Multivariant Assertion-based Guidance in Abstract Interpretation*

Isabel Garcia-Contreras[1,2], Jose F. Morales[1], and Manuel V. Hermenegildo[1,2]

[1] IMDEA Software Institute
[2] Universidad Politécnica de Madrid (UPM)

Abstract. Approximations during program analysis are a necessary evil, as they ensure essential properties, such as soundness and termination of the analysis, but they also imply not always producing useful results. Automatic techniques have been studied to prevent precision loss, typically at the expense of larger resource consumption. In both cases (i.e., when analysis produces inaccurate results and when resource consumption is too high), it is necessary to have some means for users to provide information to guide analysis and thus improve precision and/or performance. We present techniques for supporting within an abstract interpretation framework a rich set of assertions that can deal with multivariance/context-sensitivity, and can handle different run-time semantics for those assertions that cannot be discharged at compile time. We show how the proposed approach can be applied to both improving precision and accelerating analysis. We also provide some formal results on the effects of such assertions on the analysis results.

Keywords: Program Analysis · Multivariance · Context Sensitivity · Abstract Interpretation · Assertions · Static Analysis · User Guidance

1 Introduction

Abstract Interpretation [6] is a well-established technique for performing static analyses to determine properties of programs. It allows inferring at compile-time and in finite time information that is guaranteed to hold for all program executions corresponding to all possible sets of inputs to the program. Reasoning about these generally infinite sets of inputs and program paths requires *(safe) approximations* –computing over *abstract domains*– to ensure termination and soundness. If such approximations are not carefully designed, the information reported by the analyzer may not be accurate enough to be useful for the intended application, such as, e.g., performing optimizations or verifying properties. Similarly, although abstract interpretation-based analyzers are guaranteed to terminate, this does not necessarily imply that they do so in acceptable time or space, i.e., their resource usage may be higher than desirable.

* Research partially funded by Spanish MINECO grant TIN2015-67522-C3-1-R *TRACES*, the Madrid M141047003 *N-GREENS* program, and Spanish MECD grant FPU16/04811. We thank the anonymous reviewers for their useful comments.

Much work has been done towards improving both the accuracy and efficiency of analyzers through the design of automatic analysis techniques that include clever abstract domains, widening and narrowing techniques [1, 27, 28], and sophisticated fixpoint algorithms [3, 20, 25**?**]. Despite these advances, there are still cases where it is necessary for the user to provide input to the analyzer to guide the process in order to regain accuracy, prevent imprecision from propagating, and improve analyzer performance [5, 8]. Interestingly, there is comparatively little information on these aspects of analyzers, perhaps because they are perceived as internal or analyzer implementation-specific.

In this paper we focus on techniques that provide a means for the programmer to be able to optionally annotate program parts in which precision needs to be recovered. Examples are the *entry* and *trust* declarations of CiaoPP [5, 23] and the *known facts* of Astrée [7, 8] (see Sect. 6 for more related work). Such user annotations allow dealing with program constructs for which the analysis is not complete or the source is only partially available. However, as mentioned before, there is little information in the literature on these assertions beyond a sentence or two in the user manuals or some examples of use in demo sessions. In particular, no precise descriptions exist on how these assertions affect the analysis process and its results.

We clarify these points by proposing a user-guided multivariant fixpoint algorithm that makes use of information contained in different kinds of assertions, and provide formal results on the influence of such assertions on the analysis. We also extend the semantics of the assertions to control if precision can be relaxed, and also to deal with both the cases in which the program execution will and will not incorporate run-time tests for unverified assertions. Note that almost all current abstract interpretation systems assume in their semantics that the run-time checks will be run. However, due to efficiency considerations, assertion checking in often turned off in production code, specially for complex properties [15]. To the best of our knowledge this is the first precise description of how such annotations are processed within a standard parametric and multivariant fixpoint algorithm, and of their effects on analysis results.

2 Preliminaries

Program Analysis with Abstract Interpretation. Our approach is based on *abstract interpretation* [6], a technique in which execution of the program is simulated on an *abstract domain* (D_α) which is simpler than the actual, *concrete domain* (D). Although not strictly required, we assume that D_α has a lattice structure with meet ($\sqcap$), join ($\sqcup$), and less than ($\sqsubseteq$) operators. Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : D \to D_\alpha$, and *concretization* $\gamma : D_\alpha \to D$, which form a Galois connection. A description (or abstract value) $d \in D_\alpha$ *approximates* a concrete value $c \in D$ if $\alpha(c) \sqsubseteq d$ where $\sqsubseteq$ is the partial ordering on D_α. Concrete operations on D values are (over-)approximated by corresponding abstract operations on D_α values. The key result for abstract interpretation is that it guarantees that the analysis terminates, provided that D_α meets some conditions (such as finite ascending chains) and that the results are safe approximations of the

concrete semantics (provided D_α safely approximates the concrete values and operations).

Intermediate Representation. For generality, we formulate our analysis to work on a block-level intermediate representation of the program, encoded using Constrained Horn clauses (CHC). A *definite CHC program*, or *program*, is a finite sequence of clauses. A *clause* is of the form H `:-` $B_1, \ldots, B_n$ where H, the *head*, is an atom, and $B_1, \ldots, B_n$ is the *body*, a possibly empty finite conjunction of atoms. Atoms are also called *literals*. We will refer to the head and the body of a clause `cl` with `cl.head` and `cl.body` respectively. An *atom* is of the form $p(V_1, \ldots, V_n)$. It is *normalized* if the $V_1, \ldots, V_n$ are all distinct variables. Normalized atoms are also called *predicate descriptor*s. Each maximal set of clauses in the program with the same descriptor as head (modulo variable renaming) defines a *predicate* (or *procedure*). Body literals can be predicate descriptors, which represent *calls* to the corresponding predicates, or *constraints*. A *constraint* is a finite conjunction of built-in relations for some background theory. We assume that all non-builtin atoms are normalized. This is not restrictive since programs can always be put in this form, and it simplifies the presentation of the algorithm. However, in the examples we use non-normalized programs. The encoding of program semantics in CHC depends on the source language and is beyond the scope of the paper. It is trivial for (C)LP programs, and also well studied for several types of imperative programs and compilation levels (e.g., bytecode, llvm-IR, or ISA – see [2, 9, 11, 12, 16, 19, 22]).

Concrete Semantics. The concrete semantics that we abstract is that of Constraint Logic Programs – (C)LP [18]. In particular, we use the constraint extension of top-down, left-to-right SLD-resolution, which, given a *query* (*initial state*), returns the answers (*exit states*) computed for it by the program. A *query* is a pair $G : \theta$ with G a (non-empty) conjunction of atoms and θ a constraint. Executing (answering) a query with respect to a CHC program consists on determining whether the query is a logical consequence of the program and for which constraints (answers). However, since we are interested in abstracting the calls and answers (states) that occur at different points in the program, we base our semantics on the well-known notion of generalized AND trees [4]. The concrete semantics of a program P for a given set of queries $\mathcal{Q}$, $[\![P]\!]_\mathcal{Q}$, is then the set of generalized AND trees that result from the execution of the queries in $\mathcal{Q}$ for P. Each node $\langle G, \theta^c, \theta^s \rangle$ in the generalized AND tree represents a call to a predicate G (an atom), with the constraint (state) for that call, θ^c, and the corresponding success constraint θ^s (answer). The *calling_context*$(G, P, \mathcal{Q})$ of a predicate given by the predicate descriptor G defined in P for a set of queries $\mathcal{Q}$ is the set $\{\theta^c \mid \exists T \in [\![P]\!]_\mathcal{Q}\ s.t.\ \exists \langle G', \theta'^c, \theta'^s \rangle\ in\ T \wedge \exists \sigma, \sigma(G') = G, \sigma(\theta'^c) = \theta^c\}$, where σ is a *renaming* substitution, i.e., a substitution that replaces each variable in the term it is applied to with distinct, fresh variables. We use $\sigma(X)$ to denote the application of σ to X. We denote by *answers*$(P, \mathcal{Q})$ the set of success constraints computed by P for queries $\mathcal{Q}$.

Goal-dependent abstract interpretation. We use goal-dependent abstract interpretation, in particular a simplified version (PLAI-simp) of the PLAI algorithm [20, 21], which is essentially an efficient abstraction of the generalized AND

trees semantics, parametric on the abstract domain. It takes as input a program P, an abstract domain D_α, and a set of abstract initial queries $\mathcal{Q}_\alpha = \{G_i{:}\lambda_i\}$, where G_i is a normalized atom, and $\lambda_i \in D_\alpha$ is abstract constraint. The algorithm computes a set of triples $A = \{\langle G_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle G_n, \lambda_n^c, \lambda_n^s\rangle\}$. In each $\langle G_i, \lambda_i^c, \lambda_i^s\rangle$ triple, G_i is a normalized atom, and λ_i^c and λ_i^s, elements of D_α, are, respectively, abstract call and success constraints. The set of triples for a predicate cover all the concrete call and success constraints that appear during execution of the initial queries from $\gamma(Q_\alpha)$, see Def. 2.

As usual, $\bot$ denotes the abstract constraint such that $\gamma(\bot) = \emptyset$. A tuple $\langle G_j, \lambda_j^c, \bot\rangle$ indicates that all calls to predicate G_j with any constraint $\theta \in \gamma(\lambda_j^c)$ either fail or loop, i.e., they do not produce any success constraints. A represents the (possibly infinite) set of nodes of the generalized AND trees for the queries represented in $\mathcal{Q}_\alpha$ to P. In addition, A is multivariant on calls, namely, it may contain more than one triple for the same predicate descriptor G with different abstract call constraints. The PLAI algorithm provides guarantees on termination and correctness (see Thm. 1 for a more precise formulation).

Assertions. Assertions allow stating conditions on the state (current constraint store) that hold or must hold at certain points of program execution. We use for concreteness a subset of the syntax of the `pred` assertions of [5, 14, 23], which allow describing sets of *preconditions* and *conditional postconditions* on the state for a given predicate. These assertions are instrumental for many purposes, e.g. expressing the results of analysis, providing specifications, and documenting [13, 14, 24]. A `pred` assertion is of the form:

`:-` [*Status*] `pred` *Head* [`:` *Pre*] [`=>` *Post*]`.`

where *Head* is a predicate descriptor (i.e., a normalized atom) that denotes the predicate that the assertion applies to, and *Pre* and *Post* are conjunctions of *property literals*, i.e., literals corresponding to predicates meeting certain conditions which make them amenable to checking, such as being decidable for any input [23]. *Pre* expresses properties that hold when *Head* is called, namely, at least one *Pre* must hold for each call to *Head*. *Post* states properties that hold if *Head* is called in a state compatible with *Pre* and the call succeeds. Both *Pre* and *Post* can be empty conjunctions (meaning true), and in that case they can be omitted. *Status* is a qualifier of the meaning of the assertion. Here we consider: `trust`, the assertion represents an actual behavior of the predicate that the analyzer will assume to be correct; `check`, the assertion expresses properties that must hold at run-time, i.e., that the analyzer should prove or else generate run-time checks for (we will return to this in Sect. 4). `check` is the default status of assertions.

Example 1. The following assertions describe different behaviors of the `pow` predicate that computes $\mathtt{P} = \mathtt{X}^{\mathtt{N}}$: `(1)` is stating that if the exponent of a power is an even number, the result (`P`) is non-negative, `(2)` states that if the base is a non-negative number and the exponent is a natural number the result `P` also is non-negative:

```
1 :- pred pow(X,N,P) : (int(X), even(N)) => P ≥ 0.   % (1)
2 :- pred pow(X,N,P) : (X ≥ 0, nat(N))   => P ≥ 0.   % (2)
3 pow(_, 0, 1).
4 pow(X, N, P) :- N > 0,
5     N1 is N - 1, pow(X, N1, P0), P is X * P0.
```

Here, for simplicity we assume that the properties `even/1`, `int/1`, `nat/1`, and $\geq$ are *built-in properties* handled natively by the abstract domain.

In addition to predicate assertions we also consider *program-point assertions*. They can appear in the places in a program in which a literal (statement) can be added and are expressed using literals corresponding to their *Status*, i.e., `trust`(*Cond*) and `check`(*Cond*). They imply that whenever the execution reaches a state originated at the program point in which the assertion appears, *Cond* (should) hold. Example 2 illustrates their use. Program-point assertions can be translated to `pred` assertions,[3] so without loss of generality we will limit the discussion to `pred` assertions.

Definition 1 (Meaning of a Set of Assertions for a Predicate). *Given a predicate represented by a normalized atom Head, and a corresponding set of assertions* $\{a_1 \ldots a_n\}$*, with* $a_i =$ "`:- pred` *Head* `:` Pre_i `=>` $Post_i$`.`" *the set of* assertion conditions *for Head is* $\{C_0, C_1, \ldots, C_n\}$*, with:*

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

where `calls`$(Head, Pre)$[4] states conditions on all concrete calls to the predicate described by *Head*, and `success`$(Head, Pre_j, Post_j)$ describes conditions on the success constraints produced by calls to *Head* if Pre_j is satisfied.

The assertion conditions for the assertions in Example 1 are:

$$\left\{ \begin{array}{ll} \texttt{calls}(& pow(X,N,P), ((int(X), even(N)) \vee (X \geq 0, nat(N)))), \\ \texttt{success}(& pow(X,N,P), (int(X), even(N)), \qquad (P \geq 0)), \\ \texttt{success}(& pow(X,N,P), (X \geq 0, nat(N)), \qquad (P \geq 0)) \end{array} \right\}$$

Uses of assertions. We show examples of the use assertions to guide analysis.

Example 2. Regaining precision during analysis. If we analyze the following program with a simple (non-relational) intervals domain, the information inferred for `Z` would be "any integer" (line 3), whereas it can be seen that it is $Z = 2$ for any `X` and `Y`. We provide the information to the analyzer with an assertion (line 4). The analyzer will *trust* this information even if it cannot be inferred with this domain (because it cannot represent relations between variables).

[3] E.g., we can replace line 4 in Example 2, by "`assrt_aux(Z),`", and add a predicate to the program, `assrt_aux(_).`, with an assertion "`:- pred assrt_aux(Z) : Z = 2.`".

[4] We denote the calling conditions with `calls` (plural) for historic reasons, and to avoid confusion with the higher order predicate in Prolog `call/2`.

```
p(Y) :-           % (Y > 0)
    X is Y + 2,   % (X > 2,  Y > 0)
    Z is X - Y,   % (int(Z), X > 2, Y > 0)
    trust(Z = 2), % (Z = 2,  X > 2, Y > 0)
    % implementation continues
```

Example 3. Speeding up analysis. Very precise domains suffer less from loss of precision and are useful for proving complex properties, but can be very costly. In some cases less precise information in enough, e.g., this code extracted from LPdoc, the Ciao documentation generator, `html_escape` is a predicate that takes a string of characters and transforms it to html:

```
:- trust pred html_escape(S0, S) => (string(S0), string(S)).
html_escape("‘"||S0, "&ldquo;"||S) :- !, html_escape(S0, S).
html_escape("’"||S0, "&rdquo;"||S) :- !, html_escape(S0, S).
html_escape([34|S0],  "&quot;"||S)  :- !, html_escape(S0, S).
html_escape([39|S0],  "&apos;"||S)  :- !, html_escape(S0, S).
% ...
html_escape([X|S0], [X|S])          :- !, character_code(X), html_escape(S0, S).
html_escape([],[]).

% string(Str) :- list(Str, int).
```

Analyses based on regular term languages, as, e.g. `eterms` [27] infer precise regular types with subtyping, which is often costly. In this example it would be equivalent to computing an accurate regular language that over-approximates the HTML text encoding. The `trust` assertion provides a general invariant that the analyzer will take instead of inferring a more complex type.

Example 4. Defining abstract usage or specifications of libraries or dynamic predicates. When sources are not available, or cannot be analyzed, assertions can provide the missing abstract semantics. The following code illustrate the use of an assertion to describe the behavior of predicate `receive` in a `sockets` library that is written in C. The assertion in this case transcribes what is stated in natural language in the documentation of the library. Note that if no annotations were made, the analyzer would have to assume the most general abstraction ($\top$) for the library arguments.

```
:- module(sockets, []).

:- export(receive/2).
:- pred receive(S, M) : (socket(S), var(M)) => list(M, utf8).
% receive is written in C
```

Example 5. (Re)defining the language semantics for abstract domains. `trust` assertions are also a useful tool for defining the meaning (transfer function) of the basic operations of the language. In this example we define some basic properties of the product predicate in a simple types-style abstract domain:

```
:- trust pred '*'(A, B, C) : (int(A), int(B)) => int(C).
:- trust pred '*'(A, B, C) : (flt(A), int(B)) => flt(C).
:- trust pred '*'(A, B, C) : (int(A), flt(B)) => flt(C).
:- trust pred '*'(A, B, C) : (flt(A), flt(B)) => flt(C).
```

The semantics of bytecodes or machine instructions can be specified for each domain after transformation into CHCs. Assertions allow representing behaviors for the same predicate for different call descriptions (multivariance).

3 Basic fixpoint algorithm

We first present a basic, non-guided algorithm to be used as starting point –see Fig. 1. PLAI-simp is essentially the PLAI algorithm [21], but omitting some optimizations that are independent from the issues related with the guidance. The algorithm is parametric on the abstract domain D_α, given by implementing the domain-dependent operations $\sqsubseteq, \sqcap, \sqcup$, `abs_call`, `abs_proceed`, `abs_generalize`, `abs_project`, and `abs_extend` (which will be described later), and transfer functions for program built-ins, that abstract the meaning of the basic operations of the language. These operations are assumed to be monotonic and to correctly over-approximate their correspondent concrete version. As stated before, the goal of the analyzer is to capture the behavior of each procedure (function or predicate) in the program with a set A of triples $\langle G, \lambda^c, \lambda^s \rangle$, where G is a normalized atom and λ^c and λ^s are, respectively, the abstract call and success constraints, elements of D_α. For conciseness, we denote looking up in A with a partial function $a : Atom * D_\alpha \mapsto D_\alpha$, where $\lambda^s = a[G, \lambda^c]$ *iff* $\langle G, \lambda^c, \lambda^s \rangle \in A$, and modify the value of a for (G, λ^c), denoted with $a[G, \lambda^c] \leftarrow \lambda^{s'}$ by removing $\langle G, \lambda^c, \_ \rangle$ from A and inserting $\langle G, \lambda^c, \lambda^{s'} \rangle$. In A there may be more than one triple with the same G, capturing multivariance, but only one for each λ^c during the algorithm's execution or in the final results.

Operation of the algorithm. Analysis proceeds from the initial abstract queries $\mathcal{Q}_\alpha$ assuming $\bot$ as under-approximation of their success constraint. The algorithm iterates over possibly incomplete results (in A), recomputing them with any newly inferred information, until a global fixpoint is reached (controlled by flag *changes*). First, the set of captured call patterns and the clauses whose head applies (i.e., there exists a renaming σ s.t. $G = \sigma(\mathsf{cl.head})$) is stored in W. Then, each clause is solved with the following process. An "abstract unification" (`abs_call`) is made, which performs the abstract parameter passing. It includes *renaming* the variables, abstracting the parameter values (via function α), and *extending* the abstract constraint to all variables present in the head and the body of the clause. To abstractly execute a clause the function `solve_body` abstractly executes each of the literals of the body. This implies, for each literal, *projecting* the abstract constraint onto the variables of the literal (`abs_project`) and *generalizing* it if necessary (`abs_generalize`) before calling `solve`. Generalization is necessary to ensure termination since we support multivariance and infinite domains. Lastly, after returning from `solve` (returning from the literal call), `abs_extend` propagates the information given by λ^s (success abstract constraint over the variables of L) to the constraint of the variables of the clause λ^t. The `solve` function executes abstractly a literal (Fig. 2). Depending on the nature of the literal, different actions will be performed. For *built-in operations*, the corresponding transfer function (f^α) is applied. For *predicates defined in the*

ALGORITHM **Analyze**$(P, \mathcal{Q}_\alpha)$
input: $P, \mathcal{Q}_\alpha$ **output global:** $A \leftarrow \emptyset$

```
1: a[L_i, λ_i] ← ⊥ for all L_i : λ_i ∈ Q_α, changes ← true                ▷ Initial queries
2: while changes do
3:     changes ← false
4:     W ← {(G, λ^c, cl) | a[G, λ^c] is defined ∧ cl ∈ P ∧ ∃σ s.t. G = σ(cl.head)}
5:     for each (G, λ^c, cl) ∈ W do
6:         λ^t ← abs_call(G, λ^c, cl.head)
7:         λ^t ← solve_body(cl.body, λ^t)
8:         λ^{s0} ← abs_proceed(G, cl.head, λ^t)
9:         λ^{s'} ← abs_generalize(λ^{s0}, {a[G, λ^c]})
10:        if λ^{s'} ≠ λ^s then
11:            a[G, λ^c] ← λ^{s'}, changes ← true                        ▷ Fixpoint not reached yet
12: function solve_body(B, λ^t)
13:     for each L ∈ B do
14:         λ^c ← abs_project(L, λ^t)
15:         Call = {λ | a[H, λ'] is defined ∧ ∃σ s.t. σ(H) = L ∧ λ = σ(λ')}
16:         λ^{c'} ← abs_generalize(λ^c, Call)
17:         λ^s ← solve(L, λ^{c'})
18:         λ^t ← abs_extend(L, λ^s, λ^t)
19:     return λ^t
```

Global: A

Fig. 1. Baseline fixpoint analysis algorithm (PLAI-simp).

```
1: function solve(L, λ)
2:     if L is a built-in then
3:         return f^α(L, λ)                                              ▷ apply transfer function
4:     else if a[G, λ^c] is defined and ∃σ s.t. σ(G) = L then
5:         return σ(a[G, λ^c])
6:     else
7:         a[L, λ] ← ⊥
8:         return ⊥
```

Fig. 2. Pseudocode for solving a literal.

program, the answer is first looked up in A. If there is already a computed tuple that matches the abstract call, the previously inferred result is taken. Else (no stored tuple matches the abstract call), an entry with that call pattern and $\bot$ as success value is added. This will trigger the analysis of this call in the next iteration of the loop. Once a body is processed, the actions of `abs_call` have to be undone in `abs_proceed`, which performs the "abstract return" from the clause. It *projects* the temporary abstract constraint (used to solve the body) back to the variables in the head of the clause and *renames* the resulting abstract constraint back to the variables of the analyzed head. The result is then abstractly *generalized* with the previous results (either from other clauses that also unify or from previous results of the processed clause), and it is compared with the previous result to check whether the fixpoint was reached. Termination is ensured even in the case of domains with infinite ascending chains because `abs_generalize` includes performing a widening if needed, in addition to the

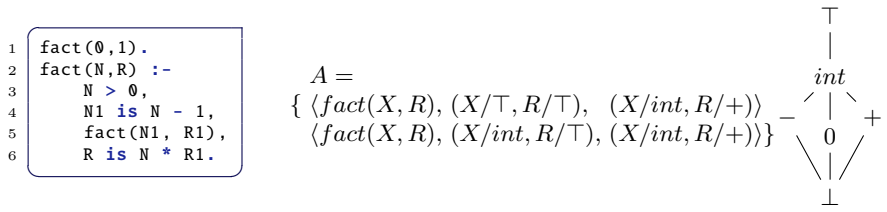

Fig. 3. Factorial program and a possible analysis result.

join operation $\sqcup$. This process is repeated for all the tuples of the analysis until the analysis results are the same in two consecutive iterations.

Fig. 3 shows a factorial program and an analysis result A for $\mathcal{Q}_\alpha = \{\texttt{fact}(X, R) : \top\}$ with an abstract domain that keeps information about signs for each of the program variables with values of the lattice shown. For example, the first tuple in A states that `fact(X,R)` may be called with any possible input and, if it succeeds, X will be an integer and R will be a positive number.

We define analysis results to be correct if the abstract call constraints cover all the call constraints (and, respectively, the abstract success constraints cover all the success constraints) which appear during the concrete execution of the initial queries in $\mathcal{Q}$. Formally:

Definition 2 (Correct analysis). *Given a program P and initial queries $\mathcal{Q}$, an analysis result A is correct for $P, \mathcal{Q}$ if:*

- $\forall G, \theta^c \in \mathit{calling\_context}(G, P, \mathcal{Q})\ \exists \langle G, \lambda^c, \lambda^s \rangle \in A$ *s.t.* $\theta^c \in \gamma(\lambda^c)$.
- $\forall \langle G, \lambda^c, \lambda^s \rangle \in A, \forall \theta^c \in \gamma(\lambda^c)$ *if* $\theta^s \in \mathit{answers}(P, \{G : \theta^c\})$ *then* $\theta^s \in \gamma(\lambda^s)$.

We recall the result from [21], adapted to the notation used in this paper.

Theorem 1. Correctness of PLAI. *Consider a program P and a set of initial abstract queries $\mathcal{Q}_\alpha$. Let $\mathcal{Q}$ be the set of concrete queries: $\mathcal{Q} = \{G : \theta \mid \theta \in \gamma(\lambda) \wedge G : \lambda \in \mathcal{Q}_\alpha\}$. The analysis result $A = \{\langle G_1, \lambda^c_1, \lambda^s_1 \rangle, \ldots, \langle G_n, \lambda^c_n, \lambda^s_n \rangle\}$ for P with $\mathcal{Q}_\alpha$ is* correct *for $P, \mathcal{Q}$.*

4 Adding assertion-based guidance to the algorithm

We now address how to apply the guidance provided by the user in the analysis algorithm. But before that we make some observations related to the run-time behavior of assertions.

Run-time semantics of assertions. Most systems make assumptions during analysis with respect to the run-time semantics of assertions: for example, Astrée assumes that they are always run, while CiaoPP assumes conservatively that they may not be (because in general they may in fact be disabled by the user). In order to offer the user the flexibility of expressing these different situations we introduce a new status for assertions, `sample-check`, as well as a corresponding program-point assertion, `sample-check(`*Cond*`)`. This `sample-check` status indicates that the properties in these assertions may or may not be checked during execution, i.e., run-time checking can be turned on or off (or done intermittently)

Status	Use in analyzer	Run-time test (if not discharged at compile-time)
`trust`	yes	no
`check`	yes	yes
`sample-check`	no	optional

Table 1. Usage of assertions during analysis.

for them. In contrast, for `check` assertions (provided that they have not been discharged statically) run-time checks must always be performed. Table 1 summarizes this behavior with respect to whether run-time testing will be performed and whether the analysis can "trust" the information in the assertion, depending on its status. The information in `trust` assertions is used by the analyzer but they are never checked at run time. `check` assertions are also checked at run time and the execution will not pass beyond that point if the conditions are not met.[5] This means that `check` assertions can also be "trusted," in a similar way to `trust` assertions, because execution only proceeds beyond them if they hold. Finally, `sample-check` assertions may or may not be checked at run-time (e.g., for efficiency reasons) and thus they cannot be used as `trust`s during analysis.

Correctly applying guidance. We recall some definitions (adapted from [24]) which are instrumental to correctly approximate the properties of the assertions during the guidance.

Definition 3 (Set of Calls for which a Property Formula Trivially Succeeds (Trivial Success Set)). *Given a conjunction L of property literals and the definitions for each of these properties in P, we define the* trivial success set *of L in P as:*

$$TS(L,P) = \{\theta | Var(L) \ s.t. \ \exists \theta' \in answers(P, \{L : \theta\}), \theta \models \theta'\}$$

where $\theta|Var(L)$ above denotes the projection of θ onto the variables of L, and $\models$ denotes that θ' is a more general constraint than θ (entailment). Intuitively, $TS(L,P)$ is the set of constraints θ for which the literal L succeeds without adding new constraints to θ (i.e., without constraining it further). For example, given the following program P:

```
1 list([]).
2 list([_|T]) :- list(T).
```

and $L = list(X)$, both $\theta_1 = \{X = [1,2]\}$ and $\theta_2 = \{X = [1,A]\}$ are in the trivial success set of L in P, since calling $(X = [1,2], list(X))$ returns $X = [1,2]$ and calling $(X = [1,A], list(X))$ returns $X = [1,A]$. However, $\theta_3 = \{X = [1|\_]\}$ is not, since a call to $(X = [1|Y], list(X))$ will further constrain the term $[1|Y]$, returning $X = [1|Y], Y = []$. We define abstract counterparts for Def. 3:

Definition 4 (Abstract Trivial Success Subset of a Property Formula). *Under the same conditions of Def. 3, given an abstract domain D_α, $\lambda^-_{TS(L,P)} \in D_\alpha$ is an* abstract trivial success subset *of L in P iff $\gamma(\lambda^-_{TS(L,P)}) \subseteq TS(L,P)$.*

[5] This strict run-time semantics for `check` assertions was used in [26].

Definition 5 (Abstract Trivial Success Superset of a Property Formula). *Under the same conditions of Def. 4, an abstract constraint* $\lambda^+_{TS(L,P)}$ *is an* abstract trivial success superset *of* L *in* P *iff* $\gamma(\lambda^+_{TS(L,P)}) \supseteq TS(L,P)$.

I.e., $\lambda^-_{TS(L,P)}$ and $\lambda^+_{TS(L,P)}$ are, respectively, safe under- and over-approximations of $TS(L,P)$. These abstractions come useful when the properties expressed in the assertions cannot be represented exactly in the abstract domain. Note that they are always computable by choosing the closest element in the abstract domain, and at the limit $\bot$ is a trivial success subset of any property formula and $\top$ is a trivial success superset of any property formula.

4.1 Including guidance in the fixpoint algorithm.

In Fig. 4 we present a version of PLAI-simp (from Fig. 1) that includes our proposed modifications to apply assertions during analysis. The additions to the algorithm are calls to functions `apply_succ` and `apply_call`, that guide analysis results with the information of the assertion conditions, and E, an analysis-like set of triples representing inferred states before applying the assertions that will be used to check whether the assertions provided by the user could be proved by the analyzer (see Sect. 5). Success conditions are applied (`apply_succ`) after the body of the clause has been abstractly executed. It receives an atom G and λ^c as parameters to decide correctly which success conditions have to be applied. Call conditions are applied (`apply_call`) before calling function `solve`. Otherwise, a less precise call pattern will be captured during the procedure (it adds new entries to the table). The last addition, E, collects tuples to be used later to check that the assertions were correct (see Sect. 5). We collect all success constraints before applying any success conditions (line 11 of Fig. 4) and all call constraints before applying any call condition (line 20 of Fig. 4).

Assuming that we are analyzing program P and the applicable assertion conditions are stored in C, the correct application of assertions is described in Fig. 5. Flag `speed-up` controls if assertions are used to recover accuracy or to (possibly) speed up fixpoint computation.

Applying call conditions. Given an atom G and an abstract call constraint λ^c, if there is a call assertion condition for G, if `speed-up` is true, $\lambda^+_{TS(Pre,P)}$ is used directly, otherwise the operation $\lambda^+_{TS(Pre,P)} \sqcap \lambda^c$ will prune from the analysis result the (abstracted) states that are outside the precondition. An over-approximation has to be made, otherwise we may remove calling states that the user did not specify.

Applying success conditions. Given an atom G, an abstract call constraint λ^c and its corresponding abstract success constraint λ^s, all success conditions whose precondition applies ($\lambda^c \sqsubseteq \lambda^-_{TS(Pre,P)}$) are collected in *app*. Making an under-approximation of *Pre* is necessary to consider the application of the assertion condition only if it would be applied in the concrete executions of the program. An over-approximation of *Post* needs to be performed since otherwise success states that actually happen in the concrete execution of the program may be

ALGORITHM **Guided_analyze**$(P, \mathcal{Q}_\alpha)$
input: $P, \mathcal{Q}_\alpha$ **global output:** $A \leftarrow \emptyset, E \leftarrow \emptyset$

1: $a[G_i, \lambda_i] \leftarrow \bot$ **for all** $G_i : \lambda_i \in \{G : \lambda^t | \lambda^t = \texttt{apply\_call}(G, \lambda), G : \lambda \in \mathcal{Q}_\alpha\}$
2: *changes* $\leftarrow$ true
3: **while** *changes* **do**
4: *changes* $\leftarrow$ false
5: $W \leftarrow \{(G, \lambda^c, \mathsf{cl}) \mid a[G, \lambda^c]$ is defined $\wedge\ \mathsf{cl} \in P\ \wedge \exists \sigma$ s.t. $G = \sigma(\mathsf{cl.head})\}$
6: **for each** $(G, \lambda^c, \mathsf{cl}) \in W$ **do**
7: $\lambda^t \leftarrow \texttt{abs\_call}(G, \lambda^c, \mathsf{cl.head})$
8: $\lambda^t \leftarrow \texttt{solve\_body}(\mathsf{cl.body}, \lambda^t)$
9: $\lambda^{s_0} \leftarrow \texttt{abs\_proceed}(G, \mathsf{cl.head}, \lambda^t)$
10: $\lambda^{s_1} \leftarrow \texttt{abs\_generalize}(\lambda^{s_0}, \{a[G, \lambda^c]\})$
11: $E \leftarrow E \cup \{\langle G, \lambda^c, \lambda^{s_1}\rangle\}$
12: $\lambda^s \leftarrow \texttt{apply\_succ}\ (G, \lambda^c, \lambda^{s_1})$
13: **if** $\lambda^s \neq a[G, \lambda^c]$ **then**
14: $a[G, \lambda^c] \leftarrow \lambda^s$, *changes* $\leftarrow$ true ▷ Fixpoint not reached yet
15: **function** $\texttt{solve\_body}(B, \lambda^t)$
16: **for each** $L \in B$ **do**
17: $\lambda^c \leftarrow \texttt{abs\_project}(L, \lambda^t)$
18: $Call = \{\lambda \mid a[H, \lambda']$ is defined $\wedge\ \exists \sigma$ s.t. $\sigma(H) = L \wedge \lambda = \sigma(\lambda')\}$
19: $\lambda^{c'} \leftarrow \texttt{abs\_generalize}(\lambda^c, Call)$
20: $E \leftarrow E \cup \{\langle L, \lambda^{c'}, \_\rangle\}$
21: $\lambda^{c'} \leftarrow \texttt{apply\_call}(L, \lambda^c)$
22: $\lambda^s \leftarrow \texttt{solve}(L, \lambda^{c'})$
23: $\lambda^t \leftarrow \texttt{abs\_extend}(L, \lambda^s, \lambda^t)$
24: **return** λ^t

Fig. 4. Fixpoint analysis algorithm using **assertion conditions**.

global flag: `speed-up`

1: **function** $\texttt{apply\_call}(L, \lambda^c)$
2: **if** $\exists \sigma, \lambda^t = \lambda^+_{TS(\sigma(Pre),P)}$ s.t. $\texttt{calls}(H, Pre) \in C, \sigma(H) = L$ **then**
3: **if** `speed-up` **return** λ^t **else return** $\lambda^c \sqcap \lambda^t$
4: **else return** λ^c
5: **function** $\texttt{apply\_succ}(G, \lambda^c, \lambda^{s_0})$
6: $app = \{\lambda \mid \exists\ \sigma, \texttt{success}(H, Pre, Post) \in C, \sigma(H) = G,$
7: $\lambda = \lambda^+_{TS(\sigma(Post),P)}, \lambda^-_{TS(\sigma(Pre),P)} \sqsupseteq \lambda^c\}$
8: **if** $app \neq \emptyset$ **then**
9: $\lambda^t = \bigsqcap app$
10: **if** `speed-up` **return** λ^t **else return** $\lambda^t \sqcap \lambda^{s_0}$
11: **else return** λ^{s_0}

Fig. 5. Applying assertions.

removed. If no conditions are applicable (i.e., *app* is empty), the result is kept as it was. Otherwise, if the flag `speed-up` is true $\lambda^+_{TS(Post,P)}$ is used, as it is; otherwise, it is used to refine the value of the computed answer λ^s.

Applying assertion conditions bounds the extrapolation (widening) performed by `abs_generalize`, avoiding unnecessary precision losses. Note that the existence

of guidance assertions for a predicate does not save having to analyze the code of the corresponding predicate if it is available, since otherwise any calls generated within that predicate would be omitted and not analyzed for, resulting in an incorrect analysis result.

4.2 Fundamental properties of analysis guided by assertions

We claim the following properties for analysis of a program P applying assertions as described in the previous sections. The inferred abstract execution states are covered by the call and (applicable) success assertion conditions.

Lemma 1. ***Applied call conditions.*** *Let* $\texttt{calls}(H, Pre)$ *be an assertion condition from program* P*, and let* $\langle G, \lambda^c, \lambda^s \rangle$ *be a triple derived for* P *and initial queries* $\mathcal{Q}_\alpha$ *by* GUIDED_ANALYZE(P, Q_α)*. If* $G = \sigma(H)$ *for some renaming* σ *then* $\lambda^c \sqsubseteq \lambda^+_{TS(\sigma(Pre),P)}$.

Proof. Function `apply_call` obtains in λ^t the trusted value for the call. It restricts the encountered call λ^c or uses it as is, in any case $\lambda^c \sqsubseteq \lambda^t = \lambda^+_{TS(Pre,P)}$. Hence if this function is applied whenever inferred call patterns are introduced in the analysis results, the lemma will hold.

The lemma holds after initialization, since the function is applied before inserting the tuples in A. Now we reason about how the algorithm changes the results. The two spots in which analysis results are updated are in function `solve` (line 7 of Fig. 2) and in the body of the loop of the algorithm (line 14 of Fig. 4). Function `solve` adds tuples to the analysis whenever new encountered call patterns are found, it is called right after `apply_call`, therefore it only inserts call patterns taking into account calls conditions. The analysis updates made in the body of the loop do not insert new call patterns, only the recomputed success abstractions for those already present (previously collected in W), therefore all call patterns encountered are added taking into account the call conditions and the lemma holds. □

Lemma 2. ***Applied success conditions.*** *Let* $\texttt{success}(H, Pre, Post)$ *be an assertion condition from program* P *and let* $\langle G, \lambda^c, \lambda^s \rangle$ *be a triple derived for* P *with* $\mathcal{Q}_\alpha$ *initial queries by* GUIDED_ANALYZE$(P, \mathcal{Q}_\alpha)$ *. If* $G = \sigma(H)$ *for some renaming* σ *then* $\lambda^c \sqsubseteq \lambda^-_{TS(\sigma(Pre),P)} \Rightarrow \lambda^s \sqsubseteq \lambda^+_{TS(\sigma(Post),P)}$.

Proof. Function `apply_succ` computes the $\sqcap$ of all applicable assertion conditions (checking $\lambda^c \sqsubseteq \lambda^-_{TS(Pre,P)}$), if existing. Since we make the $\sqcap$ of all applied conditions, $\lambda^s \sqsubseteq \bigsqcap \lambda^+_{TS(Post_i,P)} \sqsubseteq \lambda^+_{TS(Post,P)}$ for any $Post$. Hence if all results inserted in the analysis result have been previously processed by `apply_succ` the lemma holds. The lemma holds for the initialized results, because $\lambda^s = \bot \sqsubseteq \lambda^+_{TS(Post,P)}$ for any $Post$. Now we reason about how the algorithm changes the results. We have the same points in the algorithm that change the analysis result as in the proof of Lemma 1. The `solve` function initializes λ^s of the newly encountered calls with $\bot$, so it is the same situation as when initializing. In the body of the loop `apply_succ` is always called before updating the value in the result and the lemma holds. □

5 Checking correctness in a guided analysis

We discuss how assertions may introduce errors in the analysis, depending on their status. `sample-check` assertions are not used by the analyzer. Any part of the execution stopped by them will conservatively be considered to continue, keeping the analysis safe. `check` assertions stop the execution of the program if the properties of the conditions are not met. Hence it is safe to narrow the analysis results using their information. Last, `trust` assertions are not considered during the concrete executions, so they may introduce errors. Such assertion conditions express correct properties if they comply with the following definitions:

Definition 6 (Correct call condition). *Let P be a program with an assertion condition $C = \texttt{calls}(H, Pre)$. C is correct for a query $\mathcal{Q}$ to P if for any predicate descriptor G, s.t. $G = \sigma(H)$ for some renaming σ, $\forall \theta^c \in calling\_context(G, P, \mathcal{Q})$, $\theta^c \in \gamma(\lambda^+_{TS(\sigma(Pre),P)})$.*

Definition 7 (Correct success condition). *Let P be a program with an assertion condition $C = \texttt{success}(H, Pre, Post)$. C is correct for P if for any predicate descriptor G, s.t. $G = \sigma(H)$ for some renaming σ, $\theta^c \in \gamma(\lambda^-_{TS(\sigma(Pre),P)}), \theta^s \in answers(P, \{G : \theta^c\}) \Rightarrow \theta^s \in \gamma(\lambda^+_{TS(\sigma(Post),P)})$.*

Theorem 2. Correctness modulo assertions. *Let P be a program with* correct *assertion conditions C and $\mathcal{Q}_\alpha$ a set of initial abstract queries. Let $\mathcal{Q}$ be the set of concrete queries: $\mathcal{Q} = \{G : \theta \mid \ \theta \in \gamma(\lambda) \wedge G : \lambda \in \mathcal{Q}_\alpha\}$.*
The analysis result $A = \{\langle G_1, \lambda^c_1, \lambda^s_1\rangle, \ldots, \langle G_n, \lambda^c_n, \lambda^s_n\rangle\}$ computed with GUIDED_ANALYZE$(P, \mathcal{Q}_\alpha)$ *is* correct *(Def. 2) for $P, \mathcal{Q}$.*

Proof. For conciseness in the proof we omit the renaming part. Fixed program P, given an abstract description d from an assertion (*Pre* or *Post*), let $\lambda^-_d = \lambda^-_{TS(d,P)}, \lambda^+_d = \lambda^+_{TS(d,P)}$. If there are no assertion conditions, the theorem trivially holds (Thm. 1). If assertion conditions are used to generalize, the theorem also holds because $\lambda^c = \lambda^+_{Pre}$ and $\lambda^s = \lambda^+_{Post}$ are by definition (Def. 6, Def. 7, respectively) correct over-approximations. If assertion conditions are used to regain precision:
Call: We want to prove that
$\forall G, \theta^c \in calling\_context(G, P, \mathcal{Q})\ \exists \langle G, \lambda^c, \lambda^s\rangle \in A$ s.t. $\theta^c \in \gamma(\lambda^c)$(Def. 2).

We applied: $\texttt{calls}(G, Pre)$

$$\theta^c \in \gamma(\lambda^+_{Pre}) \qquad \text{(by Def. 6)}$$

$$\text{In } E: \exists \langle G, \lambda^c_E, \lambda^s_E\rangle \in E, \theta^c \in \gamma(\lambda^c_E) \qquad \text{(by algorithm (Fig. 4 line 20))}$$

$$\text{Then: } \theta^c \in \gamma(\lambda^c_E) \cap \gamma(\lambda^+_{Pre}) \subseteq \gamma(\alpha(\gamma(\lambda^c_E) \cap \gamma(\lambda^+_{Pre}))) \subseteq \gamma(\lambda^c_E \sqcap \lambda^+_{Pre})$$

$$\theta^c \in \gamma(\lambda^c_E \sqcap \lambda^+_{Pre}) = \gamma(\lambda^c) \qquad \text{(by algorithm (Fig. 5 line 3))}$$

Success: We want to prove that
$\forall\langle G, \lambda^c, \lambda^s\rangle \in A, \forall\theta^c \in \gamma(\lambda^c)$ if $\theta^s \in answers(P, \{G : \theta^c\})$ then $\theta^s \in \gamma(\lambda^s)$.

We applied: $\texttt{success}(G, Pre_i, Post_i)$

$$\lambda^c \sqsubseteq \lambda^-_{TS(Pre_i)} \implies \lambda^s \sqsubseteq \lambda^+_{Post_i} \quad \text{(by Lemma 2)}$$

$$\theta^c \in \gamma(\lambda^-_{Pre_i}), \theta^s \in answers(P, \{G : \theta^c\}) \implies \theta^s \in \lambda^+_{Post_i} \quad \text{(by Def. 7)}$$

$$\lambda^p = \bigsqcap\{\lambda^+_{Post} \mid \texttt{success}(G, Pre, Post), \forall\theta^c \in \lambda^c, \theta^c \in \gamma(\lambda^-_{Pre})\}$$

$$\theta^c \in \gamma(\lambda^c), \theta^s \in answers(P, \{G : \theta^c\}) \implies \theta^s \in \lambda^p$$

$$\exists\langle G, \lambda^c_E, \lambda^s_E\rangle \in E, \text{ s.t. } \lambda^c \sqsupseteq \gamma(\lambda^c_E) \quad \text{(unrefined abstractions)}$$

We have: $\theta^s \in \gamma(\lambda^s_E), \theta^s \in \gamma(\lambda^p)$

$$\theta^s \in \gamma(\lambda^s_E) \cap \gamma(\lambda^p) \subseteq \gamma(\alpha(\gamma(\lambda^s_E) \cap \gamma(\lambda^p))) \subseteq \gamma(\lambda^s_E \sqcap \lambda^p)$$

$$\theta^s \in \gamma(\lambda^s_E \sqcap \lambda^p) = \gamma(\lambda^s) \qquad \square$$

In other words, Theorem 2 and Lemmas 1 and 2 ensure that correct assertion conditions bound imprecision in the result, without affecting correctness. By applying the assertion conditions no actual concrete states are removed from the abstractions.

We can identify suspicious pruning during analysis. Let λ^a be the correct approximation of a condition and λ be an inferred abstract state, typically a value in the tuples of E. If $\lambda \sqcap \lambda^a = \bot$ the inferred information is incompatible with that in the condition, therefore it is likely that the assertion is *erroneous*. $\lambda \not\sqsubseteq \lambda^a$ indicates that the algorithm inferred more concrete constraint states than described in the assertion and the analysis results may be wrong. These checks can be performed while the algorithm is run or off-line, by comparing the properties of the assertion conditions against the triples stored in E, which, as mentioned earlier, stores partial analysis results with no assertions applied. A full description of this checking procedure is described in [24, 26].

6 Related work

The inference of arbitrary semantic properties of programs is known to be both undecidable and expensive, requiring user interaction in many realistic settings. Abstract interpreters allow the **selection of different domains and parameters for such domains** (e.g., polyhedra, octagons, regtypes with depth-k, etc.), as well as their widening operations (e.g., type shortening, structural widening, etc.). Other parameters include policies for partial evaluation and other transformations (loop unrolling, inlining, slicing, etc.). These parameters are orthogonal or complementary to the issues discussed in this paper. To the extent of our knowledge the use of **program-level annotations** (such as assertions) to guide abstract interpretation has not been widely studied in the literature, contrary to their (necessary) use in verification and theorem proving approaches. The **Cibai** [17] system includes *trust-style* annotations while sources are processed to encode some predefined runtime semantics. In [10] analysis is guided

by modifying the analyzed program to restrict some of its behaviors. However, this guidance affects the *order of program state exploration*, rather the analysis results, as in our case. As mentioned in the introduction, the closest to our approach is **Astrée**, that allows *assert*-like statements, where correctness of the analysis is ensured by the presence of compulsory runtime checks, and trusted (*known facts*) asserts. These refine and guide analysis operations at program points. Like in CiaoPP, the analyzer shows errors if a known fact can be falsified statically. However, as with the corresponding Ciao assertions, while there has been some examples of use [8], there has been no detailed description of how such assertions are handled in the fixpoint algorithm. We argue that this paper contributes in this direction.

7 Conclusions

We have proposed a user-guided multivariant fixpoint algorithm that makes use of check and trust assertion information, and we have provided formal results on the influence of such assertions on correctness and efficiency. We have extended the semantics of the guidance (and all) assertions to deal with both the cases in which the program execution will and will not incorporate run-time tests for unverified assertions, as well as the cases in which the assertions are intended for refining the information or instead to lose precision in order to gain efficiency. We show that these annotations are not only useful when dealing with incomplete code but also provide the analyzer with recursion/loop invariants for speeding up global convergence.

Bibliography

[1] Bagnara, R., Hill, P.M., Zaffanella, E.: Widening operators for powerset domains. In: VMCAI. LNCS, vol. 2937, pp. 135–148. Springer (2004)

[2] Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn Clause Solvers for Program Verification. In: Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday. pp. 24–51 (2015)

[3] Bourdoncle, P.: Interprocedural abstract interpretacion of block structured programs with nested procedures, aliasing and recursivity. In: PLILP, pp. 84–97. No. 456 in LNCS, Springer–Verlag (1990)

[4] Bruynooghe, M.: A Practical Framework for the Abstract Interpretation of Logic Programs. Journal of Logic Programming **10**, 91–124 (1991)

[5] Bueno, F., Cabeza, D., Hermenegildo, M.V., Puebla, G.: Global Analysis of Standard Prolog Programs. In: ESOP (1996)

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

[7] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astreé analyzer. In: ESOP 2005. pp. 21–30 (2005)

[8] Delmas, D., Souyris, J.: Astrée: From research to industry. In: SAS. pp. 437–451 (2007)

[9] Gómez-Zamalloa, M., Albert, E., Puebla, G.: Modular Decompilation of Low-Level Code by Partial Evaluation. In: SCAM. pp. 239–248. IEEE Comp. Soc. (2008)

[10] Gopan, D., Reps, T.: Guided static analysis. In: SAS. pp. 349–365. Springer (2007)
[11] Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: CAV. pp. 343–361 (2015)
[12] Henriksen, K.S., Gallagher, J.P.: Abstract Interpretation of PIC Programs through Logic Programming. In: SCAM. pp. 184–196. IEEE Computer Society (2006)
[13] Hermenegildo, M., Puebla, G., Bueno, F., García, P.L.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Comp. Progr. **58**(1–2) (2005)
[14] Hermenegildo, M.V., Puebla, G., Bueno, F.: Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In: The Logic Programming Paradigm, pp. 161–192. Springer (1999)
[15] Klemen, M., Stulova, N., Lopez-Garcia, P., Morales, J.F., Hermenegildo, M.V.: Static Performance Guarantees for Programs with Run-time Checks. In: PPDP. ACM Press (2018)
[16] Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., Lopez-Garcia, P., Grech, N., Hermenegildo, M.V., Eder, K.: Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In: LOPSTR. LNCS, vol. 8901, pp. 72–90. Springer (2014)
[17] Logozzo, F.: Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of java classes. In: VMCAI. LNCS 4349 (2007)
[18] Marriott, K., Stuckey, P.J.: Programming with Constraints: an Introduction. MIT Press (1998)
[19] Méndez-Lojo, M., Navas, J., Hermenegildo, M.: A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In: LOPSTR. LNCS, vol. 4915, pp. 154–168. Springer-Verlag (August 2007)
[20] Muthukumar, K., Hermenegildo, M.: Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In: NACLP'89. pp. 166–189. MIT Press (October 1989)
[21] Muthukumar, K., Hermenegildo, M.: Compile-time Derivation of Variable Dependency Using Abstract Interpretation. JLP **13**(2/3), 315–347 (July 1992)
[22] Navas, J., Méndez-Lojo, M., Hermenegildo, M.V.: User-Definable Resource Usage Bounds Analysis for Java Bytecode. In: BYTECODE'09. ENTCS, vol. 253, pp. 6–86. Elsevier (March 2009)
[23] Puebla, G., Bueno, F., Hermenegildo, M.V.: An Assertion Language for Constraint Logic Programs. In: Analysis and Visualization Tools for Constraint Programming, pp. 23–61. No. 1870 in LNCS, Springer-Verlag (2000)
[24] Puebla, G., Bueno, F., Hermenegildo, M.V.: Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In: Proc. of LOPSTR'99. pp. 273–292. LNCS 1817, Springer-Verlag (March 2000)
[25] Puebla, G., Hermenegildo, M.V.: Optimized Algorithms for the Incremental Analysis of Logic Programs. In: SAS'96. pp. 270–284. Springer LNCS 1145 (1996)
[26] Stulova, N., Morales, J.F., Hermenegildo, M.V.: Some Trade-offs in Reducing the Overhead of Assertion Run-time Checks via Static Analysis. Science of Computer Programming **155**, 3–26 (April 2018)
[27] Vaucheret, C., Bueno, F.: More Precise yet Efficient Type Inference for Logic Programs. In: SAS'02. pp. 102–116. No. 2477 in LNCS, Springer (2002)
[28] Zaffanella, E., Bagnara, R., Hill, P.M.: Widening Sharing. In: Nadathur, G. (ed.) PPDP. LNCS, vol. 1702, pp. 414–432. Springer-Verlag, Berlin (1999)