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
[2] Universidad Politécnica de Madrid (UPM)

Abstract. *Generic* components represent a further abstraction over the concept of modules, which introduces dependencies on other (not necessarily available) components implementing specified interfaces. It has become a key concept in large and complex software applications. Despite its undeniable advantages, generic code is known to be *anti-modular*. Precise analysis (e.g., for detecting bugs or optimizing code) requires such code to be instantiated with concrete implementations, potentially leading to a prohibitively 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* 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 the conclusion that the new incrementality-related features added to the analysis bring promising analysis performance advantages.

Keywords: Incremental Static Analysis · Assertions · Logic Programming · Generic Code · Specifications · Abstract Interpretation

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 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.

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.

* 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.

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 (also with specs), and as a placeholder for different implementations of a given functionality, in general referred to as *generic code*.

Despite its undeniable advantages, generic code is known to be in fact *anti-modular*, and the analysis of generic code poses challenges: code is not fully available and interface specifications may not be descriptive enough to verify the specifications for the whole application (e.g., proving termination if the interface specification does not enforce termination properties). 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 (widening) 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. At the same time, when considering a closed set of interface implementations, it may also be desirable to analyze the whole application together with these implementations, allowing the transfer of information with "native analysis precision" (e.g., multivariance) and specializations across the interfaces. This allows getting the most precise information, specializations, compiler optimizations, etc., but at a higher cost.

In this paper we claim that incremental (whole program) analysis can be very beneficial in this context. After providing the necessary notation and background (Section 2), we start by proposing a simple Horn-clause encoding of generic code, using *open* predicates and assertions, and introduce a novel extension for logic programming (*traits*) that is translated using open predicates (Section 3). 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. We also introduce a new *incremental* analysis algorithm (Sections 4 and 5) that, in addition to supporting and taking advantage of assertions during analysis, i.e., as part of the fixpoint calculation, offers two interesting properties: it 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, and it also *supports natively open predicates*.

Generic code offers many opportunities for this new analysis technique. For example: standalone analysis of trait-based code without particular implementations by using the (trust) assertions in the interfaces; refinement of standalone analysis for particular implementations; or reuse of analysis results when more implementations are made available. Note that fine-grained incremental analysis seems even more interesting when using generic code, where the scope of a program change may implicitly be scattered across many modules. We study a number of use cases (Section 5.2), 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, in Section 6 we provide preliminary results from the application of a prototype im-

plementation of the proposed techniques in a realistic case study: the analysis of the LPdoc documentation system and its multiple backends for generating documentation in the different formats. Finally, Section 7 discusses some related work and Section 8 presents our conclusions.

2 Background

Logic Programs. A *definite Logic 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*. 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*). We will use P and Q to denote predicates. 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 `append` predicate:

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

$\texttt{app}_1$ will denote the head of the first clause of `app/3`, $\texttt{app}_{2,1}$ will denote the first literal of the second clause of `app`, 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 [2, 12, 19], 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, 20]. A `pred` assertion is of the form:

`:- 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 [19]. *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, in other words, 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 $\texttt{calls}(Head, Pre)$[3] 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 [3]. An interesting property, specially for program analysis, is that we can distinguish between *open* and *closed* predicates. [4] Closed predicates within a module are those whose complete definition is available in the module. In contrast, open predicates (traditionally declared as `multifile` in many Prolog systems) are only partially defined within a given module, and different clauses can be scattered across different modules, and thus the complete definition is not known until all the application modules are linked (which is basically "anti-modular").

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
```

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

[4] We only consider *static* predicates and modules. Dynamic predicates whose definition may change during execution, or modules that are dynamically changed (loaded/unloaded) at execution time can also be dealt with, using various techniques, and in particular the incremental analysis proposed, but for space reasons we limit the discussion to static predicates.

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. We propose a syntactic extension for defining interfaces, or *traits*, in logic programs, which captures the essential mechanisms necessary for writing generic code, but does not require the introduction of a static type system (beyond the *typing* that modules and their interfaces already represents).

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. 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 trait
:- multifile 'T.p'/(n + 1).
:- pred 'T.p'(X, A_1, ..., A_n) : ... => ... .
% call to p/n for X implementing T
... :- ..., 'T.p'(X, A_1, ..., A_n), ... % (X as T).p(A_1, ..., A_n)
```

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` (an atom in this case, although trait syntax allows arbitrary functors) implements a `hasher`, and provides an implementation for the `dgst/2` predicate (head `(xor8 as hasher).dgst(Str, Digest)`).

The translation rules to plain predicates are as follows:

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

% bridge from interface open predicate to 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

After introducing our generic code abstraction, we now describe our proposed incremental analysis to support generic code evolution (i.e., code with traits). We start by recalling in this section the base goal-dependent abstract interpretation algorithm, and in Section 5 we describe the proposed incremental version.

4.1 Preliminaries

Program Analysis with Abstract Interpretation. Our approach is based on *abstract interpretation* [5], a technique in which 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 operations on D values are (over-)approximated by corresponding abstract operations on D_α values.

Concrete Semantics. We use 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 $\langle G, \theta\rangle$ with G an atom and θ a substitution over the variables of G. Executing (answering) a query with respect to a logic program consists on determining whether the query is a logical consequence of the program and for which substitutions (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 [1]. 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 results 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 substitution (state) for that call, θ^c, and the corresponding success substitution θ^s (answer). 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.

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 the notation $x \in P$ to denote that a node n appears in P, and $e \in P$ to denote that an edge e appears in P.

4.2 Goal-dependent abstract interpretation.

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 $\mathscr{Q} = \{\langle A_i, {\lambda^c}_i \rangle\}$, 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. $\langle G, \lambda \rangle \in \mathscr{Q} \wedge \theta \in \gamma(\lambda)$.

Analysis graphs. We will use graphs to overapproximate all possible executions of a program given an initial query. Each node in our graph is identified by a pair (P, λ) with P a predicate descriptor and $\lambda \in D_\alpha$, an element of the abstract domain, which represents that (a possibly infinite set of) calls to a predicate may occur. 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 $\lambda^c, \lambda^s \in D_\alpha$. For a given predicate P, the analysis graph can contain a number of nodes capturing different call situations. As usual, $\bot$ denotes the abstract description such that $\gamma(\bot) = \emptyset$. A call mapped to $\bot$ ($\langle A, \lambda^c \rangle \mapsto \bot$) indicates that calls to predicate A with description $\theta \in \gamma(\lambda^c)$ either fail or loop, i.e., they never succeed.

Each edge $\langle P, \lambda_1 \rangle_{c,l} \xrightarrow[\lambda^r]{\lambda^p} \langle Q, \lambda_2 \rangle$ in the graph represents a call dependency among two predicates. It represents that *calling predicate P with calling pattern λ_1 causes predicate Q (literal l of clause c) to be called with calling pattern λ_2*. It is annotated with the abstract element representing the context of the call (λ^p) and the return (λ^r) in the body of clause c and literal l of predicate P. Note that these values are introduced to ease the presentation of the algorithm, however 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 that are not relevant for the operations that we are considering. Note also that if we have 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: $\langle P, \lambda_1 \rangle_{c,l+1} \xrightarrow[\bullet]{\lambda} \langle Q', \lambda_2' \rangle$. Fig. 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.

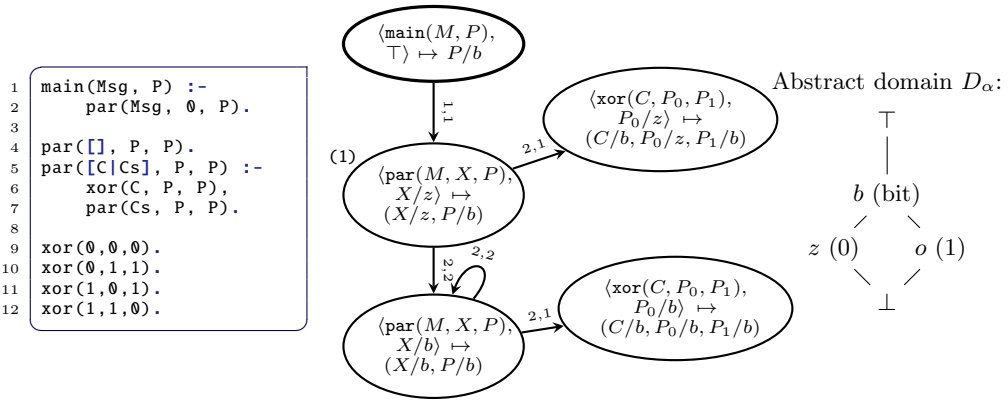


Fig. 1. A program that implements a parity function and a possible analysis result for domain D_α.

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.

The influence of assertions on the analysis result. As described earlier, assertions guarantee that incorrect or undesired behaviors of a program do not occur in the actual execution. We can take advantage of this to prune from the analysis result the states that will never be reached.

We first mention some properties of the analysis graph when no assertions are present in the program: (1) in any edge of the graph A, $\langle P, \lambda_1 \rangle_{i,j} \xrightarrow{\lambda^p} \langle Q, \lambda_2 \rangle$, the abstract substitution immediately before calling the literal (λ^p) is the same as the call to the predicate (λ_2), when projected to the variables of the literal (modulo renaming): $\lambda_2 = \sigma(\texttt{abs\_project}(\lambda^p, vars(P_{i,j}))) \wedge Q = \sigma(P)$, and (2) the answer pattern λ^s to a call to a predicate P with λ^c ($\langle P, \lambda^c \rangle \mapsto \lambda^s \in \mathscr{A}$) is the abstract union of the answer of each of its clauses, i.e., the (abstract) state after the call (λ^r) to the last literal in the body ($P_{n,last}$):
$\lambda^s = \bigsqcup\{\texttt{abs\_project}(\lambda^r, vars(P_n))$ s.t. $\langle P, \lambda^c \rangle_{n,last} \xrightarrow[\lambda^r]{} N \in \mathscr{A}\}$, with the variables restricted (projected) to the variables in the head of the clause P_n.

However, these properties do not hold when assertions are introduced in a program. (1) does not hold in general because if predicate Q has assertions that cause the analyzer to prune some over-approximated states then, it is not guaranteed to be exactly the same but $\lambda_2 \sqsubseteq \sigma(\texttt{abs\_project}(\lambda^p, vars(P_{i,j}))) \wedge Q = \sigma(P)$. (2)

does not hold for the same reason (analysis result may be pruned), and the answer pattern λ^s could be smaller than the abstract join of its clauses.

5 Incremental analysis of programs with assertions

In this section we propose an analysis algorithm that responds incrementally to changes both in the clauses and the assertions of the program. We do so by taking advantage of previous analysis algorithms.

Baseline incremental analysis algorithm. The concepts of analyzing a program incrementally and analyzing a program using assertions are not new. 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 straightforward combination of the algorithms to analyze incrementally a CHC program [13], and the analyzer that is guided by assertions [8]. This combined algorithm is detailed in Appendix A, and not included in the paper for space constraints. We will refer to it with the function $\mathscr{A}' = \textsc{IncAnalyze}(P, \Delta_{Cls}, \mathscr{Q}, \mathscr{A})$, meaning that the algorithm takes as input:

- A program $P = (Cls, As)$ as a pair of a set of clauses (Cls) and a set of assertions (As).
- A set of changes Δ_{Cls} in the form of added or deleted clauses
- A set $\mathscr{Q}$ 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 abstract the behavior of the program reacting incrementally to changes in the clauses.

The algorithm is parametric on the abstract domain D_α, given by implementing (1) the domain-dependent operations $\sqsubseteq, \sqcap, \sqcup$, `abs_project`$(\lambda, Vs)$, which restricts the abstract substitution to the set of variables Vs, and `abs_extend`$(P_{k,n}, \lambda^p, \lambda^s)$, which propagates the information of the success abstract substitution over the variables of $P_{k,n}$, λ^s, to the substitution of the variables of the clause λ^p; and (2) *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.

In addition, we assume to have two functions `apply_call`(P, λ^c) and `apply_succ`$(P, \lambda^c, \lambda^s)$ that are the transfer functions of the semantics of the assertion conditions (resp. calls and success conditions), i.e., they correctly abstract and apply the assertions to a given predicate and call description. Further details of these functions are described in Appendix A and in [8].

5.1 The incremental analyzer of programs with assertions

We extend IncAnalyze with an initial phase that will manipulate the analysis graph in a way such that we are able to call IncAnalyze to obtain results that are correct and precise, reacting incrementally to changes in assertions. This procedure is shown in Fig. 2. The phase prior to analyzing consists in inspecting all the program points 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

function IncAnalyze-w/AssrtChanges($(Cls, As), \Delta_{Cls}, \Delta_{As}, \mathscr{Q}, \mathscr{A}$)
  $R := \emptyset$
  for each $P \in Cls$ **do**
    if $\Delta_{As}[P] \neq \emptyset$ **then**
      $R := R \cup$ `update_calls_pred`(P)
      $R := R \cup$ `update_success_pred`(P)
  $\mathscr{A}' :=$ IncAnalyze$((Cls, As), \Delta_{Cls}, \mathscr{Q} \cup R, \mathscr{A})$
  del $(\mathscr{A}', \{E \mid E \in \mathscr{A}' \wedge Q \not\rightsquigarrow E \wedge Q \in \mathscr{Q}\})$ ▷ Remove unreachable calls
  return $\mathscr{A}'$

Fig. 2. High-level view of the proposed algorithm

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 affected parts in the analysis results. The pseudocode to find potential changes in the analysis results when assertions are changed is detailed in Fig. 3 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, as mentioned in section 4.2, assertions may affect the inferred call or the inferred success of predicates we have split the procedure in two functions. However the overall idea is to obtain the current substitution (i.e., 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 we get the semantics of the new assertions (using the functions `apply_call` and `apply_success`), finally we call a general procedure to treat the potential changes, `treat_change` (see Fig. 4). Specifically, in the case of `call` conditions, for a given predicate we want to review all the program points from which it is called (by checking the incoming edges of the nodes of that predicate). So, 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 semantics 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 if no assertions were present 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. Qualifying the changes in the abstract substitutions (i.e., they remain the same, they become more general, they become more concrete, or they become incompatible) becomes handy to take advantage of two key known facts about the concrete semantics of logic programs:

- Further constraining a substitution cannot cause more answers to appear.
- Generalizing a substitution cannot cause solutions to disappear.

Based on this knowledge, we define the procedure `treat_change` in Fig. 4. The goal is, given an edge that points to a literal whose success potentially changed,

1: **function** `update_calls_pred`(P)
2: $Q := \emptyset$
3: **for each** $\langle P', \lambda\rangle_{c,l} \xrightarrow{\lambda^p} \langle P, \lambda^c_{old}\rangle \in \mathscr{A}$ **do**
4: $\lambda^c := \sigma(\texttt{abs\_project}(\lambda^p, vars(P'_{c,l}))$ s.t. $\sigma(P'_{c,l}) = P$ ▷ Original call
5: $\lambda^c_{new} := \texttt{apply\_call}(P, \lambda^c)$
6: **if** $\exists \langle P', {\lambda^c}_{new}\rangle \mapsto \lambda^s \in \mathscr{A}$ **then** ▷ A node for that call already exist
7: $\lambda^{s\prime} := \lambda^s$
8: **else** $\lambda^{s\prime} := \bot$
9: $Q := Q \cup \texttt{treat\_change}(\langle P', \lambda\rangle_{c,l} \xrightarrow[\lambda^r]{\lambda^p} \langle P, {\lambda^c}_{new}\rangle, \lambda^{s\prime})$
10: **return** Q
11: **function** `update_successes_pred`(P)
12: $Q := \emptyset$
13: **for each** $\langle P, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}$ **do**
14: $\lambda := \bot$
15: **for each** $\langle P, \lambda^c\rangle_{c,last} \xrightarrow[\lambda^r]{} \langle Q, \lambda\rangle \in \mathscr{A}$ **do** ▷ Original success
16: $\lambda := \lambda \sqcup \texttt{apply\_success}(P, \lambda^c, \texttt{abs\_project}(\lambda^r, vars(P_c)))$
17: **for each** $E = N_{\bullet,\bullet} \xrightarrow[\bullet]{\bullet} \langle P, \lambda^c\rangle \in \mathscr{A}$ **do** ▷ Affected literals
18: $Q := Q \cup \texttt{treat\_change}\ (E, \lambda)$
19: **return** Q

Fig. 3. Changes in assertions (split by assertion conditions)

update the analysis result, and decide which predicates and call patterns need to be recomputed. After an initial step to update the annotation of the edge (line 3), 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 have to be restarted from the original entry points that were affected by the deletion of these potentially imprecise nodes. The case that remains (line 13) is the case in which the old and the new substitutions are the same, and, thus, nothing needs to be reanalyzed (the $\emptyset$ is returned).

5.2 Use cases

We now show some examples in which recomputing is avoided by reasoning with the changes between versions of a program. We assume that we analyze with a shape domain in which the properties that appear 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.

1: **function** `treat_change`($\langle P, \lambda\rangle_{c,l} \xrightarrow[\lambda^r]{\lambda^p} \langle Q, \lambda^c\rangle, \lambda^s$)

2: $\lambda^{r\prime} :=$ `abs_extend`(λ^p, λ^s) ▷ Obtain new info at literal return

3: $\mathsf{del}(\mathscr{A}, \langle P, \lambda\rangle_{c,l} \xrightarrow{\bullet}_{\bullet} \bullet)$

4: $\mathsf{add}(\mathscr{A}, \langle P, \lambda\rangle_{c,l} \xrightarrow[\lambda^{r\prime}]{\lambda^p} \langle Q, \lambda^c\rangle)$

5: **if** $\lambda^r \sqsubset \lambda^{r\prime}$ **then**

6: **return** $\{\langle P, \lambda\rangle\}$ ▷ Restart the analysis for this predicate and call pattern

7: **else if** $\lambda^r \not\sqsubseteq \lambda^{r\prime}$ **then** ▷ Analysis is potentially imprecise

8: $Lits := \{E \mid E = \langle P, \lambda\rangle_{c,i} \longrightarrow N \in \mathscr{A} \wedge i > l\}$ ▷ Following literals

9: $IN := \{E \mid E \rightsquigarrow L \in \mathscr{A} \wedge L \in Lits\}$ ▷ Potentially imprecise nodes

10: $Q = IN \cap \mathscr{Q}$ ▷ Entry point of potentially imprecise nodes

11: $\mathsf{del}(\mathscr{A}, IN)$

12: **return** Q

13: **else return** $\emptyset$

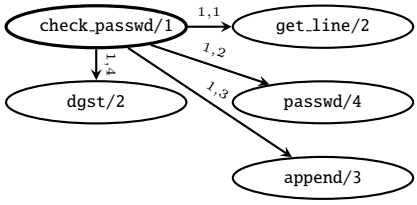
Fig. 4. Functions to determine how the analysis result needs to be recomputed.

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==").
```

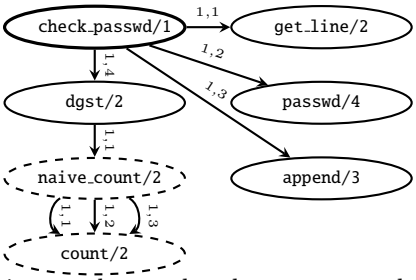


Concretely the node for `dgst/2` will represent the call $\langle$`dgst(S,D)`$, (S/$`lowercase`$, D/$`num`$)\rangle \mapsto (S/$`lowercase`$, D/$`int`$)$, in this case, `D` was inferred to be a *number* because of the success of `passwd/4`.

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,
    %% implementation continues
```



We are able to 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 avoid 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.

```
1 :- trait hasher {
2       :- 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.

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 22K 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 20K 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 [21].

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 simple groundness domain (gr), and a domain tracking dependencies via propositional clauses [7] (def). The experiment consisted on preanalyzing the tool with no backends and then adding incrementally the backends one by one. In Table 6 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.

domain	no backend	texinfo	man	html
`gr`	3.3	4.4	5.3	6.0
`gr inc`	3.5	1.8	1.3	2.7
`def`	12.2	15.2	15.7	20.8
`def inc`	12.3	2.1	1.3	2.9

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

7 Related work

Languages like C++ require specializing all parametric polymorphic code (e.g., templates [22]) to monomorphic variants. While this is more restrictive than *run-time 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 [6] (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 [17]).

Mora et al. [16] perform modular symbolic execution to prove that some (versions of) libraries are equivalent with respect to the same client. Chatterjee et al. [4] 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 [18].

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. 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 (LP) code.

References

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

2. Bueno, F., Cabeza, D., Hermenegildo, M.V., Puebla, G.: Global Analysis of Standard Prolog Programs. In: ESOP (1996)
3. Cabeza, D., Hermenegildo, M.V.: A New Module System for Prolog. In: Int'l. Conf. on Computational Logic. LNAI, vol. 1861, pp. 131–148. Springer (July 2000)
4. Chatterjee, K., Choudhary, B., Pavlogiannis, A.: Optimal Dyck reachability for data-dependence and alias analysis. PACMPL **2**(POPL), 30:1–30:30 (2018)
5. 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)
6. Donovan, A.A.A., Kernighan, B.W.: The Go Programming Language. Professional Computing, Addison-Wesley (October 2015)
7. 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)
8. Garcia-Contreras, I., Morales, J., Hermenegildo, M.V.: Multivariant Assertion-based Guidance in Abstract Interpretation. In: 28th Int'l. Symp. on Logic-based Program Synthesis and Transformation (LOPSTR'18) (January 2019)
9. 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)
10. Hermenegildo, M.V.: A Documentation Generator for (C)LP Systems. In: Int'l. Conf. CL 2000. LNAI, vol. 1861, pp. 1345–1361. Springer-Verlag (July 2000)
11. Hermenegildo, M.V., Morales, J.: The LPdoc Documentation Generator. Ref. Manual (v3.0). Tech. rep. (July 2011), available at `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: The Logic Programming Paradigm, pp. 161–192. Springer (1999)
13. Hermenegildo, M.V., Puebla, G., Marriott, K., Stuckey, P.: Incremental Analysis of Constraint Logic Programs. ACM TOPLAS **22**(2), 187–223 (March 2000)
14. Hudak, P., Peyton-Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzman, M.M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W., Peterson, J.: Report on the Programming Language Haskell. Haskell Special Issue, ACM Sigplan Notices **27**(5), 1–164 (1992)
15. Klabnik, S., Nichols, C.: The Rust Programming Language. No Starch Press, San Francisco, CA, USA (2018)
16. Mora, F., Li, Y., Rubin, J., Chechik, M.: Client-specific equivalence checking. In: 33rd ACM/IEEE Int'l. Conf. on Automated Softw. Eng., ASE. pp. 441–451 (2018)
17. Namolaru, M.: Devirtualization in GCC. In: Proceedings of the GCC Developers' Summit. pp. 125–133 (2006)
18. Puebla, G., Albert, E., Hermenegildo, M.V.: Abstract Interpretation with Specialized Definitions. In: SAS'06. pp. 107–126. No. 4134 in LNCS, Springer (2006)
19. 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)
20. 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)
21. Swift, T., Warren, D.: XSB: Extending Prolog with Tabled Logic Programming. TPLP (1-2), 157–187 (2012)
22. Vandevoorde, D., Josuttis, N.M.: C++ Templates. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)

A Full description of the base algorithm

In this section we describe the combination of the incremental analysis algorithm [13] with the algorithm that uses assertions of programs [8].

Additional graph operations. In addition to the operations introduced in the paper we will need some more operations to modify the analysis graph:

$\mathsf{upd}(g, \langle A, \lambda^c \rangle \leftarrow\!\shortmid\, \lambda^s)$: overwrites the value of $\langle A, \lambda^c \rangle$ in the mapping function and, if necessary, adding a node to g with key $\langle A, \lambda^c \rangle$.

$\mathsf{upd}(g, k_{c,l} \xrightarrow[\lambda^r]{\lambda^p} k')$: adds an edge node k to node k' with the corresponding annotation if it did not exist.

$\mathsf{upd}(g, \{e_i\})$: performs the upd operation for each of the elements of the set.

$\mathsf{ancestors}(g, k)$: obtains nodes from which there is a path to k $\{k' | k' \rightsquigarrow k \in g\}$

Algorithm usage. As mentioned in the paper, we refer to the analyzer with the function $\mathscr{A}' = \textsc{IncAnalyze}(P, \Delta_{Cls}, \mathscr{Q}, \mathscr{A})$, which is shown in Fig. 5. It takes as input:

- A program $P = (Cls, As)$ as a pair of a set of clauses (Cls) and a set of assertions (As).
- A set of changes Δ_{Cls} in the form of added or deleted clauses
- A set $\mathscr{Q}$ 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.

And produces a new $\mathscr{A}'$ that correctly abstract the behavior of the program reacting incrementally to changes in the clauses.

Additional domain operation. As mentioned earlier, the algorithm is parametric on the abstract domain (D_α), apart from the operations introduced in the paper we define an additional operation: $\texttt{abs\_call}(\lambda, P, P_k)$ 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. This operation includes the necessary variable renamings.

Events. The algorithm is centered around processing tasks triggered by events. There are two kinds of events:

- $newcall(\langle A, \lambda^c \rangle)$ indicates that a new description for atom A has been encountered.
- $arc(D)$ means that recomputation needs to be performed starting at program point (literal) indicated by dependency D.

To add events to the queue we use the function $\mathsf{add\_event}(E)$.

Operation of the algorithm. The algorithm starts by adding to the queue *newcall* events for each of the call patterns that need to be recomputed. The `process`($newcall(\langle P, \lambda^c \rangle)$) procedure initiates the processing of the clauses in the definition 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 program points from which the analysis has to be restarted. The `process`($arc(\langle P_k, \lambda^c \rangle_{l,c} \xrightarrow{\lambda^p} \langle P, \lambda^c \rangle)$) procedure performs the core of the module analysis. It performs a single step of the left-to-right traversal of a clause body. First of all the semantics of the assertions of predicate P are computed by `apply_call`. Then, if the literal $P_{k,i}$ is a *built-in*, it is added to the abstract description; otherwise, if it is an atom, 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 obtained answer is combined with the description λ^p from the program point immediately before $P_{k,i}$ to obtain the description (return) for the program point after $P_{k,i}$. This is either used to generate an *arc* event to process the next literal (if there is one), or otherwise to update the answer of the rule in `insert_answer_info`. This function combines the new answer with the semantics of any applicable assertions (in `apply_succ`), and with the previous answers, and propagates the new answer if needed.

Procedure `add_clauses` add to the queue the tasks to analyze the new clause for each predicates. This information is used to update $\mathscr{A}$ and propagated to the rest of the graph. The computation and propagation of the added rules is done simply by adding *arc* events before starting the processing of the queue.

The `delete_clauses` function selects which information can be kept in order to obtain the most precise semantics of the module, by removing all information in the L which is potentially inaccurate, i.e., the information related to the calls that depend on the deleted rules (`remove_invalid_info`), which gathers all the callers to the set of obsolete $Calls$, and the $\langle P, \lambda^c \rangle$ generated from literals that follow in a clause body any $Calls$, because they were affected by the λ^s.

B Assertions

Assertions may not be exactly represented in the abstract domain used by the analyzer. We recall some definitions (adapted from [20]) which are instrumental to correctly approximate the properties of the assertions during the analysis.

Definition 2 (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:*

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

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. 2:

Definition 3 (Abstract Trivial Success Subset of a Property Formula). *Under the same conditions of Def. 2, 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 4 (Abstract Trivial Success Superset of a Property Formula). *Under the same conditions of Def. 3, 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.

ALGORITHM **IncAnalyze**

input (global): $(Cls, As), \Delta_{Cls}, \mathscr{Q}$
global: $\mathscr{A}$

1: **for all** $\langle P, \lambda^c \rangle \in \mathscr{Q}$ **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** $\langle P, \lambda^c \rangle \mapsto \lambda^s \in \mathscr{A}$ **do**
13: λ^p :=abs_call(λ^c, P, P_k)
14: ${\lambda^c}_1$:=abs_project($\lambda^p, vars(P_{k,1})$)
15: add_event($arc(\langle P, \lambda^c \rangle_{k,1} \xrightarrow{\lambda^p} \langle pred(P_{k,1}), {\lambda^c}_1 \rangle)$)
16: **procedure** delete_clauses(Cls)
17: $Calls := \{\langle P, \lambda^c \rangle | \langle P, \lambda^c \rangle \in \mathscr{A}, (P_k$:- $Body) \in Cls\}$
18: $Ns :=$ ancestors($\mathscr{A}, Calls$)
19: del($\mathscr{A}, Ns$)
20: **function** lookup_answer($\langle P, \lambda^c \rangle$)
21: **if** $\langle P, \lambda^c \rangle \mapsto \lambda^s \in \mathscr{A}$ **then**
22: **return** λ^s
23: **else**
24: add_event($newcall(\langle P, \lambda^a \rangle)$)
25: **return** $\bot$
26: **procedure** process($newcall(\langle P, \lambda^c \rangle)$)
27: **for all** P_k :- $P_{k,1}, \ldots, P_{k,n_k} \in Cls$ **do**
28: λ^p :=abs_call(λ^c, P, P_k)
29: ${\lambda^c}_1$:=abs_project($\lambda^p, vars(P_{k,1})$)
30: add_event($arc(\langle P, \lambda^c \rangle_{k,1} \xrightarrow{\lambda^p} \langle pred(P_{k,1}), {\lambda^c}_1 \rangle)$)
31: λ^s :=initial_guess($\langle P, \lambda^c \rangle$)
32: **if** $\lambda^s \neq \bot$ **then**
33: reanalyze_updated($\langle P, \lambda^c \rangle$)
34: upd($\mathscr{A}, \langle P, \lambda^c \rangle \leftarrow\!\shortmid \lambda^s$)
35: **procedure** process($arc(\langle P, {\lambda^c}_0 \rangle_{k,i} \xrightarrow{\lambda^p} \langle Q, {\lambda^c}_1 \rangle)$)
36: $\lambda^a =$ apply_call($Q, {\lambda^c}_1$)
37: **if** $P_{k,i}$ is a *built-in* **then**
38: ${\lambda^s}_0 := f^\alpha(P_{k,i}, \lambda^a)$ ▷ Apply transfer function
39: **else** ${\lambda^s}_0$:=lookup_answer($\langle Q, \lambda^a \rangle$)
40: λ^r :=abs_extend($\lambda^p, {\lambda^s}_0$)
41: upd($\mathscr{A}, \langle P, {\lambda^c}_0 \rangle_{k,i} \xrightarrow[\lambda^r]{\lambda^p} \langle Q, \lambda^a \rangle$)
42: **if** $\lambda^r \neq \bot$ and $i \neq n_k$ **then**
43: ${\lambda^c}_2$:=abs_project($\lambda^r, vars(P_{k,i+1})$)
44: add_event($arc(\langle H, {\lambda^c}_0 \rangle_{k,i+1} \xrightarrow{\lambda^r} \langle B, {\lambda^c}_2 \rangle)$)
45: **else if** $\lambda^r \neq \bot$ and $i = n_k$ **then**
46: λ^s :=abs_project($\lambda^r, vars(P_k)$)
47: insert_answer_info($\langle P, {\lambda^c}_0 \rangle, \lambda^s$)
48: **procedure** insert_answer_info($\langle P, \lambda^c \rangle, \lambda^s$)
49: $\lambda^a :=$ apply_succ(P, λ^c, λ^s)
50: **if** $\langle P, \lambda^c \rangle \mapsto {\lambda^s}_0 \in \mathscr{A}$ **then**
51: ${\lambda^s}_1$:=abs_generalize($\lambda^a, {\lambda^s}_0$)
52: **else** ${\lambda^s}_1 = \bot$
53: **if** ${\lambda^s}_0 \neq {\lambda^s}_1$ **then**
54: upd($\mathscr{A}, \langle P, \lambda^c \rangle \leftarrow\!\shortmid {\lambda^s}_1$)
55: reanalyze_updated($\langle P, \lambda^c \rangle$)
56: **procedure** reanalyze_updated($\langle P, \lambda^c \rangle$)
57: **for all** $E := \langle Q, {\lambda^c}_0 \rangle_{k,i} \xrightarrow{\lambda^p} \langle P, \lambda^c \rangle \in \mathscr{A}$ **do**
58: add_event($arc(E)$)

Fig. 5. The generic context-sensitive, incremental fixpoint algorithm using (not changing) assertion conditions.

global flag: `speed-up`

1: **function** `apply_call`(P, λ^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** λ^t **else return** $\lambda^c \sqcap \lambda^t$
4: **else return** λ^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** λ^t **else return** $\lambda^t \sqcap \lambda^{s_0}$
11: **else return** λ^{s_0}

Fig. 6. Applying assertions.