Incremental Analysis of Logic Programs with Assertions and Open Predicates

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

[1] IMDEA Software Institute, Madrid, Spain
{isabel.garcia,josef.morales,manuel.hermenegildo}@imdea.org
[2] Universidad Politécnica de Madrid (UPM), Madrid, Spain

Abstract. *Generic* components are a further abstraction over the concept of modules, introducing dependencies on other (not necessarily available) components implementing specified interfaces. They have become a key concept in large and complex software applications. Despite undeniable advantages, generic code is also *anti-modular*. Precise analysis (e.g., for detecting bugs or optimizing code) requires such code to be instantiated with concrete implementations, potentially leading to expensive combinatorial explosion. In this paper we claim that *incremental*, whole program analysis can be very beneficial in this context, and alleviate the anti-modularity nature of generic code. We propose a simple Horn-clause encoding of generic programs, using *open* predicates and assertions, and we introduce a new *incremental, multivariant* analysis algorithm that reacts incrementally not only to changes in program clauses, but also to *changes in the assertions*, upon which large parts of the analysis graph may depend. We also discuss the application of the proposed techniques in a number of practical use cases. In addition, as a realistic case study, we apply the proposed techniques in the analysis of the `LPdoc` documentation system. We argue that the proposed traits are a convenient and elegant abstraction for modular generic programming, and that our preliminary results support our thesis that the new incrementality-related features added to the analysis bring promising advantages in this context.

Keywords: Incremental static analysis · Verification · Assertions · Generic code · Specifications · Abstract interpretation · Horn clauses · Logic programs

1 Introduction

When developing large, real-life programs it is important to ensure application reliability and coding convenience. An important component in order to achieve

Research partially funded by MINECO TIN2015-67522-C3-1-R *TRACES* project, FPU grant 16/04811, and the Madrid P2018/TCS-4339 *BLOQUES-CM* program. We are also grateful to the anonymous reviewers for their useful comments.

these goals is the availability in the language (and use in the development process) of some mechanism for expressing specifications, combined with a way of determining if the program meets the specifications or locate errors. This determination is usually achieved through some combination of compile-time analysis and verification with testing and run-time assertion checking [7,9,12,22,23].

Another relevant aspect when developing large programs is modularity. In modern coding it is rarely necessary to write everything from scratch. Modules and interfaces allow dividing the program in manageable and interchangeable parts. *Interfaces*, including specifications and dependencies, are needed in order to connect with external code (including specifications of such code), to connect self-developed code that is common with other applications, and as a placeholder for different implementations of a given functionality, in general referred to as *generic code.*

Despite undeniable advantages, generic code is known to be in fact *anti-modular*, and the analysis of generic code poses challenges: parts of the code are unavailable, and the interface specifications may not be descriptive enough to allow verifying the specifications for the whole application. Several approaches are possible in order to balance separate compilation with precise analysis and optimization. First, it is possible to analyze generic code by *trusting* its interface specifications, i.e., analyzing the client code and the interface implementations independently, flattening the analysis information inferred at the boundaries to that of the interface descriptions. This technique can reduce global analysis cost significantly at the expense of some loss of precision. Some of it may be regained by, e.g., enriching specifications manually for the application at hand. Alternatively, for a closed set of interface implementations, it may be desirable to analyze the whole application together with these implementations, keeping different specialized versions of the analysis across the interfaces. This allows getting the most precise information, specializations, compiler optimizations, etc., but at a higher cost.

Multivariant analyses maintain different information for each predicate call, depending on the caller predicate and the sequence of calls to this call. For imperative programs this implies the notions of "context-" and "path-"sensitivity. We believe that this information is specially beneficial when dealing with generic code, both for precision of the analysis results and for efficiency of the algorithm. Thus, our starting point is a (whole program) analysis that is multivariant. To treat generic code we propose a simple Horn-clause encoding, using *open* predicates and assertions, and introduce a novel extension for logic programming (*traits*) that is translated using open predicates. This abstraction addresses typical use cases of generic code in a more elegant and analysis-friendly way than the traditional alternative in LP of using *multifile* predicates. Then, we introduce a new, multivariant analysis algorithm that, in addition to supporting and taking advantage of assertions during analysis, *reacts incrementally to changes* not only in the program clauses but also *in the assertions*, upon which large parts of the analysis graph may depend, while also *supporting natively open predicates*. Generic code offers many opportunities for the application of this new analysis technique. We study a number of use cases, including editing a client (of an interface), while keeping the interface unchanged (e.g., analyzing a program reusing the analysis of a –family of– libraries) and keeping the client code unchanged, but editing the interface implementation(s) (e.g., modifying one implementation of an interface). In addition, we provide experimental results in a realistic case study: the analysis of the `LPdoc` documentation system and its multiple backends for generating documentation in different formats. Related work is discussed in Sect. 7.

2 Background

Logic Programs. A *definite Logic Program*, or *program*, is a finite sequence of *Horn clauses* (*clauses* for short). 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*. An *atom* is of the form $p(V_1, \ldots, V_n)$, where p is a symbol of arity n. It is *normalized* if the $V_1, \ldots, V_n$ are all distinct variables. Normalized atoms are also called *predicate descriptors*. Each maximal set of clauses in the program with the same descriptor as head (modulo variable renaming) defines a *predicate* (or *procedure*). p/n refers to a predicate p of arity n. Body literals can be predicate descriptors, which represent *calls* to the corresponding predicates, or *built-ins*. A *built-in* is a predefined relation for some background theory. Note that built-ins are not necessarily normalized. In the examples we may use non-normalized programs. We denote with $vars(A)$ the set of variables that appear in the atom A.

For presentation purposes, the heads of the clauses of each predicate in the program will be referred to with a unique subscript attached to their predicate name (the clause number), and the literals of their bodies with dual subscript (clause number, body position), e.g., P_k `:-` $P_{k,1}, \ldots P_{k,n_k}$. The clause may also be referred to as clause k of predicate P. For example, for the predicate `app/3`:

```
app(X,Y,Z):- X=[],  Y=Z.
app(X,Y,Z):- X=[U|V], Z=[U|W], app(V,Y,W).
```

`app/3`$_1$ denotes the head of the first clause of `app/3`, `app/3`$_{2,1}$ denotes the first literal of the second clause of `app/3`, i.e., the unification `X=[U|V]`.

Assertions. Assertions allow stating conditions on the state (current substitution) 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 [12,21], 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 [9,12,22]. A `pred` assertion is of the form:

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

where *Head* is a predicate descriptor 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 [21]. *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.

Example 1. The following assertions describe different behaviors of an implementation of a hashing function `dgst`: `(1)` states that, when called with argument `Word` a string and `N` a variable, then, if it succeeds, `N` will be a number, `(2)` states that calls for which `Word` is a string and `N` is an integer are allowed, i.e., it can be used to check if `N` is the hash of `Word`.

```
:- pred dgst(Word,N) : (string(Word), var(N)) => num(N).  % (1)
:- pred dgst(Word,N) : (string(Word), int(N)).            % (2)
dgst(Word,N) :-
% implementation of the hashing function
```

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 $\texttt{calls}(Head, Pre)$[1] states conditions on all concrete calls to the predicate described by *Head*, and $\texttt{success}(Head, Pre_j, Post_j)$ describes conditions on the success substitutions produced by calls to *Head* if Pre_j is satisfied.

3 An Approach to Modular Generic Programming: *Traits*

In this section we present a simple approach to modular generic programming for logic programs without static typing. To that end we introduce the concept of *open* predicates. Then we show how they can be used to deal with generic code, by proposing a simple syntactic extension for logic programs for writing and using generic code (*traits*) and its translation to plain clauses.

Open vs. Closed Predicates. We consider a simple module system for logic programming where predicates are distributed in modules (each predicate symbol belongs to a particular module) and where module dependencies are explicit in the program [2]. An interesting property, specially for program analysis, is

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

that we can distinguish between *open* and *closed* predicates.[2] Closed predicates within a module are those whose complete definition is available in the module. In contrast, the definition of open predicates (traditionally declared as `multifile` in many Prolog systems) can be can be scattered across different modules, and thus not known until all the application modules are linked (note that programs still use the closed world assumption). Despite its flexibility, open predicates are "anti-modular" (in a similar way to typeclasses in Haskell).

Open as "multifile." The following example shows an implementation of a generic password-checking algorithm in Prolog:

```
:- multifile dgst/3.

check_passwd(User) :-
    get_line(Plain),                 % Read plain text password
    passwd(User,Hasher,Digest,Salt), % Consult password database
    append(Plain,Salt,Salted),       % Append salt
    dgst(Hasher,Salted,Digest).      % Compute and check digest
```

The code above is generic w.r.t. the selected hashing algorithm (`Hasher`). Note that there is no explicit dependency between `check_passwd/1` and the different hashing algorithms. The special *multifile* predicate `dgst/3` acts as an *interface* between implementations of hashing algorithms and `check_passwd/1`. While this type of encoding is widely used in practice, the use of multifile predicates is semantically obscure and error-prone. Instead we propose *traits* as a syntactic extension that captures the essential mechanisms necessary for writing generic code.[3]

Traits. A *trait* is defined as a collection of predicate specifications (as predicate assertions). For example:

```
:- trait hasher { :- pred dgst(Str, Digest) : string(Str) => int(Digest). }.
```

defines a trait `hasher`, which specifies a predicate `dgst/2`, which must be called with an instantiated string, and obtains an integer in `Digest`.

As a minimalistic syntactic extension, we introduce a new head and literal notation $(X\ \texttt{as}\ T).\texttt{p}(A_1, \ldots, A_n)$, which represents the predicate p for X implementing trait T. Basically, this is equivalent to $p(X, A_1, \ldots, A_n)$, where X is used to select the trait implementation. In literals, X is annotated with a trait, which can be different for each call due to dynamic typing and multiple trait implementations for the same data. When X (the implementation) is unknown

[2] For space reasons we only consider *static* predicates and modules. Predicates whose definition may change during execution, or modules that are dynamically loaded/unloaded at run time can also be dealt with, using various techniques, and in particular the incremental analysis proposed.

[3] In this paper we only focus on traits as interfaces. The actual design in Ciao supports default implementations, which makes them closer to traits in Rust.

at compile-time, this is equivalent to dynamic dispatch. The `check_passwd/1` predicate using the trait above is:

```
check_passwd(User) :-
    get_line(Plain),
    passwd(User,Hasher,Digest,Salt),
    append(Plain,Salt,Salted),
    (Hasher as hasher).dgst(Salted,Digest).
```

The following translation rules convert code using traits to plain predicates. Note that we rely on the underlying module system to add module qualification to function and trait (predicate) symbols. Calls to trait predicates are done through the interface (open) predicate, which also carries the predicate assertions declared in the trait definition:

```
% open predicates and assertions for each p/n in the trait
:- multifile 'T.p'/(n+1).
:- pred 'T.p'(X,A1,...,An) : ... => ... .
% call to p/n for X implementing T
... :- ..., 'T.p'(X,A1,...,An), ... % (X as T).p(A1,...,An)
```

A trait *implementation* is a collection of predicates that implements a given trait, indexed by a specified functor associated with that implementation. E.g.:

```
:- impl(hasher, xor8/0).
(xor8 as hasher).dgst(Str, Digest) :- xor8_dgst(Xs, 0, Digest).

xor8_dgst([], D, D).
xor8_dgst([X|Xs], D0, D) :- D1 is D0 # X, xor8_dgst(Xs, D1, D).
```

declares that `xor8` implements a `hasher`. In this case `xor8` is an atom, but trait syntax allows arbitrary functors. The implementation for the `dgst/2` predicate is provided by `(xor8 as hasher).dgst(Str, Digest)`.

The translation rules to plain predicates are as follows:

```
% the implementation is a closed predicate (head renamed)
'<f/k as T>.p'(f(...),A1,...,An) :- ... % (f(...) as T).p(A1,...,An)

% bridge from interface (open predicate) to the implementation
'T.p'(X,A1,...,An) :- X=f(...), '<f/k as T>.p'(X,A1,...,An).
```

Adding new implementations is simple:

```
:- impl(hasher, sha256/0).
(sha256 as hasher).dgst(Str, Digest) :- ...
```

This approach still preserves some interesting modular features: trait names can be local to a module (and exported as other predicate/function symbols), and trait implementations (e.g., `sha256/0`) are just function symbols, which can also be made local to modules in the underlying module system.

4 Goal-Dependent Abstract Interpretation

We recall some basic concepts of abstract interpretation of logic programs.

Program Analysis with Abstract Interpretation. Our approach is based on *abstract interpretation* [4], a technique in which the execution of the program is simulated (over-approximated) 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 Semantics. In out context, running a program consists of making a *query*. Executing (answering) a query is determining for which substitutions (answers) the query is a logical consequence of the program if any. A *query* is a pair $\langle G,\theta\rangle$ with G an atom and θ a substitution over the variables of G. For concreteness, we focus on top-down, left-to-right SLD-resolution. We base our semantics on the well-known notion of generalized AND trees [1]. The concrete semantics of a program P for a given set of queries $\mathcal{Q}$, $[\![P]\!]_{\mathcal{Q}}$, is the set of generalized AND trees that results from the execution of the queries in $\mathcal{Q}$ for P. Each node $\langle G,\theta^c,\theta^s\rangle$ in the tree represents a call to a predicate G (an atom), with the substitution (state) for that call, θ^c, and the success substitution 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 $\{\theta^c \mid \langle G',\theta'^c,\theta'^s\rangle \in T \;\forall\; T \in [\![P]\!]_{\mathcal{Q}} \wedge \exists\sigma, \sigma(G') = G \wedge \sigma(\theta'^c) = \theta^c\}$, where σ is a *renaming* substitution. I.e., a substitution that replaces each variable in the term with distinct, fresh variables. We denote by *answers*$(P,\mathcal{Q})$ the set of success substitutions computed by P for queries $\mathcal{Q}$.

Graphs and Paths. We denote by $G = (V,E)$ a finite *directed graph* (henceforward called simply a graph) where V is a set of nodes and $E \subseteq V \times V$ is an edge relation, denoted with $u \to v$. A *path* P is a sequence of edges $(e_1,\ldots,e_n)$ and each $e_i = (x_i,y_i)$ is such that $x_1 = u$, $y_n = v$, and for all $1 \leq i \leq n-1$ we have $y_i = x_{i+1}$, we also denote paths with $u \rightsquigarrow v \in G$. We use $n \in P$ and $e \in P$ to denote, respectively, that a node n and an edge e appear in P.

4.1 Goal-Dependent Program Analysis

We perform goal-dependent abstract interpretation, whose result is an abstraction of the generalized AND tree semantics. This technique derives an analysis result from a program P, an abstract domain D_α, and a set of initial abstract queries $\mathcal{Q} = \{\langle A_i, {\lambda^c}_i\rangle\}$, where A_i is a normalized atom, and ${\lambda^c}_i \in D_\alpha$. An *analysis result* encodes an abstraction of the nodes of the generalized AND trees derived from all the queries $\langle G,\theta\rangle$ s.t. $\langle G,\lambda\rangle \in \mathcal{Q} \wedge \theta \in \gamma(\lambda)$.

Analysis Graphs. We use graphs to overapproximate all possible executions of a program given an initial query. Each node in the graph is identified by a pair (P, λ) with P a predicate descriptor and $\lambda \in D_\alpha$, an element of the abstract domain, representing the possibly infinite set of calls encountered. The analysis result defines a mapping function $ans : Pred \times D_\alpha \to D_\alpha$, denoted with $\langle P, \lambda^c \rangle \mapsto \lambda^s$ which over-approximates the answer to that abstract predicate call. It is interpreted as "*calls to predicate P with calling pattern λ^c have the answer pattern λ^s*" with $\lambda^c, \lambda^s \in D_\alpha$. The analysis graph is *multivariant*. Thus, it may contain a number of nodes for the same predicate capturing different call situations, for different contexts or different paths. As usual, $\bot$ denotes the abstract description such that $\gamma(\bot) = \emptyset$. A call mapped to $\bot$ ($\langle P, \lambda^c \rangle \mapsto \bot$) indicates that calls to predicate P with any description $\theta \in \gamma(\lambda^c)$ either fail or loop, i.e., they never succeed.

Edges in the graph represent a call dependency among two predicates. An edge is of the form $\langle P, \lambda_1 \rangle_{c,l} \xrightarrow[\lambda^r]{\lambda^p} \langle Q, \lambda_2 \rangle$, and is interpreted as "*calling predicate P with substitution λ_1 causes predicate Q (literal l of clause c) to be called with substitution λ_2*". Substitutions λ^p and λ^r are, respectively, the call and return context of the call. These values are introduced to ease the presentation of the algorithm, but they can be reconstructed with the identifiers of the nodes (i.e., predicate descriptor and abstract value) and the source code of the program. For simplicity, we may write $\bullet$ to omit the values when they are not relevant to the discussion. Note that the edges that represent the calls to a literal l and the following one $l+1$, $\langle P, \lambda_1 \rangle_{c,l} \xrightarrow[\lambda]{\bullet} \langle Q, \lambda_2 \rangle$ the result at the return of the literal is the call substitution of the next literal: $\langle P, \lambda_1 \rangle_{c,l+1} \xrightarrow[\bullet]{\lambda} \langle Q', \lambda_2' \rangle$. Figure 1 shows a possible analysis graph for a program that checks/computes the parity of a message. The following operations defined over an analysis result g allow us to inspect and manipulate analysis results to partially reuse or invalidate.

Graph Consultation Operations

$\langle P, \lambda^c \rangle \in g$: there is a node in the call graph of g with key $\langle P, \lambda^c \rangle$.

$\langle P, \lambda^c \rangle \mapsto \lambda^s \in g$: there is a node in g with key $\langle P, \lambda^c \rangle$ and the answer mapped to that call is λ^s.

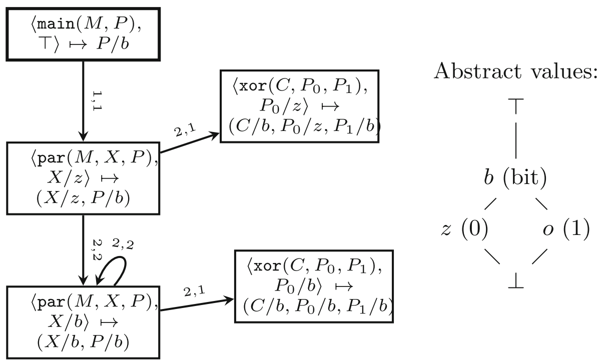
$\langle P, \lambda^c \rangle_{c,l} \xrightarrow[\lambda^r]{\lambda^p} \langle Q, \lambda^{c\prime} \rangle \in g$: there are two nodes ($k = \langle P, \lambda^c \rangle$ and $k' = \langle Q, \lambda^{c\prime} \rangle$) in g and there is an annotated edge from k to k'.

Graph Update Operations

$\mathsf{add}(g, \{k_{c,l} \xrightarrow[\lambda^p]{\lambda^r} k'\})$: adds an edge from node k to k' (creating node k' if necessary) annotated with λ^p and λ^r for clause c and literal l.

$\mathsf{del}(g, \{k_{c,l} \xrightarrow[\bullet]{\bullet} k'\})$: removes the edge from node k to k' annotated for clause c and literal l.

```
main(Msg, P) :-
    par(Msg, 0, P).

par([], P, P).
par([C|Cs], P0, P) :-
    xor(C, P0, P1),
    par(Cs, P1, P).

xor(0,0,0).
xor(0,1,1).
xor(1,0,1).
xor(1,1,0).
```



$\langle \texttt{main}(M, P), \top\rangle \mapsto P/b$
1,1
$\langle \texttt{par}(M, X, P), X/z\rangle \mapsto (X/z, P/b)$
2,1
$\langle \texttt{xor}(C, P_0, P_1), P_0/z\rangle \mapsto (C/b, P_0/z, P_1/b)$
2,2
2,2
$\langle \texttt{par}(M, X, P), X/b\rangle \mapsto (X/b, P/b)$
2,1
$\langle \texttt{xor}(C, P_0, P_1), P_0/b\rangle \mapsto (C/b, P_0/b, P_1/b)$
Abstract values:
$\top$
b (bit)
z (0)
o (1)
$\bot$

Fig. 1. A program that implements a parity function and a possible analysis result for domain D_α.

5 Incremental Analysis of Programs with Assertions

Baseline Incremental Analysis Algorithm. We want to take advantage of the existing algorithms to design an analyzer that is sensible to changes in assertions also. We will use as a black box the combination of the algorithms to analyze incrementally a logic program [13], and the analyzer that is guided by assertions [8]. We will refer to it with the function $\mathscr{A}' = \textsc{IncAnalyze}(P, \mathscr{Q}_\alpha, \Delta_{Cls}, \mathscr{A})$, where the inputs are:

- A program $P = (Cls, As)$ with Cls a set of clauses and As a set of assertions.
- A set of changes Δ_{Cls} in the form of added or deleted clauses.
- A set $\mathscr{Q}_\alpha$ of initial queries that will be the starting point of the analyzer.
- A previous result of the algorithm $\mathscr{A}$ which is a well formed analysis graph.

The algorithm produces a new $\mathscr{A}'$ that correctly abstracts the behavior of the program reacting incrementally to changes in the clauses. It is parametric on the abstract domain D_α, given by implementing (1) the domain-dependent operations $\sqsubseteq, \sqcap, \sqcup$, `Aproj`$(\lambda, Vs)$, which restricts the abstract substitution to the set of variables Vs, `Aextend`$(P_{k,n}, \lambda^p, \lambda^s)$ propagates the success abstract substitution over the variables of $P_{k,n}$, λ^s to the substitution of the variables of the clause λ^p, `Acall`(λ, P, P_k) performs the abstract unification of predicate descriptor P with the head of the clause P_k, including in the new substitution abstract values for the variables in the body of clause P_k, and `Ageneralize`$(\lambda, \{\lambda_i\})$ performs the generalization of a set of abstract substitutions $\{\lambda_i\}$ and λ; and (2) *transfer functions* for program built-ins, that abstract the meaning of the basic operations of the language. Functions `apply_call`(P, λ^c, As) and `apply_succ`$(P, \lambda^c, \lambda^s, As)$ abstract the meaning of the assertion conditions (respectively calls and success conditions). Further details of these functions are described in Appendix A and in [8]. These operations are assumed to be monotonic and to correctly over-approximate their correspondent concrete version.

Operation of the Algorithm. The algorithm is centered around processing events. It starts by queueing a *newcall* event for each of the call patterns that need to be recomputed. This triggers `process`($newcall(\langle P, \lambda^c\rangle)$), which processes the clauses of predicate P. For each of them an *arc* event is added for the first literal. The `initial_guess` function returns a guess of the λ^s to $\langle P, \lambda^c\rangle$. If possible, it reuses the results in $\mathscr{A}$, otherwise returns $\bot$. Procedure `reanalyze_updated` propagates the information of new computed answers across the analysis graph by creating *arc* events with the literals from which the analysis has to be restarted. `process`($arc(\langle P_k, \lambda^c\rangle_{l,c} \xrightarrow{\lambda p} \langle P, \lambda^c\rangle)$) performs a single step of the left-to-right traversal of a clause body. First, the meaning of the assertion conditions of P is computed by `apply_call`. Then, if the literal $P_{k,i}$ is a *built-in*, its transfer function is computed; otherwise, an edge is added to $\mathscr{A}$ and the λ^s is looked up (a process that includes creating a *newcall* event for $\langle P, \lambda^c\rangle$ if the answer is not in the analysis graph). The answer is combined with the description λ^p from the literal immediately before $P_{k,i}$ to obtain the description (return) for the literal after $P_{k,i}$. This is used either to generate an *arc* event to process the next literal, or to update the answer of the predicate in `insert_answer_info`. This function combines the new answer with the semantics of any applicable assertions (in `apply_succ`), and the previous answers, propagating the new answer if needed.

Procedure `add_clauses` adds *arc* events for each of the new clauses. These trigger the analysis of each clause and the later update of $\mathscr{A}$ by using the edges in the graph.

The `delete_clauses` function selects the information to be kept in order to obtain the most precise semantics of the program, by removing all information which is potentially inaccurate (Fig. 2).

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

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

From [13] and [8] we have that:

Theorem 1 (Correctness of IncAnalyze from scratch). *Let P be a program, and $\mathscr{Q}_\alpha$ a set of abstract queries. Let $\mathscr{Q}$ be the set of concrete queries: $\mathscr{Q} = \{\langle G, \theta\rangle \mid \theta \in \gamma(\lambda) \wedge \langle G, \lambda\rangle \in \mathscr{Q}_\alpha\}$. The analysis result $\mathscr{A} = \text{IncAnalyze}(P, \mathscr{Q}_\alpha, \emptyset, \emptyset)$ for P with $\mathscr{Q}_\alpha$ is* correct *for $P, \mathscr{Q}$.*

Additionally, assertions ensure that certain executions never occur. This information is included in the analysis in the following way (adapted from [8]):

Theorem 2 (Applied assertion conditions). *Let P be a program, and $\mathscr{Q}_\alpha$ a set of abstract queries. Let $\mathscr{A} = \text{IncAnalyze}(P, \mathscr{Q}_\alpha, \emptyset, \emptyset)$.*

(a). The call assertion conditions cover all the inferred states:
$\forall \langle P, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}.\lambda^c \sqsubseteq$ `apply_call`$(P, \top, As)$.

```
IncAnalyze($P, \mathcal{Q}_\alpha, \Delta_{Cls}, \mathcal{A}$)
 1: for all $\langle P, \lambda^c\rangle \in \mathcal{Q}_\alpha$ do
 2:     add_event($newcall(\langle P, \lambda^c\rangle)$)
 3: if $\Delta_{Cls} = (Dels, Adds) \neq (\emptyset, \emptyset)$ then
 4:     delete_clauses($Dels$)
 5:     add_clauses($Adds$)
 6: analysis_loop()
 7: procedure analysis_loop()
 8:     while $E$ := next_event() do
 9:         process($E$)
10: procedure add_clauses($Cls$)
11:     for all $P_k$ :- $P_{k,1}, \ldots, P_{k,n_k} \in Cls$ do
12:         for all $\langle P, \lambda^c\rangle \mapsto \lambda^s \in \mathcal{A}$ do
13:             $\lambda^p$ := Acall($\lambda^c, P, P_k$)
14:             $\lambda^c_1$ := Aproj($\lambda^p, vars(P_{k,1})$)
15:             add_event($arc(\langle P, \lambda^c\rangle_{k,1} \xrightarrow{\lambda^p} \langle P_{k,1}, \lambda^c_1\rangle)$)
16: procedure delete_clauses($Cls$)
17:     $Calls$ := $\{\langle P, \lambda^c\rangle | \langle P, \lambda^c\rangle \in \mathcal{A}, (P_k$ :- $\ldots) \in Cls\}$
18:     $Ns$ := $\{N \in \mathcal{A} | N \rightsquigarrow C \in \mathcal{A}, C \in Calls\}$
19:     del($\mathcal{A}, Ns$)
20: function lookup_answer($\langle P, \lambda^c\rangle$)
21:     if $\langle P, \lambda^c\rangle \mapsto \lambda^s \in \mathcal{A}$ then
22:         return $\lambda^s$
23:     else
24:         add_event($newcall(\langle P, \lambda^a\rangle)$)
25:         return $\bot$
26: procedure reanalyze_updated($\langle P, \lambda^c\rangle$)
27:     for all $E$ := $\langle Q, \lambda^c_0\rangle_{k,i} \xrightarrow{\lambda^p} \langle P, \lambda^c\rangle \in \mathcal{A}$ do
28:         add_event($arc(E)$)
29: procedure process($newcall(\langle P, \lambda^c\rangle)$)
30:     for all $P_k$ :- $P_{k,1}, \ldots, P_{k,n_k} \in Cls$ do
31:         $\lambda^p$ := Acall($\lambda^c, P, P_k$)
32:         $\lambda^c_0$ := Aproj($\lambda^p, vars(P_{k,1})$)
33:         $Calls$ := $\{\lambda \mid \langle P, \lambda\rangle \in \mathcal{A}\}$
34:         $\lambda^c_1$ := Ageneralize($\lambda^c_0, Calls$)
35:         add_event($arc(\langle P, \lambda^c\rangle_{k,1} \xrightarrow{\lambda^p} \langle P_{k,1}, \lambda^c_1\rangle)$)
36:     $\lambda^s$ := initial_guess($\langle P, \lambda^c\rangle$)
37:     if $\lambda^s \neq \bot$ then
38:         reanalyze_updated($\langle P, \lambda^c\rangle$)
39:     upd($\mathcal{A}, \langle P, \lambda^c\rangle \leftarrow\!\!\shortmid \lambda^s$)
40: procedure process($arc(\langle P, \lambda^c_0\rangle_{k,i} \xrightarrow{\lambda^p} \langle Q, \lambda^c_1\rangle)$)
41:     $\lambda^a$ := apply_call($Q, \lambda^c_1, As$)
42:     if $P_{k,i}$ is a built-in then
43:         $\lambda^s_0$ := $f^\alpha(P_{k,i}, \lambda^a)$        ▷ Apply transfer function
44:     else $\lambda^s_0$ := lookup_answer($\langle Q, \lambda^a\rangle$)
45:     $\lambda^r$ := Aextend($\lambda^p, \lambda^s_0$)
46:     upd($\mathcal{A}, \langle P, \lambda^c_0\rangle_{k,i} \xrightarrow[\lambda^r]{\lambda^p} \langle Q, \lambda^a\rangle$)
47:     if $\lambda^r \neq \bot$ and $i \neq n_k$ then
48:         $\lambda^c_2$ := Aproj($\lambda^r, vars(P_{k,i+1})$)
49:         add_event($arc(\langle H, \lambda^c_0\rangle_{k,i+1} \xrightarrow{\lambda^r} \langle B, \lambda^c_2\rangle)$)
50:     else if $\lambda^r \neq \bot$ and $i = n_k$ then
51:         $\lambda^s$ := Aproj($\lambda^r, vars(P_k)$)
52:         insert_answer_info($\langle P, \lambda^c_0\rangle, \lambda^s$)
53: procedure insert_answer_info($\langle P, \lambda^c\rangle, \lambda^s$)
54:     $\lambda^a$ := apply_succ($P, \lambda^c, \lambda^s, As$)
55:     if $\langle P, \lambda^c\rangle \mapsto \lambda^s_0 \in \mathcal{A}$ then
56:         $\lambda^s_1$ := Ageneralize($\lambda^a, \{\lambda^s_0\}$)
57:     else $\lambda^s_1$ := $\bot$
58:     if $\lambda^s_0 \neq \lambda^s_1$ then
59:         upd($\mathcal{A}, \langle P, \lambda^c\rangle \leftarrow\!\!\shortmid \lambda^s_1$)
60:         reanalyze_updated($\langle P, \lambda^c\rangle$)
```

Fig. 2. The generic context-sensitive, incremental fixpoint algorithm using (not changing) assertion conditions.

```
 1: function IAwAC($(Cls, As), \Delta_{Cls}, \Delta_{As}, \mathcal{Q}_\alpha, \mathcal{A}$)
 2:     $R$ := preprocess($Cls, As, \mathcal{A}$)
 3:     $\mathcal{A}'$ := IncAnalyze($(Cls, As), \mathcal{Q}_\alpha \cup R, \Delta_{Cls}, \mathcal{A}$)
 4:     del ($\mathcal{A}', \{E \mid E \in \mathcal{A}' \wedge Q \not\rightsquigarrow E \wedge Q \in \mathcal{Q}_\alpha\}$)
 5:     return $\mathcal{A}'$
 6: function preprocess($Cls, As, \mathcal{A}$)
 7:     $R$ := $\emptyset$
 8:     for each $P \in Cls$ do
 9:         if $\Delta_{As}[P] \neq \emptyset$ then
10:             $R$ := $R \cup$ update_calls_pred($P, As, \mathcal{A}$)
11:             $R$ := $R \cup$ update_success_pred($P, As, \mathcal{A}$)
12:     return $R$
```

Fig. 3. High-level view of the proposed algorithm

(b). The inferred abstract success states are covered by the success assertion conditions: $\forall \langle P, \lambda^c\rangle \mapsto \lambda^s \in \mathcal{A}.\ \lambda^s \sqsubseteq \texttt{apply\_succ}(P, \lambda^c, \top, As)$

We introduce a new proposition about the algorithm that will be of use later.

Proposition 1 (Correctness starting from a partial analysis). *Let P be a program, $\mathscr{Q}_\alpha$ a set of abstract queries, and $\mathscr{A}_0$ any analysis graph. Let $\mathscr{A} = \textsc{IncAnalyze}(P, \mathscr{Q}_\alpha, \Delta_{Cls}, \mathscr{A}_0)$. $\mathscr{A}$ is correct for P and a query $Q \in \mathscr{Q}_\alpha$ if for any node $N \in \mathscr{A}_0$ such that there is a path $Q \rightsquigarrow N$ in $[\![P]\!]_{\gamma(Q)}$, $N \in \mathscr{Q}_\alpha$.*

Proof. This follows from the creation of a *newcall* event for each of the queries, which will trigger the recomputation and later update of all the nodes of the analysis graph that are potentially under the fixpoint.

Note that here we are not assuming that $\mathscr{A}_0$ is the (correct) output of a previous analysis, it can be any partial analysis (below the fixpoint).

5.1 The Incremental Analyzer of Programs with Assertions

We propose to inspect and update the analysis graph to guarantee that a call to IncAnalyze produces results that are correct and precise. We call this new analyzer IAwAC, short for IncAnalyze-w/AssrtChanges (Fig. 3). The preprocess phase consists in inspecting all the literals affected by the changes in the assertions, collecting which call patterns need to be reanalyzed by the incremental analysis, i.e., it may be different from the set of initial queries $\mathscr{Q}$ originally requested by the user. In addition, after the analysis phase, the unreachable abstract calls that were safe to reuse may not be reachable anymore, so they need to be removed from the analysis result.

Detecting the Affected Parts in the Analysis Results. The steps to find potential changes in the analysis results when assertions are changed are detailed in Fig. 4 with procedures `update_calls_pred` and `update_successes_pred`. The goal is to identify which edges and nodes of the analysis graph are not precise or correct. Since assertions may affect the inferred call or the inferred success of predicates, we have split the procedure into two functions. However, the overall idea is to obtain the current substitution, which encodes the semantics of the assertions in the previous version of the program, and the abstract substitution that would have been inferred if no assertions were present. Then functions `apply_call` and `apply_success` obtain the meaning of the new assertions. Finally, we call a general procedure to treat the potential changes, `treat_change` (see Fig. 5). Specifically, in the case of `call` conditions, we review all the program points from which it is called, by checking the incoming edges of the nodes of that predicate. For each node we project the substitution of the clause (λ^p) to the variables of the literal to obtain the call patterns if no assertions would be specified (line 4). We then detect if the call pattern produced by the new meaning of the assertions already existed in the analysis graph to reuse its result, and, last, we call the procedure to treat the change. In the case of `success` conditions we obtain the substitution including the new meaning of the assertion by joining the return substitution at the last literal of each of the clauses of the predicate, previously projected to the variables of the head (line 16).

Amending the Analysis Results. The procedure `treat_change` (Fig. 5), given an edge that points to a literal whose success potentially changed, updates the

```
1: function update_calls_pred($P, As, \mathscr{A}$)
2:   $Q := \emptyset$
3:   for each $\langle P', \lambda\rangle_{c,l} \xrightarrow[\bullet]{\lambda^p} \langle P, \lambda^c_{old}\rangle \in \mathscr{A}$ do
4:     $\lambda^c := \sigma(\texttt{Aproj}(\lambda^p, vars(P'_{c,l}))$ s.t. $\sigma(P'_{c,l}) = P$
5:     $\lambda^c_{new} :=$ apply_call($P, \lambda^c, As$)
6:     if $\exists\langle P', \lambda^c{}_{new}\rangle \mapsto \lambda^s \in \mathscr{A}$ then
7:       $\lambda^{s\prime} := \lambda^s$
8:     else $\lambda^{s\prime} := \bot$
9:     $Q := Q \cup$ treat_change($\langle P', \lambda\rangle_{c,l} \xrightarrow[\lambda^r]{\lambda^p} \langle P, \lambda^c{}_{new}\rangle, \lambda^{s\prime}, \mathscr{A}$)
10:  return $Q$

11: function update_successes_pred($P, As, \mathscr{A}$)
12:   $Q := \emptyset$
13:   for each $\langle P, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}$ do
14:     $\lambda := \bot$
15:     for each $\langle P, \lambda^c\rangle_{c,last} \xrightarrow[\lambda^r]{\bullet} \langle Q, \lambda\rangle \in \mathscr{A}$ do
16:       $\lambda := \lambda \sqcup$ apply_succ($P, \lambda^c, \texttt{Aproj}(\lambda^r, vars(P_c)), As$)
17:     for each $E = N_{\bullet,\bullet} \xrightarrow[\bullet]{\bullet} \langle P, \lambda^c\rangle \in \mathscr{A}$ do
18:       $Q := Q \cup$ treat_change ($E, \lambda, \mathscr{A}$)
19:   return $Q$
```

Fig. 4. Changes in assertions (split by assertion conditions)

analysis result, and decides which predicates and call patterns need to be recomputed. After updating the annotation of the edge (line 4), we study how the abstract substitution changed. If the new substitution ($\lambda^{r\prime}$) is more general than the previous one (λ^r), this means that the previous assertions where pruning more concrete states than the new one, and, thus, this call pattern needs to be reanalyzed. Else, if $\lambda^r \not\sqsubseteq \lambda^{r\prime}$, i.e., the new abstract substitution is more concrete or incompatible, some parts of the analysis graph may not be accurate. Therefore, we have to eliminate from the graph the literals that were affected by the change (i.e., the literals following the program point with a change) and all the dependent code from this call pattern. Also, the analysis has to be restarted from the original entry points that were affected by the deletion of these potentially imprecise nodes. In the last case (line 3) the old and the new substitutions are the same, and, thus, nothing needs to be reanalyzed (the $\emptyset$ is returned).

```
1: function treat_change($\langle P, \lambda\rangle_{c,l} \xrightarrow[\lambda^r]{\lambda^p} \langle Q, \lambda^c\rangle, \lambda^s, \mathscr{A}$)
2:   $\lambda^{r\prime} :=$ Aextend($\lambda^p, \lambda^s$)          ▷ Obtain new abstraction at literal return
3:   if $\lambda^r = \lambda^{r\prime}$ then return $\emptyset$
4:   del($\mathscr{A}, \langle P, \lambda\rangle_{c,l} \xrightarrow[\bullet]{\bullet} \bullet$)
5:   add($\mathscr{A}, \langle P, \lambda\rangle_{c,l} \xrightarrow[\lambda^{r\prime}]{\lambda^p} \langle Q, \lambda^c\rangle$)          ▷ Update the analysis graph
6:   if $\lambda^r \sqsubset \lambda^{r\prime}$ then return $\{\langle P, \lambda\rangle\}$
7:   else if $\lambda^r \not\sqsubseteq \lambda^{r\prime}$ then          ▷ Analysis is potentially imprecise
8:     $Lits := \{E \mid E = \langle P, \lambda\rangle_{c,i} \longrightarrow N \in \mathscr{A} \land i > l\}$
9:     $IN := \{E \mid E \rightsquigarrow L \in \mathscr{A} \land L \in Lits\}$          ▷ Potentially imprecise nodes
10:    del($\mathscr{A}, IN$)
11:    return $IN$
```

Fig. 5. Procedure to determine how the analysis result needs to be recomputed.

Correctness of the Algorithm

Proposition 2 (IAwAC from scratch). *Let P be a program, $\mathscr{Q}_\alpha$ a set of abstract queries. Let $\mathscr{Q}$ be the set of concrete queries: $\mathscr{Q} = \{\langle G, \theta \rangle \mid \theta \in \gamma(\lambda) \wedge \langle G, \lambda \rangle \in \mathscr{Q}_\alpha\}$. The analysis $\mathscr{A} = \mathrm{IAwAC}(P, \mathscr{Q}_\alpha, \emptyset, \emptyset, \emptyset)$ for P with $\mathscr{Q}_\alpha$ is* correct *for $P, \mathscr{Q}$.*

Proof. Since the preprocessing phase only modifies information that is already in the initial analysis and it is empty, correctness follows from Theorem 1.

In terms of precision, we want to ensure that the meaning of the new assertions is precisely included in the analysis result.

Proposition 3 (Precision after `update_calls_pred`). *Let Cls be a set of clauses, As be a set of assertions, and $\mathscr{A}$ any analysis graph. For any predicate G of Cls, let $\mathscr{A}'$ be the state of $\mathscr{A}$ after* `update_calls_pred`$(G, As, \mathscr{A})$*. Then, for any $\langle G, \lambda^c \rangle \mapsto \lambda^s \in \mathscr{A}'.\lambda^c \sqsubseteq$* `apply_call`$(G, \top, As)$.

Proof. Given a predicate G, the function `update_calls_pred` looks at each edge that finishes in a node G, and obtains the new meaning of the conditions (line 5). Then, in line 4 of `treat_change`, the node is removed if it is different. Because `apply_call` is assumed to be monotonic, for any λ^c. `apply_call`$(G, \lambda^c, As) \sqsubseteq$ `apply_call`$(G, \top, As)$.

Proposition 4 (Precision after `update_successes_pred`). *Under the conditions of Proposition 3, for any predicate G of Cls, let $\mathscr{A}'$ be the state of $\mathscr{A}$ after* `update_successes_pred`$(G, As, \mathscr{A})$*. Then, for any $\langle G, \lambda^c \rangle \mapsto \lambda^s \in \mathscr{A}'.\lambda^s \sqsubseteq$* `apply_succ`$(G, \lambda^c, \top, As)$.

Proof. Given a predicate G, the function `update_successes_pred` looks at the last literal of each clause of G, and obtains the new meaning of the conditions (line 16). Then, in line 4 of `treat_change`, the node is removed if it is different. Because `apply_call` is assumed to be monotonic, for any pair (λ^c, λ^s). `apply_succ`$(G, \lambda^c, \lambda^s, As) \sqsubseteq$ `apply_call`$(G, \lambda^c, \lambda^s, As)$.

As shown in Proposition 1, given any partial analysis result, we can ensure correctness of the reanalysis if we guarantee that all literals that need to be reanalyzed are included in $\mathscr{Q}_\alpha$. We want to show that the set Q of queries collected in `treat_change` is enough to guarantee the correctness of the result.

Proposition 5 (Queries collected in preprocess). *Let $P = (Cls, As_0)$ be a program, $\mathscr{Q}_\alpha$ a set of abstract queries. Let $\mathscr{Q}$ be the set of concrete queries: $\mathscr{Q} = \{\langle G, \theta \rangle \mid \theta \in \gamma(\lambda) \wedge \langle G, \lambda \rangle \in \mathscr{Q}_\alpha\}$. Let $\mathscr{A} = \mathrm{IAwAC}(P, \mathscr{Q}_\alpha, \emptyset, \emptyset, \emptyset)$ be the correct analysis for $P, \mathscr{Q}$. If P changes to $P' = (Cls, As)$, ${\mathscr{Q}_\alpha}' = \textsc{preprocess}(Cls, As, \mathscr{A})$ guarantees that $\mathscr{A}' = \textsc{IncAnalyze}((Cls, As), {\mathscr{Q}_\alpha}', \emptyset, \mathscr{A})$ is correct for P' and $\mathscr{Q}$.*

Proof. We split the proof into two cases: (a) The assertions change only for one predicate: because $\mathscr{A}$ is correct, by Theorem 1, since $\mathscr{A}$ is an over-approximation of $[\![P]\!]_{\mathscr{Q}}$, and Proposition 1 is true.

(b) The assertions change for more than one predicate: after processing the first predicate $\mathscr{A}$ may not be correct, as `treat_change` removes nodes. However, every node that is removed is added to the set of queries. This means that the nodes that are unreachable when processing the following predicates were already stored before, and therefore, Proposition 1 also holds.

Theorem 3 collects all correctness and precision properties of the algorithm.

Theorem 3 (Correctness of IAwAC). *Let P_0 and $P = (Cls, As)$ be programs that differ in Δ_{Cls} and Δ_{As}, $\mathscr{Q}_\alpha$ be a set of abstract queries. Let $\mathscr{Q}$ be the set of concrete queries $\mathscr{Q} = \{\langle G, \theta\rangle \mid \theta \in \gamma(\lambda) \land \langle G, \lambda\rangle \in \mathscr{Q}_\alpha\}$. Given $\mathscr{A}_0 = \mathrm{IAwAC}(P_0, \mathscr{Q}_\alpha, \emptyset, \emptyset, \emptyset)$, and the analysis $\mathscr{A} = \mathrm{IAwAC}(P, \mathscr{Q}_\alpha, \Delta_{Cls}, \Delta_{As}, \mathscr{A}_0)$.*

(a). $\mathscr{A}$ is correct *or P and $\mathscr{Q}$.*
(b). $\forall \langle G, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}.\lambda^c \sqsubseteq \texttt{apply\_call}(G, \top, As)$.
(c). $\forall \langle G, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}.\ \lambda^s \sqsubseteq \texttt{apply\_succ}(G, \lambda^c, \top, As)$

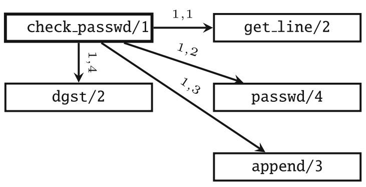
Proof. (a) follows from Theorem 2 and Proposition 5. (b) and (c) follow from Lemma 2 and Propositions 3 and 4.

5.2 Use Cases

We show some examples of the algorithm. We assume that we analyze with a shape domain in which the properties in the assertions can be exactly represented.

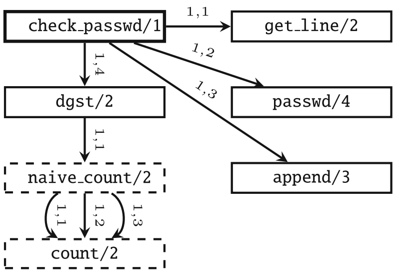
Example 2 (Reusing a preanalyzed generic program). Consider a slightly modified version the program that checks a password as shown earlier, that only allows the user to write passwords with *lowercase* letters. Until we have a concrete implementation for the hasher we will not be able to analyze precisely this program. However, we can preanalyze it by using the information of the assertion of the trait to obtain the following simplified analysis graph:

```
:- trait hasher { :- pred dgst(Str, Digest)
        : lowercase(Str) => int(Digest). }.

check_passwd(User) :-
    get_line(Plain),
    passwd(User,Hasher,Digest,Salt),
    append(Plain,Salt,Salted),
    (Hasher as hasher).dgst(Salted,Digest).

passwd(don,xor8,0x6d,"eNfwuBhtN9CUHxg==").
```



The node for `dgst/2` represents the call $\langle \texttt{dgst(S,D)}, (S/\texttt{lowercase}, D/\texttt{num})\rangle \mapsto (S/\texttt{lowercase}, D/\texttt{int})$, in this case, `D` was inferred to be a *number* because of the success of `passwd/4`. If we add a very naive implementation that consists on counting the number of some letters in the password, reanalyzing will cause adding to the graph some new nodes, shown with a dashed line:

```
:- impl(hasher, naive/0).
(naive as hasher).dgst(Str, Digest) :-
    naive_count(Xs, 0, Digest).

naive_count(L, D0, D) :-
    count(L,'a',Na), D1 is D0 + Na*1,
    count(L,'b',Nb), D2 is D1 + Nb*2,
    count(L,'c',Nc), D3 is D2 + Nc*3,
```



We detect that none of the previous nodes need to be recomputed due to tracking dependencies for each literal. The analysis was performed by going directly to the program point of `dgst/2` and inspecting the new clause (that was generated automatically by the translation) that calls `naive_count/2`. By analyzing `naive_count/2` we obtain nodes $\langle$`naive_count(S,D)`$, (S/$`lowercase`$, D/$`num`$)\rangle \mapsto (S/$`lowercase`$, D/$`int`$)$, and $\langle$`count(L,C,N)`$, (S/$`lowercase`$, C/$`char`$)\rangle \mapsto (S/$`lowercase`$, C/$`char`$, N/$`int`$)$. As no information needs to be propagated because the head does not contain any of the variables of the call to digest, we are done, and we avoided reanalyzing any caller to `check_passwd/2`, if existed.

Example 3 (Weakening assertion properties). Consider the program and analysis result of Example 2. We realize that allowing the user to write a password only with lowercase letters is not very secure. We can change the assertion of the trait to allow any string as a valid password.

```
:- trait hasher { :- pred dgst(Str, Digest) : string(Str) => int(Digest). }.
```

When reanalyzing, node $\langle$`dgst(S,D)`$, (S/$`lowercase`$, D/$`num`$)\rangle$ will disappear to become $\langle$`dgst(S,D)`$, (S/$`string`$, D/$`num`$)\rangle$, and the same for `naive_count/3`. A new call pattern will appear for `count/3` $\langle$`count(L,C,N)`$, (S/$`string`$, C/$`char`$)\rangle \mapsto (S/$`string`$, C/$`char`$, N/$`int`$)$, leading to the same result for `dgst/2`. I.e., we only had to partially analyze the library, instead of the whole program.

Table 1. Analysis time for `LPdoc` adding one backend at a time (time in seconds).

Domain	No backend	+ `texinfo`	+ `man`	+ `html`
`reachability`	1.7	2.1	3.4	3.9
`reachability` *inc*	1.7	1.2	1.0	1.6
`gr`	2.1	2.2	2.3	2.6
`gr` *inc*	2.1	1.4	0.9	1.8
`def`	6.0	7.1	7.8	9.7
`def` *inc*	6.0	2.2	1.3	3.5
`sharing`	27.2	28.1	24.2	28.5
`sharing` *inc*	27.2	3.9	1.4	5.1

6 Experiments

We have implemented the proposed analysis algorithm within the `CiaoPP` system [9] and performed some preliminary experiments to test the use case described in Example 2. Our test case is the `LPdoc` documentation generator tool [10,11], which takes a set of Prolog files with assertions and machine-readable comments and generates a reference manual from them. `LPdoc` consists of around 150 files, of mostly (`Ciao`) Prolog code,with assertions (most of which, when written, were only meant for documentation generation), as well as some auxiliary scripts in Lisp, JavaScript, bash, etc. The Prolog code analyzed is about 22 K lines. This is a tool in everyday use that generates for example all the manuals and web sites for the `Ciao` system (http://ciao-lang.org, http://ciao-lang.org/documentation.html) and as well as for all the different *bundles* developed internal or externally, processing around 20 K files and around 1M lines of Prolog and interfaces to another 1M lines of C and other miscellaneous code). The `LPdoc` code has also been adapted as the documentation generator for the XSB system [24].

`LPdoc` is specially relevant in our context because it includes a number of backends in order to generate the documentation in different formats such as **`texinfo`**, Unix **`man`** format, **`html`**, **`ascii`**, etc. The front end of the tool generates a documentation tree with all the content and formatting information and this is passed to one out of a number of these backends, which then does the actual, specialized generation in the corresponding typesetting language. We analyzed all the `LPdoc` code with a **`reachability`** domain, a groundness domain (`gr`), a domain tracking dependencies via propositional clauses [6] (`def`), and a **`sharing`** domain with cliques [19]. The experiment consisted in preanalyzing the tool with no backends and then adding incrementally the backends one by one. In Table 1 we show how much time it took to analyze in each setting, i.e., for the different domains and with the incremental algorithm or analyzing from scratch. The experiments were run on a MacBook Pro with an Intel Core i5 2.7 GHz processor, 8GB of RAM, and an SSD disk. These preliminary results support our hypothesis that the proposed incremental analysis brings performance advantages when dealing with these use cases of generic code.

7 Related Work

Languages like C++ require specializing all parametric polymorphic code (e.g., templates [25]) to monomorphic variants. While this is more restrictive than *runtime polymorphism* (variants must be statically known at compile time), it solves the analysis precision problem, but not without additional costs. First, it is known to be slow, as templates must be instantiated, reanalized, and recompiled for each compilation unit. Second, it produces many duplicates which must be removed later by the linker. Rust [15] takes a similar approach for *unboxed* types.

Runtime polymorphism or dynamic dispatch can be used in C++ (virtual methods), Rust (boxed traits), Go [5] (interfaces), or Haskell's [14] type classes.

However, in this case compilers and analyzers do not usually consider the particular instances, except when a single one can be deduced (e.g., in C++ devirtualization [18]).

Mora et al. [17] perform modular symbolic execution to prove that some (versions of) libraries are equivalent with respect to the same client. Chatterjee et al. [3] analyze libraries in the presence of callbacks incrementally for data dependence analysis. I.e., they preanalyze the libraries and when a client uses it reuses the analysis and adds incrementally possible calls made by the client. We argue that when using our Horn clause encoding, both high analysis precision and compiler optimizations can be achieved more generally by combining the incremental static global analysis that we have proposed with abstract specialization [20].

8 Conclusions

While logic programming can intrinsically handle generic programming, we have illustrated a number of problems that appear when handling generic code with the standard solutions provided by current (C)LP module systems, namely, using multifile predicates. We argue that the proposed traits are a convenient and elegant abstraction for modular generic programming, and that our preliminary results support the conclusion that the novel incremental analysis proposed brings promising analysis performance advantages for this type of code. Our encoding is very close to the underlying mechanisms used in other languages for implementing dynamic dispatch or run-time polymorphism (like Go's interfaces, Rust's traits, or a limited form of Haskell's type clases), so we believe that our techniques and results can be generalized to other languages. This also includes our proposed algorithm for incremental analysis with assertion changes, which can be applied to different languages through the standard technique of translation to Horn-clause representation [16]. Traits are also related to higher-order code (e.g., a "callable" trait with a single "call" method). We also claim that our work contributes to the specification and analysis of higher-order code.

A Assertions

Assertions may not be exactly represented in the abstract domain used by the analyzer. We recall some definitions (adapted from [22]) which are instrumental to correctly approximate the properties of the assertions during the analysis (Fig. 6).

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:*

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

```
global flag: speed-up
1: function apply_call($P, \lambda^c$)
2:   if $\exists \sigma, \lambda^t = \lambda^+_{TS(\sigma(Pre),P)}$ s.t. calls($H, Pre$) $\in C, \sigma(H) = P$ then
3:     if speed-up return $\lambda^t$ else return $\lambda^c \sqcap \lambda^t$
4:   else return $\lambda^c$
5: function apply_succ($P, \lambda^c, \lambda^{s_0}$)
6:   $app = \{\lambda \mid \exists\ \sigma,$ success$(H, Pre, Post) \in C, \sigma(H) = P,$
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 $\lambda^t$ else return $\lambda^t \sqcap \lambda^{s_0}$
11:  else return $\lambda^{s_0}$
```

Fig. 6. Applying assertions.

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:

```
list([]).
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 Definition 3:

Definition 4 (Abstract Trivial Success Subset of a Property Formula). *Under the same conditions of Definition 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 $\gamma(\lambda^-_{TS(L,P)}) \subseteq TS(L, P)$.*

Definition 5 (Abstract Trivial Success Superset of a Property Formula). *Under the same conditions of Definition 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.

References

1. Bruynooghe, M.: A practical framework for the abstract interpretation of logic programs. J. Logic Program. **10**, 91–124 (1991)

2. Cabeza, D., Hermenegildo, M.: A new module system for prolog. In: Lloyd, J., et al. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 131–148. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44957-4_9
3. Chatterjee, K., Choudhary, B., Pavlogiannis, A.: Optimal Dyck reachability for data-dependence and alias analysis. PACMPL **2**(POPL), 1–30 (2018)
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of POPL 1977, pp. 238–252. ACM Press (1977)
5. Donovan, A.A.A., Kernighan, B.W.: The Go Programming Language, Professional Computing. Addison-Wesley, Boston (2015)
6. Dumortier, V., Janssens, G., Simoens, W., García de la Banda, M.: Combining a definiteness and a freeness abstraction for CLP languages. In: Workshop on LP Synthesis and Transformation (1993)
7. Flanagan, C.: Hybrid type checking. In: 33rd ACM Symposium on Principles of Programming Languages (POPL 2006), pp. 245–256, January 2006
8. Garcia-Contreras, I., Morales, J.F., Hermenegildo, M.V.: Multivariant assertion-based guidance in abstract interpretation. In: Mesnard, F., Stuckey, P.J. (eds.) LOPSTR 2018. LNCS, vol. 11408, pp. 184–201. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-13838-7_11
9. Hermenegildo, M., Puebla, G., Bueno, F., Lopez-Garcia, P.: Integrated program debugging, verification, and optimization using abstract interpretation (and The Ciao System Preprocessor). Sci. Comput. Program. 58(1–2), 115–140 (2005)
10. Hermenegildo, M.: A documentation generator for (C)LP systems. In: Lloyd, J., et al. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 1345–1361. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44957-4_90
11. Hermenegildo, M.V., Morales, J.: The LPdoc documentation generator. Ref. Manual (v3.0). Technical report, July 2011. http://ciao-lang.org
12. Hermenegildo, M.V., Puebla, G., Bueno, F.: Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging. In: Apt, K.R., Marek, V.W., Truszczynski, M., Warren, D.S. (eds.) The Logic Programming Paradigm, pp. 161–192. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-642-60085-2_7
13. Hermenegildo, M.V., Puebla, G., Marriott, K., Stuckey, P.: Incremental analysis of constraint logic programs. ACM TOPLAS **22**(2), 187–223 (2000)
14. Hudak, P., et al.: Report on the programming language Haskell. Haskell special issue. ACM SIGPLAN Not. **27**(5), 1–164 (1992)
15. Klabnik, S., Nichols, C.: The Rust Programming Language. No Starch Press, San Francisco (2018)
16. Méndez-Lojo, M., Navas, J., Hermenegildo, M.V.: A flexible, (C)LP-based approach to the analysis of object-oriented programs. In: King, A. (ed.) LOPSTR 2007. LNCS, vol. 4915, pp. 154–168. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78769-3_11
17. Mora, F., Li, Y., Rubin, J., Chechik, M.: Client-specific equivalence checking. In: 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 441–451. ASE (2018)
18. Namolaru, M.: Devirtualization in GCC. In: Proceedings of the GCC Developers' Summit, pp. 125–133 (2006)
19. Navas, J., Bueno, F., Hermenegildo, M.: Efficient top-down set-sharing analysis using cliques. In: Van Hentenryck, P. (ed.) PADL 2006. LNCS, vol. 3819, pp. 183–198. Springer, Heidelberg (2005). https://doi.org/10.1007/11603023_13

20. Puebla, G., Albert, E., Hermenegildo, M.: Abstract interpretation with specialized definitions. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 107–126. Springer, Heidelberg (2006). https://doi.org/10.1007/11823230_8
21. Puebla, G., Bueno, F., Hermenegildo, M.: An assertion language for constraint logic programs. In: Deransart, P., Hermenegildo, M.V., Małuszynski, J. (eds.) Analysis and Visualization Tools for Constraint Programming. LNCS, vol. 1870, pp. 23–61. Springer, Heidelberg (2000). https://doi.org/10.1007/10722311_2
22. Puebla, G., Bueno, F., Hermenegildo, M.: Combined static and dynamic assertion-based debugging of constraint logic programs. In: Bossi, A. (ed.) LOPSTR 1999. LNCS, vol. 1817, pp. 273–292. Springer, Heidelberg (2000). https://doi.org/10.1007/10720327_16
23. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop, pp. 81–92 (2006)
24. Swift, T., Warren, D.: XSB: extending prolog with tabled logic programming. TPLP **12**(1–2), 157–187 (2012). https://doi.org/10.1017/S1471068411000500
25. Vandevoorde, D., Josuttis, N.M.: C++ Templates. Addison-Wesley Longman Publishing Co. Inc., Boston (2002)