Reducing the Overhead of Assertion Run-time Checks via Static Analysis*

Nataliia Stulova[1] José F. Morales[1] Manuel V. Hermenegildo[1,2]

[1] IMDEA Software Institute
{nataliia.stulova,josef.morales,
manuel.hermenegildo}@imdea.org

[2] School of Computer Science
Technical University of Madrid (UPM)
manuel.hermenegildo@upm.es

ABSTRACT

In order to aid in the process of detecting incorrect program behaviors, a number of approaches have been proposed which include a combination of language-level constructs (such as procedure-level assertions/contracts, program-point assertions, gradual types, etc.) and associated tools (such as code analyzers and run-time verification frameworks). However, it is often the case that these constructs and tools are not used to their full extent in practice due to a number of limitations such as excessive run-time overhead and/or limited expressiveness. Verification frameworks that combine static and dynamic techniques offer the potential to bridge this gap. In this paper we explore the effectiveness of abstract interpretation in detecting parts of program specifications that can be statically simplified to true or false, as well as the impact of such analysis in reducing the cost of the run-time checks required for the remaining parts of these specifications. Starting with a semantics for programs with assertion checking, and for assertion simplification based on static analysis information, we propose and study a number of practical assertion checking modes, each of which represents a trade-off between code annotation depth, execution time slowdown, and program safety. We also propose techniques for taking advantage of the run-time checking semantics to improve the precision of the analysis. Finally, we study experimentally the performance of these techniques. Our experiments illustrate the benefits and costs of each of the assertion checking modes proposed as well as the benefit of analysis for these scenarios.

1. INTRODUCTION

Detecting incorrect program behaviors is an important part of the software development life cycle. It is also a complex and tedious one, in which dynamic languages bring special challenges. A number of techniques have been proposed to aid in the process, among which we center our attention on the use of language-level constructs to describe expected program behavior, and of associated tools to compare actual program behavior against expectations, such as static code analyzers/verifiers and run-time verification frameworks. Approaches that fall into this category are the assertion-based frameworks used in (Constraint) Logic Programming [8, 36, 4, 2, 14, 38, 19, 15, 26], soft/gradual typing approaches in functional programming [5, 10, 47, 7, 39, 45, 46], and contract-based extensions in object-oriented programming [20, 21, 9]. These tools are aimed at detecting violations of the expected behavior or certifying the absence of any such violations, and often involve a certain degree of run-time testing, specially for non-trivial properties.

A practical limitation of many of these tools is that they can incur significant run-time performance overhead, even in the simple case of performing just type checks between typed and untyped parts of programs [39, 46]. In [26] reductions were obtained by limiting the points at which the tests are performed and the instrumentation, and by inlining, but some types of tests still incurred significant costs. Some approaches have opted for limiting the expressiveness of the assertion language in order to reduce the overhead (see [40] for some recent case studies). Recently, some proposals have been made for reducing the run-time overhead of assertion checking based on optimizing the run-time checking mechanisms themselves, at the expense of increased memory consumption [18, 44]: repeated checks on immutable recursive data structures are converted from execution time overhead to increased memory use via caching and/or tabling techniques.

*This research has been partially funded by EU FP7 agreement 318337 ENTRA, Spanish MINECO project TIN2015-67522-C3-1-R *TRACES*, and Madrid Region program M141047003 *N-GREENS*. We would also like to thank the anonymous reviewers for providing valuable comments and suggestions.

However, despite these advances, run-time overhead often remains impractically high, specially for complex properties, such as, for example, deep data structure tests. This reduces the attractiveness of run-time checking to programmers, which may allow sporadic checking of very simple conditions, but tend to turn off run-time checking for more complex properties.

Motivated by this problem, assertion-based frameworks have been proposed where static analysis is used to minimize the number and cost of the run-time checks that need to be placed in the program to detect incorrect program behaviors. This idea was pioneered by the Ciao system [36, 4, 14, 38, 15, 35, 17] where a number of (abstract interpretation-based) static analyses are combined in order to verify assertions to the largest extent possible at compile time, and for simplifying and reducing the number of remaining properties that that need to be introduced in the program as run-time checks. Intuitively, this model can offer a more appealing trade-off of performance vs. safety guarantees. However, while there has been evidence from use, there has been little systematic experimental work presented to date verifying this hypothesis, i.e., measuring the actual impact of analysis on reducing run-time checking overhead. For example, in [26, 27] the overhead of run-time checking was studied but without taking into account analysis information.

In order to bridge this gap, in this work we explore the effectiveness of abstract interpretation-based compile-time analysis in detecting parts of program specifications that can be simplified before they are turned into run-time checks. Again, the objective of such simplification is to achieve a system that can detect the same (or a larger) set of incorrect behaviors in a program, but with a significant reduction in the impact on the running time of the program.

Starting with a semantics for programs with assertion checking and for assertion simplification based on analysis information obtained via abstract interpretation, we propose and study a number of practical *assertion checking modes*, each of which represents a trade-off between code annotation depth, execution time slowdown, and program behavior safety guarantees. The proposed modes are specially tailored to the scenario of annotating and pre-processing libraries to ensure their correctness prior to their use by client programs. We also define a transformation-based approach in order to implement each one of these modes.

We then concentrate on the reduction of the number of run-time tests via (abstract interpretation-based) program analysis. To this end we propose a technique that enhances analysis precision by taking into account that any assertions that cannot be proved statically will be the subject of run-time testing. We then report on an implementation of the proposed techniques (within the CiaoPP system) and study their impact in practice, by measuring the reduction in run-time checking overhead achieved.

We develop the discussion in the context of *(Horn Clause) Logic Programs*, to take advantage of the availability of mature program analysis and transformation tools, and a well developed assertion language and assertion processing framework (in particular, that of the Ciao system). However, we believe the results are applicable to other programming paradigms, either directly (to, e.g., other forms of declarative programming), or, following recent work, to imperative programs, via transformation into Horn Clauses [25, 12, 1, 13]. Examples of this include cost analysis of Java bytecode programs [33, 34], or inferring energy bounds in binaries from C-style programs [23, 22].

The rest of the paper is structured as follows: Section 2 presents the run-time checking part of our approach. After introducing some notation and the basic semantics in Section 2.1, Section 2.2 presents the assertion language and Section 2.3 the operational semantics with run-time checking of such assertions. Section 3 then presents the run-time assertion checking modes proposed, including a discussion of the transformations required to implement the different modes. Section 4 then addresses the issue of optimizing run-time checks via static analysis. Section 4.1 presents the basic abstract interpretation-based analysis approach used and the memo table representation of the analysis results. Section 4.2 describes how run-time tests are optimized using the information in the analysis memo table. Section 5 then presents our approach for taking advantage of the run-time checking semantics to improve the precision of the analysis. Section 6 describes our experimental harness and presents our results for the different options (with and without analysis, with and without improved analysis precision). Section 7 finally presents our conclusions.

2. RUN-TIME CHECKING OF ASSERTIONS

2.1 Basic notation and standard semantics

We revisit here some basic notation and the standard program semantics, where we use the formalization of [44, 43, 38].

An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol of arity n and $t_1, ..., t_n$ are terms. A *constraint* is a conjunction of expressions built from predefined predicates (such as term equations or inequalities over the reals) whose arguments are constructed using predefined functions (such as real addition). A *literal* is either an atom or a constraint. A *goal* is a finite sequence of literals. A *rule* is of the form H :- B where H, the *Head*, is an atom and B, the *body*, is a possibly empty finite sequence of literals. A *constraint logic program*, or *program*, is a finite set of rules.

The *definition* of an atom A in a program, $\mathsf{defn}(A)$, is the set of variable renamings of the program rules s.t. each renaming has A as a Head and has distinct new local variables. We assume that all rule Heads are *normalized*, i.e., H is of the form $p(X_1, ..., X_n)$ where the $X_1, ..., X_n$ are distinct free variables. Let $\exists_L \theta$ be the constraint θ restricted to the variables of the syntactic object L. We denote *constraint entailment* by $\models$, so that $\theta_1 \models \theta_2$ denotes that θ_1 entails θ_2. Then, we say that θ_2 is *weaker* than θ_1.

The operational semantics of a program is given in terms of its *derivations*, which are sequences of *reductions* between *states*. A *state* $\langle G \mid \theta \rangle$ consists of a goal G and a constraint store (or *store* for short) θ. We use :: to denote concatenation of sequences and we assume for simplicity that the underlying constraint solver is complete. A state $S = \langle L :: G \mid \theta \rangle$ where L is a literal can be *reduced* to a state S' as follows:

1. $\langle L :: G \mid \theta \rangle \rightsquigarrow \langle G \mid \theta \wedge L \rangle$ if L is a constraint and $\theta \wedge L$ is satisfiable.
2. $\langle L :: G \mid \theta \rangle \rightsquigarrow \langle B :: G \mid \theta \rangle$ if L is an atom of the form $p(t_1, \ldots, t_n)$,
   for some rule $(L$:- $B) \in \mathsf{defn}(L)$.

We use $S \rightsquigarrow S'$ to indicate that a reduction can be applied to state S to obtain state S'. Also, $S \rightsquigarrow^* S'$ indicates that there is a sequence of reduction steps from state S to state S'. We denote by $D_{[i]}$ the i-th state of the derivation. As a shorthand, given a non-empty derivation D, $D_{[-1]}$ denotes the last state. A *query* is a pair (L, θ), where L is a literal and θ a store, for which the constraint logic programming system starts a computation from state $\langle L \mid \theta \rangle$. The set of all derivations from the query Q is denoted $\mathsf{derivs}(Q)$. A finished derivation from a query (L, θ) is *successful* if the last state is of the form $\langle \square \mid \theta' \rangle$, where $\square$ denotes the empty goal sequence. In that case, the constraint $\exists_L \theta'$ is an *answer* to S. We denote by $\mathsf{answers}(Q)$ the set of answers to a query Q.

2.2 Assertion Language

We assume that program specifications are provided by means of assertions: linguistic constructions that allow expressing properties of programs. In particular, we would like to specify certain conditions on the constraint store that must hold at certain points of program derivations. For concreteness we will use the `pred` assertions of the Ciao assertion language [36, 14, 37, 38, 17]. The main intent behind the construction of a specification for a predicate using `pred`-assertions is to define the set of all admissible preconditions for this prediate, and for each such precondition in turn specify the respective postcondition. I.e., `pred`-assertions allow stating sets of related *preconditions* and *conditional postconditions* for a given predicate.

These pre- and postconditions are formulas containing literals corresponding to predicates that are specially labeled as *properties*. The design of this language is such that properties and the other predicates composing the program are written in the same language. This approach is motivated by the direct correspondence between the declarative and operational semantics of constraint logic programs and it provides a direct link between the properties used in assertions and the corresponding run-time tests, which constitute (instrumented) calls to the predicates defining the properties. This also allows defining specifications that are more general than, e.g., the traditional notions of types.

More formally, the set of assertions for a given predicate represented by $Head$ is composed of all statements of the form:[1]

$$\begin{array}{l} \texttt{:- pred } Head \texttt{ : } Pre_1 \texttt{ => } Post_1 \texttt{.} \\ \ldots \\ \texttt{:- pred } Head \texttt{ : } Pre_n \texttt{ => } Post_n \texttt{.} \end{array}$$

where $Head$ is the same normalized atom, that denotes the predicate that the assertions apply to, and the Pre_i and $Post_i$ are conjunctions[2] of *prop* literals that refer to the variables of $Head$.

A set of assertions as above states that in any execution state $\langle Head :: G \mid \theta \rangle$ at least one of the Pre_i conditions should hold, and that, given the $(Pre_i, Post_i)$ pair(s) where Pre_i holds, then, if $Head$ succeeds, the corresponding $Post_i$ should hold upon success. More formally, given a predicate represented by a normalized atom $Head$, and the corresponding set of assertions is $\mathcal{A}(Head) = \{A_1 \ldots A_n\}$, with $A_i =$ "`:- pred` $Head$ `:` Pre_i `=>` $Post_i$`.`" such assertions are normalized into a set of *assertion conditions* for that predicate, denoted as $\mathcal{A}_C(Head) = \{C_0, C_1, \ldots, C_n\}$ s.t.:

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

where c_i is a unique assertion condition identifier. If there are no assertions associated with $Head$ then the corresponding set of assertion conditions is empty. The set of assertion conditions for a program is the union of the assertion conditions for each of the predicates in the program.

The $\mathsf{calls}(Head, \ldots)$ conditions encode the checks that ensure that the calls to the predicate represented by the $Head$ literal are within those admissible by the set of assertions, and we thus call them the *calls assertion conditions*. The $\mathsf{success}(Head_i, Pre_i, Post_i)$ conditions encode the checks for compliance of the successes for particular sets of calls, and we thus call them the *success assertion conditions*.

2.3 Semantics with assertions

We now recall the operational semantics with assertions, which checks whether assertion conditions hold or not while computing the derivations from a query. In order to keep track of any violated assertion conditions, we use the identifiers of the assertion conditions. Given the atom L_a that is a renaming of some normalized atom L s.t. $L_a = \sigma(L)$ and the corresponding set of assertion conditions $\mathcal{A}_C(L)$, the assertion conditions for L_a are obtained as follows: if $\exists C \in \mathcal{A}_C(L), C = c.\mathsf{calls}(L, Pre)$ (or $C = c.\mathsf{success}(L, Pre, Post)$), then $C_a = \sigma(C) = c_a.\mathsf{calls}(L_a, \sigma(Pre))$ (or $C_a = c_a.\mathsf{success}(L_a, \sigma(Pre), \sigma(Post))$). We also introduce an extended program state of the form $\langle G \mid \theta \mid \mathcal{E} \rangle$, where $\mathcal{E}$ denotes the set of identifiers for falsified assertion condition instances and $|\mathcal{E}| \leq 1$. For the sake of readability, we write labels in *negated* form when they appear in the error set. A finished derivation from a query (L, θ) now is *successful* if the last state is of the form $\langle \square \mid \theta' \mid \varnothing \rangle$ ($\varnothing$ denotes the empty set), and *failed* if the last state is of the form $\langle L' \mid \theta' \mid \{\bar{c}\} \rangle$. We also extend the set of literals with syntactic objects of the form $\mathsf{acheck}(L, c)$ where L is a literal and c is an identifier for an assertion condition instance, which we call *check literals*. Thus, a *literal* is now a constraint, an atom or a check literal. A literal L *succeeds trivially* for θ in program P, denoted $\theta \Rightarrow_P L$, iff $\exists \theta' \in \mathsf{answers}((L, \theta))$ such that $\theta \models \theta'$. We can now recall the notion of *Reductions in Programs with Assertions* from [43], which is our starting point: a state $S = \langle L :: G \mid \theta \mid \varnothing \rangle$, where L is a literal, can be *reduced* to a state S', denoted $S \rightsquigarrow_{\mathcal{A}} S'$, as follows:

1. If L is a constraint then the new state is $S' = \langle G' \mid \theta' \mid \varnothing \rangle$ where G' and θ' are obtained in a same manner as in $\langle L :: G \mid \theta \rangle \rightsquigarrow \langle G' \mid \theta' \rangle$
2. If L is an atom and $\exists (L \texttt{:-} B) \in \mathsf{defn}(L)$, then the new state S' is obtained as

$$S' = \begin{cases} \langle L \mid \theta \mid \{\bar{c}\} \rangle & \text{if } \exists\ c.\mathsf{calls}(L, Pre) \in \mathcal{A}_C(L) \\ & \quad \wedge\ \theta \not\Rightarrow_P Pre \\ \langle B :: G' \mid \theta \mid \varnothing \rangle & \text{otherwise} \end{cases}$$

and $G' = \mathsf{acheck}(L, c_1) :: \ldots :: \mathsf{acheck}(L, c_n) :: G$ such that
$c_i.\mathsf{success}(L, Pre_i, Post_i) \in \mathcal{A}_C(L)\ \wedge\ \theta \Rightarrow_P Pre_i$.

[1] We follow the more compact formalization of [43], using only `pred` assertions. See also [38] for the original presentation using `calls` and `success` assertions. We are also not dealing herein with `comp` assertions and properties.

[2] In the general case *Pre* and *Post* can be DNF formulas of *prop* literals but we limit them to conjunctions herein for simplicity of presentation.

3. If L is a check literal $\mathsf{acheck}(L', c)$, then S' is obtained as

$$S' = \begin{cases} \langle L' \mid \theta \mid \{\bar{c}\}\rangle & \text{if } c.\mathsf{success}(L', \_, Post) \in \mathcal{A}_C(L') \\ & \quad \wedge\ \theta \not\Rightarrow_P Post \\ \langle G \mid \theta \mid \varnothing\rangle & \text{otherwise} \end{cases}$$

3. ASSERTION CHECKING MODES

When a program is being instrumented with run-time checks the choice of instrumentation strategy is determined by several factors and considerations. Most of these factors can typically be generalized to a compromise between thoroughness of the code annotation (complexity of the properties, annotation depth) and the resulting performance penalties (execution time slowdown, code bloat, increases in memory use).

We propose a view on this compromise that differentiates between various levels of behavioral safety guarantees embodied in several *assertion checking modes*. In the following we will explain these modes for concreteness in a client-library interaction scenario, and from the perspective of the client. However, the proposed view is not limited to this particular case and can be applied to other kinds of interaction between program components.

Unsafe Checking Mode.

This checking mode corresponds to a scenario where no execution time slowdown is tolerated at run time, even at the cost of no program behavior safety guarantees. In this case no run-time checks are generated from the assertions of the library. Formally, this corresponds to using the standard semantics of Section 2.1, and thus ignoring the assertions in the program. This of course eliminates any overhead but at the cost of correctness. However, we still consider it because it represents a baseline and is in fact used sometimes in practice for production code, in order to avoid overhead, if, e.g., it is perceived that sufficient testing was carried out prior to delivery.

Client-safe Checking Mode.

In this mode the library provides the client with behavior guarantees on its interface, but does not check the assertions for the internal procedures. Run-time checks are thus generated only for the assertion conditions for the exported predicates of the library. More formally, assuming that the set of (atoms of) exported predicates is given by Exp:

1. If L is a constraint or L is an atom such that $L \notin Exp$, then the new state $S' = \langle G' \mid \theta' \mid \varnothing\rangle$ where G' and θ' are obtained in a same manner as in $\langle L :: G \mid \theta\rangle \rightsquigarrow \langle G' \mid \theta'\rangle$
2. If L is an atom such that $L \in Exp$, and $\exists(L\texttt{:-}B) \in \mathsf{defn}(L)$, then the new state S' is obtained as in step 2 of the semantics with run-time checks of Section 2.3.
3. If L is a check literal $\mathsf{acheck}(L', c)$, then S' is obtained as in step 3 of the semantics with run-time checks of Section 2.3.

The modified semantics above ensures that checks are performed only for the predicates in the library interface. However, all calls within the library to the exported predicates, including recursive calls, would also be checked, unnecessarily. In order to avoid this, and to ensure that the checks are performed only on the external calls, we assume that the program transformation given in Fig. 1 is applied to all exported predicates. The transformation introduces intermediate *link* predicates for exported predicates so that the module interface is preserved but all the internal calls are replaced by calls to the wrapper predicates, for which no checks are performed. This combination of program transformation and run-time checking policy allows obtaining safety guarantees at the library boundaries with minimal run-time checking execution time overhead.

```
1  :- module(_, [p]). % exported predicate p
2
3  :- check pred p : Pre => Post.
4
5  % c0.calls(p, Pre)            ∧ status(c0, check)
6  % c1.success(p, Pre, Post) ∧ status(c1, check)
7
8  p :- body.
9
10 q :- p.
```

(a) Initial program fragment.

```
1  :- module(_, [p]).
2
3  % c0.calls(p, Pre)            ∧ status(c0, check)
4  % c1.success(p, Pre, Post) ∧ status(c1, check)
5
6  p :- p_inner. % (the link clause)
7
8  p_inner :- body.
9
10 q :- p_inner.
```

(b) The same program fragment after the transformation.

Figure 1: Client-safe program transformation.

Safe-RT Execution Mode.

In this mode the library provides behavior guarantees both on its interface and its internals. Run-time checks are generated for all assertions of the library. This corresponds to using the semantics with assertions of Section 2.3. The performance penalty here is the largest.

Transformations.

The checking modes described above require different source transformations to be performed on a program during compile time (see Fig. 2). Before any such transformations take place, the assertions are normalized and expanded into assertion conditions. This allows assuring that no syntactic errors are present in the assertion conditions and that no undefined properties (i.e., properties that are not defined in the program or imported from libraries) appear in such conditions.

In the *Unsafe* mode nothing is done and the assertion conditions are simply ignored during compilation. In the *Safe-RT* mode the source transformation is quite straightforward: all the assertion conditions for all assertions in the program are turned into run-time checks directly. In the *Client-safe* mode, as mentioned before, the program transformation of Figure 1 is first performed for all the exported predicates, and then run-time checks are generated only for the assertion conditions of those exported predicates.

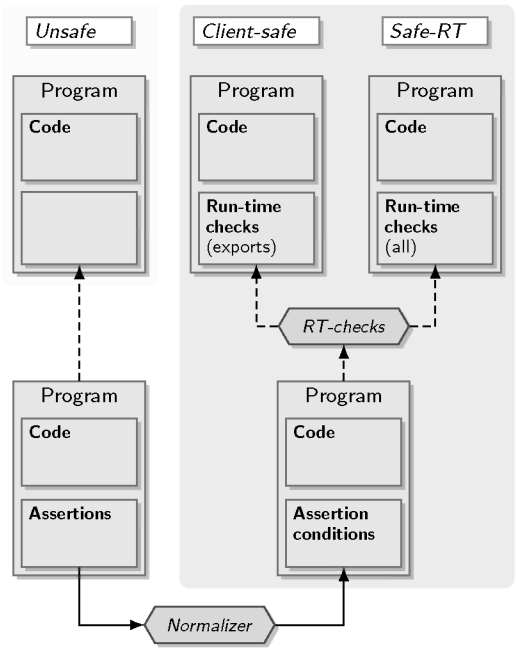

Figure 2: Source transformation differences per checking mode.

4. OPTIMIZING RUN-TIME CHECKS VIA STATIC ANALYSIS

We now return to the issue of optimizing run-time checks via (abstract interpretation-based) static program analysis, in order to reduce the number of run-time tests and thus the overhead from run-time testing, following the Ciao model. To this end, we recall the basic abstract interpretation-based analysis approach used and the memo table representation of the analysis results and describe how run-time tests are optimized using the information in the analysis memo table. Based on this in the following section we will present our approach for taking advantage of the run-time checking semantics to improve the precision of the analysis.

Herein we will refer to this combination of static and dynamic checking as the ***Safe-CT-RT Checking Mode***, i.e., as a variation on the *Safe-RT* run-time checking mode, where static verification is performed in order to eliminate as many of the properties in the program assertions to be checked at run time as possible. Run-time checks are still generated for all program assertions but in contrast to the *Safe-RT* case the assertions are simplified before the checks are generated from them (see Fig. 3). In this mode the run-time checks for the *calls* assertion conditions of the exported predicates are left untouched though, to ensure calls safety in an open context.[3]

4.1 Abstract Interpretation-based Analysis

For analysis we use the technique of abstract interpretation [6], which simulates the execution of a program on an *abstract domain* (D_α) which is simpler than the actual, *concrete domain*[4] (D). Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : 2^D \to D_\alpha$, and *concretization* $\gamma : D_\alpha \to 2^D$. The operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) over abstract literals λ mimic those of 2^D in a precise sense:

$$\forall \lambda, \lambda' \in D_\alpha : \ \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$$
$$\forall \lambda_1, \lambda_2, \lambda' \in D_\alpha : \ \lambda_1 \sqcup \lambda_2 = \lambda' \Leftrightarrow \gamma(\lambda_1) \cup \gamma(\lambda_2) = \gamma(\lambda')$$
$$\forall \lambda_1, \lambda_2, \lambda' \in D_\alpha : \ \lambda_1 \sqcap \lambda_2 = \lambda' \Leftrightarrow \gamma(\lambda_1) \cap \gamma(\lambda_2) = \gamma(\lambda')$$

As usual in abstract interpretation, $\bot$ denotes the abstract constraint such that $\gamma(\bot) = \emptyset$, whereas $\top$ denotes the most general abstract constraint, i.e., $\gamma(\top) = D$.

The concrete framework that we will use in the static analysis component is the Ciao PLAI abstract interpretation system [28, 29, 31]. Below we adapt some definitions and notation from [38] to illustrate the analysis process implemented by PLAI.

The goal dependent abstract interpretation performed by PLAI takes as input a program P, an abstract domain D_α, and a description $\mathcal{Q}_\alpha$ of the possible initial queries to the program given as a set of abstract queries. An *abstract query* is a pair (L, λ), where L is an atom (for one of the exported predicates) and $\lambda \in D_\alpha$ an abstract constraint which describes the initial stores for L. A set of abstract queries $\mathcal{Q}_\alpha$ represents a set of queries, denoted $\gamma(\mathcal{Q}_\alpha)$, which is defined as $\gamma(\mathcal{Q}_\alpha) = \{(L, \theta) \mid (L, \lambda) \in \mathcal{Q}_\alpha \wedge \theta \in \gamma(\lambda)\}$. Such an abstract interpretation computes a set of triples $Analysis(P, \mathcal{Q}_\alpha, D_\alpha) = \{\langle L_p, \lambda^c, \lambda^s \rangle \mid p \text{ is a predicate of } P\}$, where λ^c and λ^s are abstract constraints that describe calls and success patterns for p.

The analysis (as the assertion language, to be introduced later) is designed to discern among the various usages of

[3]Although modular analysis can also eliminate the module interface checks, this is to be consistent with our simple library scenario.

[4]In what follows we assume the concrete domains to have a powerset structure, but the framework is not limited to such domains and can be applied to domains of arbitrary structure.

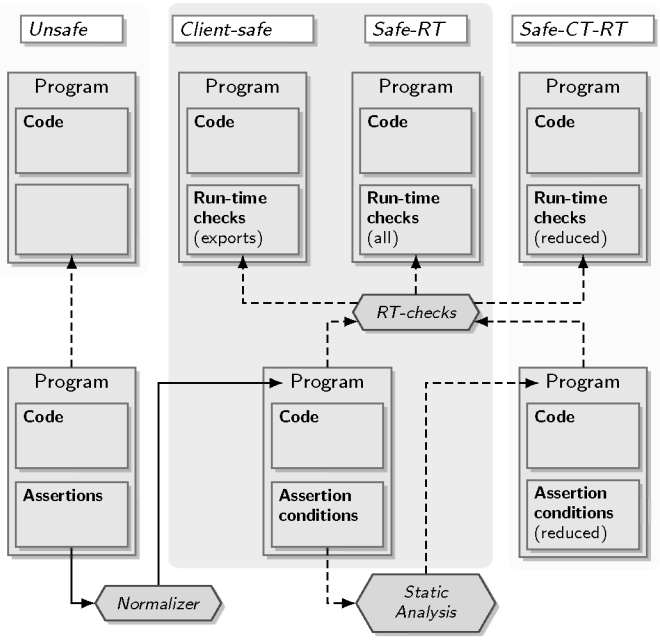

Figure 3: Source transformation differences per checking mode, including compile-time analysis.

Table 1: Assertion status.

Status	Source	Description
`check`	user	The assertion expresses part of the intended semantics. It may or may not hold in the current version of the program. It is the default status that is assumed for assertions written without and explicit status.
`checked`	static checking	The assertion was a `check` assertion which has been proved to actually hold in the current version of the program for any valid initial call.
`false`	static checking	Similarly, a `check` assertion is rewritten with the status `false` when it is proved not to hold for some valid initial query.
`true`	static analyses or user	Such an assertion expresses (a part of) the actual semantics of the program, normally automatically inferred by analysis.Can also be provided by the programmer.

a predicate. Thus, multiple usages of a procedure can result in multiple descriptions in the analysis output, i.e., for a given predicate P multiple $\langle P, \lambda^c, \lambda^s \rangle$ triples may be inferred. More precisely, the analysis is said to be *multivariant on calls* if more than one triple $\langle P, \lambda^c_1, \lambda^s_1 \rangle$, ..., $\langle P, \lambda^c_n, \lambda^s_n \rangle$ $n \geq 0$ with $\lambda^c_i \neq \lambda^c_j$ for some i, j may be computed for the same predicate. However, for simplicity of presentation, we assume that the analysis computes exactly one tuple $\langle L_p, \lambda^c, \lambda^s \rangle$ for each (reachable) predicate p.

4.2 Optimizing Assertions with Analysis Results

The steps of the verification process are represented by associating a notion of "status" to each assertion:

```
:- [Status] pred Head :  Pre_1 => Post_1 .
...
:- [Status] pred Head :  Pre_n => Post_n .
```

This optional *Status* flag indicates whether the assertion refers to intended or actual properties, and possibly some additional information, as shown in the top part of Table 1 (see also Figure 6).

The reasoning about the statuses of assertion conditions is performed in the following terms. Given a literal L and a program P, the *trivial success set* of L in P is $TS(L,P) = \{\bar{\exists}_L\theta \mid \theta \Rightarrow_P L\}$. We also recall here the auxiliary partial functions prestep and step from [43] which are instrumental in reasoning about program state reductions:

$$\mathsf{prestep}(L_a, D) = (\theta, \sigma) \equiv D_{[-1]} = \langle L :: G \mid \theta \rangle \wedge \exists \sigma\ L = \sigma(L_a)$$

$$\mathsf{step}(L_a, D) = (\theta, \sigma, \theta') \equiv D_{[-1]} = \langle G \mid \theta' \rangle \wedge \exists \sigma\ L = \sigma(L_a) \wedge\ \exists i\ D_{[i]} = \langle L :: G \mid \theta \rangle$$

Given a derivation whose current state is a call to L_a (normalized atom), the prestep function returns the substitution σ for L_a, and the constraint store θ at the predicate *call* (i.e., just before the literal is reduced). Given a derivation whose current state corresponds exactly to the return from a call to L_a, the step function returns the substitution σ for L_a, the constraint store θ at the call to L_a, and the constraint store θ' at L_a's *success* (i.e., just after all literals introduced from the body of L_a have been fully reduced).

An abstract constraint $\lambda^-_{TS(L,P)}$ is an *abstract trivial success subset* of L in P iff $\gamma(\lambda^-_{TS(L,P)}) \subseteq TS(L,P)$. An abstract constraint $\lambda^+_{TS(L,P)}$ is an *abstract trivial success superset* of L in P iff $\gamma(\lambda^+_{TS(L,P)}) \supseteq TS(L,P)$. Given the program P, the concrete and abstract sets of queries $\mathcal{Q}$ and $\mathcal{Q}_\alpha$[5] respectively, where $\gamma(\mathcal{Q}_\alpha) \supseteq \mathcal{Q}$, and $\langle L, \lambda^c, \lambda^s \rangle \in Analysis(P, \mathcal{Q}_\alpha, D_\alpha)$, the status of an assertion condition C, associated with it by the mapping $\mathsf{status}(c, Status)$ where c is the corresponding identifier, is determined as follows:

- $C = c.\mathsf{calls}(L, Precond) \wedge \mathsf{status}(c, \mathtt{checked})$ if $\lambda^c \sqsubseteq \lambda^-_{TS(Precond,P)}$.
- $C = c.\mathsf{success}(L, Pre, Post) \wedge \mathsf{status}(c, \mathtt{checked})$ if (1) $\lambda^c \sqcap \lambda^+_{TS(Pre,P)} = \bot$ or (2) $\lambda^s \sqsubseteq \lambda^-_{TS(Post,P)}$;
- $C = c.\mathsf{calls}(L, Precond) \wedge \mathsf{status}(c, \mathtt{false})$ if $\exists D \in \mathsf{derivs}(\mathcal{Q})$ s.t. $\mathsf{prestep}(L, D) = (\theta, \sigma) \wedge \bar{\exists}_L\theta \neq \emptyset$ and $\lambda^c \sqcap \lambda^+_{TS(Precond,P)} = \bot$.
- $C = c.\mathsf{success}(L, Pre, Post) \wedge \mathsf{status}(c, \mathtt{false})$ if $\lambda^c \sqcap \lambda^-_{TS(Pre,P)} \neq \bot$ and $\lambda^s \sqcap \lambda^+_{TS(Post,P)} = \bot$ and $\exists\ \theta \in \gamma(\lambda^c \sqcap \lambda^-_{TS(Pre,P)}) : \exists D \in \mathsf{derivs}(\mathcal{Q})$ s.t. $\mathsf{step}(L, D) = (\theta, \sigma, \theta') \wedge \bar{\exists}_L\theta' \neq \emptyset$.

The compile-time checking process can be seen as a revision of the assertion statuses where for each predicate literal L its *annotation* composed from the respective assertion conditions $\mathcal{A}^{usr}_C(L) = \{C \mid c.C \in \mathcal{A}_C(L) \wedge \mathsf{status}(c, S) \wedge S \in \{\mathtt{check}, \mathtt{true}\}\}$ given the analysis output of the form $\mathcal{A}^{ana}_C(L) = \{C \mid \forall c.C \quad \mathsf{status}(c, \mathtt{true})\}$ is rewritten into $\{C \mid c.C \in \mathcal{A}^{usr}_C(L) \wedge \mathsf{status}(c, S) \wedge S \in \{\mathtt{check}, \mathtt{checked}, \mathtt{false}\}\}$.

5. TAKING ADVANTAGE OF THE RUN-TIME CHECKING SEMANTICS DURING ANALYSIS

The standard analysis introduced in Section 4.1 safely approximates the traditional semantics (i.e., the semantics without assertions or run-time checks).[6] However, if we know that run-time checks will be performed for sure for a certain set of (`check`) assertions (as, e.g., for all assertions in the Safe-RT execution mode, or the ones corresponding to interface predicates in the client-safe mode), it is possible to use this information during analysis to improve precision,

[5] In the implementation of PLAI, $\mathcal{Q}_\alpha$ is obtained from the calls conditions of the assertions of exported predicates (or, if no such assertions are present, a "topmost" abstract state is assumed), or from specific "`entry`" assertions.

[6] Actually, assertions with `trust` status (which are typically used to provide information to improve analysis precision), and assertions with `true` status are in fact read and applied by the traditional analysis during its fixpoint calculation. However, in this discussion we refer to incorporating into the analysis the information present in `check` assertions, i.e., from the assertions being checked at compile time or run time. These assertions are not normally taken into account by the analysis since they may or may not hold and, in general, run-time tests may or may not be included in the compiled program.

```
1  :- module(_,[p/2]).        % p/2 is exported
2  :- use_module(foo,[e/2]). % e/2 is imported
3
4  p(X,Y) :- q(X,Y).
5
6  :- pred q(X,Y) : int(X) => int(Y).
7  q(X,Y) :- r(X,Y).
8
9  :- pred r(X,Y) : int(X) => int(Y).
10 r(X,Y) :- e(X,Y).
```

Figure 4: Example for analysis improvement.

i.e., to assume that the calls assertion conditions hold after the predicate has entered the predicate definition (since, according to the semantics of Section 2.3 either the checks for these calls assertion conditions have already succeeded or the program has exited with error), and to assume the relevant success assertion conditions after the predicate has exited (since, again, at this point either these success assertion conditions have already succeeded or the program has exited with error).

As an example, consider the program of Figure 4.[7] `p/2` is an exported predicate, `q/2` and `r/2` are local predicates, and `e/2` is imported. We allow both `p/2` and `e/2` to be called without any restriction, and we do not specify any constraints either regarding their successes. However, we want to enforce (through the two assertions) that `q/2` and `r/2` always be called with their first argument `X` bound to an integer, and that their second argument `Y` be bound to an integer upon success. Since any type of call is allowed to `p/2`, and without information on the presence of run-time checks, analysis cannot infer anything about the calls conditions for `q/2` and `r/2`, or for the success conditions of these two predicates, and will report warnings for unchecked conditions for all of them (and the two assertions will remain in `check` status).

However, note that, assuming we are generating run-time checks for all assertion conditions, the call to `r/2` in the body of `q/2` can only be reached if the calls condition for `q/2` holds, i.e., if `X` is bound to an integer (since otherwise execution would have been aborted). Thus, this information can be incorporated into the analysis and propagated to the call to `r/2`, and it can be determined that the calls condition for `r/2` (i.e., that its first argument be also bound to an integer) always holds. Thus, this calls condition for `r/2` gets status `checked` and no run-time test needs to be generated for it.

Similarly, the run-time test for the success condition for `r/2` ensures that if the call to `r/2` in the body of `q/2` returns, then its second argument is guaranteed to be bound to an integer. Thus, the success condition for `1/2` will also get status `checked` and no run-time test needs to be generated for it either.

Transformation.

A straightforward method to incorporate the information from successful checks into the analysis, so that it takes the semantics with run-time checking into account, would be to analyze the transformed program (i.e., the program including the code that performs the run-time tests) instead of the original one. This is the approach implied by the original transformational definitions of the assertion language. On the other hand, programs transformed for run-time testing contain numerous optimizations and instrumentation that make their analysis less efficient and can potentially affect precision. An alternative would be to use a very simple (even if inefficient) run-time checking transformation just for analysis. Inspired by this idea, we propose herein a different, even more direct approach, based on introducing additional assertions and link predicates in the program that together capture the run-time checking semantics and provide the additional information source for the analysis, in order to increase precision. This is performed as a program transformation T that precedes the analysis and is applied to every annotated predicate in a program:

$$T(L) = \langle \{L\texttt{:-}L_{inner}\} \cup \mathsf{defn}(L_{inner}), \mathcal{A}_C^{link} \cup \mathcal{A}_C^{inner} \rangle$$

where $L = p(\vec{X})$, and the literal $L_{inner} = p_{inner}(\vec{X})$ is obtained with a new predicate symbol p_{inner}, and:

$$\begin{aligned}
\mathsf{defn}(L_{inner}) &= \{L_{inner}\texttt{:-}B \mid L\texttt{:-}B \in \mathsf{defn}(L)\} \\
\mathcal{C} &= \{c.C \in \mathcal{A}_C(L) \mid \mathsf{status}(c, \texttt{check})\} \\
\mathcal{A}_C^{link} &= \{c^l.C \mid c.C \in \mathcal{C}\} \text{ and } \forall c^l.C \in \mathcal{A}_C^{link} \text{ we extend} \\
&\quad \text{the status relation s.t. } \mathsf{status}(c^l, S^l), \text{ where:} \\
S^l &= \begin{cases} \texttt{check} & \text{if } C = \mathsf{calls}(_,_) \\ \texttt{true} & \text{if } C = \mathsf{success}(_,_,_) \end{cases} \\
\mathcal{A}_C^{inner} &= \{c^i.C \mid c.C \in \mathcal{C}\} \text{ and } \forall c^i.C \in \mathcal{A}_C^{inner} \text{ we extend} \\
&\quad \text{the status relation s.t. } \mathsf{status}(c^i, S^i), \text{ where:} \\
S^i &= \begin{cases} \texttt{true} & \text{if } C = \mathsf{calls}(_,_) \\ \texttt{check} & \text{if } C = \mathsf{success}(_,_,_) \end{cases}
\end{aligned}$$

The objective of the transformation is to improve the precision and reduce the cost of the analysis, while preserving program behavior when the `check` assertion conditions are expanded into run-time checks. The transformation modifies all predicates with `check` assertions. For each such predicate p, the original predicate symbol is renamed into p_{inner} and a single-clause wrapper predicate for p (which we will refer to as a *link* clause), is introduced which calls the p_{inner} predicate.

The set of assertion conditions for the initial predicate p is duplicated for the p_{inner} counterpart, including their original statuses. However, the statuses of the `success` assertion conditions for p in the link clause and the `calls` assertion conditions of p_{inner} are set to `true`. As a result, the calls assertion conditions for p (i.e., $c^l.\mathsf{calls}(L,\_)$ with $\mathsf{status}(c^l, \texttt{true})$) will still be checked in the version with run-time checks, but they will be assumed in p_{inner} (i.e.,

```
1  :- check calls   q(X,Y) : int(X).
2  :- true  success q(X,Y) : int(X) => int(Y).
3  q(X,Y) :- q_inner(X,Y).
4
5  :- true  calls   q_inner(X,Y) : int(X).
6  :- check success q_inner(X,Y) : int(X) => int(Y).
7  q_inner(X,Y) :- r(X,Y).
```

Figure 5: CTRT program transformation example (output).

[7] In the examples we use just simple types as properties for conciseness, but even in this case please note that the use of types is *moded*, i.e., the assertions here express *states of instantiation*.

c^i.calls(L_{inner},_) with status(c^i, true)).

For the success part the assertion conditions will still be checked for the inner predicate (i.e., c^i.success(L_{inner},_,_) with status(c^i, check)) and the information will be assumed upon exiting p (i.e., c^i.success(L,_,_) with status(c^i, true)). The transformation guarantees that the same run-rime tests will be performed, that no duplication of checks will occur (since there are no intermediate states between the calls to p and p_{inner} and exits from p_{inner} to p), and that the analysis will gather the right information.

An example of the CTRT transformation for the q/2 predicate from the program in Fig. 4 is shown in Fig. 5. The true assertions here correspond to the additional information that can be safely used in the analysis. Since all predicates with assertions undergo this transformation, a number of inner calls coming from the link clauses is added to the program. Yet such calls are relatively inexpensive and the resulting runtime overhead is negligible. Even more, should the analysis verify the calls assertion condition of the link clause or the success assertion condition of the inner clause, the link clause then becomes unnecessary and can be completely removed.

Lemma 1 (Correctness of the CTRT Transformation) *Let P be a program and $\mathcal{Q} = (L, \theta)$ a query to P. Then $\forall\ D \in$ derivs(Q) the final state $D_{[-1]}$ is the same in the versions of P with and without the CTRT transformation.*

PROOF. First, let us prove the correctness of the transformation for the *calls* assertion conditions.

Let $\mathcal{A}_C(L) = \{C\}$ where $C = c$.calls(L, Pre) s.t. status$(c,$ check$)$ and $\exists(L$:- $B) \in$ defn(L). The possible reduction sequences from the $S_0 = \langle L :: G \mid \theta \mid \emptyset\rangle$ state are:

$S_0 \leadsto_{\mathcal{A}} \langle B :: G \mid \theta \mid \emptyset\rangle = S_{succ}$ if $\theta \Rightarrow_P Pre$

$S_0 \leadsto_{\mathcal{A}} \langle L \mid \theta \mid \{\bar{c}\}\rangle = S_{err}$ if $\theta \not\Rightarrow_P Pre$

Now let us add the *link* clause for L and rename its other clauses s.t. defn$(L) = \{L$:- $L_{inner}\}$ and $\exists L_{inner}$:- $B \in$ defn(L_{inner}), and let's add an assertion condition for L_{inner}: $C^{inner} = c^i$.calls(L_{inner}, Pre) with status$(c^i,$ check$)$. The possible reduction sequences from the S_0 state now are:

$S_0 \leadsto_{\mathcal{A}} \langle L_{inner} :: G \mid \theta \mid \emptyset\rangle \leadsto_{\mathcal{A}} S_{succ}$ if $\theta \Rightarrow_P Pre$

$S_0 \leadsto_{\mathcal{A}} S_{err}$ if $\theta \not\Rightarrow_P Pre$

The $S_0 \leadsto_{\mathcal{A}} \langle L_{inner} :: G \mid \theta \mid \emptyset\rangle \leadsto_{\mathcal{A}} \langle L_{inner} \mid \theta \mid \{\bar{c}^i\}\rangle$ reduction sequence is impossible since it would require $\theta \Rightarrow_P Pre$ to hold in the first reduction step and $\theta \not\Rightarrow_P Pre$ to hold in the second reduction step.

This way in both assertion checking modes $D_{[-1]} \in \{S_{succ}, S_{err}\}$ and run-time checks for the *calls* assertion condition C^{inner} (namely, checks for $\theta \Rightarrow_P Pre$ after the checks for $\theta \Rightarrow_P Pre$) could be safely removed by setting status$(c^i,$ true$)$.

Next, let's consider the case of *success* assertion conditions.

Let $\mathcal{A}_C(L) = \{C\}$ where $C = c$.success$(L, Pre, Post)$ s.t. status$(c,$ check$)$ and $\exists(L$:- $B) \in$ defn(L). The possible reduction sequences from the $S_0 = \langle L :: G \mid \theta \mid \emptyset\rangle$ state are:

$S_0 \leadsto_{\mathcal{A}} \langle B :: \text{acheck}(L, c) :: G \mid \theta \mid \emptyset\rangle \leadsto^*_{\mathcal{A}} \langle G \mid \theta \mid \emptyset\rangle = S_{succ}$
if $\theta \Rightarrow_P Post$

$S_0 \leadsto_{\mathcal{A}} \langle B :: \text{acheck}(L, c) :: G \mid \theta \mid \emptyset\rangle \leadsto^*_{\mathcal{A}} \langle \text{acheck}(L, c) \mid \theta \mid \bar{c}\rangle = S_{err}$
if $\theta \not\Rightarrow_P Post$

Now let us add the *link* clause for L and rename its other clauses s.t. defn$(L) = \{L$:- $L_{inner}\}$ and $\exists L_{inner}$:- $B \in$ defn(L_{inner}), and let's add an assertion condition for L_{inner}: $C^{inner} = c^i$.success$(L_{inner}, Pre, Post)$ with status$(c^i,$ check$)$. We also now consider C as C^{link} with its identifier c^l. The possible reduction sequences from the S_0 state now are:

$S_0 \leadsto_{\mathcal{A}} \langle B :: \text{acheck}(L, c^i) :: \text{acheck}(L, c^l) :: G \mid \theta \mid \emptyset\rangle \leadsto^*_{\mathcal{A}} \langle G \mid \theta \mid \emptyset\rangle = S_{succ}$
if $\theta \Rightarrow_P Post$

$S_0 \leadsto_{\mathcal{A}} \langle B :: \text{acheck}(L, c^i) :: \text{acheck}(L, c^l) :: G \mid \theta \mid \emptyset\rangle \leadsto^*_{\mathcal{A}} \langle \text{acheck}(L, c^l) \mid \theta \mid \bar{c}^l\rangle = S_{err}$
if $\theta \not\Rightarrow_P Post$

Although the assertion condition identifiers for the two S_{err} are different, the checks performed in these states are equal ($\theta \not\Rightarrow_P Post$).

This way the run-time checks for the c^l assertion condition are duplicating the checks for c^i and could be safely removed by setting status$(c^l,$ true$)$. □

6. EXPERIMENTS

As stated throughout the paper, our objective is to explore the effectiveness of abstract interpretation in detecting parts of program specifications that can be statically simplified to true or false, and to quantify the impact of this application of analysis towards reducing the cost of the run-time checks. In particular, we have studied these issues for the different assertion checking modes that we have defined.

6.1 Experimental Setup

We have built an experimental harness by extending the Ciao preprocessor, CiaoPP, which implements our baseline assertion verification framework. The architecture of this framework is shown in Figure 6. In that figure, hexagons represent system tools and components and arrows indicate the communication paths among them. Most of this communication is performed also in terms of assertions.

The input to the verification process is the user program, optionally including a set of assertions; this set always includes any assertions present for predicates exported by any libraries used. The check and trust/true assertions are normalized and the program is expanded to kernel form (simple horn clauses), and they are all given as input to the *static analysis*.

We have introduced new front-end passes implementing the new transformations (marked in Figure 6) which thus support the defined scenarios, as well as some other minor adaptations and extensions to the interface to select these different scenarios.

The results of analysis over the different abstract domains selected are provided in the form of true assertions. Then, for every predicate p in the program the framework performs compile-time checking of assertions by *comparing* the check assertions in the program (their assertion conditions) with the analysis results.

As a possible result of the comparison, assertions may be proved to hold, in which case they get checked status. As another possible result, assertions can be proved not to hold, in which case they get false status and a compile-time error is issued. Finally, if it is not possible to prove nor to disprove (part of) an assertion, then such assertion (or the relevant subset) is left as a check assertion, and the run-time check annotator introduces run-time checking code in the program for the assertion conditions as required by the scenario. In particular, the program transformations used in our experiments are those of [44], with no caching.

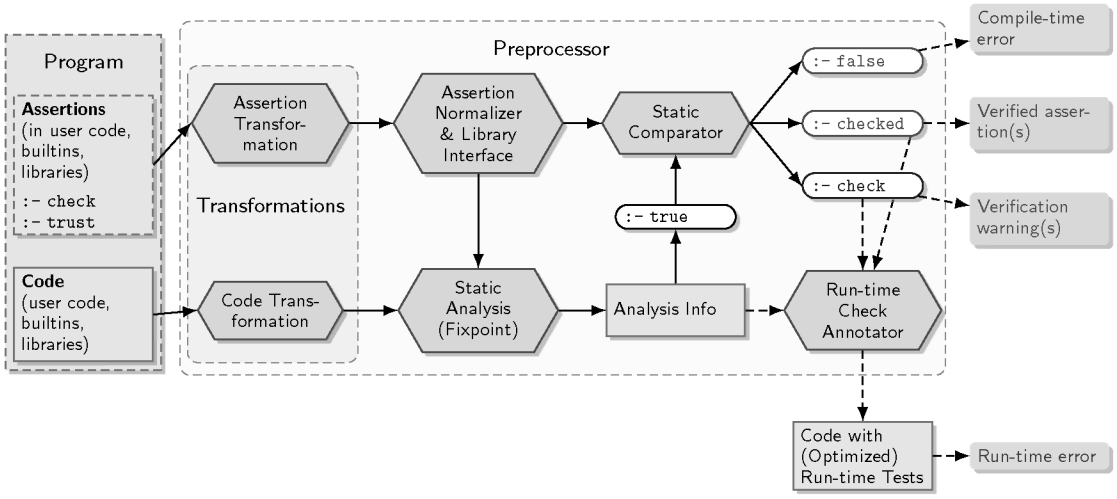

Figure 6: Adding the transformations to the Ciao Preprocessor.

6.2 Properties and Analysis Domains

In our experiments we concentrate on two classes of properties. The first one is the *state of variable instantiation*, i.e., which variables are bound to ground terms, or unbound, and, if they are unbound, the sharing (aliasing) patterns in order to be able to transfer accurately grounding information ("strong update"). These properties are approximated safely and quite accurately using the *sharing and freeness* domain [30]. The second class of properties we will be using refers to the shapes of the data structures constructed by the program in memory. To this end we use the *eterms* [48] abstract domain which infers safely these shapes as regular trees.[8]

6.3 Benchmarks

To study the differences in the run-time overhead levels observed in different assertion checking modes we have selected a set of benchmarks, listed in Table 2.[9] These benchmarks represent simple yet diverse programs that represent frequently-occurring programming patterns such as performing symbolic or arithmetic computations, problem solving in fixed domains, processing stream data, etc. In general, they include recursion, search, irregular/dynamic

Table 2: Benchmarks.

`boyer`	a theorem prover implementation based on Lisp by R. Boyer (nqthm system), performs symbolic evaluation of a given formula;
`boyerx`	a variant of `boyer` (using generic term manipulation predicates for formula rewrites);
`crypt`	cryptomultiplication puzzles solver;
`deriv`	a program that performs symbolic differentiation of a given formula;
`exp`	exponential calculation;
`factorial`	recursive factorial calculation;
`fft`	fast Fourier transformation calculation;
`fib`	a program that finds N-th Fibonacci number;
`guardians`	prison guards game;
`hamming`	a program that generates the sequence of Hamming numbers;
`hanoi`	hanoi towers puzzle solver for N disks that are moved over three rods;
`jugs`	the water jugs problem;
`knights`	N knights chess problem;
`mmatrix`	matrix multiplication for two matrices with dimensions $n \times n$;
`nreverse`	naive list reversal;
`poly`	a program that raises a polynomial $(1 + x + y + z)$ to the 10th power symbolically;
`primes`	a program that computes N first prime numbers;
`progeom`	a program that constructs a perfect difference set of order N;
`queens`	the N queens program, the number of the queens being the input;
`qsort`	the quicksort program;
`serialize`	a palindrome program;
`tak`	a program that computes the *tak* function;
`witt`	the WITT clustering system implementation;

[8] Note that these notions are more general and powerful than the traditional notions of types of modes. Also, comparing to the *traditional* notion of type inference in statically-typed languages, not only the types are generalized to any property supported by an abstract domain, but also the overall approach is quite different: there all the type definitions must be present in the program, and the inference problem just amounts to assigning one of these types to each program element in a single pass. If this assignment is not possible the program is rejected. In the Ciao approach, no type definitions are required. The purpose of analysis is precisely to infer, in a closed form (e.g., regular types) the shapes of the data structures that are built in memory for the whole program, which is done via a fixpoint calculation. Also, note that the regular types inferred and checked allow sub-typing. The situation is similar for the sharing+freeness domain versus, for example, traditional modes. This is also a strong difference with other approaches within logic programming, such as Mercury or Gödel, which also require the type definitions to be provided and that the program be typeable. See [16] for a further discussion to the very interesting topic of how to best straddle the dynamic vs. static language boundaries.

[9] Source available at https://cliplab.org/papers/optchk-ppdp2016/

Table 3: Benchmark metrics.

Benchmarks			Assertions
Name	LOC	Size (KB)	total
boyer	853	70	25
boyerx	853	50	23
crypt	76	10	16
deriv	29	9	3
exp	28	6	5
factorial	13	4	3
fft	104	13	19
fib	11	5	5
guardians	78	9	13
hamming	71	9	19
hanoi	44	6	4
jugs	132	10	9
knights	49	8	13
mmatrix	48	6	6
nreverse	14	5	5
poly	81	12	14
primes	33	6	8
progeom	71	8	16
qsort	46	6	8
queens	47	6	10
serialize	81	10	9
tak	18	5	3
witt	651	50	83

Table 4: Benchmarks: full compilation time.

Benchmark	Compilation time, ms			
	Unsafe	Safe		
		Client	RT	CT+RT
boyer	215	1 413	1 222	393 869
boyerx	194	1 193	1 191	25 275
crypt	148	753	755	3 776
deriv	142	655	702	993
exp	141	691	684	976
factorial	139	650	658	1 207
fft	154	835	827	2 757
fib	139	574	664	1 154
guardians	146	767	775	1 323
hamming	162	796	713	1 435
hanoi	142	662	700	834
jugs	142	728	748	1 277
knights	153	748	724	1 347
mmatrix	146	697	514	982
nreverse	139	662	686	882
poly	147	626	788	1 755
primes	139	690	696	998
progeom	140	776	606	1 492
qsort	136	712	692	985
queens	149	700	554	1 143
serialize	145	735	721	1 353
tak	140	604	621	751
witt	219	1 418	1 558	149 855

data structures, etc. The relative internal complexity despite their generally small size make them good candidates to answer our main questions, allowing us to concentrate on the properties of interest in each case. All the benchmarks have been carefully annotated with reasonable program assertions that describe the expected behaviour. E.g., this is a selection of the `fft` code:

```
1 :- regtype complex/1. % A complex number
2 complex((A,B)) :- num(A), num(B).
3
4 :- pred complex_mul(A, B, C) % Multiplication
5     : complex * complex * term
6    => complex * complex * complex.
7 complex_mul((Ra,Ia), (Rb,Ib), (Rs,Is)) :-
8     Rs is Ra*Rb-Ia*Ib,
9     Is is Ra*Ib+Rb*Ia.
```

Table 3 presents some quantitative characteristics of the benchmarks, such as lines of code (LOC) and size metrics, and also the total number of program assertions.

While the effectiveness of our assertion-based approach is not directly the objective of this paper, it is worth noticing that during our experiments the analysis on one of the more complex programs, `boyer`, has allowed us to spot bugs in the original translation from LISP that had been around for 30 years.

6.4 Experimental Results

Table 4 shows the compilation time for the benchmarks under the different assertion checking modes.[10] The compilation time for the benchmarks under the *Safe-CT-RT* mode includes the total static analysis and assertion checking times. The experiments were run on a MacBook Pro with 2.3GHz Intel Core i7 processor, 16GB RAM, and under the Mac OS X 10.11.1 operating system.

Table 5 shows the detailed analysis times for the *Safe-CT-RT* mode for the benchmarks. The *prep* columns indicate the time needed to load and prepare the analysis, and the *shfr* and *eterms* columns the time to perform sharing and freeness and regular type analyses, respectively. The analysis is actually relatively inexpensive compared to the rest of the compilation passes for most of the benchmarks. The regular type analysis is expensive in `boyer` and `boyerx`. The analysis of the formula rewrite predicates generates many large types whose manipulation is expensive. The `witt` benchmark, despite having more regular data structures (tables of sets and matrices), is also expensive to analyze due to a large number of operations. Note that in any case more efficient –but less precise– domains are available for these cases, many within CiaoPP, such as, for example, several widenings for sharing [32, 24], pair sharing domains [42, 41], or other type inference domains [11, 3].

Finally, Table 6 reports on the execution times for each benchmark in the different assertion checking modes, together with the statistics of assertion checking results. For some of the benchmarks, inputs were varied and this is reflected by the notation $Name(Input)$. The 'Checked/Total

[10]Times for compilation and analysis assume that the compiler and analyzer are already loaded in memory and ready to execute. Thus, we removed the compiler and CiaoPP start-up time. In the current implementation, the engine needs around 1.4 seconds to load all the necessary bytecode but can then process different programs (e.g., interactively, from within the development environment) without having to be restarted. There exist in any case many solutions to significantly reduce this startup time time (keeping code in memory, optimizing the bytecode reader, reduced versions of CiaoPP that contain only the necessary domains, lazy load, etc.).

Table 5: Static analysis time for benchmarks using the *Safe-CT-RT* checking mode (part of total compilation time).

Benchmark	Analysis time, ms				Assertion checking
	prep	shfr	prep	eterms	
boyer	8.185	56.577	8.469	634.059	521.067
boyerx	5.401	42.500	5.456	467.579	343.906
crypt	1.397	8.100	1.359	35.164	115.782
deriv	0.711	2.426	0.669	13.877	24.630
exp	0.402	1.593	0.373	12.471	42.513
factorial	0.276	0.906	0.193	9.300	11.032
fft	1.544	7.425	1.758	36.885	143.532
fib	0.266	1.083	0.215	11.240	13.475
guardians	1.059	4.455	1.049	19.958	46.498
hamming	1.044	6.460	1.022	21.417	60.509
hanoi	0.496	2.267	0.477	11.629	17.453
jugs	0.837	4.039	0.981	22.661	101.270
knights	0.774	3.483	0.748	22.240	46.713
mmatrix	0.480	2.359	0.449	11.671	23.010
nreverse	0.278	2.673	0.253	2.763	7.489
poly	1.414	41.274	1.419	42.363	79.432
primes	0.571	2.278	0.486	12.547	25.364
progeom	0.933	5.600	0.891	21.057	47.704
qsort	0.526	3.725	0.504	6.562	18.491
queens	0.614	3.448	0.587	15.489	35.767
serialize	1.076	11.775	0.984	17.515	46.347
tak	0.327	1.427	0.271	9.986	19.747
witt	13.622	104 625.161	14.430	666.404	1 079.152

Assrts' column reports the numbers of statically checked assertions (in the *Safe-CT-RT* checking mode) and total number of assertions for each benchmark. In the worst case the overhead in the *Safe-RT* checking mode is two orders of magnitude higher than in *Client-safe*, but *Safe-CT-RT* removes one order of magnitude (`boyerx`, `fft`, `knights`, `witt`). This is expected since run-time checks of complex properties like data shapes cannot be performed in constant time. The checking changes the complexity of the programs and the overhead increases as the size of the input grows. Note that the *Client-safe* mode also represents the theoretically lowest overhead that we could obtain (assumming a fixed implementation of the instrumentation), by removing internal checks. Sometimes, due to measurement imprecision, the time in *Safe-CT-RT* mode is smaller than in *Client-safe* mode (`crypt`, `exp`, `primes`). In practice, in many programs *Safe-CT-RT* is able to remove most of the checks, except those corresponding to the external predicates. We included in the benchmarks two versions of `boyer`. The original translation (which we call here `boyerx`) uses `functor/3` and `arg/3` to implement rewrites of arbitrary terms representing formulas. This makes the domains lose precision. The `boyer` version used instead a larger predicate that explicitly enumerates possible formula terms.

7. CONCLUSIONS

We have addressed the problem of run-time overhead reduction in the context of verification frameworks that combine static and dynamic verification, i.e., systems that combine compile-time and run-time checking of user-provided assertions. Our ultimate objective is to construct an automatic verification system for non-trivial, structural properties, that can be used routinely in production code.

We have defined four practical assertion checking modes, and studied the corresponding trade-offs between the level of guarantees provided by each one and the corresponding execution time slowdown. For these checking modes we have explored the effectiveness of abstract interpretation in detecting the parts of the program's (partial) specifications that can be statically simplified to true or false, as well as the practical impact of such analysis in reducing the cost of the run-time checks required for the remaining parts of the specifications.

Our experiments have shown that there is indeed a significant advantage in using analysis to reduce run-time tests. We argue that the results are encouraging, supporting the hypothesis that the combination of run-time checking with analysis can reduce checking overhead sufficiently to allow providing full safety in production code, for non-trivial properties. It is worth noticing that the analysis on more complex code like `boyer` allowed us to spot bugs in the original translation from LISP that had been around for 30 years.

We have presented for concreteness our approach in the context of the Ciao 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 should carry over to other (dynamic) declarative or imperative languages.

8. REFERENCES

[1] N. Bjørner, F. Fioravanti, A. Rybalchenko, and V. Senni, editors. *Workshop on Horn Clauses for Verification and Synthesis*, July 2014. To appear in Electronic Proceedings in Theoretical Computer Science.

Table 6: Benchmark execution times under the different modes and checked vs. total assertions.

Benchmark	Execution time, ms				Checked/Total Assrts	
	Unsafe	Safe			CT+RT	Total
		Client	RT	CT+RT		
boyer	10.404	10.390	2 412.197	12.545	25	25
boyerx	15.185	15.511	2 284.085	1 098.419	21	23
crypt	0.098	0.108	6.196	0.105	15	16
deriv	0.110	0.730	4.640	0.640	2	3
exp	3.838	3.971	63.210	3.899	4	5
factorial	0.008	0.013	0.748	0.013	2	3
fft	23.234	23.972	25 879.525	202.655	17	19
fib	0.067	0.073	13.069	0.072	4	5
guardians	2.960	2.980	5 805.539	3.221	12	13
hamming	15.259	15.071	7 431.143	17.012	18	19
hanoi (2)	0.001	0.011	0.139	0.011	3	4
hanoi (4)	0.002	0.012	1.324	0.013		
hanoi (8)	0.054	0.064	106.649	0.078		
jugs	0.014	0.022	1.660	0.023	8	9
knights	201.278	192.987	17 979.597	202.061	12	13
mmatrix (2)	0.001	0.007	0.127	0.008	5	6
mmatrix (3)	0.002	0.010	0.319	0.010		
mmatrix (4)	0.005	0.015	0.662	0.016		
nreverse	2.243	2.313	8 217.188	3.125	4	5
poly	1.084	1.272	356.360	1.302	13	14
primes	0.027	0.041	8.860	0.036	7	8
progeom (2)	0.002	0.006	0.630	0.005	15	16
progeom (4)	0.089	0.093	24.424	0.103		
progeom (8)	5.274	5.268	1 890.538	5.934		
qsort (8)	0.003	0.006	0.929	0.007	7	8
qsort (16)	0.008	0.013	2.396	0.015		
qsort (32)	0.021	0.028	7.477	0.033		
queens (4)	0.006	0.009	1.076	0.010	9	10
queens (6)	0.122	0.123	20.605	0.139		
queens (8)	2.302	2.281	360.660	2.559		
serialize (9)	0.002	0.007	0.803	0.008	8	9
serialize (16)	0.005	0.011	2.207	0.012		
serialize (25)	0.010	0.017	5.029	0.019		
tak	2.855	2.866	963.481	3.665	2	3
witt	15.886	15.947	1 511.059	316.063	69	83

[2] J. Boye, W. Drabent, and J. Małuszyński. Declarative Diagnosis of Constraint Programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linköping, Sweden, May 1997. U. of Linköping Press.

[3] M. Bruynooghe and J. Gallagher. Inferring Polymorphic Types from Logic Programs. In *International Symposium on Logic-based Program Synthesis and Transformation (LOPST R'04)*. Preproceedings, July 2004.

[4] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. 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*, pages 155–170. U. Linköping Press, May 1997.

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

[6] 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.

[7] C. Dimoulas and M. Felleisen. On contract satisfaction in a higher-order world. *ACM Trans. Program. Lang. Syst.*, 33(5):16, 2011.

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

[9] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software*, FoVeOOS'10, pages 10–30, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In M. Wand and S. L. P. Jones, editors, *ICFP*, pages 48–59. ACM, 2002.

[11] J. Gallagher and D. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proc. of the 11th*

International Conference on Logic Programming, pages 599–613. MIT Press, 1994.

[12] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A Software Verifier Based on Horn Clauses - (Competition Contribution). In C. Flanagan and B. König, editors, *TACAS*, volume 7214 of *LNCS*, pages 549–551. Springer, 2012.

[13] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The seahorn verification framework. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.

[14] M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

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

[16] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. Morales, and G. Puebla. The Ciao Approach to the Dynamic vs. Static Language Dilemma. In *Proceedings for the International Workshop on Scripts to Programs, STOP'11*, New York, NY, USA, 2011. ACM.

[17] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012. http://arxiv.org/abs/1102.5497.

[18] E. Koukoutos and V. Kuncak. Checking Data Structure Properties Orders of Magnitude Faster. In B. Bonakdarpour and S. A. Smolka, editors, *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 263–268. Springer International Publishing, 2014.

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

[20] L. Lamport and L. C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, May 1999.

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

[22] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder. Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR. In M. V. Eekelen and U. D. Lago, editors, *Foundational and Practical Aspects of Resource Analysis. Fourth International Workshop FOPARA 2015, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2016. To appear.

[23] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In G. Gupta and R. Peña, editors, *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Revised Selected Papers*, volume 8901 of *Lecture Notes in Computer Science*, pages 72–90. Springer, 2014.

[24] M. Méndez-Lojo, O. Lhoták, and M. Hermenegildo. Efficient Set Sharing using ZBDDs. In *21st Int'l. WS on Languages and Compilers for Parallel Computing (LCPC'08)*, LNCS. Springer-Verlag, August 2008.

[25] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*, number 4915 in Lecture Notes in Computer Science, pages 154–168. Springer-Verlag, August 2007.

[26] E. Mera, P. López-García, and M. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *25th Int'l. Conference on Logic Programming (ICLP'09)*, number 5649 in LNCS, pages 281–295. Springer-Verlag, July 2009.

[27] E. Mera, T. Trigo, P. López-García, and M. Hermenegildo. Profiling for Run-Time Checking of Computational Properties and Performance Debugging. In *Practical Aspects of Declarative Languages (PADL'11)*, volume 6539 of *Lecture Notes in Computer Science*, pages 38–53. Springer-Verlag, January 2011.

[28] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

[29] K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.

[30] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *International Conference on Logic Programming (ICLP 1991)*, pages 49–63. MIT Press, June 1991.

[31] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[32] J. Navas, F. Bueno, and M. Hermenegildo. Efficient top-down set-sharing analysis using cliques. In *Eight International Symposium on Practical Aspects of Declarative Languages*, number 2819 in LNCS, pages

183–198. Springer-Verlag, January 2006.

[33] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications. In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, pages 29–32, April 2008. Extended Abstract.

[34] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 65–82. Elsevier - North Holland, March 2009.

[35] P. Pietrzak, J. Correas, G. Puebla, and M. Hermenegildo. Context-Sensitive Multivariant Assertion Checking in Modular Programs. In *13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'06)*, number 4246 in LNCS, pages 392–406. Springer-Verlag, November 2006.

[36] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *ILPS'97 WS on Tools and Environments for (C)LP*, October 1997. `ftp://cliplab.org/pub-/papers/assert_lang_tr_discipldeliv.ps.gz`.

[37] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[38] G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, March 2000.

[39] A. Rastogi, N. Swamy, C. Fournet, G. M. Bierman, and P. Vekris. Safe & efficient gradual typing for typescript. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 167–180. ACM, 2015.

[40] T. W. Schiller, K. Donohue, F. Coward, and M. D. Ernst. Case studies and tools for contract specifications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 596–607, New York, NY, USA, 2014. ACM.

[41] S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *12th International Symposium Static Analysis Symposium (SAS'05)*, volume 3672 of *Lecture Notes in Computer Science*. Springer, 2005.

[42] H. Søndergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

[43] N. Stulova, J. F. Morales, and M. V. Hermenegildo. Assertion-based Debugging of Higher-Order (C)LP Programs. In *16th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'14)*. ACM Press, September 2014.

[44] N. Stulova, J. F. Morales, and M. V. Hermenegildo. Practical Run-time Checking via Unobtrusive Property Caching. volume 15, pages 726–741. Cambridge U. Press, September 2015.

[45] A. Takikawa, D. Feltey, E. Dean, M. Flatt, R. B. Findler, S. Tobin-Hochstadt, and M. Felleisen. Towards practical gradual typing. In J. T. Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 4–27. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[46] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? In R. Bodík and R. Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 456–468. ACM, 2016.

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

[48] C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *International Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.