Some Trade-offs in Reducing the Overhead of Assertion Run-time Checks via Static Analysis[☆]

Nataliia Stulova[a,b,*], José F. Morales[a], Manuel V. Hermenegildo[a,b]

[a] *IMDEA Software Institute, Madrid, Spain*
[b] *Universidad Politécnica de Madrid (UPM), Madrid, Spain*

Abstract

A number of approaches have been proposed for helping programmers detect incorrect program behaviors, which are based on combining language-level constructs (such as procedure-level assertions/contracts, program-point assertions, or gradual types) with a number of associated tools (such as code analyzers and run-time verification frameworks) that automatically check the validity of such constructs. However, these constructs and tools are often not used to their full extent in practice due to a number of limitations which may include excessive run-time overhead, limited expressiveness, and/or limitations in the effectiveness of the associated tools. Verification frameworks that combine static and dynamic verification techniques and are based on abstraction 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 obtained

[☆] This paper is an extended version of [1], presented at PPDP 2016.
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*

[*] Corresponding author.
Email addresses: `nataliia.stulova@imdea.org` (Nataliia Stulova), `josef.morales@imdea.org` (José F. Morales), `manuel.hermenegildo@imdea.org` (Manuel V. Hermenegildo), `manuel.hermenegildo@upm.es` (Manuel V. Hermenegildo)

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 safety. We then explore these modes in two typical, library-oriented scenarios. We also propose program transformation-based methods 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 benefits obtained from analysis and the proposed transformations in these scenarios.

Keywords: Abstract Interpretation, Assertions, Run-time Checking, Verification, Logic Programming, Horn Clauses

1. Introduction

Detecting and avoiding 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 this process, among which we center our attention on the use of language-level constructs to describe expected program behavior, and of associated tools to automatically 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 [2–10], soft/gradual typing approaches in functional programming [11–17], and contract-based extensions in object-oriented programming [18–20]. 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 complex 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 [15, 17]. In [10] overhead reductions were obtained by limiting the points at which the tests are performed and the instrumentation, as well as by inlining, but some types of tests still incurred significant costs. Other approaches opt for limiting the expressiveness of the assertion language in order to reduce the overhead (see [21] 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 [22, 23]: the time overhead of repeated checks on immutable recursive data structures is traded for increased memory use via caching and/or tabling techniques.

However, despite these advances, run-time overhead often remains impractically high, for example for properties which require deep data structure tests. This reduces the attractiveness of run-time checking to programmers, which may activate 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 [3, 4, 6, 7, 9, 24, 25] 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 supporting this hypothesis from the regular use of the system, there has been little systematic experimental work presented to date verifying this, i.e., measuring the actual impact of analysis on reducing run-time checking overhead. For example, in [10, 26] 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 ab-

stract 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*, which allows us 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 argue that the results are applicable to other programming paradigms, either directly (e.g., to other forms of declarative programming), or to imperative programs, via transformation into Horn clauses. The use of Horn clauses in the Ciao system as an intermediate language to support programs in other languages was described in [27]. Some concrete examples of the application of this approach that we have explored within Ciao include cost analysis of Java

bytecode programs [28, 29] and energy bound inference in binaries stemming from C-style programs [30, 31]. Recently [32] proposed an approach for using Horn clauses as an intermediate language which is quite similar to Ciao's [27].

The Horn clause-based transformational approach is currently receiving considerable interest (see, e.g., [33–35]), and is even the subject of the "Horn clause-based Verification and Synthesis" workshop series [36]. In [37] encouraging results are reported for the direct inference of the verification conditions of safety properties for C programs based on their (C)LP representation. Similar approaches have been used to translate to other formalisms, such as term rewrite systems [38].

Regarding the Ciao model of simplifying run-time checks using analysis information, that we base our work on, there has been recent work in the context of run-time monitoring frameworks for imperative programs that uses similar ideas to exploit static analyses in order to reduce the run-time overhead of the monitors as, e.g., proposed in [39] for Java programs, in addition to the already mentioned approaches for reducing run-time checking overhead via caching [22, 23].

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 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 transformational approach for taking advantage of the run-time checking semantics to improve the precision of the analysis. Section 6 discusses the application of this approach when optimizing run-time checks for the calls across client-library boundaries. Section 7 describes our experimental harness and presents our results for the different options (with

and without analysis, as well as with and without improved analysis precision). Section 8 finally presents some 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 [7, 23, 40].

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 $\bar{\exists}_L \theta$ 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 $(L\texttt{:-}B) \in \mathsf{defn}(L)$.

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 finite derivation from a query (L, θ) is *finished* if the last state in the derivation cannot be reduced. 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 $\bar{\exists}_L \theta'$ 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 [3, 6, 7, 25, 41]. 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., classical types.

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

$$\texttt{:- pred } Head \texttt{ : } \quad Pre_1 \texttt{ => } Post_1\texttt{.}$$
$$\ldots$$
$$\texttt{:- pred } Head \texttt{ : } \quad Pre_n \texttt{ => } Post_n\texttt{.}$$

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 $A_i =$ "$\texttt{:- pred } Head \texttt{ : } Pre_i \texttt{ => } Post_i\texttt{.}$" 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.:

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

We also assign unique identifiers to each assertion condition, represented by the c_i above. 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

[1] We follow the more compact formalization of [40], using only `pred` assertions. See also [7] for the original presentation using `calls` and `success` assertions. We are also not dealing herein with `comp` assertions and `comp` 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.

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_σ that is a renaming of some normalized atom L s.t. $L_\sigma = \sigma(L)$ and the corresponding set of (also renamed apart) assertion conditions $\mathcal{A}_C(L)$, the assertion conditions for L_σ 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 $C_\sigma = \sigma(C) = c_\sigma.\mathsf{calls}(L_\sigma, \sigma(Pre))$ (or $C_\sigma = c_\sigma.\mathsf{success}(L_\sigma, \sigma(Pre), \sigma(Post))$). 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 [40], 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 and $\theta \wedge L$ is satisfiable then the new state is $S' = \langle G \mid \theta \wedge L \mid \varnothing \rangle$, in the 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 $c_i.\mathsf{success}(L, Pre_i, Post_i) \in \mathcal{A}_C(L)\ \wedge\ \theta \Rightarrow_P Pre_i$.
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 (increases in execution time, code size, and memory use).

We propose a view on this compromise that differentiates among various levels of behavioral safety guarantees embodied in different *assertion checking modes*. We consider for concreteness the context of developing a standalone library that provides an *open* interface to its clients. By this we mean that at the time of analyzing and instrumenting the library the clients are not known and can be expected to call the library in both correct and incorrect ways, i.e., we do not require the clients to verify that the calls to the library adhere to the interface. Also, we do not expect the library to be recompiled (or reanalyzed)

depending on the needs of each client.[3] Thus, the library has to be analyzed and checked independently of the clients. We define three scenarios in this context, depending on the level of guarantees that the library provides to the clients that use it.

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 providing no safety guarantees to the clients. I.e., 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 all the assertions in the code. This of course eliminates any overhead but at the cost of not being able to ensure correctness. However, we still consider it, first because it represents a baseline to compare to, and also because of the frequent –even if not recommendable– practice of turning off run-time checks for production code, in order to avoid overhead, which is typically done if it is perceived that sufficient testing was carried on the code out prior to delivery.

Client-safe Checking Mode. In this checking mode the library provides the client with behavior guarantees on its interface, but does not check any of the assertions for the internal procedures. Run-time checks are thus generated only for

[3] This is all in contrast with the scenario in which the whole set of modules involved is available and can be processed as a whole, monolithically or modularly [42, 43]. Similarly, we also do not address directly in this work link-time optimizations.

```
1  :- module(_, [p]). % p is exported
2
3  :- check pred p : Pre => Post.
4
5
6  p :- body. % no calls to p/1
7             % for simplicity
8
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.

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, the run-time semantics under such mode is:

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 the 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:

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

   and $G' = \mathsf{acheck}(L, c_1) :: \ldots :: \mathsf{acheck}(L, c_n) :: G$ such that $c_i.\mathsf{success}(L, Pre_i, Post_i) \in \mathcal{A}_C(L)\ \wedge\ \theta \Rightarrow_P Pre_i$.
3. If L is a check literal $\mathsf{acheck}(L', c)$, then S' is obtained as

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

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, which is not required by the definition of the scenario, which only establishes the checking of the calls that cross the interface. 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. This transformation introduces intermediate *link* predicates for the 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

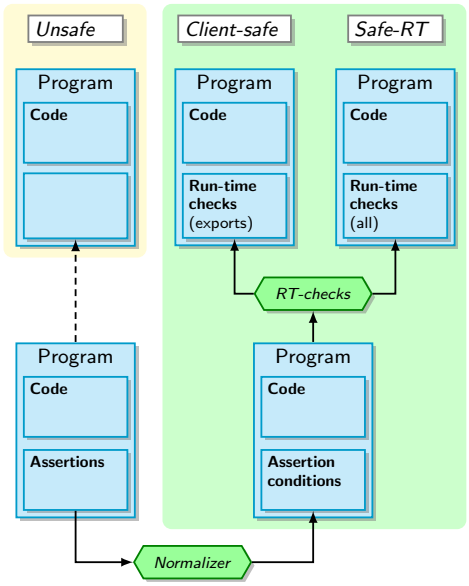

Figure 2: Source transformation differences per checking mode.

obtaining safety guarantees at the library boundaries with minimal run-time checking execution time overhead.

Safe-RT Execution Mode. In this mode the library provides behavior guarantees both on its interface and its internals. Run-time checks are thus 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 ensuring 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.

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. In this mode the run-time checks for the *calls* assertion conditions of the exported predicates are left untouched in any case, in order to ensure the safety of calls in our open-library context.

4.1. Abstract Interpretation-based Analysis

For analysis we use the technique of abstract interpretation [44], which safely approximates the execution of a program on an *abstract domain* (D_α) which is

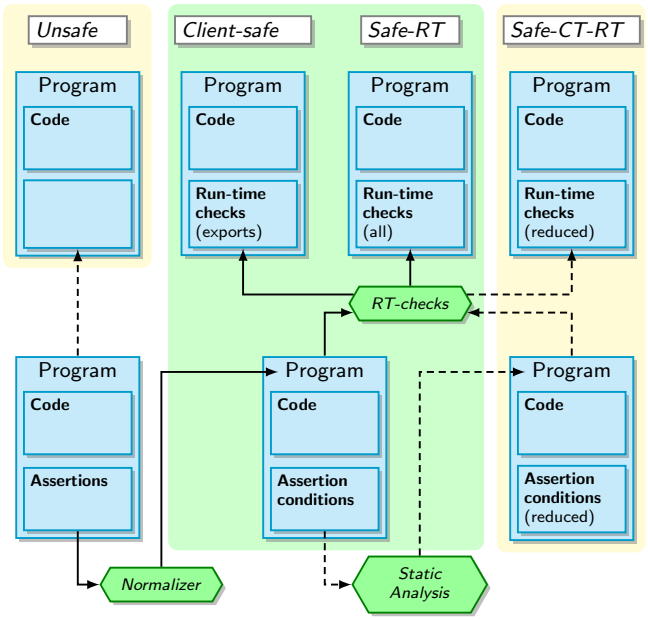

Figure 3: Source transformation differences per checking mode, including compile-time analysis.

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* $\gamma : D_\alpha \to 2^D$.[5] The operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) over abstract values 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 $\gamma(\bot) = \emptyset$ (and represents unreachable code), 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

[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.
[5] Strictly, only the concretization function is required.

is the Ciao PLAI abstract interpretation system [45–47]. Below we adapt some definitions and notation from [7] 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,[6] and a description $\mathcal{Q}_\alpha$ of the possible initial queries to the program, given as a set of *abstract queries*. Each such abstract query is a pair (L, λ), where L is an atom (for one of the exported predicates) and $\lambda \in D_\alpha$ an abstraction of a set of concrete initial program states (e.g., substitutions or constraints). Thus, a set of abstract queries $\mathcal{Q}_\alpha$ represents a set of concrete queries, denoted $\gamma(\mathcal{Q}_\alpha)$, which is defined as $\gamma(\mathcal{Q}_\alpha) = \{(L, \theta) \mid (L, \lambda) \in \mathcal{Q}_\alpha \wedge \theta \in \gamma(\lambda)\}$. The PLAI abstract interpretation process computes a set of (connected) 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 sets calls (entry) and success (exit) states for p such that λ^c safely approximates a set of call states at p and λ^s safely approximates the set of success states at p for all calls contained in λ^c. In what follows we will refer to such triplets also as *memo table entries*.[7]

The analysis (as the assertion language, to be introduced later) is designed to discern among the various usages of a predicate. Thus, multiple usages (contexts) of a procedure can result in multiple descriptions in the analysis output, i.e., for a given predicate p multiple $\langle L_p, \lambda^c, \lambda^s\rangle$ triples may be inferred. More precisely, the analysis is said to be *multivariant on calls* if more than one triple $\langle L_p, \lambda^c_1, \lambda^s_1\rangle, \ldots, \langle L_p, \lambda^c_n, \lambda^s_n\rangle$ $n \geq 0$ with $\lambda^c_i \neq \lambda^c_j$ for some i, j may be computed for the same predicate. Independently of the number of triples computed, the set of all λ^c_i together (i.e., the union of the concretizations of all the λ^c_i) safely approximate the set of possible concrete calls made to p during any program execution. In any case, for simplicity of presentation, we assume

[6] In fact, the analysis supports analysis using a number of different abstract domains, but, for simplicity, and without loss of generality –a set of abstract domains can always be encoded as a single domain– we use only one domain in the presentation.

[7] The analysis also provides information at body literals (also referred to as "program points"), as we will discuss in Section 4.2.

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 an 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 (for the given Q_α).
`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 (for the given Q_α).
`true`	static analyses	Such an assertion expresses (a part of) the actual semantics of the program, normally automatically inferred by analysis. In particular, each triple (memo table entry) $\langle L_p, \lambda^c, \lambda^s \rangle$ computed by the analysis is presented to the user by including a corresponding assertion of the form "`:- true pred` P `:` λ^c `=>` λ^s`.`" in the program.
`trust`	user	Provided by the user (or other tools) in order to guide analysis (increase precision).[8]

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 8).

[8] We will use only `true` assertions in the rest of the paper for simplicity.

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 [40] 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 $\mathcal{Q}_\alpha$[9] respectively, where $\gamma(\mathcal{Q}_\alpha) \supseteq \mathcal{Q}$, and $\langle L, \lambda^c, \lambda^s\rangle \in \mathit{Analysis}(P, \mathcal{Q}_\alpha, D_\alpha)$, the status of an assertion condition C, associated with it by the mapping $\mathsf{status}(c, \mathit{Status})$ where c is the corresponding identifier, is determined as follows:

- $C = c.\mathsf{calls}(L, \mathit{Precond}) \wedge \mathsf{status}(c, \texttt{checked})$ if $\lambda^c \sqsubseteq \lambda^-_{TS(\mathit{Precond},P)}$.
- $C = c.\mathsf{success}(L, \mathit{Pre}, \mathit{Post}) \wedge \mathsf{status}(c, \texttt{checked})$ if (1) $\lambda^c \sqcap \lambda^+_{TS(\mathit{Pre},P)} = \bot$ or (2) $\lambda^s \sqsubseteq \lambda^-_{TS(\mathit{Post},P)}$;

[9] 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.

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

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 $\mathcal{A}^{usr}_C(L) = \{C \mid (C = c.\mathsf{calls}(L, \_) \vee C = c.\mathsf{success}(L, \_, \_)) \wedge C \in \mathcal{A}_C(L) \wedge \mathsf{status}(c, S) \wedge S \in \{\texttt{check}, \texttt{true}\}\}$ given the analysis output of the form $\mathcal{A}^{ana}_C(L) = \{C \mid \forall C \text{ s.t. } (C = c.\mathsf{calls}(L, \_) \vee C = c.\mathsf{success}(L, \_, \_)) \wedge \mathsf{status}(c, \texttt{true})\}$ is rewritten into $\{C \mid (C = c.\mathsf{calls}(L, \_) \vee C = c.\mathsf{success}(L, \_, \_)) \wedge C \in \mathcal{A}^{usr}_C(L) \wedge \mathsf{status}(c, S) \wedge S \in \{\texttt{check}, \texttt{checked}, \texttt{false}\}\}$.

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).[10] 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:

- It is possible 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).

[10] Assertions with `true` and `trust` status (Table 1) 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.

- It is also possible 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).

```
:- module(_,[p/2]).        % p/2 is exported
:- use_module(lib,[e/2]). % e/2 is imported

p(X,Y) :- q(X,Y).


:- pred q(X,Y) : (int(X), X>3) => (int(Y), Y>0).

q(X,Y) :- r(X,Y).


:- pred r(X,Y) : (int(X), X>0) => (int(Y), Y>16).

r(X,Y) :- e(X,Y).
```

Figure 4: Example for analysis improvement.

As an example, consider the Ciao Prolog program of Figure 4.[11] There, `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` will always be called with their first argument `X` bound to an integer greater than 3, and that their second argument `Y` be bound to a positive integer upon success. Since any type of call is allowed to `p/2`, without information on the presence of run-time checks the 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, if we know that we will be generating run-time checks for those assertion conditions, the call to `r/2` in the body of `q/2` can only be

[11] In the examples we use just simple types (and in some cases constraints) as properties for simplicity of presentation, but even in this case please note that the use of types is *moded*, i.e., the assertions here express *states of instantiation*.

reached if the calls condition for `q/2` holds, i.e., if `X` is bound to an integer, and greater than 3 (since otherwise execution would have been aborted by the failing run-time check). 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 will be bound to a positive 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 and greater than 16. Thus, the success condition for `q/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:

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

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 which it is known that run-time checks will be generated. 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 $\mathsf{status}(c^l, \texttt{check})$) will still be checked in the version with run-time checks, but they will be assumed in p_{inner} (i.e., $c^i.\mathsf{calls}(L_{inner}, \_)$ with $\mathsf{status}(c^i, \texttt{true})$).

```
1  :- check calls   q(X,Y) : (int(X), X>3).
2  :- true  success q(X,Y) : (int(X), X>3) => (int(Y), Y>0).
3
4  q(X,Y) :- q_inner(X,Y).
5
6
7  :- true  calls   q_inner(X,Y) : (int(X), X>3).
8  :- check success q_inner(X,Y) : (int(X), X>3) => (int(Y), Y>0).
9
10 q_inner(X,Y) :- r(X,Y).
```

Figure 5: CTRT program transformation example (output).

For the success part the assertion conditions will still be checked for the inner predicate (i.e., c^i.success$(L_{inner}, _, _)$ with status$(c^i, \texttt{check})$) and the information will be assumed upon exiting p (i.e., c^i.success$(L, _, _)$ with status$(c^i, \texttt{true})$). The transformation guarantees that the same run-time 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 are 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.*

The proof of the lemma can be found in Appendix A.

6. Optimizing Checks at the Client-Library Boundaries

We now consider another aspect of our library scenario: optimizing the checks at the client-library boundaries. We will remain within the case in which the library provides an open interface to its clients, i.e., the clients are not known when analyzing and compiling the library, these clients can be expected to call the library in arbitrary ways, and we do not want the library to be reanalyzed or recompiled for each particular client. As seen in Section 3, in this scenario the reusability of the library forces us at least in principle to keep the run-time checks for the assertions at the library interface to ensure correctness. However, on the client side it may be possible to detect places where there is a call in the client module to a library predicate, such that the checks or analysis performed in the client module guarantee that the calls conditions of the library predicate will be satisfied. Detecting this could allow us to optimize away the checks at the client-library boundaries, and thus reduce run-time checking overhead.

Again, while inter-modular analysis could be used to this end, the advantage of fixing the library boundaries is that the library modules, once analyzed and compiled, can be reused without repeating the analysis or re-analyzing for new abstract call states. This reanalysis may not be really practical in the case of pre-compiled libraries, and also implies in any case additional cost, which may be prohibitive for some applications. Also, in inter-modular analysis and optimization the module boundaries change dynamically during analysis and this can happen after a change in any module. Another advantage of fixing the library boundaries is thus that it avoids having to recompile the client if there are changes in the library source code (and vice-versa), provided that the interface of the library itself is not changed. I.e., there are advantages to being able to fix the interface at certain boundaries.

The alternative that we propose is to provide a fixed interface, but one that provides two entry points for each predicate exported by the library: the standard one, that performs the run-time checks for the assertions in the library interface, and another one that provides direct access to the exported

```
:- module(mod,[p/2]).
:- use_module(lib,[e/2]).

:- pred p(X,Y) : int(X) => int(Y).
p(X,Y) :- e(X,Y).

:- pred q(X,Y) => int(Y).
q(X,Y) :- e(X,Y).
```

```
:- module(lib,[e/2]).

:- pred e(X,Y) : int(X) => int(Y).

e(X,Y) :- ...
```

Figure 6: A client-library program.

predicates bypassing the boundary assertion checks (in particular, the `_inner` versions produced by the CTRT transformation). We also propose a matching transformation for the client module that allows selecting, for each literal in the client that calls a library predicate, which of the two versions of that predicate exported by the library interface can safely be used.[12]

On the client side, we assume that the source code of the library predicates that are being imported by the client module is in general not accessible from the client during the analysis in the client. However, we assume that the interface of the library includes also the assertions of its exported predicates (as is the case in Ciao/CiaoPP). Thus, analysis on the client side has to rely solely on the information available in the interface of the library. This is not an issue however, if the library is compiled with the CTRT transformation, as in this case the transformation includes the assertions for the exported predicates (more specifically, the *link* clause assertions) in the library interface.

As an example, consider the client-library program in Figure 6 (using just moded types for brevity). There, in the client module, `mod`, `p/2` is an exported predicate and `q/2` is a local predicate, and `e/2` is imported from the library `lib`. We want to enforce through the assertions that `p/2` always be called with its

[12] This can obviously be generalized to providing several entry points under several conditions, but we will keep the discussion limited to two entry points per predicate for simplicity).

first argument `X` bound to an integer, and that its second argument `Y` be bound to an integer upon success (i.e., returning a free variable is not allowed). At the same time, we do not enforce any call-specific way to invoke `q/2`, and we enforce that its second argument `Y` should be bound to an integer upon success.

Both `p/2` and `q/2` call predicate `e/2`, imported from the library. Since `e/2` is an exported predicate in the `lib` module, the check for its calls condition (that its first argument `X` is bound to an integer) will always be performed. But notice that at the point where `e/2` is called from `p/2` the check for its first argument being an integer at run time has already taken place, as the same check was required by the calls conditions for the `p/2` predicate. This check duplication can be avoided if we replace at compile-time the call to `e/2` in the body of `p/2` with a call to `e_inner/2`, which is visible from `mod` during the pre-compilation analysis time. In principle this inner predicate would have to be exported but in practice it is done through the internal visibility mechanism in the compiler, which the user cannot bypass. At the same time we would like to keep the check for the calls condition of `e/2` when it is called from the body of `q/2`, as in that case nothing ensures that its first argument will be bound to an integer.

The optimization that we seek requires us to be able to reason about individual call sites in the bodies of the clauses in the program predicates, also referred to as "program points". For this, we need the analysis information (abstract states) to be available not just at the whole predicate level (call and success) but also at the level of the clause literals. This information is indeed provided by the PLAI analysis that we are using as reference (Section 4.1). We also need the interface of the transformed library to be extended by making accessible the *link* predicates generated for all its annotated exported predicates, together with their respective assertions. As mentioned before, such interface extension will provide us with (at least) two different versions of the library exported predicates, that can be called at different program points in the client. For this kind of reasoning we also require the static analysis performed to be in effect *multivariant* on calls.

Let `ppt` denote a program point identifier, which refers to a particular literal

```
:- module(mod,[p/2, q/2]).
:- use_module(lib,[e/2,e_inner/2]).

:- check calls   p(X,Y) : int(X).
:- true  success p(X,Y) : int(X) => int(Y).

p(X,Y) :- p_inner(X,Y).

:- true  calls   p_inner(X,Y) : int(X).
:- check success p_inner(X,Y) : int(X) => int(Y).

p_inner(X,Y) :- e_inner(X,Y).

:- check calls   q(X,Y) : term(X).
:- true  success q(X,Y) : term(X) => num(Y).

q(X,Y) :- q_inner(X,Y).

:- true  calls   q_inner(X,Y) : term(X).
:- check success q_inner(X,Y) : term(X) => num(Y).

q_inner(X,Y) :- e(X,Y).

```

```
:- module(lib,[e/2, e_inner/2]).

:- check calls   e(X,Y) : int(X).
:- true  success e(X,Y) : int(X) => int(Y).

e(X,Y) :- e_inner(X,Y).

:- true calls    e_inner(X,Y) : int(X).
:- check success e_inner(X,Y) : int(X) => int(Y).

e_inner(X,Y) :- ...

```

Figure 7: A two-module program after the transformations.

position in the body of a particular clause in the program. Let L^{ppt} denote the literal L that is located at program point ppt. We assume thus that the analysis provides the following information:

- The $\langle L_p, \lambda_i^c, \lambda_i^s \rangle$ triples for the predicates in the program, as before.
- In addition, triples $\langle L_p^{\mathsf{ppt}}, \lambda^c, \lambda^s \rangle$ that provide, for each literal L_p, the abstract state before and after the calls to such literal at each program point ppt in which L_p occurs in the body of a clause.

We further adapt our notation to *program point*-level reasoning as follows:

- Let $\mathsf{status}^{\mathsf{ppt}}(c, S)$ denote the status of some assertion condition $c.\mathsf{calls}(L, \_)$ or $c.\mathsf{success}(L, \_, \_)$ for the literal L at program point ppt.

Now with the information from the multivariant analysis and the statuses of assertions after the checking phase it is straightforward to apply a program-point literal substitution. Since we are considering programs that undergo the CTRT transformation by the time the static analysis and assertion checking are performed, L^{ppt} should be either L or L_{inner}, depending on the abstract state at the program point and the result of the program point assertion checks:

$$L^{\mathsf{ppt}} = \begin{cases} L_{inner} & \text{if } \forall c.C \in \mathcal{A}_C(L) \ s.t. \ C = c.\mathsf{calls}(L, \_) \\ & \quad \mathsf{status}^{\mathsf{ppt}}(c, \mathtt{checked}) \text{ holds} \\ L & \text{otherwise} \end{cases}$$

A result of such program transformation can be seen in Figure 7 for the program in Figure 6.

7. 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 and for the two scenarios.

7.1. 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 8. 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

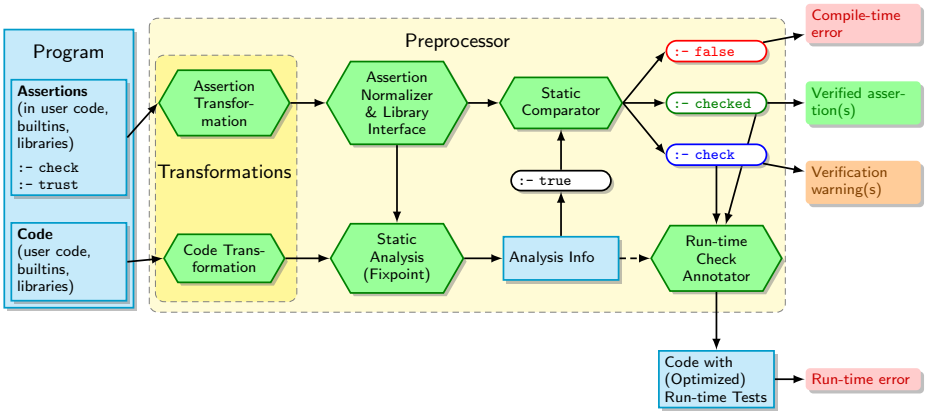

Figure 8: Adding the transformations to the Ciao Preprocessor.

exported by any libraries used. Any `check`, `trust`, or `true` assertions are normalized and the program is expanded to kernel form (simple Horn clauses), and the result is given as input to the *static analysis*.

We have introduced new front-end passes implementing the new transformations (marked in Figure 8) 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, as mentioned in Section 4. 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 for introducing the run-time checks are those of [23], with no caching.

7.2. Properties and Analysis Domains

In our experiments we consider several classes of properties, that are typically of interest to describe the intended semantics of (logic) programs:

- 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 ("pointer aliasing") patterns in order to be able to determine independence and transfer accurately grounding information ("strong update"). These properties are approximated safely and quite accurately using the CiaoPP *sharing and freeness* abstract domain [48].

- 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 CiaoPP *eterms* [49] abstract domain which infers safe approximations of these shapes as regular trees.

- The third class of properties that we consider refers to the numerical relations among program variables (constraints), in particular linear inequalities over real (floating point) numbers, which are useful to describe properties of numerical parts of programs. To this end we apply CiaoPP's *polyhedra* abstract domain, using the Parma Polyhedra Library (PPL) [50] as back-end solver.

Note that both the Ciao language of assertions and the analyzers in the system support a wide class of additional properties, including sized types, determinacy, non-failure, cardinality, constraints, size relations between variables, consumption of a variety of resources, etc. [9, 51]. However, we consider the three classes above a suitable study set for our experiments.[13]

[13] Clearly, these properties are more general and powerful than the traditional notions of types, modes, etc. Also, comparing to the *traditional* notion of type inference in statically-

7.3. 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.[14]

Given a concrete program, the CiaoPP assertion checking system checks the properties appearing in the assertions in the program and automatically chooses the appropriate abstract domains that have to be used during analysis on order to prove those properties [9]. In our experiments, however, in order to be able to study separately the impact on our proposals for different kinds of properties/domains, we have done the domain selection manually for each benchmark, as follows.

The benchmarks above the horizontal line in the table are symbolic and the properties of their predicates are more naturally expressed using the *eterms* and *sharing and freeness* abstract domains, The benchmarks in the second set are classical numerical benchmarks and their properties are more naturally expressed using the *polyhedra* abstract domain (as well as *sharing and freeness* for describing inputs/outputs and absence of sharing/pointer aliasing).

These benchmarks are relatively 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 data

typed languages, not only the notion of type is 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 and widening. 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. And of course, other properties like sized types, size relations between variables, or cost depart even further. See [52] for a further discussion of the very interesting topic of how to best straddle the dynamic vs. static language boundaries.

[14] Source available at https://cliplab.org/papers/optchk-scp2017/

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;
`ackerman`	Ackerman function computation;
`array`	a generic array API implementation;
`factA`	factorial with multiplication as addition in a loop;
`factM`	factorial with direct multiplication;
`incr`	variable increment;
`mc`	MacCarthy91 program;
`symstairs`	synchronous increment/decrement for two variables;

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 behavior. E.g., Figure 9 shows a fragment of the `fft` code. Table 3 presents some quantitative characteristics of

```
:- regtype complex/1. % A complex number
complex((A,B)) :- num(A), num(B).

:- pred complex_mul(A, B, C) % Multiplication
    : complex * complex * term
   => complex * complex * complex.
complex_mul((Ra,Ia), (Rb,Ib), (Rs,Is)) :-
    Rs is Ra*Rb-Ia*Ib,
    Is is Ra*Ib+Rb*Ia.
```

Figure 9: Complex number operations (fragment).

the benchmarks, such as lines of code (LOC), excluding empty and commented lines, size metrics of the benchmark object file after the compilation, and also the total number of program `pred` assertions. Regarding the sizes after the transformations, note that these transformations only add binary wrapper predicates that incur very little run-time overhead (since arguments do not change order the wrapper predicates translate to a single call instruction, with no argument overhead), so they do not significantly alter the benchmark metrics.

In order to measure the run-time overhead reduction in the client-library interaction scenario (i.e., measuring the gains from eliminating the redundant run-time checks on calls to the library predicates for the *Safe-CTRT* assertion– Section 6) we have adapted several of our benchmarks, splitting them into client and library parts. We have selected primarily those benchmarks where such separation is meaningful, i.e., where it is straightforward to identify a part of the benchmark module with a library-like structure that can be placed naturally in a separate module. As an example, we separated the `fft` benchmark into a library for arithmetic operations on complex numbers and the FFT calculations themselves as the client. We have also concentrated on benchmarks in which there are different call sites to the (now) library predicates, and where some of them required keeping the checks on the calls in the imported predicates and others did not, presenting thus good opportunity for study. Table 4 lists these benchmarks and the boundary at which the client-library split of each individual benchmark was performed.

Note that the *lists* library is listed as used in every benchmark of this client-

Table 3: Benchmark metrics.

Code			Assertions
Name	LOC	Size (KB)	total
boyer	853	70	13
boyerx	853	50	12
crypt	76	10	8
deriv	29	9	2
exp	28	6	3
factorial	13	4	2
fft	104	13	10
fib	11	5	3
guardians	78	9	7
hamming	71	9	10
hanoi	44	6	3
jugs	132	10	5
knights	49	9	7
mmatrix	48	6	4
nreverse	14	5	3
poly	81	12	7
primes	33	6	5
progeom	71	8	9
qsort	46	6	6
queens	47	6	7
serialize	81	10	6
tak	18	5	2
witt	651	50	43
ackerman	16	4	1
array	24	6	1
factA	15	4	1
factM	9	4	1
incr	9	4	2
mc	8	4	1
symstairs	15	4	1

Table 4: Benchmarks used for the client-library interaction use case.

fft	fast Fourier transformation calculation	lists, complex numbers
hamming	Hamming numbers calculation	lists, queues
hanoi	hanoi towers puzzle encoding	lists
nreverse	naive list reversal	lists
qsort	the quicksort algorithm	lists
witt	the WITT clustering algorithm	lists, sets

library interaction study subset. This is because this library provides some of the regular types that are used in the assertions of the client parts of the benchmarks.

7.4. Experimental Results (base scenario)

Tables 5 and 6 show the compilation time for the benchmarks under the different assertion checking modes that we have defined.[15] Note that the compilation time for the benchmarks under the *Safe-CT-RT* mode includes the total static analysis and assertion checking times. In all cases the compilation times include the cost of the proposed transformations, except in the unsafe mode, in which no transformations are performed and thus serves also as baseline. The experiments were run on a MacBook Pro with 2.6 GHz Intel Core i5 processor, 8GB RAM, and under the Mac OS X 10.12.5 operating system.

Tables 7 and 8 show more detail on the analysis and assertion checking times for the *Safe-CT-RT* mode for the different benchmarks. The *load* and *prep* columns indicate the time needed to load the source files and prepare the analyses, and the *shfr*, *eterms*, and *polyhedra* columns the time to perform sharing+freeness, shape (regular types), and numerical analyses, respectively.

The analyses are 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 the *eterms* abstract domain can be controlled in several ways within CiaoPP but we left the analyzer use the automatic, default settings for these experiments. Also note that more efficient –but less precise– domains are available to control analysis cost, many within CiaoPP, such as, for example, several widenings for sharing [53, 54], pair sharing domains [55, 56], or other type inference

[15] 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: Benchmarks: full compilation time (including *eterms* and *sharing and freeness* analysis, assertion checking, and transformations).

Benchmark	Compilation time, ms			
	Unsafe	Safe		
		Client	RT	CT+RT
boyer	242	1,271	1,444	469,807
boyerx	221	1,070	1,244	31,426
crypt	174	638	629	2,286
deriv	193	797	765	1,031
exp	167	741	734	1,075
factorial	148	686	1,006	865
fft	181	901	808	3,429
fib	157	608	722	933
guardians	169	673	736	1,580
hamming	187	852	1,085	1,987
hanoi	164	638	635	1,142
jugs	179	827	855	1,590
knights	162	852	974	1,751
mmatrix	174	825	722	1,085
nreverse	163	799	690	989
poly	181	941	909	2,156
primes	173	676	651	1,536
progeom	173	934	845	1,974
qsort	167	770	909	1,341
queens	169	821	1,037	1,405
serialize	167	849	822	1,636
tak	161	959	686	1,035
witt	281	1,866	1,938	180,353

Table 6: Benchmarks: full compilation time (including *polyhedra* analysis, assertion checking, and transformations).

Benchmark	Compilation time, ms			
	Unsafe	Safe		
		Client	RT	CT+RT
ackerman	183	651	460	523
array	159	580	568	730
factA	146	518	448	649
factM	162	505	525	601
incr	152	485	480	617
mc	149	505	643	632
symstairs	153	583	481	661

domains [57, 58].

Tables 9, 10, and 11 report on the actual execution times for each benchmark using the different assertion checking modes, together with data on the results of assertion checking. For some of the benchmarks, measurements were taken for calls with several input values and this is expressed using the notation $Name(Input)$. The 'Checked Assertion Conditions' column reports the ratios

Table 7: Static analysis time for benchmarks using the *Safe-CT-RT* checking mode with *eterms* and *sharing and freeness* analyses (part of total compilation time).

Benchmark	`load`	Analysis time, ms				Assertion checking
		prep	*shfr*	prep	*eterms*	
`boyer`	757.43	9.21	62.59	9.36	737.73	614.33
`boyerx`	686.38	6.40	53.63	6.68	556.77	408.75
`crypt`	528.04	1.43	8.46	1.41	39.37	138.05
`deriv`	478.51	1.04	4.32	1.46	17.36	32.74
`exp`	460.00	0.49	2.12	0.45	15.17	54.11
`factorial`	493.96	0.29	1.61	0.25	11.21	16.75
`fft`	515.95	1.84	9.54	1.91	43.05	162.43
`fib`	477.50	0.41	2.71	0.90	13.06	17.47
`guardians`	481.39	1.08	9.24	1.22	28.86	63.74
`hamming`	536.77	1.22	9.10	1.22	27.85	81.34
`hanoi`	477.13	0.59	2.91	0.47	15.65	21.79
`jugs`	494.40	1.02	5.08	1.19	26.75	126.10
`knights`	527.57	0.90	4.19	1.37	32.31	58.59
`mmatrix`	482.28	1.28	3.85	0.80	15.12	27.86
`nreverse`	524.33	0.50	3.23	0.30	3.50	9.01
`poly`	494.44	1.67	50.94	1.49	52.26	103.17
`primes`	527.36	0.80	2.64	0.55	17.34	33.09
`progeom`	481.82	1.30	7.18	1.00	27.66	59.31
`qsort`	496.13	1.00	5.77	0.64	8.23	22.18
`queens`	512.68	0.71	4.77	1.32	22.13	48.62
`serialize`	496.34	1.33	15.28	1.41	24.30	52.89
`tak`	519.57	0.44	1.75	0.43	13.76	26.67
`witt`	580.95	15.52	124,284.60	15.88	847.20	1,250.04

Table 8: Static analysis time for benchmarks using the *Safe-CT-RT* checking mode with *polyhedra* numerical analysis (part of total compilation time).

Benchmark	`load`	Analysis time, ms		Assertion checking
		prep	*polyhedra*	
`ackerman`	476.19	1.02	11.99	13.21
`array`	460.07	0.86	38.55	21.37
`factA`	451.58	1.01	33.67	13.43
`factM`	459.06	0.72	4.01	9.07
`incr`	448.13	0.79	5.19	6.96
`mc`	493.90	0.70	5.41	6.75
`symstairs`	460.86	0.79	7.34	10.12

of statically checked *calls* and *success* assertion conditions in the *Safe-CT-RT* checking mode to the total number of respective assertion conditions in the *Safe-RT* checking mode for each benchmark (i.e., N/M means that N out of the M assertion conditions are checked).

In the worst case the overhead in the *Safe-RT* checking mode is three 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

Table 9: Benchmark execution times under the different modes and checked vs. total assertions (all benchmarks).

Benchmark	Execution time, ms				Checked Assertion Conditions	
	Unsafe	Safe				
		Client	RT	CT+RT	calls	success
`boyer`	11.665	11.350	3,215.894	14.010	13/13	12/12
`boyerx`	17.541	17.755	2,621.203	1,254.041	11/12	10/11
`crypt`	0.106	0.118	6.601	0.114	7/8	8/8
`deriv`	0.013	0.062	4.629	0.071	1/2	1/1
`exp`	4.359	4.363	73.321	4.427	2/3	2/2
`factorial`	0.008	0.014	0.803	0.015	1/2	1/1
`fft`	28.419	32.702	32,112.845	254.773	9/10	8/9
`fib`	0.080	0.086	16.052	0.094	2/3	2/2
`guardians`	3.637	3.255	6,521.171	3.866	6/7	6/6
`hamming`	17.793	18.288	9,860.070	20.197	9/10	9/9
`hanoi` (8)	0.057	0.070	122.730	0.086	1/2	2/2
`jugs`	0.017	0.026	1.529	0.026	4/5	4/4
`knights`	232.922	232.940	18,842.485	250.993	6/7	6/6
`mmatrix` (4)	0.005	0.016	0.742	0.017	2/3	3/3
`nreverse`	2.438	2.699	10,596.668	3.640	2/3	2/2
`poly`	1.172	1.371	428.480	1.404	6/7	7/7
`primes`	0.033	0.044	11.066	0.040	3/4	4/4
`progeom` (8)	5.702	5.694	2,222.974	6.378	7/8	8/8
`qsort` (32)	0.022	0.030	7.382	0.035	4/5	3/3
`queens` (8)	2.522	2.527	545.413	2.846	5/6	4/4
`serialize` (25)	0.012	0.025	4.998	0.029	4/5	4/4
`tak`	2.980	2.991	980.910	3.457	1/2	1/1
`witt`	24.027	17.488	1,853.552	389.750	31/43	38/40

checks of complex properties like data shapes cannot be performed in constant time. The run-time checking process 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 (assuming a fixed implementation of the instrumentation), by removing all the internal checks, but keeping the library interface checks.

We can observe performance variations due to secondary effects (code layout, cache alignment), due to which sometimes the time in *Safe-CT-RT* mode can be slightly smaller than in *Client-safe* mode (`crypt`). To reduce the measurement noise (also influenced by the computations performed by other processes) we execute each benchmark several times and report the minimal time.[16]

[16] The current measurements depend on the C `getrusage()` function, that on Mac OS has

Table 10: Benchmark execution times under the different modes and checked vs. total assertions (benchmarks subset, varied output).

Benchmark	Execution time, ms				Checked Assertion Conditions	
	Unsafe	Safe				
		Client	RT	CT+RT	calls	success
`hanoi` (2)	0.000	0.012	0.161	0.013		
`hanoi` (4)	0.002	0.014	1.517	0.015	1/2	2/2
`hanoi` (8)	0.057	0.070	122.730	0.086		
`mmatrix` (2)	0.001	0.010	0.148	0.010		
`mmatrix` (3)	0.002	0.011	0.358	0.013	2/3	3/3
`mmatrix` (4)	0.005	0.016	0.742	0.017		
`progeom` (2)	0.002	0.005	0.615	0.005		
`progeom` (4)	0.096	0.098	28.118	0.111	7/8	8/8
`progeom` (8)	5.702	5.694	2,222.974	6.378		
`qsort` (8)	0.002	0.008	0.839	0.008		
`qsort` (16)	0.008	0.014	2.664	0.016	4/5	3/3
`qsort` (32)	0.022	0.030	7.382	0.035		
`queens` (4)	0.007	0.009	1.248	0.011		
`queens` (6)	0.133	0.136	29.527	0.153	5/6	4/4
`queens` (8)	2.522	2.527	545.413	2.846		
`serialize` (9)	0.004	0.008	0.881	0.011		
`serialize` (16)	0.006	0.013	2.343	0.014	4/5	4/4
`serialize` (25)	0.012	0.025	4.998	0.029		

In practice, in many programs *Safe-CT-RT* is able to remove most of the checks, except of course 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 uses instead a larger predicate that explicitly enumerates possible formula terms.

The benefits of applying the CTRT transformation are not so prominent in the case of numerical analysis, mainly due to the fact that the numerical checks are usually much less costly than the data shape checks. However, in programs that include arithmetic operations that are not captured well by the *polyhedra* abstract domain the overhead reduction is still noticeable (e.g., compare the running times of the `factA` and `factM` benchmarks, which differ only in the way they perform multiplication). Another challenge for the domain are complex

microsecond resolution.

Table 11: Benchmark (`polyhedra`) execution times under the different modes and checked vs. total assertions.

Benchmark	Execution time, ms				Checked Assertion Conditions	
	Unsafe	Safe				
		Client	RT	CT+RT	calls	success
`ackerman`	0.042	0.043	4.049	0.045	1/1	1/1
`array`	0.004	0.003	0.043	0.003	1/1	1/1
`factA`	0.002	0.008	0.032	0.018	0/1	1/1
`factM`	0.001	0.008	0.031	0.043	0/1	0/1
`incr`	0.001	0.007	0.128	0.032	1/2	2/2
`mc`	0.007	0.015	0.737	0.312	0/1	1/1
`symstairs`	0.006	0.012	0.601	0.309	0/1	1/1

Table 12: Benchmarks: full compilation time (client-library scenario).

Benchmark	Compilation time, ms		
	Client		Library
	Unoptimized	Optimized	
`fft`	3,253	3,385	1,674
`hamming`	1,565	1,542	1,463
`hanoi`	1,219	1,297	1,035
`nreverse`	1,040	1,203	1,089
`qsort`	1,377	1,223	1,105
`witt`	169,944	164,121	2,617

benchmarks like `ackerman` (double recursion) and `mc`.

7.5. Experimental Results (client-library scenario)

Table 12 shows the compilation time for the client-library scenario benchmarks from Table 4. As mentioned before, each of the benchmarks was split into client and library modules, and then two versions were generated of the client module: one without any optimization of the calls to the library and the other applying the program-point calls optimization ('Unoptimized' and 'Optimized' columns, respectively). All files were compiled in the *Safe-CT-RT* checking mode. One can notice that sum of the compilation times of client and libraries is proportional to the compilation time of the 'monolithic' version.

Table 13 provides the details for the analysis times of the client-library scenario benchmarks. The 'Part=C-u' rows report the analysis times for the client modules without optimizations of the calls to the library modules and the 'Part=C-o' ones report the times for the client modules with the optimized calls.

Table 13: Static analysis time for benchmarks (client-library scenario, part of total compilation time).

Benchmark	Part	load	Analysis time, ms				Assertion checking
			prep	`shfr`	prep	`eterms`	
`fft`	C-u	486.76	1.39	8.38	1.49	40.45	125.55
	C-o	518.09	1.33	8.24	1.31	38.07	121.75
	L	484.69	0.75	5.45	0.64	25.72	82.77
`hamming`	C-u	483.03	0.92	4.50	0.87	19.09	87.67
	C-o	512.81	1.08	5.11	0.89	19.86	53.74
	L	451.23	0.61	7.28	0.64	8.68	22.41
`hanoi`	C-u	470.81	0.45	2.36	0.34	11.66	23.17
	C-o	509.23	0.42	2.39	0.32	12.10	15.29
	L	454.93	0.37	2.00	0.31	3.30	11.86
`nreverse`	C-u	456.46	0.33	2.04	0.26	2.23	10.32
	C-o	492.82	0.50	2.88	0.32	3.25	7.27
	L	447.63	0.26	2.03	0.21	1.87	5.33
`qsort`	C-u	480.46	0.73	3.87	0.68	7.33	21.92
	C-o	498.62	0.52	3.61	0.51	7.11	16.74
	L	485.56	0.61	3.40	0.39	3.48	8.06
`witt`	C-u	505.32	12.32	118,807.15	14.41	722.83	1,491.66
	C-o	605.32	14.77	117,395.04	12.43	663.12	1,128.95
	L	488.70	2.08	64.39	2.21	41.74	88.74

The 'Part=L' rows provide the analysis times for the library modules. The sum of the analysis times of this client-library separated benchmark versions is comparable to the analysis time of the 'monolithic' benchmark versions reported above. The slight increase in the analysis time is expected, since processing a module and the modules at its interface takes some time.

The fact that the analysis times in the two-module scenario do not differ much from the analysis times of the 'monolithic' version of our benchmarks provides evidence supporting the scalability of the transformations that we have proposed, in the sense that, since changes in the client code do not affect the library part any more, only that part of the program will have to be recompiled should some changes be made. Even if the largest part of the cost is in the client (e.g., `witt`), note that the observation before is also true with respect to changes in the library, i.e., the client will not have to be reanalyzed for changes in the library.

The actual execution times for the benchmarks in the client-server scenario are given in Table 14. Here we are of course interested in the effect of the optimization of the checks at the module boundaries, i.e., in comparing the

Table 14: Benchmark execution times in the client-library scenario.

Benchmark	Execution time, ms	
	Unoptimized	Optimized
`fft`	2,199.29	271.78
`hamming`	146.85	60.47
`hanoi` (2)	0.03	0.01
`hanoi` (4)	0.16	0.02
`hanoi` (8)	3.21	0.10
`nreverse`	22.13	3.39
`qsort` (8)	0.08	0.01
`qsort` (16)	0.16	0.02
`qsort` (32)	0.32	0.04
`witt`	466.34	426.21

'Unoptimized' and 'Optimized' results. The results show that in the optimized case the execution times are reduced and comparable to those in the previous 'monolithic' setup (i.e., to the times in Tables 9, 10). The minor deviations from that case are due to the noise in the measurements and the use of additional predicate wrappers in the interface of the library (that was not present in the 'monolithic' versions). These wrappers are necessary to distinguish internal from external calls within the library. This effect can be observed in the execution times of the `hamming` benchmark: the current compilation mechanism introduces these wrapper predicates that add some overhead, and since in `hamming` the operations are very simple this overhead becomes noticeable. However, this overhead does not have a big impact in other benchmarks. In the case where we have not optimized the checks at the boundaries of the module (the 'Unoptimized' column) execution times are higher than in the 'monolithic' setup and are only superseded by the times with all run-time checks enabled (the *Safe-CT-RT* mode). These experiments clearly demonstrate the positive effect of eliminating run-time checks at module boundaries. It is quite interesting that we are able to achieve these performance gains without generating more versions or specializing the program (which is important in some contexts).

8. Conclusions

Our overall objective is to construct automatic verification and debugging systems for non-trivial properties, that can be used routinely as part of the development process for both prototyping and production code. Our concrete approach is the use of frameworks that combine static and dynamic verification, i.e., systems that combine compile-time and run-time checking of user-provided assertions. In this paper we have addressed the study of how run-time overhead can be reduced in different scenarios and, specially, via static analysis.

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, concentrating on the practical impact of such analysis in reducing the cost of the run-time checks required for the remaining parts of the specifications. We have also addressed the application of our approach when optimizing run-time checks for the calls across client-library boundaries. We have described a typical client-library use case and discussed the possibilities for optimizing the run-time checks in this context using an illustrative example. Also, we have proposes a new program point source transformation for avoiding the duplication of run-time checks.

We have also proposed program transformations that allow incorporating the run-time checking semantics into the analysis phase and demonstrated that this approach can increase analysis precision and allow for better and more fine-grained (program-point) check elimination.

Our experiments have shown that there is indeed a significant advantage in using analysis to reduce the overhead implied by the 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.

While evaluating the effectiveness of our assertion-based approach in finding errors in programs was not directly the objective of this paper (we concentrated here on measuring the reduction in run-time overhead due to analysis and the enhancements proposed), during our experiments a good number of program errors were flagged by the system. In particular, it is worth mentioning that the analysis of one of the more complex programs, `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 Horn clauses, and in particular 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.

References

[1] N. Stulova, J. F. Morales, M. V. Hermenegildo, Reducing the Overhead of Assertion Run-time Checks via static analysis, in: 18th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'16), ACM Press, 2016, pp. 90–103.

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

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

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

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

[6] M. Hermenegildo, G. Puebla, 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, D. S. Warren (Eds.), The Logic Programming Paradigm: a 25–Year Perspective, Springer-Verlag, 1999, pp. 161–192.

[7] G. Puebla, F. Bueno, M. Hermenegildo, Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs, in: Logic-based Program Synthesis and Transformation (LOPSTR'99), no. 1817 in LNCS, Springer-Verlag, 2000, pp. 273–292.

[8] C. Laï, Assertions with Constraints for CLP Debugging, in: P. Deransart, M. V. Hermenegildo, J. Maluszynski (Eds.), Analysis and Visualization Tools for Constraint Programming, Vol. 1870 of Lecture Notes in Computer Science, Springer, 2000, pp. 109–120.

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

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

[11] R. Cartwright, M. Fagan, Soft Typing, in: PLDI'91, SIGPLAN, ACM, 1991, pp. 278–292.

[12] R. B. Findler, M. Felleisen, Contracts for higher-order functions, in: M. Wand, S. L. P. Jones (Eds.), ICFP, ACM, 2002, pp. 48–59.

[13] S. Tobin-Hochstadt, M. Felleisen, The Design and Implementation of Typed Scheme, in: POPL, ACM, 2008, pp. 395–406.

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

[15] A. Rastogi, N. Swamy, C. Fournet, G. M. Bierman, P. Vekris, Safe & efficient gradual typing for typescript, in: S. K. Rajamani, D. Walker (Eds.), Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015, ACM, 2015, pp. 167–180. doi:10.1145/2676726.2676971.
URL http://doi.acm.org/10.1145/2676726.2676971

[16] A. Takikawa, D. Feltey, E. Dean, M. Flatt, R. B. Findler, S. Tobin-Hochstadt, M. Felleisen, Towards practical gradual typing, in: J. T. Boyland (Ed.), 29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic, Vol. 37 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 4–27. doi:10.4230/LIPIcs.ECOOP.2015.4.
URL http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.4

[17] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, M. Felleisen, Is sound gradual typing dead?, in: R. Bodík, R. Majumdar (Eds.), Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, ACM, 2016, pp. 456–468. doi:10.1145/2837614.2837630.
URL http://doi.acm.org/10.1145/2837614.2837630

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

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

[20] M. Fähndrich, F. Logozzo, Static contract checking with abstract interpretation, in: Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software, Vol. 6528 of FoVeOOS'10, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 10–30.
URL http://dl.acm.org/citation.cfm?id=1949303.1949305

[21] T. W. Schiller, K. Donohue, F. Coward, M. D. Ernst, Case studies and tools for contract specifications, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 596–607. doi:10.1145/2568225.2568285.
URL http://doi.acm.org/10.1145/2568225.2568285

[22] E. Koukoutos, V. Kuncak, Checking Data Structure Properties Orders of Magnitude Faster, in: B. Bonakdarpour, S. A. Smolka (Eds.), Runtime Verification, Vol. 8734 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 263–268. doi:10.1007/978-3-319-11164-3_22.
URL http://dx.doi.org/10.1007/978-3-319-11164-3_22

[23] N. Stulova, J. F. Morales, M. V. Hermenegildo, Practical Run-time Checking via Unobtrusive Property Caching, Theory and Practice of Logic Programming, 31st Int'l. Conference on Logic Programming (ICLP'15) Special Issue 15 (04-05) (2015) 726–741.
URL http://arxiv.org/abs/1507.05986

[24] P. Pietrzak, J. Correas, G. Puebla, M. Hermenegildo, Context-Sensitive

Multivariant Assertion Checking in Modular Programs, in: 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'06), no. 4246 in LNCS, Springer-Verlag, 2006, pp. 392–406.

[25] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. Morales, G. Puebla, An Overview of Ciao and its Design Philosophy, Theory and Practice of Logic Programming 12 (1–2) (2012) 219–252, http://arxiv.org/abs/1102.5497. doi:doi:10.1017/S1471068411000457.

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

[27] M. Méndez-Lojo, J. Navas, 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), no. 4915 in Lecture Notes in Computer Science, Springer-Verlag, 2007, pp. 154–168.

[28] J. Navas, M. Méndez-Lojo, M. Hermenegildo, Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications, in: The Sixth NASA Langley Formal Methods Workshop (LFM 08), 2008, pp. 29–32, extended Abstract.

[29] J. Navas, M. Méndez-Lojo, 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), Vol. 253 of Electronic Notes in Theoretical Computer Science, Elsevier - North Holland, 2009, pp. 65–82.

[30] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. Hermenegildo, K. Eder, Energy Consumption Analysis of Programs

based on XMOS ISA-level Models, in: G. Gupta, R. Peña (Eds.), Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Revised Selected Papers, Vol. 8901 of Lecture Notes in Computer Science, Springer, 2014, pp. 72–90. doi:10.1007/978-3-319-14125-1_5.
URL http://dx.doi.org/10.1007/978-3-319-14125-1_5

[31] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, K. Eder, Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR, in: M. V. Eekelen, U. D. Lago (Eds.), Foundational and Practical Aspects of Resource Analysis: 4th International Workshop, FOPARA 2015, London, UK, April 11, 2015. Revised Selected Papers, Vol. 9964 of Lecture Notes in Computer Science, Springer, 2016, pp. 81–100. arXiv:1511.01413, doi:10.1007/978-3-319-46559-3_5.
URL http://dx.doi.org/10.1007/978-3-319-46559-3_5

[32] G. Gange, J. A. Navas, P. Schachte, H. S, P. J. Stuckey, Horn Clauses as an Intermediate Representation for Program Analysis and Transformation, TPLP 15 (2015) 526–542.

[33] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, A. Rybalchenko, HSF(C): A Software Verifier Based on Horn Clauses - (Competition Contribution), in: C. Flanagan, B. König (Eds.), TACAS, Vol. 7214 of LNCS, Springer, 2012, pp. 549–551.

[34] A. Gurfinkel, T. Kahsai, A. Komuravelli, J. A. Navas, The SeaHorn Verification Framework, in: D. Kroening, C. S. Pasareanu (Eds.), Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I, Vol. 9206 of Lecture Notes in Computer Science, Springer, 2015, pp. 343–361. doi:10.1007/978-3-319-21690-4_20.
URL http://dx.doi.org/10.1007/978-3-319-21690-4_20

[35] M. Madsen, M. Yee, O. Lhoták, From Datalog to FLIX: a declarative language for fixed points on lattices, in: C. Krintz, E. Berger (Eds.), Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016, ACM, 2016, pp. 194–208. doi:10.1145/2908080.2908096.
URL http://doi.acm.org/10.1145/2908080.2908096

[36] N. Bjørner, F. Fioravanti, A. Rybalchenko, V. Senni (Eds.), Workshop on Horn Clauses for Verification and Synthesis, 2014, to appear in Electronic Proceedings in Theoretical Computer Science.
URL http://vsl2014.at/meetings/HCVS-index.html

[37] E. De Angelis, F. Fioravanti, A. Pettorossi, M. Proietti, Semantics-based Generation of Verification Conditions via Program Specialization, Science of Computer Programmingdoi:http://dx.doi.org/10.1016/j.scico.2016.11.002.
URL http://www.sciencedirect.com/science/article/pii/S016764231630199X

[38] J. Giesl, R. Thiemann, P. Schneider-Kamp, S. Falke, Automated termination proofs with aprove, in: Proc. International Conference on Rewriting Techniques and Applications (RTA), Aachen, Germany, 2004, pp. 210–220.

[39] E. Bodden, P. Lam, L. Hendren, Partially Evaluating Finite-state Runtime Monitors Ahead of Time, ACM Transactions on Programming Languages and Systems (TOPLAS) 34 (2) (2012) 7:1–7:52. doi:10.1145/2220365.2220366.
URL http://www.bodden.de/pubs/blh12partially.pdf

[40] N. Stulova, J. F. Morales, 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, 2014.

[41] G. Puebla, F. Bueno, M. Hermenegildo, An Assertion Language for Constraint Logic Programs, in: P. Deransart, M. Hermenegildo, J. Maluszynski (Eds.), Analysis and Visualization Tools for Constraint Programming, no. 1870 in LNCS, Springer-Verlag, 2000, pp. 23–61.

[42] G. Puebla, J. Correas, M. Hermenegildo, F. Bueno, M. García de la Banda, K. Marriott, P. J. Stuckey, A Generic Framework for Context-Sensitive Analysis of Modular Programs, in: M. Bruynooghe, K. Lau (Eds.), Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development, no. 3049 in LNCS, Springer-Verlag, Heidelberg, Germany, 2004, pp. 234–261.

[43] J. Correas, G. Puebla, M. Hermenegildo, F. Bueno, Experiments in Context-Sensitive Analysis of Modular Programs, in: 15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05), no. 3901 in LNCS, Springer-Verlag, 2006, pp. 163–178.

[44] P. Cousot, 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.

[45] K. Muthukumar, M. Hermenegildo, Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation, in: 1989 North American Conference on Logic Programming, MIT Press, 1989, pp. 166–189.

[46] K. Muthukumar, 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).
URL ftp://cliplab.org/pub/papers/tr153-90.mcc.ps.Z

[47] K. Muthukumar, M. Hermenegildo, Compile-time Derivation of Variable

Dependency Using Abstract Interpretation, Journal of Logic Programming 13 (2/3) (1992) 315–347.

[48] K. Muthukumar, M. Hermenegildo, Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation, in: International Conference on Logic Programming (ICLP 1991), MIT Press, 1991, pp. 49–63.

[49] C. Vaucheret, F. Bueno, More Precise yet Efficient Type Inference for Logic Programs, in: International Static Analysis Symposium, Vol. 2477 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 102–116.

[50] R. Bagnara, P. M. Hill, E. Zaffanella, The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems, Science of Computer Programming 72 (1–2).
URL http://www.cs.unipr.it/ppl/Documentation/BagnaraHZ08SCP.pdf

[51] A. Serrano, P. Lopez-Garcia, F. Bueno, M. Hermenegildo, Sized Type Analysis for Logic Programs (technical communication), in: T. Swift, E. Lamma (Eds.), Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement, Vol. 13, Cambridge U. Press, 2013, pp. 1–14.

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

[53] J. Navas, F. Bueno, M. Hermenegildo, Efficient top-down set-sharing analysis using cliques, in: Eight International Symposium on Practical Aspects of Declarative Languages, no. 2819 in LNCS, Springer-Verlag, 2006, pp. 183–198.

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

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

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

[57] J. Gallagher, D. de Waal, Fast and precise regular approximations of logic programs, in: P. Van Hentenryck (Ed.), Proc. of the 11th International Conference on Logic Programming, MIT Press, 1994, pp. 599–613.

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

Appendix A. Proof of Lemma 5

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

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

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

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

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

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

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

The $S_0 \rightsquigarrow_{\mathcal{A}} \langle L_{inner} :: G \mid \theta \mid \varnothing \rangle \rightsquigarrow_{\mathcal{A}} \langle L_{inner} \mid \theta \mid \{\bar{c}^i\} \rangle$ 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 $\mathsf{status}(c^i, \mathtt{true})$.

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

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

$S_0 \rightsquigarrow_{\mathcal{A}} \langle B :: \mathsf{acheck}(L, c) :: G \mid \theta \mid \varnothing \rangle \rightsquigarrow^*_{\mathcal{A}} \langle G \mid \theta \mid \varnothing \rangle = S_{succ}$
if $\theta \Rightarrow_P Post$

$S_0 \rightsquigarrow_{\mathcal{A}} \langle B :: \mathsf{acheck}(L, c) :: G \mid \theta \mid \varnothing \rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{acheck}(L, c) \mid \theta \mid \{\bar{c}\} \rangle = S_{err}$
if $\theta \not\Rightarrow_P Post$

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

sequences from the S_0 state now are:

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

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

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 $\mathsf{status}(c^l, \texttt{true})$. □

List of Updates with respect to the First Revision

Following the recommendations of the reviewers and the editor we have improved the experimental evaluation and the comparison to related work, as well as addressing the other comments and suggestions for improvement of the reviewers.

More concretely, we have included the following new material w.r.t. the first (journal) version of the paper:

- Regarding the *improvement of the experimental evaluation*:
  - (page 30) We have added a third analysis domain, *polyhedra* (in addition to *eterms* and *sharing+freenes*), and studied it experimentally alongside the other two domains w.r.t. the proposed compilation modes. This provides additional experimental data and also illustrates further that the Ciao assertion model and its implementation in CiaoPP is far more general than traditional modes+types, following the suggestions of the reviewers.
  - (page 32) We have also added new benchmarks, mostly of numeric nature and stemming from imperative programs. Apart from extending the benchmark set, the intention is to illustrate the wider applicability of the method and also to include programs where the new polyhedra domain is essential. The sources of the new benchmarks have been made available on-line alongside the previous ones.
  - (page 32) A better justification of the choices made in benchmark selection has also been provided.
  - (page 33) We have revised the metrics used for describing the benchmarks characteristics and code (the tables describing size, number of assertions and percentage checked, etc.) to make them easier to understand and reproduce, following the reviewer suggestions.
  - (page 39) We have also improved the explanations of the measurements and timing techniques used to produce the results in the tables.

  - The tables with the experimental results have been reorganized and moved to the respective subsections, following the reviewer suggestions.

- Regarding the improvement of the discussion of related work on combining analysis and assertion checking we have added all the additional references suggested by the reviewers and compared our work to them (pages 4, 5), as suggested by the reviewers.

- Finally, (page 54) the proof of lemma 5 has been moved to an appendix, as suggested by reviewers.

Original List of Updates with respect to the Conference Version

Already in the first round we included the following new material w.r.t the conference proceedings version of this paper:

- We have added a new section (Section 6) discussing the application of our approach when optimizing run-time checks for the calls across client-library boundaries. We describe a typical client-library use case and discuss the possibilities for optimizing the run-time checks in this context using an illustrative example.
- Also, we propose there a new program point source transformation for avoiding the duplication of run-time checks. The approach proposed uses a different level of granularity in the analysis information (information at program points).
- In line with this, in the experimental section we have added a new set of benchmarks for evaluating the impact of optimizing run-time checks for the calls across client-library boundaries (see Table 4). We have evaluated experimentally the new transformation for these benchmarks (Section 7.5) and provided the results for the compilation (see Table 12), analysis (see Table 13), and running times (see Table 14). We have also added a discussion of these results.

Other changes:

- We have clarified better the different scenarios and the relation with other possible approaches (such as all-out inter-modular analysis).
- We have improved the discussion of the experimental results.
- We have clarified further the status of assertions, specially the relation between true assertions and the analysis memo tables.
- We have updated the abstract and conclusions to reflect the new contributions.

- We have added more references, including to work in supporting analysis and verification of imperative programs via translation to Horn clauses.
- We have introduced many minor clarifications, improvements in wording, and fixes.