97.92%)	94/96 (97.92%)	299/303 (98.68%)	148/150 (98.67%)
hanoi	19/24 (79.17%)	19/24 (79.17%)	19/24 (79.17%)	37/42 (88.10%)	52/57 (91.23%)
jugs	42/45 (93.33%)	42/45 (93.33%)	42/45 (93.33%)	66/69 (95.65%)	54/57 (94.74%)
knights	58/60 (96.67%)	58/60 (96.67%)	58/60 (96.67%)	105/107 (98.13%)	70/72 (97.22%)
mmatrix	14/17 (82.35%)	14/17 (82.35%)	14/17 (82.35%)	14/17 (82.35%)	14/17 (82.35%)
nreverse	13/15 (86.67%)	13/15 (86.67%)	13/15 (86.67%)	22/24 (91.67%)	**46/48 (95.83%)**
poly	60/63 (95.24%)	60/63 (95.24%)	60/63 (95.24%)	159/162 (98.15%)	249/252 (98.81%)
primes	28/30 (93.33%)	28/30 (93.33%)	28/30 (93.33%)	52/54 (96.30%)	37/39 (94.87%)
progeom	67/69 (97.10%)	67/69 (97.10%)	67/69 (97.10%)	230/232 (99.14%)	98/100 (98.00%)
qsort	27/28 (96.43%)	27/28 (96.43%)	27/28 (96.43%)	36/37 (97.30%)	57/58 (98.28%)
queens	24/26 (92.31%)	24/26 (92.31%)	24/26 (92.31%)	54/56 (96.43%)	41/43 (95.35%)
serialize	19/21 (90.48%)	19/21 (90.48%)	19/21 (90.48%)	29/31 (93.55%)	29/31 (93.55%)
tak	8/12 (66.67%)	8/12 (66.67%)	8/12 (66.67%)	8/12 (66.67%)	**20/24 (83.33%)**
total	631/700 (90.14%)	654/700 (93.41%)	653/700 (93.28%)	**1480/1529 (96.80%)**	1254/1301 (96.39%)
atom_concat	0/5 (0.00%)	0/5 (0.00%)	0/5 (0.00%)	**13/15 (86.67%)**	0/20 (0.00%)
formulae	12/21 (57.14%)	12/21 (57.14%)	16/21 (76.19%)	26/35 (74.29%)	44/70 (62.86%)
iso_char	13/39 (33.33%)	22/39 (56.41%)	13/39 (33.33%)	56/87 (64.37%)	37/93 (39.78%)
lists	51/106 (48.11%)	50/106 (47.17%)	51/106 (48.11%)	108/183 (59.02%)	88/174 (50.57%)
numlists	6/11 (54.55%)	6/11 (54.55%)	6/11 (54.55%)	11/16 (68.75%)	11/16 (68.75%)
random_utils	34/56 (60.71%)	35/56 (62.50%)	35/56 (62.50%)	52/73 (71.23%)	52/73 (71.23%)
sets	0/46 (0.00%)	0/46 (0.00%)	0/46 (0.00%)	2/49 (4.08%)	0/46 (0.00%)
amqueue	7/34 (20.59%)	7/34 (20.59%)	7/34 (20.59%)	**108/147 (73.47%)**	32/123 (26.02%)
binary_tree	6/19 (31.58%)	9/19 (47.37%)	9/19 (47.37%)	**33/38 (86.84%)**	19/38 (50.00%)
btree	timeout	78/113 (69.03%)	68/113 (60.18%)	345/426 (80.99%)	467/558 (83.69%)
heap	77/90 (85.56%)	79/90 (87.78%)	85/90 (94.44%)	187/192 (97.40%)	451/456 (98.90%)
rbtree	101/132 (76.52%)	122/132 (92.42%)	124/132 (93.94%)	112/120 (93.33%)	112/120 (93.33%)
sets_ops	32/77 (41.56%)	32/77 (41.56%)	32/77 (41.56%)	103/161 (63.98%)	112/158 (70.89%)
total	339/749 (45.26%)	382/749 (51.00%)	446/749 (59.55%)	**1156/1542 (74.97%)**	1425/1945 (73.26%)

but the usage of multivariant information yields overall better percentages. Notably, different modules show varied improvements based on whether the multivariant information inferred from *Sharing-Freeness* or *Eterms* is used for the versions transformation.

Table 2 presents the execution times for the programs, running with run-time checks activated, for the different techniques. As a baseline, column **rt** shows the execution times without any optimization. We also show in parenthesis the speedups obtained with respect to our baseline.

The results indicate that applying any of the proposed techniques significantly improves program execution times. The fixpoint analysis with run-time semantics yields results similar to those obtained with the transformation technique proposed by Stulova et al. (2018). In a few cases, however, the **ctrt** technique outperforms our fixpoint implementation (*e.g.*, `fft`). This difference is caused by the widening operators in the *Eterms* abstract domain being overly conservative very soon in the fixpoint iteration. In that example, only one property was not checked under the **ctrt** transformation, yet this caused the fixpoint approach to be 32 times slower. Even so, it remained 12 times faster than the baseline. This example highlights the importance of discharging as many properties as possible at compile-time. It also raises questions about whether additional improvements, or new widening operators are needed, and whether more precise operations could further improve execution times. Exploiting the multivariant information to *automatically* generate program versions allows speed-ups of up to 118× compared to the **ctrt** transformation (*e.g.*, `tak`). In some cases, this corresponds to a total speed-up of up to 240× with respect to the baseline (**rt**) execution. Unlike **ctrt**, which requires manually crafting specialized

Table 2: Execution time, ms (speedup *w.r.t.* **rt**).

module	rt	ct	ctrt	ctrt_f	ctrt_f_vers_etsh	ctrt_f_vers_shet
exp	38.093	5.419 (7×)	5.454 (7×)	5.419 (7×)	5.422 (7×)	5.421 (7×)
factorial	0.357	0.183 (2×)	0.183 (2×)	0.182 (2×)	0.181 (2×)	0.027 (13×)
fft	15,057.10	11,468.4 (1.3×)	39.954 (376×)	1,303.05 (11×)	1,285.54 (12×)	1,281.08 (12×)
fib	6.871	0.043 (159×)	0.048 (143×)	0.043 (159×)	0.043 (159×)	0.043 (159×)
guardians	3,308.72	2.084 (1587×)	2.222 (1489×)	2.082 (1589×)	2.075 (1594×)	2.075 (1594×)
hamming	4,098.18	8.210 (499×)	9.274 (441×)	8.298 (493×)	8.166 (501×)	8.123 (504×)
hanoi	64.731	1.462 (44×)	1.476 (43×)	1.462 (44×)	1.462 (44×)	**0.045 (1438×)**
jugs	0.712	0.012 (59×)	0.012 (59×)	0.012 (59×)	0.012 (59×)	0.012 (59×)
knights	8,440.13	106.201 (79×)	110.018 (76×)	111.269 (75×)	107.094 (78×)	105.934 (79×)
mmatrix	1.471	0.092 (16×)	0.093 (16×)	0.092 (16×)	0.092 (16×)	0.092 (16×)
nreverse	5,020.80	10.800 (464×)	11.179 (449×)	10.793 (465×)	10.786 (465×)	**1.141 (4400×)**
poly	192.200	0.767 (251×)	0.831 (231×)	0.766 (250×)	0.736 (261×)	0.736 (261×)
primes	5.269	0.015 (351×)	0.017 (309×)	0.015 (351×)	0.015 (351×)	0.015 (351×)
progeom	204,112	727.256 (280×)	793.123 (257×)	727.265 (280×)	726.545 (280×)	727.287 (280×)
qsort	18.089	0.369 (49×)	0.379 (48×)	0.380 (48×)	0.370 (49×)	**0.062 (292×)**
queens	326,621	1,358.14 (240×)	1,515.82 (215×)	1,359.46 (240×)	1,359.02 (240×)	1,359.40 (240×)
serialize	2.056	0.021 (98×)	0.022 (93×)	0.021 (98×)	0.021 (98×)	0.021 (98×)
tak	436.523	213.487 (2×)	214.284 (2×)	213.103 (2×)	213.183 (2×)	**1.824 (239×)**
geom. mean	184.282	2.953 (62×)	2.263 (81×)	2.628 (70×)	2.607 (71×)	**1.185 (156×)**
atom_concat	14.052	14.068 (1×)	14.886 (1×)	14.071 (1×)	7.044 (2×)	18.153 (1×)
formulae	1,365.66	0.375 (3641×)	0.377 (3622×)	0.376 (3633×)	0.376 (3633×)	0.378 (3613×)
iso_char	5.965	5.967 (1×)	3.476 (2×)	5.965 (1×)	2.293 (2.5×)	5.523 (1×)
lists	730.89	21.678 (34×)	25.239 (29×)	21.690 (34×)	**18.402 (40×)**	21.472 (34×)
numlists	390.163	229.500 (2×)	229.387 (2×)	229.392 (2×)	230.229 (2×)	230.589 (2×)
sets	271.956	271.861 (1×)	271.289 (1×)	271.861 (1×)	265.793 (1×)	267.915 (1×)
binary_tree	0.033	0.033 (1×)	0.045 (0.7×)	0.033 (1×)	0.015 (2×)	0.033 (1×)
btree	0.248	timeout	0.149 (2×)	0.208 (1×)	0.135 (2×)	0.129 (2×)
heap	0.287	0.089 (3×)	0.084 (3×)	0.038 (8×)	0.038 (8×)	0.038 (8×)
rbtree	0.274	0.119 (2×)	0.030 (9×)	0.031 (9×)	0.031 (9×)	**0.031 (9×)**
sets_ops	0.676	0.238 (3×)	0.226 (3×)	0.228 (3×)	0.198 (3×)	0.171 (4×)
geom. mean	7.439	2.264 (3×)	1.542 (5×)	1.487 (5×)	**1.112 (7×)**	1.408 (5×)

Table 3: Number of reduced properties (%) for LPdoc.

module	ct	ctrt	ctrt_f	ctrt_f_vers_etsh	ctrt_f_vers_shet
doctree	16/48 (33.33%)	19/48 (39.58%)	22/48 (45.83%)	**62/90 (68.89%)**	**83/117 (70.94%)**
filesystem	16/32 (50.00%)	19/32 (59.38%)	21/32 (65.62%)	21/32 (65.62%)	21/32 (65.62%)
html	6/22 (27.27%)	14/22 (63.64%)	21/22 (95.45%)	21/22 (95.45%)	21/22 (95.45%)
html_assets	1/1 (100.00%)	1/1 (100.00%)	1/1 (100.00%)	1/1 (100.00%)	1/1 (100.00%)
html_template	4/12 (33.33%)	6/12 (50.00%)	6/12 (50.00%)	6/12 (50.00%)	6/12 (50.00%)
images	5/12 (41.67%)	5/12 (41.67%)	5/12 (41.67%)	5/12 (41.67%)	5/12 (41.67%)
index	5/5 (100.00%)	5/5 (100.00%)	5/5 (100.00%)	5/5 (100.00%)	5/5 (100.00%)
lpdoc_help	0/2 (0.00%)	0/2 (0.00%)	0/2 (0.00%)	0/2 (0.00%)	0/2 (0.00%)
man	0/4 (0.00%)	0/4 (0.00%)	4/4 (100.00%)	4/4 (100.00%)	4/4 (100.00%)
nil	0/4 (0.00%)	0/4 (0.00%)	4/4 (100.00%)	4/4 (100.00%)	4/4 (100.00%)
refsdb	17/29 (58.62%)	18/29 (62.07%)	18/29 (62.07%)	18/29 (62.07%)	**27/38 (71.05%)**
single_mod	0/8 (0.00%)	0/8 (0.00%)	0/8 (0.00%)	0/8 (0.00%)	0/8 (0.00%)
texinfo	12/16 (75.00%)	13/16 (81.25%)	16/16 (100.00%)	16/16 (100.00%)	16/16 (100.00%)
total	82/195 (42.05%)	100/195 (51.28%)	123/195 (63.08%)	**163/237 (68.78%)**	**193/273 (70.70%)**

versions, the new technique materializes versions systematically. This is especially relevant in more realistic modules, where multiple assertions and call patterns would make a manual, per-predicate specialization impractical. For programs where more moderate speed-ups were observed (ranging from 2× to 10×), execution times were comparable to the more precise approach. In all cases, both techniques were still several times, or even orders of magnitude, faster than the baseline execution. We also observe that the order in which the information of the versions from different domains is applied matters, as expected (*e.g.*, `factorial`, `nreverse`, `qsort`, and `tak`, vs., *e.g.*, `atom_concat`, `iso_char`, or `lists`). In practice we would thus use a combined domain approach, or simply try both orders and keep the best.

As a final experiment, we tested the approach on a more complex application: the LPdoc documenter (Hermenegildo 2000; Hermenegildo and Morales 2011). LPdoc is a documentation generator for LP systems, used in both Ciao and XSB. In addition to generating all the Ciao system manuals and XSB manuals, it is used to generate many web-sites, class slides and exercises, user program documentation, etc. It comprises approximately 22K lines of Prolog code, plus the use of many Ciao libraries. The results on property reduction are shown in Table 3. Perhaps the most notable (and encouraging) conclusion from this experiment is that these results do not differ qualitatively from those on the smaller benchmarks, and similar positive overall patterns are observed. Overall, the **ctrt_f_vers_shet** analysis verified 193 properties out of a total of 273, *i.e.*, 70% of all the properties appearing in the assertions in the code. Furthermore, the verification process detected some errors in LPdoc that had gone unnoticed for many years.

6 Related approaches

The pattern of folding assertion preconditions into a multivariant abstract state and discharging postconditions per clause appears, at least partially, in other approaches. Perhaps the closest to the CiaoPP model (Hermenegildo et al. 1999) is soft contract verification in higher-order functional languages (Nguyen et al. 2014; 2017), as well as the gradual-typing-optimization line (Takikawa et al. 2016; Rastogi et al. 2015), which has also aimed in part at generating residual programs with optimized run-time checks. Also related, but not necessarily aimed at producing optimized programs, are assume-based imperative abstract interpreters such as Astrée (Cousot et al. 2005), Frama-C/EVA (Kirchner et al. 2015), and Goblint (Schwarz et al. 2021). Deductive Verification Condition (VC)-based verifiers (*e.g.*, Dafny (Leino 2010), Why3 (Filliâtre and Paskevich 2013), F* (Swamy et al. 2016)) can in principle discharge sub-clauses via independent VCs, but differ in several respects: their objective is a binary verification verdict, they do not produce optimized programs with simplified checks, and they inherit the solver's incompleteness on recursive and quantified goals. In contrast, our approach based on abstract interpretation always terminates (guaranteed by widening) and unverified properties are turned into run-time checks rather than blocking deployment. In this context, our contribution is to integrate the run-time checking semantics directly into the multivariant top-down fixpoint via the alternative call and success abstractions (Algorithms 1 and 2), avoiding the auxiliary program transformation of Stulova et al. (2018); to exploit the inferred multivariant information to materialize specialized predicate versions (Algorithm 3); and to provide performance results for this approach.

7 Conclusions and future work

We have addressed the problem of reducing run-time overhead in the context of verification frameworks that combine compile-time and run-time checking of user-provided assertions. To this end, we have described how the multivariant, top-down abstract interpretation algorithm can be adapted to more precisely analyze programs that are executed under optional run-time assertion semantics. We have studied how the multivariant information abstracted by these frameworks can be exploited to increase the number of

program properties that are checked at compile time. Our experimental results show that these techniques are not only effective in increasing the number of properties that are checked at compile time, but also in reducing execution times under run-time assertion checking semantics. We presented our approach in the context of the Ciao Prolog language, but the Ciao approach to combining static and dynamic analysis is general and system-independent, as well as the techniques used herein, so we expect the results to carry over to other (dynamic) logic languages. Future work includes investigating further the impact of abstract specialization, and, as mentioned previously, studying the effects of other domains and domain combinations.

Competing interests: The authors declare none.

References

Boye, J., Drabent, W., and Małuszyński, J. Declarative Diagnosis of Constraint Programs: an Assertion-based Approach. In *Proc. of the 3rd. Int'l WS on Automated Debugging–AADEBUG'97* 1997, pp. 123–141. U. of Linköping Press.

Bruynooghe, M. 1991. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, *10*, 91–124.

Bueno, F., Deransart, P., Drabent, W., Ferrand, G., Hermenegildo, M., Maluszynski, J., and Puebla, G. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd Int'l. WS on Automated Debugging–AADEBUG* 1997, pp. 155–170. U. Linköping Press.

Cartwright, R. and Fagan, M. Soft Typing. In *PLDI'91* 1991, pp. 278–292. SIGPLAN, ACM.

Cousot, P. and Cousot, R. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77* 1977, pp. 238–252. ACM Press.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. The ASTREÉ analyzer. In *14th European Symposium on Programming, ESOP 2005* 2005, pp. 21–30.

De Angelis, E., Fioravanti, F., Gallagher, J. P., Hermenegildo, M. V., Pettorossi, A., and Proietti, M. 2021. Analysis and Transformation of Constrained Horn Clauses for Program Verification. *TPLP*, *22*, 6, 1–69.

Dimoulas, C. and Felleisen, M. 2011. On Contract Satisfaction in a Higher-Order World. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *33*, 5, 1–29.

Drabent, W., Nadjm-Tehrani, S., and Małuszyński, J. The Use of Assertions in Algorithmic Debugging. In *Intl. Conf. on Fifth Generation Computer Systems* 1988, pp. 573–581.

Fähndrich, M. and Logozzo, F. Static Contract Checking with Abstract Interpretation. In *Int'l. Conf. on Formal Verification of Object-oriented Software, FoVeOOS'10* 2011, volume 6528 of *LNCS*, pp. 10–30. Springer.

Filliâtre, J.-C. and Paskevich, A. Why3 — where programs meet provers. In Felleisen, M. and Gardner, P., editors, *Programming Languages and Systems* 2013, pp. 125–128, Berlin, Heidelberg. Springer Berlin Heidelberg.

Findler, R. B. and Felleisen, M. Contracts for Higher-Order Functions. In *Int'l. Conf. on Functional Programming (ICFP)* 2002, pp. 48–59. ACM.

García de la Banda, M., Hermenegildo, M., Bruynooghe, M., Dumortier, V., Janssens, G., and Simoens, W. 1996. Global Analysis of Constraint Logic Programs. *ACM Trans. on Programming Languages and Systems*, *18*, 5, 564–615.

Hanus, M. Combining static and dynamic contract checking for curry. In *Logic-Based Program Synthesis and Transformation* 2017, volume 10855 of *LNCS*, pp. 323–340. Springer.

Henriksen, K. S. and Gallagher, J. P. Abstract Interpretation of PIC Programs through Logic Programming. In *SCAM '06* 2006, pp. 184–196. IEEE Computer Society.

Hermenegildo, M. A Documentation Generator for (C)LP Systems. In *International Conference on Computational Logic, CL2000* 2000, number 1861 in LNAI, pp. 1345–1361. Springer-Verlag.

Hermenegildo, M. and Morales, J. 2011. The LPdoc Documentation Generator. Ref. Manual (v3.0). Technical report, UPM. Available at https://ciao-lang.org.

Hermenegildo, M., Puebla, G., and Bueno, F. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In *The Logic Programming Paradigm: a 25–Year Perspective* 1999, pp. 161–192. Springer-Verlag.

Hermenegildo, M., Puebla, G., Bueno, F., and Garcia, P. L. 2005. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, *58*, 1–2, 115–140.

Jaffar, J., Maher, M., Marriott, K., and Stuckey, P. 1998. The semantics of constraint logic programs. *Journal of Logic Programming*, *37*, 1, 1–46.

Kelly, A., Marriott, K., Søndergaard, H., and Stuckey, P. 1998. A Practical Object-Oriented Analysis Engine for CLP. *Software: Practice and Experience*, *28*, 2, 188–224.

Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. 2015. Frama-c: A software analysis perspective. *Form. Asp. Comput.*, *27*, 3, 573–609.

Koukoutos, E. and Kuncak, V. Checking Data Structure Properties Orders of Magnitude Faster. In *Runtime Verification* 2014, volume 8734 of *LNCS*, pp. 263–268. Springer.

Laï, C. Assertions with Constraints for CLP Debugging. In Deransart, P., Hermenegildo, M. V., and Maluszynski, J., editors, *Analysis and Visualization Tools for Constraint Programming* 2000, volume 1870 of *Lecture Notes in Computer Science*, pp. 109–120. Springer.

Lamport, L. and Paulson, L. C. 1999. Should Your Specification Language be Typed? *ACM Transactions on Programming Languages and Systems*, *21*, 3, 502–526.

Le Charlier, B. and Van Hentenryck, P. 1994. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM TOPLAS*, *16*, 1, 35–101.

Leavens, G. T., Leino, K. R. M., and Müller, P. 2007. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.*, *19*, 2, 159–189.

Leino, K. R. M. Dafny: An automatic program verifier for functional correctness. In Clarke, E. M. and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning* 2010, pp. 348–370, Berlin, Heidelberg. Springer Berlin Heidelberg.

Méndez-Lojo, M., Navas, J., and Hermenegildo, M. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *LOPSTR* 2007a, volume 4915 of *LNCS*, pp. 154–168. Springer-Verlag.

Méndez-Lojo, M., Navas, J., and Hermenegildo, M. An Efficient, Parametric Fixpoint Algorithm for Analysis of Java Bytecode. In *ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)* 2007b.

Mera, E., Lopez-Garcia, P., and Hermenegildo, M. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *ICLP'09* 2009, volume 5649 of *LNCS*, pp. 281–295. Springer-Verlag.

Muthukumar, K. and Hermenegildo, M. 1990. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Comp. Tech. Corp. (MCC).

Muthukumar, K. and Hermenegildo, M. 1992. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *JLP*, *13*, 2/3, 315–347.

Nguyen, P. C., Gilray, T., Tobin-Hochstadt, S., and Van Horn, D. 2017. Soft contract verification for higher-order stateful programs. *Proc. ACM Program. Lang.*, *2*, POPL.

Nguyen, P. C., Tobin-Hochstadt, S., and Van Horn, D. Soft contract verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* 2014, ICFP '14, 139–152, New York, NY, USA. Association for Computing Machinery.

Puebla, G., Bueno, F., and Hermenegildo, M. An Assertion Language for Debugging of Constraint Logic Programs. In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming* 1997.

Puebla, G., Bueno, F., and Hermenegildo, M. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming* 2000a, number 1870 in LNCS, pp. 23–61. Springer-Verlag.

Puebla, G., Bueno, F., and Hermenegildo, M. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Proc. of LOPSTR'99* 2000b, LNCS 1817, pp. 273–292. Springer-Verlag.

Rastogi, A., Swamy, N., Fournet, C., Bierman, G., and Vekris, P. Safe & Efficient Gradual Typing for TypeScript. In *42nd POPL* 2015, pp. 167–180. ACM.

Schwarz, M., Saan, S., Seidl, H., Apinis, K., Erhard, J., and Vojdani, V. Improving thread-modular abstract interpretation. In Drăgoi, C., Mukherjee, S., and Namjoshi, K., editors, *Static Analysis* 2021, pp. 359–383, Cham. Springer International Publishing.

Seidl, H. and Vogler, R. 2021. Three improvements to the top-down solver. *Math. Struct. Comput. Sci.*, *31*, 9, 1090–1134.

Stulova, N., Morales, J. F., and Hermenegildo, M. 2015. Practical Run-time Checking via Unobtrusive Property Caching. *Theory and Practice of Logic Programming*, *15*, 04-05, 726–741.

Stulova, N., Morales, J. F., and Hermenegildo, M. 2018. Some Trade-offs in Reducing the Overhead of Assertion Run-time Checks via Static Analysis. *Science of Computer Programming*, *155*, 3–26.

Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.-Y., Kohlweiss, M., Zinzindohoue, J.-K., and Zanella-Béguelin, S. Dependent types and multi-monadic effects in f*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* 2016, POPL '16, 256–270, New York, NY, USA. Association for Computing Machinery.

Takikawa, A., Feltey, D., Dean, E., Flatt, M., Findler, R., Tobin-Hochstadt, S., and Felleisen, M. Towards Practical Gradual Typing. In *29th ECOOP* 2015, pp. 4–27.

Takikawa, A., Feltey, D., Greenman, B., New, M., Vitek, J., and Felleisen, M. Is Sound Gradual Typing Dead? In *POPL 2016* 2016, pp. 456–468.

Tilscher, S., Stade, Y., Schwarz, M., Vogler, R., and Seidl, H. The Top-Down Solver—An Exercise in A^2I. In *Challenges of Software Verification* 2023, volume 238, chapter 9, pp. 157–179. Springer, Singapore.

Tobin-Hochstadt, S. and Felleisen, M. The Design and Implementation of Typed Scheme. In *POPL* 2008, pp. 395–406. ACM.

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

Vazou, N., Tanter, É., and Horn, D. V. 2018. Gradual liquid type inference. *Proc. ACM Program. Lang.*, *2*, OOPSLA, 132:1–132:25.

Warren, R., Hermenegildo, M., and Debray, S. K. On the Practicality of Global Flow Analysis of Logic Programs. In *JICSLP* 1988, pp. 684–699. MIT Press.