Incremental and Modular Context-sensitive Analysis[*]

ISABEL GARCIA-CONTRERAS
IMDEA Software Institute, Madrid, Spain
Universidad Politécnica de Madrid (UPM), Madrid, Spain
(*e-mail:* isabel.garcia@imdea.org)

JOSÉ F. MORALES
IMDEA Software Institute, Madrid, Spain
(*e-mail:* josef.morales@imdea.org)

MANUEL V. HERMENEGILDO
IMDEA Software Institute, Madrid, Spain
Universidad Politécnica de Madrid (UPM), Madrid, Spain
(*e-mail:* manuel.hermenegildo@imdea.org)

submitted 20 July 2019; revised 12 December 2020; accepted 19 December 2020

Abstract

Context-sensitive global analysis of large code bases can be expensive, which can make its use impractical during software development. However, there are many situations in which modifications are small and isolated within a few components, and it is desirable to reuse as much as possible previous analysis results. This has been achieved to date through incremental global analysis fixpoint algorithms that achieve cost reductions at fine levels of granularity, such as changes in program lines. However, these fine-grained techniques are neither directly applicable to modular programs nor are they designed to take advantage of modular structures. This paper describes, implements, and evaluates an algorithm that performs efficient context-sensitive analysis incrementally on modular partitions of programs. The experimental results show that the proposed modular algorithm shows significant improvements, in both time and memory consumption, when compared to existing non-modular, fine-grain incremental analysis techniques. Furthermore, thanks to the proposed intermodular propagation of analysis information, our algorithm also outperforms traditional modular analysis even when analyzing from scratch.

KEYWORDS: program analysis, incremental analysis, modular analysis, constrained Horn clauses, abstract interpretation, fixpoint algorithms, logic and constraint programming

1 Introduction and motivation

Large, real-life programs typically have a complex structure combining a number of modules with system libraries. Context-sensitive global analysis of such large code bases can be expensive, and this can be specially problematic in interactive uses of analyzers. An

[*] Research partially funded by MINECO MICINN PID2019-108528RB-C21 *ProCode* project, FPU grant 16/04811, and the Madrid M141047003 *N-GREENS* and P2018/TCS-4339 *BLOQUES-CM* programs. We are also grateful to the anonymous reviewers, editors, and to Ignacio Fábregas for their comments.

example is detecting and reporting bugs as the program is being edited, by running the analysis in the background at small intervals, for example, each time a set of changes is made, when a file is saved, or when a commit is made in the version control system. Other such scenarios include reanalyzing after performing source-to-source transformations and/or optimizations, or updating analysis results after dynamic program modifications (reanalysis at runtime). In these scenarios, triggering a complete reanalysis for each change set is often too costly for larger programs. However, a key observation is that very often changes in the program are small and isolated inside a small number of components. Ideally, this characteristic can be taken advantage of to reduce the cost of re-analysis in two ways: reusing as much information as possible from previous analyses, and avoiding the maintenance of analysis information for unaffected components.

In the field of abstract interpretation, there have been proposals to deal with the following two cases: (a) context-sensitive incremental fixpoint algorithms (Hermenegildo *et al.* 1995, 2000; Puebla and Hermenegildo 1996; Kelly *et al.* 1997; Albert *et al.* 2012; Arzt and Bodden 2014; Szabó *et al.* 2016), which reuse information but still need to work with the program as a whole (incremental but *monolithic* analyzers)and (b) *modular* algorithms, aimed at reducing the memory consumption or working set size (Bueno *et al.* 2001; Cousot and Cousot 2002; Puebla *et al.* 2004; Correas *et al.* 2006; Cousot *et al.* 2009; Fähndrich and Logozzo 2011), which work on a module at a time but do not support changes in the program. Surprisingly, the combination of both techniques has not been explored to date. The monolithic incremental analyzers are not directly applicable in the modular setting due to two issues: first, these analyzers do not deal with code that is partially available, that is, they have no provisions to make assumptions about code that is external. Even though one could see built-in operations of the language as external calls, as they are obviously not defined in the module, the semantics of these are typically "hardwired" in the analyzer as *transfer functions*. This leads to the second issue: even though the monolithic analyzers can make assumptions using this mechanism, these algorithms are not prepared to deal in a correct and precise way with updates to these assumptions.

In order to bridge this gap, using a monolithic incremental analysis algorithm as a starting point, we develop a modular, incremental analyzer capable of performing fine-grain incremental analysis across modular program partitions. Our algorithm is based on computing local fixpoints on one module at a time, identifying, invalidating, and re-computing only those parts of the analysis results that are affected by these fine-grain program changes, and propagating the fine-grained analysis information across module boundaries. Our contributions are extending the incremental (global) fixpoint algorithm of Hermenegildo *et al.* (2000) with widening (Section 4.1), providing a formal description of the modular analysis algorithm of Puebla *et al.* (2004) with correctness results (Section 4.2), and providing a new analysis algorithm that is modular and incremental, also with correctness results (Sections 5 and 6). Additionally, we have implemented the proposed approach within the Ciao/CiaoPP system (Hermenegildo *et al.* 2005, 2012) and benchmarked it. The experimental results observed show good cost performance trade-offs, in both time and memory consumption, and suggest that this is an interesting and practically relevant approach.

2 Preliminaries and notation

CHCs as Intermediate Representation. For generality, we will formulate our algorithm to work on a block-level intermediate representation of the program, encoded using (constrained) Horn clauses. A constrained Horn clause (CHC) program, or constraint logic program (CLP), is a set of *clauses* of the form H `:-` $A_1, \ldots, A_n$, where $A_1, \ldots, A_n$ are *literals* and H is an *atom* said to be the *head* of the clause. For simplicity, and without loss of generality, we assume that each head atom is normalized, that is, it is of the form $p(x_1, \ldots, x_m)$ where p is an m-ary predicate symbol and $x_1, \ldots, x_m$ are distinct variables. However, in the examples, we will sometimes show programs unnormalized for brevity. A set of clauses with the same head is called a *predicate* (procedure). To refer to predicates, we will use normalized atoms and sometimes will call them *predicate descriptors.* A *literal* is an atom or a *primitive constraint* (which we will also refer to as a *built-in*). A primitive constraint is defined by the underlying abstract domain(s) and is of the form $c(e_1, \ldots, e_k)$ where c is a k-ary predicate symbol and the $e_1, \ldots, e_k$ are expressions. 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 subscripts (clause number, body position), for example, A_k`:-`$A_{k,1}, \ldots A_{k,n_k}$. The clause may also be referred to as clause k of predicate A. For example, for the following predicate, `p/3`:

$$\texttt{p}(X, Y, Z) \texttt{ :- } X =< 0, Y = Z.$$
$$\texttt{p}(X, Y, Z) \texttt{ :- } X > 0, X1 = X-1, Y1 = Y{*}X, \texttt{p}(X1, Y1, Z).$$

$\texttt{p/3}_1$ denotes the head of the first clause of `p/3`, and $\texttt{p/3}_{2,1}$ denotes the first literal of the second clause of `p/3`, that is, the constraint $X > 0$.

We assume that programs are converted to this Horn clause-based representation, on a modular basis. The conversion itself is beyond the scope of the paper (and dependent on the source language). It is trivially direct in the case of (C)LP programs or (eager) functional programs, and for imperative programs we refer the reader to, for example, Henriksen and Gallagher (2006), Méndez-Lojo *et al.* (2007), Albert *et al.* (2007), Gallagher *et al.* (2020). In Navas *et al.* (2007), the base algorithms that we extend in this work were shown to be directly applicable to Java bytecode. In fact, Horn clauses have since been used successfully as intermediate representations for many different programming languages and compilation levels (e.g., bytecode, llvm-IR, ISA, . . .), in a good number of analysis and verification tools (Banda and Gallagher 2009; Navas *et al.* 2008, 2009; Grebenshchikov *et al.* 2012; Jaffar *et al.* 2012; Albert *et al.* 2012; Bjørner *et al.* 2013, 2015; Liqat *et al.* 2014, 2016; De Angelis *et al.* 2014; Gurfinkel *et al.* 2015; Madsen *et al.* 2016; de Moura and Bjørner 2008; Kafle *et al.* 2016; Lopez-Garcia *et al.* 2018; Perez-Carrasco *et al.* 2020) (see Section 8 for other related work). We note that some of these approaches use the *bottom-up* semantics on the CHC side, and then typically the *small-step semantics* in the translation, while others, including ours, exploit the complementary approach of using the *top-down* semantics on the CHC side, and then typically the *big-step semantics* in the translation, but some combine, for example, big-step with bottom-up (Gurfinkel *et al.* 2015). Big-step and small-step are nicknames often used to refer to, respectively, Kahn's natural semantics (Kahn 1987) and Plotkin's structural operational semantics (Plotkin 1981, 2004). In the big-step semantics approach, the clause-based encoding is equivalent to a block-based control flow graph, which is in turn a well-established intermediate representation for program analysis. Each block is represented by a clause, constraints or built-ins in a clause represent the primitives of the language (bytecodes, machine instructions, commands, etc.), literals represent calls to other blocks, and predicates with multiple clauses implement alternatives such as conditionals, case statements, dynamic dispatch, etc. (see, e.g., Méndez-Lojo *et al.* 2007; Lopez-Garcia *et al.* 2018). This approach is particularly well suited for programs with structured control flow, although program transformations allow supporting other program structures. See Gallagher *et al.* (2020) for a recent overview of the subject. In the following, we revisit the *top-down* semantics and establish our baseline.

Selection of the Concrete Semantics. The semantics of CHC programs that we use as starting point is goal-dependent (i.e., query-dependent or "top-down"), and based on SLD resolution (Robinson 1965), and its generalization to CLP (Jaffar and Lassez 1987; Marriott and Stuckey 1998), where constraint domains and constraint solving extend the domain of Herbrand terms with unification. The traditional description of this resolution procedure (Lloyd 1987; Apt 1990; Jaffar and Lassez 1987) builds a tree structure in which the nodes contain *resolvents*. However, when used as a basis for top-down program analyses, this construction is typically adorned so that nodes in the resolution tree include representations of the constraints both before and after completing the branch in which they appear. These are then called the *call* and *success* states for that node. This is because the aim of goal-directed, top-down program analysis is usually to obtain information on the constraints before and after each program point. This idea of storing call and success states is present, for example, in the notion of *generalized and trees* of Bruynooghe (1991). However, such trees only describe the *successful* derivation trees, that is, a query that eventually fails will have an empty tree. In practice, it is useful to generalize this notion to collect also those parts of the execution trees that lead to false, that is, the *calls* made to predicates in the program also during computations that eventually fail or loop, as in Muthukumar and Hermenegildo (1990, 1992). We will refer to these trees simply as AND trees. It is also often interesting to consider trees with also OR nodes, that is, AND-OR *trees*, rather than considering sets of AND trees, to capture analyses such as determinacy (Lopez-Garcia *et al.* 2010; King *et al.* 2006), cardinality (Braem *et al.* 1994), non-failure (Debray *et al.* 1997), etc., but for simplicity we limit the discussion herein to semantics based on AND trees.

Concrete Semantics. An AND tree represents the execution of a *query* (corresponding to one of more entry points to the program), and each node in such a tree represents a call to a predicate, adorned on the left with the state for that call, and on the right with the corresponding success state. The concrete semantics of a program P for a given set of queries $\mathcal{Q}$, $[\![P]\!]_{\mathcal{Q}}$, is the set of AND trees that represent the execution of the queries in $\mathcal{Q}$ for P. Queries are of the form $Q = \langle A, \theta^c \rangle$ where A is a normalized atom corresponding to a predicate in the program and θ^c is the *calling* or *initial constraint.* Nodes in an AND tree are of the form $\langle A, \theta^c, \theta^s \rangle$, where A is the call to a predicate p in P, and θ^c and θ^s are, respectively, the call and success constraints over the variables of A. Nodes that are part of failing (or looping) branches (i.e., that never "return") will have empty success fields: $\langle A, \theta^c, \emptyset \rangle$. The *calling context* of a predicate given by the predicate descriptor A defined in P for a set of queries Q is the set $\mathsf{calling\_context}(A, P, Q) = \{\theta^c \mid \exists T \in [\![P]\!]_Q \ s.t.\ \exists \langle A', \theta^c, \theta^s \rangle \ in\ T \wedge \exists \sigma A' = \sigma(A)\}$, where σ is a *renaming* substitution over variables in the program, that is, a substitution that replaces each variable in the term it is applied to with distinct, fresh variables. In the following, we will use σ to denote such renaming substitutions. We denote by $\mathsf{answers}(P, Q)$ the set of answers (success constraints) computed by P for queries Q, that is, $\mathsf{answers}(P, Q)$ is $\{\theta^s \mid s.t.\ \exists T \in [\![P]\!]_Q \wedge \langle A, \theta^c, \theta^s \rangle = root(T)\}$.

Modular Partitions of Programs. A partition of a program is said to be modular when its source code is distributed in several source units, each defining its interface with other such units of the program. We will refer to these units as *modules*. The interface of a module contains the names of the predicates it exports and the names of the modules it imports. Modular partitions of programs may be synthesized or specified by the programmer, for example, via a strict module system, that is, a system in which modules can only communicate via their interface. We will use M and M' to denote modules. Given a module M:

- $\mathsf{exports}(M)$ denotes the set of predicate names exported by module M,
- $\mathsf{imports}(M)$ is the set of modules which M imports, and
- $\mathsf{mod}(A)$ denotes the module in which the predicate corresponding to atom A is defined. We sometimes abuse notation and denote the module of a query as $\mathsf{mod}(Q)$, to refer to the module of the predicate called in the query, that is, if $Q = \langle A, \lambda^c \rangle$ then $\mathsf{mod}(Q) = \mathsf{mod}(A)$.

3 Analysis graphs in goal-dependent abstract interpretation

In this section, we present the main abstraction object that is used in goal-dependent abstract interpretation: the analysis graph. Later sections will address the procedures for constructing such graphs.

Program Analysis by Abstract Interpretation. Abstract interpretation (Cousot and Cousot 1977) is a technique for static program analysis in which the execution of the program is simulated on an abstract domain (D_α) which is simpler than the concrete domain (D). Values in the abstract domain and sets of values in the concrete domain are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : D \rightarrow D_\alpha$, and *concretization* $\gamma : D_\alpha \rightarrow D$ which form a Galois connection. An 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. We refer to these abstract values interchangeably as *descriptions* or *patterns*. The correctness of abstract interpretation guarantees that the descriptions inferred (by computing a fixpoint through a Kleene sequence) approximate all the actual values or traces which occur during any possible execution of the program, and that this fixpoint computation process will terminate given some conditions on the description domains (such as being finite, or of finite height, or without infinite ascending chains) or by the use of a *widening* operator ∇ (Cousot and Cousot 1977).

Abstract Domain Operations for the Algorithms. The abstract interpretation-based algorithms that we will present are all *parametric on the abstract domain*, that is, they are independent of the (data)abstractions used. Each such abstract domain is defined by providing: the basic operations of the domain lattice mentioned above ($\sqsubseteq, \sqcap, \sqcup$ and, optionally, the widening ∇ operator); the abstract semantics (*transfer functions*, f^α) of the constraints (representing the *built-ins*, or basic operations of the source language); and the following additional instrumental operations, following Hermenegildo *et al.* (2000):

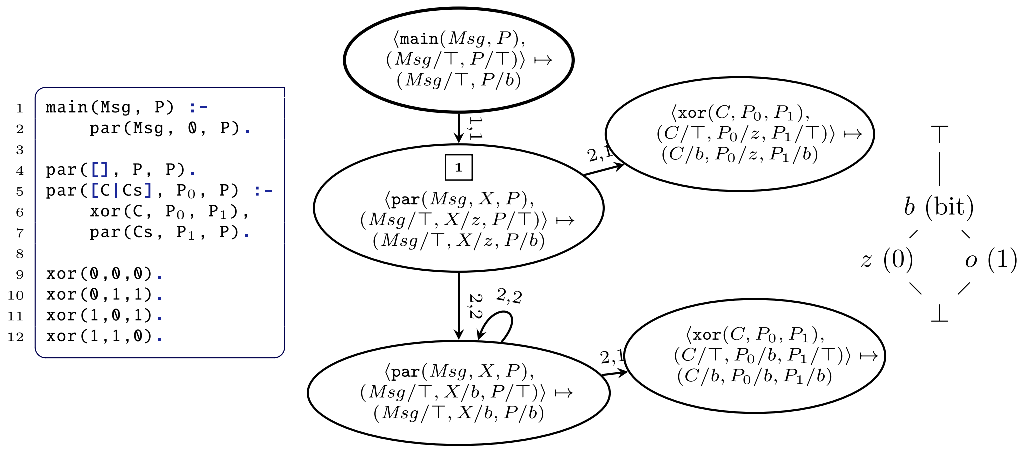
- `Aproj`(λ, Vs) restricts the abstract constraint to the set of variables Vs.
- `Aextend`$(A_{k,n}, \lambda^p, \lambda^s)$ propagates the information in the success abstract constraint λ^s, which is defined over the variables of $A_{k,n}$, to an abstract constraint λ^p that includes all the variables of the clause A_k.
- `Acall`(λ, A, A_k) performs the abstract unification (conjunction) of predicate descriptor A with the head of clause A_k, including in the new constraint abstract values for the variables in the body of clause A_k.
- `Ageneralize`$(\lambda, \{\lambda_i\})$ joins λ together with the set of abstract constraints $\{\lambda_i\}$. To produce an abstract constraint that is greater or equal than λ. It will either perform the least upper bound ($\sqcup$) or the widening operation over the set together with λ, depending on termination or performance needs, typically determined by the abstract domain.[1]

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 $u \rightarrow v$. A *path* P is a sequence of edges $(e_1, \dots, e_n)$ 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 a path P.

Analysis graphs. We perform *goal-dependent abstract interpretation*, whose result is an *abstraction* of the AND tree semantics, $[\![P]\!]_Q$. The discussion essentially follows the PLAI algorithm (Muthukumar and Hermenegildo 1990, 1992), using the presentation of Hermenegildo *et al.* (2000). The purpose of this abstraction is to represent as a finite object the (possibly infinite) set of (possibly infinite) AND trees in $[\![P]\!]_Q$. As mentioned before, the abstract interpretation technique guarantees that this process terminates and that the *concretization* of the resulting abstraction will be a safe (over-)approximation of the AND trees of the concrete semantics.

The *input* to this abstract interpretation process is a program P, an abstract domain D_α, and a set of initial *abstract queries* $Q_\alpha = \{\langle A_i, {\lambda^c}_i\rangle\}$, where each A_i is a normalized atom, and ${\lambda^c}_i \in D_\alpha$. Q_α defines the (typically infinite) set of concrete queries Q that the analysis will be valid for. With some abuse of notation, we represent this set as $\gamma(Q_\alpha)$, that is, $Q = \gamma(Q_\alpha) = \{\langle A, \theta\rangle \mid \theta \in \gamma(\lambda) \wedge \langle A, \lambda\rangle \in Q_\alpha\}$. This also determines the concrete semantics to be safely approximated, which is then the set of AND trees $[\![P]\!]_Q = [\![P]\!]_{\gamma(Q_\alpha)}$.

[1] The implementation of the classical algorithm includes options for activating or deactivating multivariance on calls and also on success. We leave the latter out herein for simplicity; however, our results also apply since this is equivalent to turning the affected domains into power domains.



```
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).
```

$\langle \mathtt{main}(Msg, P), (Msg/\top, P/\top)\rangle \mapsto (Msg/\top, P/b)$
1,1
$\boxed{1}$
$\langle \mathtt{par}(Msg, X, P), (Msg/\top, X/z, P/\top)\rangle \mapsto (Msg/\top, X/z, P/b)$
2,1
$\langle \mathtt{xor}(C, P_0, P_1), (C/\top, P_0/z, P_1/\top)\rangle \mapsto (C/b, P_0/z, P_1/b)$
2,2
2,2
$\langle \mathtt{par}(Msg, X, P), (Msg/\top, X/b, P/\top)\rangle \mapsto (Msg/\top, X/b, P/b)$
2,1
$\langle \mathtt{xor}(C, P_0, P_1), (C/\top, P_0/b, P_1/\top)\rangle \mapsto (C/b, P_0/b, P_1/b)$
$\top$
b (bit)
z (0)
o (1)
$\bot$

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

An *analysis result* is a call graph and a mapping function from predicate descriptors and call descriptions to answer descriptions, both elements of D_α. We also call this structure an *analysis graph*.

A *node* in an analysis graph represents that a call to a predicate ($\langle A, \lambda^c\rangle$) is possibly made, and it has an associated answer λ^s, through the mapping, $\langle A, \lambda^c\rangle \mapsto \lambda^s$, with $\lambda^c, \lambda^s \in D_\alpha$. This represents that *the answer pattern for calls to predicate A with calling pattern λ^c is λ^s*, and it implies that for any node in the concrete trees in $[\![P]\!]_Q$ of the form $\langle A, \theta^c, \theta^s\rangle$, there must exist a node $\langle A, \lambda^c\rangle \mapsto \lambda^s$ in the analysis graph such that $\theta^c \in \gamma(\lambda^c)$ and $\theta^s \in \gamma(\lambda^s)$. Therefore, analysis graphs must capture all the call–success pairs, that is, all the nodes in the AND trees of the concrete semantics (these conditions are formulated more precisely in Section 4.1.1). For a given predicate A, the analysis graph may contain more than one node capturing different call situations. As usual, $\top$ denotes the most general abstract description, which is equivalent to "I do not know," and $\bot$ denotes the abstract description such that $\gamma(\bot) = \emptyset$. A call mapped to $\bot$ ($\langle A, \lambda^c\rangle \mapsto \bot$) indicates that all calls to predicate A with description $\theta \in \gamma(\lambda^c)$ either fail or loop, that is, they never succeed.

An *edge* in an analysis graph is of the form $\langle A, \lambda^c\rangle \rightarrow_{k,i} \langle B, \lambda^{c\prime}\rangle$. This represents that *calling predicate A with calling pattern λ^c may cause predicate B to be called (via the literal $A_{k,i}$) with calling pattern $\lambda^{c\prime}$*. Correctness with respect to the concrete semantics requires that if in any concrete tree in $[\![P]\!]_Q$ the clause A_k is executed with a calling pattern θ^c that causes predicate B (the literal $A_{k,i}$) to be called with some calling pattern $\theta^{c'}$, then there must be an edge in the graph $\langle A, \lambda^c\rangle \rightarrow_{k,i} \langle B, \lambda^{c\prime}\rangle$ and $\theta^c \in \gamma(\lambda^c)$, $\theta^{c'} \in \gamma(\lambda^{c'})$. These edges capture the dependencies between the immediate calls of a predicates, that is, given a node in the tree, the immediately following nodes. For simplicity, in the rest of the paper, we omit k, i when not relevant in the context.

Example 1
Figure 1 shows an analysis graph (center) for a program that computes the parity of a message (left) with an abstract domain that infers for each variable whether it takes values of 0 or 1 (right) and initial abstract query $Q_\alpha = \{\langle \mathtt{main}(Msg, P), (Msg/\top, P/\top)\rangle\}$. In the examples, we will mark with a bold outline the initial nodes (i.e., the nodes in

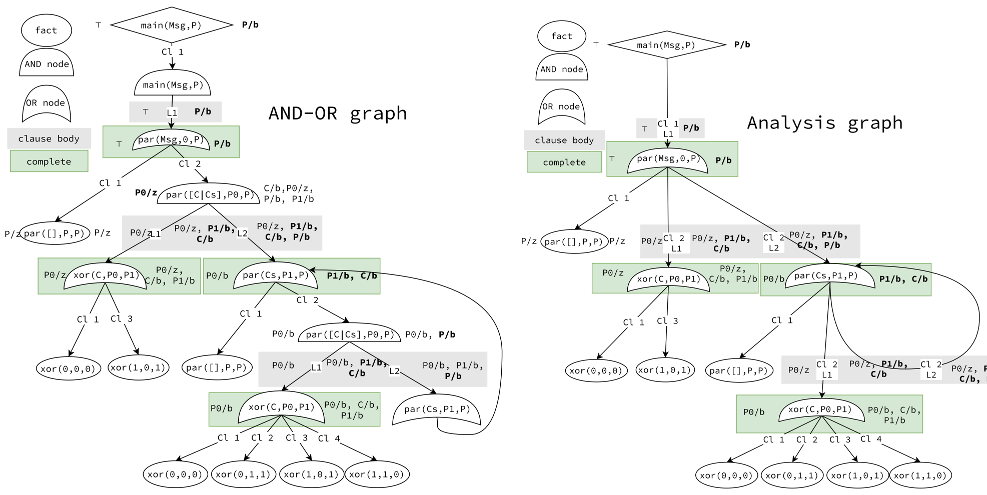


Fig. 2. The analysis graph versus the AND-OR graph – compacting representation.

Q_α). *Node* $\boxed{1}$ ($\langle \texttt{par}(Msg, X, P), (Msg/\top, X/z, P/\top)\rangle \mapsto (Msg/\top, X/z, P/b)$) captures that **par/3** may be called with X bound to any in $\gamma(z) = \{0\}$ and, if it succeeds, the third argument P will be bound to any of $\gamma(b) = \{1, 0\}$. Note that a different node (the one below) captures that there are other calls to **par** where X/z holds. The edges in the graph represent the $\langle A, \lambda^c\rangle \rightarrow_{k,i} \langle B, \lambda^{c'}\rangle$ relation. For example, two such edges exist starting at node $\boxed{1}$, denoting (right) that it may call **xor/3** and (below) that it may call itself with a different call description. Figure 2 illustrates for the example in Figure 1, the evolution from AND-OR *graphs* (left) to the compact representation of the analysis graphs: AND *nodes* are made implicit (right) by keeping the references to the clauses and literals. The information in the AND-OR *graph* can be reconstructed by renaming and projecting abstract descriptions of the analysis graph, which keeps the information only at the predicate and literal level. Last, please note that although in this simple example we are using a domain with a simple structure of tuples of *Variable*/*AbstractValue* pairs, the domain structure can be arbitrary and in particular includes *relational* domains.

Multivariance (a.k.a., context and path sensitivity). As seen in the example, these analysis graphs allow representing the different call patterns encountered during the execution, separating the cases in which such calls differ, even if some of them subsume others. This feature is traditionally referred to as *multivariance* in the context of logic program analysis, and, in our context, it serves two purposes:

1. *Precision:* Different calling patterns to the same predicate are stored depending from which exact clause and literal this predicate is called from and with which call pattern. This idea of storing multiple calling contexts in this way is used in recent implementations of context sensitivity in imperative program analyses (e.g., Khedker and Karkare 2008; Thakur and Nandivada 2020) where it is referred to as keeping *multiple value contexts*.
2. *Efficiency:* For the same literal and clause in the program, storing different calling patterns allows keeping the fixpoint computation localized to only those patterns that change.

```
% ⟨main(Msg, P), (Msg/⊤, P/⊤)⟩ ↦ (Msg/⊤, P/b)
main(Msg, P) :-
    par_1(Msg, 0, P).

% ⟨par(Msg, X, P), (Msg/⊤, X/z, P/⊤)⟩ ↦ (Msg/⊤, X/z, P/b)
par_1([], P, P).
par_1([C|Cs], P0, P) :-
    true([Msg/T,P/T]),
    xor_1(C, P0, P1).
    par_2(Cs, P1, P).

% ⟨par(Msg, X, P), (Msg/⊤, X/b, P/⊤)⟩ ↦ (Msg/⊤, X/b, P/b)
par_2([], P, P).
par_2([C|Cs], P0, P) :-
    xor_2(C, P0, P1),
    par_2(Cs, P1, P).

% ⟨xor(C, P0, P1), (C/⊤, P0/z, P1/⊤)⟩ ↦ (C/b, P0/z, P1/b)
xor_1(0,0,0).
xor_1(0,1,1).               % After abstract partial eval.:
xor_1(1,0,1).               % xor_1(0,0,0).
xor_1(1,1,0).               % xor_1(1,0,1).

% ⟨xor(C, P0, P1), (C/⊤, P0/b, P1/⊤)⟩ ↦ (C/b, P0/b, P1/b)
xor_2(0,0,0).
xor_2(0,1,1).
xor_2(1,0,1).
xor_2(1,1,0).
```

Fig. 3. The program specialization implicit in the analysis, after *version materialization*.

While beyond the scope of this paper, note also that multivariance is a form of *multiple specialization* of predicates. For example, the graph in Figure 1 contains two *versions* of predicate `par/3` and another two of `xor/3` and implies the specialization shown in Figure 3. This is referred to as *materializing* the versions in the analysis graph (Muthukumar and Hermenegildo 1992).

Reconstructing the paths of concrete executions. The analysis graph, through the *edges* ($\langle A, \lambda^c\rangle \rightarrow_{k,i} \langle B, \lambda^{c'}\rangle$) relation, also provides an abstraction of the *paths* explored by the concrete executions through the program, represented by the concrete trees. In particular, it is possible to reconstruct, for every node, all possible (and possibly infinite) execution trees that lead to the call pattern described by the node, by following the edges of the analysis graph. The analysis graph thus embodies two different abstractions (two different abstract domains): the graph itself is a *regular approximation* of the paths through the program, using a domain of regular structures. Separately, the abstract values (call and success patterns) contained in the graph nodes are finite representations of the states occurring at each point in the program paths, by means of the *data abstract domain.* Note that the path abstraction implicit in the graph is more powerful than the call stack representation in the well-known call strings method introduced of Sharir and Pnueli (1978) (see, e.g., Khedker and Karkare 2008; Thakur and Nandivada 2020 for two recent examples of use), as this method only keeps track of the *callers* of the abstracted call, and typically as a limited-length sequence (Sharir and Pnueli 1978), whereas we infer, as a regular tree, all the arbitrarily large sequences of procedures *executed* before that call, that is, not only its direct callers or a limited-depth sequence. Note also that, as mentioned before, our analysis includes also the call patterns and paths leading to failure or non-termination in the concrete semantics (for all of which the answer pattern will be $\perp$ (s.t. $\gamma(\perp) = \emptyset$).

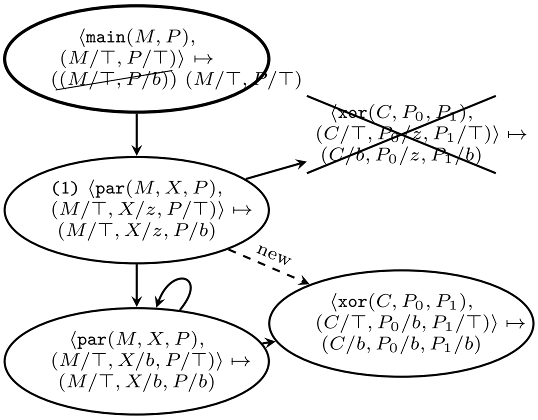

⟨main(M, P), (M/⊤, P/⊤)⟩ ↦ ~~((M/⊤, P/b))~~ (M/⊤, P/⊤)
~~⟨xor(C, P0, P1), (C/⊤, P0/z, P1/⊤)⟩ ↦ (C/b, P0/z, P1/b)~~
(1) ⟨par(M, X, P), (M/⊤, X/z, P/⊤)⟩ ↦ (M/⊤, X/z, P/b)
new
⟨xor(C, P0, P1), (C/⊤, P0/b, P1/⊤)⟩ ↦ (C/b, P0/b, P1/b)
⟨par(M, X, P), (M/⊤, X/b, P/⊤)⟩ ↦ (M/⊤, X/b, P/b)

Fig. 4. Graph after the modification operations.

Notation for and operations on analysis results. The following operations defined over an analysis result g allow us to inspect and manipulate analysis results.

$\langle A, \lambda^c \rangle \in g$:	there is a node in the call graph of g with key $\langle A, \lambda^c \rangle$.
$\langle A, \lambda^c \rangle \mapsto \lambda^s \in g$:	there is a node in g with key $\langle A, \lambda^c \rangle$ and the answer mapped to that call is λ^s.
$\langle A, \lambda^c \rangle \rightarrow \langle B, \lambda^{c'} \rangle \in g$:	there are two nodes ($n = \langle A, \lambda^c \rangle$ and $n' = \langle B, \lambda^{c'} \rangle$) in g and there is an edge from n to n'.
$\mathsf{del}(g, \{n_i\})$:	removes from g nodes n_i and its incoming and outgoing edges and unsets the element in the mapping function (it becomes undefined for all n_i).
$\mathsf{upd}(g, \langle A, \lambda^c \rangle \mapsto \lambda^s)$:	overwrites the value of $\langle A, \lambda^c \rangle$ in the mapping function and, if necessary, adds a node to g with key $\langle A, \lambda^c \rangle$.
$\mathsf{upd}(g, \{n \rightarrow n'\})$:	adds an edge from node n to node n' if it did not exist.
$\mathsf{upd}(g, \{e_i\})$:	performs $\mathsf{upd}(g, e_i)$ for each element of $\{e_i\}$.

Example 2
To illustrate the graph operations, we show some examples of operations done to the analysis graph of Figure 1, that we will refer to with $\mathscr{A}$.

- Check if there is a call to `par/3` with the second argument as 0: $\langle \texttt{par}(M, X, P), (M/\top, X/z, P/\top) \rangle \in \mathscr{A}$. This is true (node $\boxed{1}$).
- Check if there is a call to `main/2`, that, if it succeeds the second argument is a *bit*: $\langle \texttt{main}(M, P), \lambda^c \rangle \mapsto (M/\top, P/b) \in \mathscr{A}$. This is true (entry node).
- Check if there is a literal with `xor/3` in any of the clauses of `main/2`: $\langle \texttt{main}(M, P), \_ \rangle \rightarrow \langle \texttt{xor}(C, P_0, P_1), \_ \rangle \in \mathscr{A}$. This is false, there is a path from `main/2` to nodes containing `xor/3` but there is not a direct call.

These operations do not modify the graph.

Example 3
To illustrate the graph modification operations, we show some examples of operations done to the analysis graph of Figure 1, referred to again with $\mathscr{A}$.

- Remove the node for the abstract call $\langle \texttt{xor}(C, P_0, P_1), (C/\top, P_0/z, P_1/\top)\rangle$: $\mathsf{del}(\mathscr{A}, \{\langle \texttt{xor}(C, P_0, P_1), (C/\top, P_0/z, P_1/\top)\rangle\})$.
- Update the node for **main/2** with a more general success pattern: $\mathsf{upd}(\mathscr{A}, \langle \texttt{main}(M, P), (M/\top, P/\top)\rangle \mapsto (M/\top, P/\top))$.
- Add an edge from node $\boxed{1}$ to the remaining node for **xor/3**: $\mathsf{upd}(\mathscr{A}, \{\langle \texttt{par}(M, X, P), (M/\top, X/z, P/\top)\rangle \rightarrow \langle \texttt{xor}(C, P_0, P_1), (C/\top, P_0/b, P_1/\top)\rangle\})$.

After these operations, the state of the analysis graph is depicted in Figure 4.

4 The baseline analysis algorithms

The *analysis algorithms* are the fixpoint-calculating procedures that infer the analysis graphs, described in the previous section, so that they safely approximate the given program semantics. *Incremental* algorithms are those that can modify and recalculate such analysis graphs after program changes, without having to start the process from scratch. *Modular* algorithms (in contrast to *monolithic* algorithms) are those that are capable of analyzing a modular partition of a program (see Section 2) without having to load or treat the whole program at any given step.

In this section, we present our baseline algorithms, which already include some improvements with respect to previous descriptions, while in Section 5 we will present the incremental and modular algorithm that is our main contribution.

4.1 The monolithic and incremental fixpoint algorithm

We now present our first baseline, the monolithic incremental analysis algorithm of Hermenegildo *et al.* (2000), extended with widening to ensure termination in the presence of infinite abstract domains. This algorithm (Figure 5) takes as input a program P, a set of initial abstract queries Q_α, the differences Δ of P with respect to a previous version P', and an analysis result that is *correct* for P'. We will refer to this algorithm with $\mathscr{A} = \textbf{IncAnalyze}(P, Q_\alpha, \Delta, \mathscr{A}_0)$. Note that if the algorithm is called with $\mathscr{A}_0$ an empty analysis, that is, *from scratch*, then it is the same as the traditional PLAI algorithm (Muthukumar and Hermenegildo 1992). As mentioned before, we will refer to these to algorithms as *monolithic* because they assume that all the predicates executed in the target program P are provided to the analyzer, that is, these algorithms treat only whole programs.

Operation of the algorithm. The algorithm is centered around processing two kinds of events: *newcall* events, which control which predicates and clauses of the program that need reanalysis, and *arc* events, which process the body of one clause for a call pattern, starting at a certain literal. The algorithm starts by queueing a *newcall* event for each of the call patterns that need to be (re)computed. This triggers $\texttt{process}(\mathit{newcall}(\langle A, \lambda^c\rangle))$, which processes all the clauses of predicate A. For each of them, the abstract call is performed (`Acall`, which includes the renaming) and an *arc* event is added for the first literal. The initial-guess function returns a guess of the answer, λ^s, to $\langle A, \lambda^c\rangle$. If possible, it reuses the results in $\mathscr{A}$, otherwise returns $\perp$. Procedure `reanalyze_updated` propagates the information of new computed answers across the analysis graph by

```
ALGORITHM IncAnalyze($P, Q_\alpha, \Delta, \mathscr{A}$)
1:  for all $\langle A, \lambda^c\rangle \in Q_\alpha$ do
2:      add-event($newcall(\langle A, \lambda^c\rangle)$)
3:  delete_clauses($\Delta$)
4:  add_clauses($\Delta$)
5:  analysis_loop()
6:  return $\mathscr{A}$
7:  procedure analysis_loop()
8:      while events() $\neq \emptyset$ do
9:          $E$ := next-event()
10:         process($E$)
11: procedure add_clauses($Cls$)
12:     for all $A_k$ :- $A_{k,1}, \ldots, A_{k,n_k} \in Cls$ do
13:         for all $\langle A, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}$ do
14:             $\lambda^p$ := Acall($\lambda^c, A, A_k$)
15:             ${\lambda^c}_1$ := Aproj($\lambda^p, vars(A_{k,1})$)
16:             add-event($arc(\langle A, \lambda^c\rangle \rightarrow_{k,1} \langle A_{k,1}, {\lambda^c}_1\rangle)$)
17: procedure delete_clauses($Cls$)
18:     $Calls := \{\langle A, \lambda^c\rangle | \langle A, \lambda^c\rangle \in \mathscr{A}, (A_k$ :- $\ldots) \in Cls\}$
19:     $Ns := \{N \in \mathscr{A} | N \rightsquigarrow C \in \mathscr{A}, C \in Calls\}$
20:     del($\mathscr{A}, Ns$)
21: function lookup_answer($\langle A, \lambda^c\rangle$)
22:     if $\langle A, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}$ then
23:         return $\lambda^s$
24:     else
25:         add-event($newcall(\langle A, \langle A, \lambda^c\rangle\rangle)$)
26:         return $\perp$
27: procedure reanalyze_updated($\langle A, \lambda^c\rangle$)
28:     for all $E := \langle B, {\lambda^c}_0\rangle \rightarrow_{k,i} \langle A, \lambda^c\rangle \in \mathscr{A}$ do
29:         add-event($arc(E)$)
30: procedure process($newcall(\langle A, \lambda^c\rangle)$)
31:     for all $A_k$ :- $A_{k,1}, \ldots, A_{k,n_k} \in Cls$ do
32:         $\lambda^p$ := Acall($\lambda^c, A, A_k$)
33:         ${\lambda^c}_1$ := Aproj($\lambda^p, vars(A_{k,1})$)
34:         add-event($arc(\langle A, \lambda^c\rangle \rightarrow_{k,1} \langle A_{k,1}, {\lambda^c}_1\rangle)$)
35:     $\lambda^s$ := initial-guess($\langle A, \lambda^c\rangle$)
36:     if $\lambda^s \neq \perp$ then
37:         reanalyze_updated($\langle A, \lambda^c\rangle$)
38:     upd($\mathscr{A}, \langle A, \lambda^c\rangle \mapsto \lambda^s$)
39: procedure process($arc(\langle A, {\lambda^c}_0\rangle \rightarrow_{k,i} \langle B, {\lambda^c}_1\rangle)$)
40:     $Calls := \{\lambda \mid \langle A, \_\rangle \rightarrow_{k,i} \langle B, \lambda\rangle \in \mathscr{A}\}$
41:     $\lambda^c$ := Ageneralize(${\lambda^c}_1, Calls$)
42:     if $B$ is a built-in then
43:         ${\lambda^s}_0 := f^\alpha(\langle B, \lambda^c\rangle)$
44:     else ${\lambda^s}_0$ := lookup_answer($\langle B, \lambda^c\rangle$)
45:     upd($\mathscr{A}, \langle A, {\lambda^c}_0\rangle \rightarrow_{k,i} \langle B, \lambda^c\rangle$)
46:     $\lambda^r$ := Aextend($\lambda^p, {\lambda^s}_0$)
47:     if $\lambda^r \neq \perp$ and $i \neq n_k$ then
48:         ${\lambda^c}_2$ := Aproj($\lambda^r, vars(A_{k,i+1})$)
49:         add-event($arc(\langle H, {\lambda^c}_0\rangle \rightarrow_{k,i+1} \langle B, {\lambda^c}_2\rangle)$)
50:     else if $\lambda^r \neq \perp$ and $i = n_k$ then
51:         $\lambda^s$ := Aproj($\lambda^r, vars(A_k)$)
52:         insert_answer_info($\langle A, {\lambda^c}_0\rangle, \lambda^s$)
53: procedure insert_answer_info($\langle A, \lambda^c\rangle, \lambda^s$)
54:     if $\langle A, \lambda^c\rangle \mapsto {\lambda^s}_0 \in \mathscr{A}$ then
55:         ${\lambda^s}_1$ := Ageneralize($\lambda^s, \{{\lambda^s}_0\}$)
56:     else ${\lambda^s}_0 := \perp$, ${\lambda^s}_1 := \lambda^s$
57:     if ${\lambda^s}_0 \neq {\lambda^s}_1$ then
58:         upd($\mathscr{A}, \langle A, \lambda^c\rangle \mapsto {\lambda^s}_1$)
59:         reanalyze_updated($\langle A, \lambda^c\rangle$)
```

Fig. 5. The monolithic, context-sensitive, incremental fixpoint algorithm.

creating *arc* events with the literals from which the analysis has to be restarted. `process`($arc(\langle A_k, \lambda^c\rangle \rightarrow_{k,i} \langle B, \lambda^c\rangle)$) performs a single step of the left-to-right traversal of a clause body. Since the algorithm is multivariant, an infinite number of different call patterns may be encountered, even if the domain has finite height. In this case, the call patterns are generalized, via a widening operator, denoted by the `Ageneralize` operation. Then, if the literal $A_{k,i}$ is a *built-in*, its transfer function is applied; otherwise, an edge is added to $\mathscr{A}$ and the λ^s is looked up, which includes creating a *newcall* event for $\langle A, \lambda^c\rangle$ if the answer is not in the analysis graph. The answer is combined with the description λ^p from the literal immediately before $A_{k,i}$ to obtain the description (return) for the literal after $A_{k,i}$. 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 the previous answers. To ensure termination when analyzing with abstract domains with infinite ascending chains, this answer needs to be generalized, also with a widening operator (`Ageneralize`). Lastly, the new answer is propagated 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 $\mathscr{A}$ 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 (all the dependent nodes in the graph).

Differences w.r.t. the original incremental algorithm. The algorithm presented in Figure 5 differs from the one described in Hermenegildo *et al.* (2000) only in lines 40 and 41, which perform the widening of the encountered call patterns for the cases in which the abstract domain has infinite width, and in line 55, that performs the widening on the success for the cases in which the abstract domain is of infinite height. The abstract interpretation technique guarantees that generalization with a widening operation preserves soundness and guarantees termination at the expense of losing of precision. Since widening may not be necessary for all domains, it may be disabled in the algorithm by:

- removing line 40,
- replacing line 41 by "$\lambda^c := \lambda^c{}_1$",
- and replacing line 55 by "$\lambda^s{}_1 := \lambda^s \sqcup \lambda^s{}_0$".

4.1.1 Correctness

We now formulate the correctness results of the algorithm *with generalization*, that is, as presented in Figure 5.

Definition 1 (*Correctly approximated calls*)
Let P be a program, Q a set of initial concrete queries, and $\mathscr{A}$ an analysis graph. We say that $\mathscr{A}$ *correctly approximates the calls in* $[\![P]\!]_Q$ if all encountered call patterns during the concrete execution are contained in $\mathscr{A}$. That is, for all predicates A in P:

$$\forall \theta^c \in \mathsf{calling\_context}(A, P, Q). \exists \langle A, \lambda^c \rangle \mapsto \lambda^s \in \mathscr{A} \text{ s.t. } \theta^c \in \gamma(\lambda^c).$$

Definition 2 (*Correctly approximated answers*)
Let P be a program, Q a set of initial concrete queries, and $\mathscr{A}$ an analysis graph. We say that *the answers in* $\mathscr{A}$ *correctly approximate the answers in* $[\![P]\!]_Q$ if they abstract all the answer patterns to the encountered call patterns. That is, for all predicates A of P:

$$\forall \langle A, \lambda^c \rangle \mapsto \lambda^s \in \mathscr{A}, \forall \theta^c \in \gamma(\lambda^c) \text{ if } \theta^s \in \mathsf{answers}(P, \{\langle A, \theta^c \rangle\}) \text{ then } \theta^s \in \gamma(\lambda^s).$$

Definition 3 (*Correct global analysis*)
Let P be a program, Q a set of initial concrete queries, and $\mathscr{A}$ an analysis graph. $\mathscr{A}$ *is correct for* P, Q if

(a) $\mathscr{A}$ correctly approximates the calls for P, Q (Definition 1) and
(b) $\mathscr{A}$ correclty approximates the answers for P, Q (Definition 2).

Given these definitions, the following Theorems 1, 2, and 3 from Hermenegildo *et al.* (2000) hold, because, as stated earlier, generalization via a widening guarantees correctness:

Theorem 1 (*Correctness of* IncAnalyze *from scratch*)
Let P be a program and Q_α a set of abstract queries. The analysis result $\mathscr{A} = \text{IncAnalyze}(P, Q_\alpha, \emptyset, \emptyset)$ for P with Q_α is *correct* for P and $\gamma(Q_\alpha)$.

Theorem 2 (*Correctness of* IncAnalyze *adding clauses*)
Let P and P' be two programs such that s.t. $\Delta = (C_{add}, \emptyset)$, $P = (P' \cup C_{add})$, and Q_α a set of abstract queries. If $\mathscr{A}_0 = \text{IncAnalyze}(P', Q_\alpha, \emptyset, \emptyset)$, then the analysis result $\mathscr{A} = \text{IncAnalyze}(P, Q_\alpha, \Delta, \mathscr{A}_0)$ for P with Q_α *correct* for P and $\gamma(Q_\alpha)$.

Theorem 3 (*Correctness of* IncAnalyze *deleting clauses*)
Let P and P' be two programs such that s.t. $\Delta = (\emptyset, C_{del})$, $P = P' \setminus C_{del}$, and Q_α a set of abstract queries. If $\mathscr{A}_0 = \textsc{IncAnalyze}(P', Q_\alpha, \emptyset, \emptyset)$, then the analysis result $\mathscr{A} = \textsc{IncAnalyze}(P, Q_\alpha, \Delta, \mathscr{A}_0)$ for P with Q_α *correct* for P and $\gamma(Q_\alpha)$.

We introduce a new theorem that generalizes Theorems 1, 2, and 3.

Theorem 4 (*Correctness of* IncAnalyze *starting from a partial analysis*)
Let P be a program, Q_α a set of abstract queries, and $\mathscr{A}_0$ *any analysis graph*. Let $\mathscr{A} = \textsc{IncAnalyze}(P, Q_\alpha, \emptyset, \mathscr{A}_0)$. $\mathscr{A}$ is *correct* for P and $\gamma(Q_\alpha)$ if for all concrete queries $Q \in \gamma(Q_\alpha)$ all nodes N from which there is a path in the concrete execution $Q \rightsquigarrow N$ in $[\![P]\!]_Q$that are abstracted in the analysis $\mathscr{A}_0$ are included in Q_α, that is:

$$\forall Q, N. Q \in \gamma(Q_\alpha) \wedge Q \rightsquigarrow N \in [\![P]\!]_Q, \forall N_\alpha \in \mathscr{A}_0 . N \in \gamma(N_\alpha) \Rightarrow N_\alpha \in Q_\alpha.$$

Intuitively, the algorithm is correct for any query Q not already abstracted in $\mathscr{A}_0$. If $\mathscr{A}_0$ contains already information about Q, it needs to be rechecked by recomputing the analysis of all the nodes in which Q depends by including them in Q_α. Theorem 4 is a generalization because, implicitly, procedures `add_clauses` and `delete_clauses` are doing exactly, this: either removing the analysis so that it is computed from scratch again or adding the necessary queries (directly by creating the corresponding *newcall* events) to guarantee that the analysis is correct.

Proof
This follows from the creation of a *newcall* event for each of the queries Q_α. The processing of the events trigger the recomputation and later update of all the nodes of the analysis graph that are potentially under the fixpoint. □

Note that $\mathscr{A}_0$ is not assumed to be the (correct) output of a previous analysis, it can be any analysis (below, above, or incomparable with the fixpoint). Also note that if all nodes in the analysis graph are included, together with the original queries, in Q_α, the result is guaranteed to be correct.

4.1.2 Precision

If generalization is removed from the algorithm, as indicated in Section 4.1, and assume that initial-guess returns a value below the least fixed point, the following precision result from Hermenegildo *et al.* (2000) is preserved when analyzing with finite abstract domains:

Theorem 5 (*Precision of* IncAnalyze)
Let P and P' be programs, such that P differs from P' by Δ, let Q_α a set of abstract queries, and $\mathscr{A}_0 = \textsc{IncAnalyze}(P', Q_\alpha, \emptyset, \emptyset)$ an analysis graph. The following hold

- If $\mathscr{A} = \textsc{IncAnalyze}(P, Q_\alpha, \emptyset, \emptyset)$, then $\mathscr{A}$ is the *least program analysis graph* for P and $\gamma(Q_\alpha)$, and
- $\textsc{IncAnalyze}(P, Q_\alpha, \Delta, \mathscr{A}_0) = \textsc{IncAnalyze}(P, Q_\alpha, \emptyset, \emptyset)$.

That is, when analyzing from scratch, always the most precise result is produced, and when reusing a least program analysis graph in the incremental analysis, the new result is the least program analysis graph as well. This means that there is no analysis graph with smaller call or answer patterns that correctly over-approximates the behavior of the program.

Theorem 5 shows that, if the $\mathscr{A}_0$ is a correct and precise analysis, then the incremental analysis result is correct and precise. However, the conditions on $\mathscr{A}_0$ can be relaxed if we strengthen the conditions on the queries and still guarantee the same precision/correctness results. The following new theorem states the general condition for guaranteeing precision when (re)starting from a partial analysis result.

Theorem 6 (*Precision of* IncAnalyze *starting from a partial analysis*)
Let P be a program, Q_α a set of abstract queries, and $\mathscr{A}_0$ an analysis graph *below the least fixed point (lfp)*, that is, $\forall \langle A, \lambda^c \rangle \mapsto {\lambda^s}_0 \in \mathscr{A}_0 . \langle A, \lambda^c \rangle \mapsto \lambda^s \in \mathscr{A} \wedge {\lambda^s}_0 \sqsubseteq \lambda^s$, and the conditions on Q_α of Theorem 4 hold then:

$$\text{IncAnalyze}(P, Q_\alpha, \emptyset, \emptyset) = \text{IncAnalyze}(P, Q_\alpha, \emptyset, \mathscr{A}_0).$$

Proof
The abstract interpretation technique (Cousot and Cousot 1977) guarantees that the fixed point of a set of monotonic equations can be computed by repeatedly applying each of the equations in a chaotic iteration manner. If the iteration is started at $\perp$, it is guaranteed that the least fixed point of the equations is found. In our case, the equations are the Horn clauses that encode the (concrete) semantics of the program P. Let $f_P(X)$ be one step of the chaotic iteration, that is, *applying semantics of one clause of P to the current value of the sequence.* When starting from an empty analysis, IncAnalyze will compute the *lfp* by applying $f_P(X)$ a number of times:

$$\perp \sqsubseteq f_P(\perp) \sqsubseteq f_P(f_P(\perp)) \sqsubseteq f_P^3(\perp) \sqsubseteq \ldots \sqsubseteq f_P^k(\perp) = \ldots = f_P^{k+n}(\perp) = \mathit{lfp}(P).$$

In the sequence above, the fixpoint value is reached in the k-th step of the iteration. However, this value is not confirmed yet to be the fixpoint. The chaotic iteration process needs to continue until all the equations have been exhaustively applied and the value of the fixpoint is kept, and this is represented by the n steps after f_P^k. Note that the number of steps k and n will depend highly on the strategy for the chaotic iteration. In our case, we safely reduce them by keeping the dependencies between clauses.

Starting from a partial analysis is equivalent to computing the Kleene fixpoint of the original program including a new equation, which is a constant, representing the initial results. Let us call this equation $\mathscr{A}_0$. Our goal is to prove that chaotic iteration of f_P with $\mathscr{A}_0$ also results in the $\mathit{lfp}(P)$ if $\mathscr{A}_0 \sqsubseteq \mathit{lfp}(P)$.

By definition, for any k-th step of the iteration $f_P^k(\perp) \sqsubseteq \mathit{lfp}(P)$, also, by hypothesis, $\mathscr{A}_0 \sqsubseteq \mathit{lfp}(P)$. Therefore, for any k and applying any random clause, $\mathscr{A}_0 \sqcup f_P^k(\perp) \sqsubseteq \mathit{lfp}(P)$. So, if we "plug in" the initial analysis $\mathscr{A}_0$ at any point of the chaotic iteration over f_P, because the equations of P are monotonic, for any k, $f_P(\mathscr{A}_0 \sqcup f_P^k(\perp)) \sqsubseteq f_P(\mathit{lfp}(P))$, and precision is preserved. Concretely, this also implies that precision is preserved if we start from $f_P(\mathscr{A}_0)$.

The condition imposed on the set of queries guarantees that the chaotic iteration includes all the equations that the iteration needs to be rerun with (see Theorem 4). This justifies not reprocessing the equations that are not affected by the changes in the algorithm, since the corresponding steps can be skipped safely. □

Note that these precision results imply also correctness since the *lfp* is obtained, which was already proved in Section 4.1.1. Nevertheless, precision has been included separately because it does not hold in the presence of generalization: using widening, as required for dealing with infinite domains, implies not being able to guarantee that the least fixed point is obtained, and given that this operator is not only assumed to be associative but also does not guarantee the analysis result will be the same (i.e., that the same imprecision is obtained), as this depends on how the processing of the events is scheduled.

4.2 The modular fixpoint algorithm

We now present the reference algorithm for analyzing modular programs, described in Puebla *et al.* (2004). As expected, the approach consists in analyzing partitions of programs making assumptions about the code that is external to each partition. Several possibilities were proposed in that work for making such assumptions, including, for example, assuming that nothing is known about the answer ($\top$), computing the "topmost" abstraction of the call (as before but taking into account any local information available), or strategies with better precision but, in general, more costly, such as assuming $\perp$ temporarily for the unknown answers and later reanalyzing whenever a better abstraction of the answer is available. In this work, we fix the strategy to the latter one in order to obtain the best precision. Also, module analysis order may affect the speed at which the fixpoint computation converges. Some scheduling policies were studied in Correas *et al.* (2006). We provide a new pseudocode for the algorithm of Puebla *et al.* (2004), specialized for the case in which the maximum precision is aimed for. Then, we provide new formal results about correctness and precision of this algorithm. Also, both for generality and reusability, although not required for our results, we propose a formulation of the algorithm that is parametric on the analysis used within each modular partition, which in our case is instantiated to IncAnalyze.

Modular analysis results. To store the overall analysis result of the program and keep track of fine-grain dependencies between modules, we propose to use also an analysis graph structure at the intermodular level. One can see this as a sort of "projection" of the *monolithic* analysis graph, described in Section 3, in which only the information about the predicates in the boundaries of the modules is kept. Nodes represent calls to predicates and edges capture the relations between the predicates in the boundaries of the partitions (exported/imported predicates) with arcs $\langle A, \lambda^c\rangle \rightarrow \langle B, \lambda^{c'}\rangle$ meaning *a call to A in $\mathsf{mod}(A)$ with description λ^c may cause a call to B with description $\lambda^{c'}$ and* $\mathsf{mod}(B) \in \mathsf{imports}(\mathsf{mod}(A))$. From this point on, we will use $\mathcal{G}$ to denote the modular (global) analysis graph and $\mathscr{L}$ to denote the analysis of a single module (local analysis graph).

Figure 6 shows a modular version of the program and analysis results of Figure 1. The nodes of this (global) analysis graph encode that calling the exported predicate `main/1` of module `main` may cause a call to `xor/3` exported by module `bitops` with two different call descriptions (two edges).

Operation of the algorithm. The algorithm takes as input a (partitioned) program $P = \{M_i\}$, some initial queries Q_α to any exported predicate of the program, that is, any

```
:- module(main, [main/1]).

:- use_module(bitops).
main(Msg, P) :-
    par(Msg, 0, P).

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

```
:- module(bitops, [xor/3]).

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

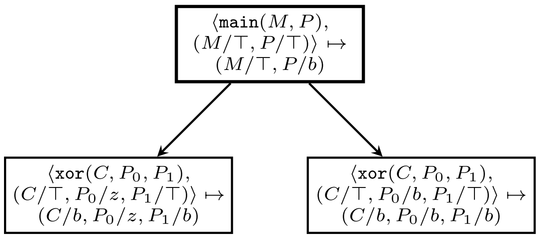

$\langle \mathtt{main}(M, P), (M/\top, P/\top)\rangle \mapsto (M/\top, P/b)$
$\langle \mathtt{xor}(C, P_0, P_1), (C/\top, P_0/z, P_1/\top)\rangle \mapsto (C/b, P_0/z, P_1/b)$
$\langle \mathtt{xor}(C, P_0, P_1), (C/\top, P_0/b, P_1/\top)\rangle \mapsto (C/b, P_0/b, P_1/b)$

Fig. 6. Modular version of Figure 1 with a possible modular analysis result.

$\langle A, \lambda^c\rangle \in Q_\alpha, A \in \mathsf{exports}(\mathsf{mod}(A))$. If there are recursive dependencies between modules, the modules in each clique will be grouped and analyzed as a whole module (after doing the necessary renamings). This decision is based on the observation that, if we choose to not group modules that are in the same recursive clique, then, after program changes within the clique, we will have to delete all the internal analysis information, as we will see later, and this is essentially equivalent to considering the clique a single module. Alternatively, it would be possible in principle to pass more detailed information across modules, but then again this is essentially equivalent to doing monolithic incremental for the clique.

The pseudocode of the algorithm is detailed in Figure 7. Each of the modules in the program will be analyzed independently, and possibly several times. The algorithm keeps a queue of all the call patterns that need to be (re)analyzed for each module. To distinguish between the queries defined by the user and the intermediate queries done internally by the modular analysis algorithm, we will call the latter *entries* and they will be referred to with E. The queue is initialized with an entry for each of the abstract queries. Modular analysis is controlled by this queue that contains the call patterns with possibly incomplete answers (added with procedure add-entries). At each iteration of the loop, a module is reanalyzed independently for its set of annotated entries (E) extracted from the queue. This is done by procedure next-entries which extracts from the queue entries that are reachable from the initial Q_α in $\mathcal{G}$. In every iteration, modules are analyzed *from scratch*. This means that, in principle, the analysis of module M with entries E should be performed by $\mathscr{L} = \textsc{IncAnalyze}(M, E, \emptyset, \emptyset)$. However, IncAnalyze assumes that all code is available for analysis. Since this is not so in this modular case, IncAnalyze needs to be provided with an abstraction of the predicates imported by M. To this end, in line 5 (**PreloadImported**), the nodes and answers of the global graph $\mathcal{G}$ of predicates imported by M are added to $\mathscr{L}$. After this, $\mathcal{G}$ is updated, by propagating the newly computed answers (**StoreAnswers**), provided that a generalization is made before to ensure termination and updating the dependencies of the predicates in the boundary of the modules (**UpdateDependencies**), adding entries for the newly encountered call patterns (**ScheduleNewCalls**), and also generalizing them if necessary.

```
ALGORITHM ModAnalyze($P = \{M_i\}, Q_\alpha$)
1:  add-entries($\{k \in Q_\alpha \mid k \notin \mathcal{G}\}$), upd($\mathcal{G}, \{k \mapsto \bot \mid k \in Q_\alpha\}$)
2:  while entries($\mathcal{G}, Q_\alpha$) $\neq \emptyset$ do
3:      $(M, E)$ := next-entries($\mathcal{G}, Q_\alpha$)
4:      $\mathscr{L} := \emptyset$
5:      upd($\mathscr{L}, \{\langle A, \lambda^c\rangle \mapsto \lambda^s \in \mathcal{G} \mid$ mod($A$) $\in$ imports($M$)$\}$)      ▷ PreloadImported
6:      $\mathscr{L}$ := IncAnalyze($M, E, \emptyset, \mathscr{L}$)
7:      for $\langle P, \lambda^c\rangle \mapsto {\lambda^s}_l \in \mathscr{L}.\langle P, \lambda^c\rangle \mapsto {\lambda^s}_g \in \mathcal{G} \Rightarrow {\lambda^s}_l \neq {\lambda^s}_g$ do
8:          $\lambda^s :=$ Ageneralize($({\lambda^s}_l, \{{\lambda^s}_g\}$)
9:          upd($\mathcal{G}, \langle P, \lambda^c\rangle \mapsto \lambda^s$)      ▷ StoreAnswers
10:         add-entries($\{k \mid k \to \langle P, \lambda^c\rangle \in \mathcal{G}\}$)
11:     del($\mathcal{G}, \{\langle P, \lambda^c\rangle \to k' \in \mathcal{G}\}$)      ▷ UpdateDependencies
12:     $R = \{\langle P, \lambda^c\rangle \to k' \mid \exists\ \langle P, \lambda^c\rangle \rightsquigarrow k \in \mathscr{L}, k' = \langle A, \lambda^c\rangle,$ mod($A$) $\neq M\}$
13:     for $k \to \langle Q, {\lambda^c}_t\rangle \in R$ do
14:         $Calls := \{\lambda \mid \langle A, \lambda\rangle \in \mathcal{G}\}$
15:         $\lambda^c :=$ Ageneralize(${\lambda^s}_t, Calls$)
16:         if $\langle Q, \lambda^c\rangle \notin \mathcal{G}$ then      ▷ ScheduleNewCalls
17:             add-entries($\langle Q, \lambda^c\rangle\}$)
18:         upd($\mathcal{G}, \{k \to \langle Q, \lambda^c\rangle\}$)
19: return $\mathcal{G}$
```

Fig. 7. Modular fixpoint algorithm.

4.2.1 Correctness

We now formalize the notion of *correct modular analysis*. Let `first_ext_calls` $(E, [\![P]\!]_Q)$ be a function that, given a set of execution trees $[\![P]\!]_Q$, returns the set of calls reachable from any $e \in E$ that are the first reachable predicate that is imported by mod(e), together with E. That is:

$$\{C \mid \exists e \in E \text{ and } (e \rightsquigarrow C) \in [\![P]\!]_Q . \forall l \in (e \rightsquigarrow C).\mathsf{mod}(l) = \mathsf{mod}(e) \wedge \mathsf{mod}(C) \neq \mathsf{mod}(e)\} \cup E.$$

Definition 4 (*Correctly approximated intermodular calls*)
Let P be a program and Q a set of concrete queries, $\mathcal{G}$ an analysis graph, and E a set of entries, and let I be the transitive closure of `first_ext_calls`$(E, [\![P]\!]_Q)$. We say that $\mathcal{G}$ *correctly approximates the intermodular calls of* $[\![P]\!]_Q$, if it abstracts all the call patterns in the transitive closure of I. That is:

$$\forall \langle A, \theta^c\rangle \in I . \exists \langle A, \lambda^c\rangle \in \mathcal{G} \wedge \theta^c \in \gamma(\lambda^c).$$

That is, $\mathcal{G}$ contains all the calls of the exported predicates that were originated from a different module in which they are defined, and that are reachable from Q. Note that this set in the concrete execution may be infinite, for example, in the case in which an imported predicate is called inside a loop.

Definition 5 (*Correct modular analysis*)
Given a program P, split in modules M_i, and initial concrete queries Q, we say a modular analysis graph $\mathcal{G}$ *is correct for* P, Q if:

(a) it approximates the intermodular calls correctly (see Definition 4) and
(b) it approximates the answers correctly (see Definition 2).

As mentioned earlier, IncAnalyze assumes that either the procedures executed by a program are defined in the clauses provided to the analyzer or they are basic, built-in operations of the language, that is, they are interpreted applying their corresponding transfer function. This is not the case when analyzing programs module by module, and assumptions need to be made about the imported code. The following lemma states that the analysis graph inferred by IncAnalyze is correct assuming the answers of $\mathscr{L}_0$ if it only contains abstractions of the imported predicates. In other words, if $\mathscr{L}_0$ correctly over-approximates the behavior of the imported predicates, then the analysis of the module is correct.

Lemma 1 (*Correctness of* IncAnalyze *modulo imported predicates*)
Let M be a module of program P and E a set of abstract queries. Let $\mathscr{L}_0$ be an analysis graph such that $\forall \langle A, \lambda^c \rangle \in \mathscr{L}_0 . \mathsf{mod}(A) \in \mathsf{imports}(M)$. The analysis result:

$$\mathscr{L} = \textsc{IncAnalyze}(M, E, \emptyset, \mathscr{L}_0)$$

is *correct* (see Definition 3) for M and $\gamma(E)$ assuming $\mathscr{L}_0$.

Proof
By Theorem 1, IncAnalyze produces a correct analysis whenever the initial analysis graph is empty. Since, $\mathscr{L}_0$ contains only information about the imported predicates, the analysis graph inferred is correct for all the predicates in M, assuming that the original information in $\mathscr{L}_0$ is correct. □

Theorem 7 (*Correctness of* ModAnalyze)
Let P be a modular program and Q_α a set of abstract queries. The modular analysis graph:

$$\mathcal{G} = \textsc{ModAnalyze}(P, Q_\alpha)$$

is *correct* (Definition 5) for P and $\gamma(Q_\alpha)$.

Proof
By induction on the number of modular partitions, if there is only one partition, the conditions in Definition 4 hold trivially because the only intermodular call patterns are the Q_α (added in line 1). Since $\mathscr{L}$ is correct by Theorem 1 and the results are updated in line 9, the conditions in Definition 2 hold. And no further iteration is required.

If the program P is partitioned into n modules, we need to prove that if analyzing $n-1$ modules finishes, then analyzing all n modules also finishes. Assuming that the analysis of the first $n-1$ modules finishes and is correct, the result of these $n-1$ modules could be seen as one module, reducing this general case to the case of two modules. To prove this, the following invariant of the algorithm is used:

Before extracting from the queue via next-entries *(line 2), either the results in $\mathcal{G}$ are correct, or the queue is not empty.*

This invariant trivially holds immediately after initializing the queue with the queries in line 1. Then, at each iteration of the while loop, since there are only two modules, when one is extracted from the queue, the queue is empty. After analyzing (line 4), we know $\mathscr{L}$ is correct if $\mathcal{G}$ was correct. If no answers changed w.r.t. $\mathcal{G}$, no modules are added and the fixed point was reached. If the results change, every answer that changed is generalized and updated in $\mathcal{G}$, which results in adding an entry to it (line 9). Then, since there are only two modules, there can be at most one module in the queue, since the one being processed is extracted. If after processing one module, the nodes and answers (excluding the answers to Q_α) stay the same, no new events will be added to the queue. In this case, then the analysis is already correct, by Lemma 1, because IncAnalyze was performed assuming already correct information. Else, if new answers were encountered it means that the previous information was incomplete, these answers are stored (line 9), and the entries that depend on these answers are added to the queue, so the invariant holds. If new call patterns were encountered, then it means that the analysis was not completed yet. The algorithm, after generalization, schedules them to be reanalyzed (line 17), and therefore the invariant holds as well. $\square$

As mentioned earlier, the goal of this algorithm was not to perform incremental analysis but rather to reduce the working set of the basic (monolithic) analyzer. In fact, in Puebla *et al.* (2004), the authors neither provide a clear strategy of how to tackle the problem of reusing the analysis result after making modifications to the program nor perform experiments.

4.2.2 Precision

We now show the precision guarantees when analyzing with finite abstract domains if the generalization step is removed, that is, by:

- replacing line 8 by $\lambda^s := \lambda^s{}_l \sqcup \lambda^s{}_g$,
- removing line 14, and
- replacing line 15 by $\lambda^c := \lambda^s{}_t$.

Lemma 2 (*Precision of* IncAnalyze *modulo imported predicates*)
Let M be a module of program P and E a set of abstract queries. Let $\mathscr{L}_0$ be an analysis graph such that $\forall \langle A, \lambda^c \rangle \in \mathscr{L}_0.\mathsf{mod}(A) \in \mathsf{imports}(M)$ if $\mathscr{L}_0$ contains *the least fixed point* as defined in Theorem 6. The analysis result:

$$\mathscr{L} = \text{IncAnalyze}(M, E, \emptyset, \mathscr{L}_0)$$

is the *least program analysis graph* for M and $\gamma(E)$ assuming $\mathscr{L}_0$.

Proof
Since all values reused are the least fixed point, no imprecision is introduced by $\mathscr{L}_0$. Correctness follows from Lemma 1. $\square$

Theorem 8 (*Precision of* ModAnalyze)
Let P be a modular program and Q_α a set of abstract queries. The modular analysis result:

$$\mathcal{G} = \text{ModAnalyze}(P, Q_\alpha)$$

is the *least modular analysis graph* for P and $\gamma(Q_\alpha)$.

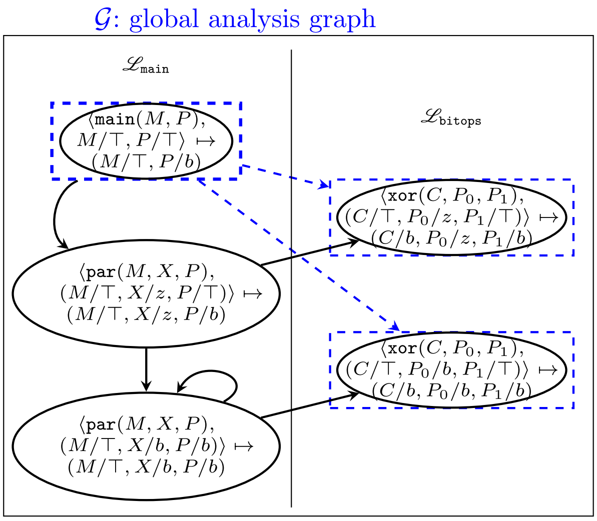
```
:- module(main, [main/1]).

:- use_module(bitops).
main(Msg, P) :-
    par(Msg, 0, P).

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

```
:- module(bitops, [xor/3]).

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



$\mathcal{G}$: global analysis graph
$\mathscr{L}_{\texttt{main}}$
$\mathscr{L}_{\texttt{bitops}}$
$\langle \texttt{main}(M, P), M/\top, P/\top\rangle \mapsto (M/\top, P/b)$
$\langle \texttt{par}(M, X, P), (M/\top, X/z, P/\top)\rangle \mapsto (M/\top, X/z, P/b)$
$\langle \texttt{par}(M, X, P), (M/\top, X/b, P/b)\rangle \mapsto (M/\top, X/b, P/b)$
$\langle \texttt{xor}(C, P_0, P_1), (C/\top, P_0/z, P_1/\top)\rangle \mapsto (C/b, P_0/z, P_1/b)$
$\langle \texttt{xor}(C, P_0, P_1), (C/\top, P_0/b, P_1/\top)\rangle \mapsto (C/b, P_0/b, P_1/b)$

Fig. 8. A modular version of Figure 1 keeping a local analysis graph per module.

Proof
Since no imprecision is introduced during the modular processing, and all answers are started assuming $\perp$ (line 1), each of the calls to IncAnalyze will produce results that are below or exactly the least fixed point. Correctness follows from Theorem 7. □

5 The algorithm for incremental and modular context-sensitive analysis

We now propose an algorithm that performs a goal-directed, top-down, incremental abstract interpretation of modular Horn clause programs. The analyzer takes a program (target), a set of initial call states, and, optionally, analysis results of a previous version of the program, and information about the changes w.r.t. the target program. The analyzer will annotate the program with information about the current state of the variables at each clause and literal whenever they are reached when executing the calls described by the initial call states, reusing as much of the provided analysis results as possible.

Analysis graphs for modular and incremental analysis. To have an algorithm that processes partitions of programs modularly but, at the same time, is able to update localized information we propose to keep, in addition to $\mathcal{G}$, a local analysis graph per modular partition M, referred to with $\mathscr{L}_M$. The analysis result then consists on a set of graphs $\{\mathcal{G}, \{\mathscr{L}_i\}\}$. An example of an analysis result of this shape is depicted in Figure 8. The information of the local analysis graphs is drawn in black and with nodes as ellipses. The left box corresponds to the `main` module, $\mathscr{L}_{\texttt{main}}$, and the box on the left to the `bitops` module, $\mathscr{L}_{\texttt{bitops}}$. The nodes in blue, dashed, and with rectangles show the information in the global analysis graph $\mathcal{G}$, which coincides with Figure 6.

5.1 Operation of the algorithm

The algorithm takes as input a (partitioned) program $P = \{M_i\}$, some initial queries Q_α, a previous correct analysis result $\{\mathcal{G}, \{\mathscr{L}_i\}\}$, and a set of program edits in the form of

```
ALGORITHM ModIncAnalyze($P = \{M_i\}, Q_\alpha, \mathcal{G}, \{\mathscr{L}_i\}, \Delta$)
 1: add-entries($\{k = \langle A, \lambda^c \rangle \mid k \in \mathcal{G}, \Delta_{\mathsf{mod}(A)} \neq \emptyset\}$)        ▷ AnalyzeOutdated
 2: add-entries($\{k \in Q_\alpha \mid k \notin \mathcal{G}\}$), upd($\mathcal{G}, \{k \mapsto \bot \mid k \in Q_\alpha\}$)        ▷ AnalyzeNew
 3: while entries($\mathcal{G}, Q_\alpha$) $\neq \emptyset$ do
 4:     $(M, E)$ := next-entries($\mathcal{G}, Q_\alpha$)
 5:     $I := \{\langle A, \lambda^c \rangle \mapsto \lambda^s \in \mathscr{L}_M \mid$ mod$(A) \in$ imports$(M)\}$        ▷ Imported
 6:     $I_p := \{k \mid k \mapsto \lambda^s \in I, n \mapsto \lambda^{s\prime} \in \mathcal{G}, \lambda^s \not\sqsubseteq \lambda^{s\prime}\}$        ▷ ImpreciseImported
 7:     $I_c := \{k' \mid k' \rightsquigarrow k \in \mathscr{L}_M,\ n \mapsto \lambda^s \in I, n \mapsto \lambda^{s\prime} \in \mathcal{G}, \lambda^{s\prime} \sqsubset \lambda^s\})$
 8:                                                                        ▷ IncorrectImported
 9:     del($\mathscr{L}_M, \{k \mid k_c \in I_p, k \rightsquigarrow k_c \in \mathscr{L}_M$ or $(n_a \rightsquigarrow k_c \in \mathscr{L}_M \wedge k_a \rightsquigarrow k \in \mathscr{L}_M)\}$)
10:                                                                        ▷ DelImprecise
11:     upd($\mathscr{L}_M, \{\langle A, \lambda^c \rangle \mapsto \lambda^s \in \mathcal{G} \mid$ mod$(A) \in$ imports$(M)\}$)        ▷ PreloadImported
12:     $\mathscr{L}_M :=$ IncAnalyze($M, E \cup I_c, \Delta_M, \mathscr{L}$), $\Delta_M \leftarrow \emptyset$
13:     del($\mathscr{L}_M, \{\langle A, \lambda^c \rangle \mid \nexists k', k' \rightarrow \langle A, \lambda^c \rangle \in \mathscr{L}_M$, mod$(A) \neq M\}$)        ▷ RemoveUnused
14:     for $\langle A, \lambda^c \rangle \mapsto {\lambda^s}_l \in \mathscr{L}_M.\langle A, \lambda^c \rangle \mapsto {\lambda^s}_g \in \mathcal{G} \Rightarrow {\lambda^s}_l \neq {\lambda^s}_g$, mod$(A) \neq M$ do
15:         $\lambda^s :=$ Ageneralize(${\lambda^s}_l, \{{\lambda^s}_g\}$)
16:         upd($\mathcal{G}, \langle A, \lambda^c \rangle \mapsto \lambda^s$)        ▷ StoreAnswers
17:         add-entries($\{k \mid k \rightarrow \langle A, \lambda^c \rangle \in \mathcal{G}\}$)
18:     del($\mathcal{G}, \{\langle A, \lambda^c \rangle \rightarrow k' \in \mathcal{G}\}$)        ▷ UpdateDependencies
19:     $R = \{\langle A, \lambda^c \rangle \rightarrow k' \mid \exists \langle P, \lambda^c \rangle \rightsquigarrow k \in \mathscr{L}_M, k' = \langle A, \lambda^c \rangle$, mod$(A) \neq M\}$
20:     for $k \rightarrow \langle Q, {\lambda^c}_t \rangle \in R$ do
21:         $Calls := \{\lambda \mid \langle A, \lambda \rangle \in \mathcal{G}\}$
22:         $\lambda^c :=$ Ageneralize(${\lambda^s}_t, Calls$)
23:         if $\langle Q, \lambda^c \rangle \notin \mathcal{G}$ then        ▷ ScheduleNewCalls
24:             add-entries($\langle Q, \lambda^c \rangle\}$)
25:         upd($\mathcal{G}, \{k \rightarrow \langle Q, \lambda^c \rangle\}$)
26: return $\mathcal{G}, \{\mathscr{L}_i\}$
```

Fig. 9. Incremental and modular fixpoint algorithm.

additions and deletions (Δ_{M_i}), which collect the differences w.r.t. the previous state for each module. The pseudocode of the algorithm is detailed in Figure 9. The steps required to perform local analysis incrementally are presented in **blue**, that is, those steps that were added or modified in the modular, non-incremental algorithm depicted in Figure 7. Before starting the analysis process, the entries of edited modules and new queries are marked to be (re)analyzed. Each of the scheduled modules will be analyzed independently, and possibly several times. Modular analysis is, again, controlled by a queue to which entries with possibly incomplete answer descriptions are added (with the procedure add-entries). At each iteration of the loop, a module is reanalyzed independently for its set of annotated entries (E) extracted from the queue. This is done by procedure next-entries which extracts from the queue entries that are reachable from the initial Q_α in $\mathcal{G}$. Incrementally analyzing a module consists of updating the information about the calls to imported predicates in $\mathscr{L}_M$, by removing possibly inaccurate results and adding the newly computed ones, and calling IncAnalyze. Finally, $\mathcal{G}$ is updated, which includes updating the newly computed answers, updating the dependencies of the predicates in the

```
DelImprecise
        $Calls = \{k \mid k \mapsto \lambda^s \in \mathscr{L}_M, k \in I_p\}$
        $\{M'_i\}, I_{M'_i} = \textsf{split-sources-scc}(M, I_p)$
        $\{\mathcal{G}', \{\mathscr{L}'_i\}\} = \textsf{split-in-scc}(\mathscr{L}_M)$
        $\{\mathcal{G}''\{\mathscr{L}'_i\}\} := \textsc{ModIncAnalyze}(\{M'_i\}, I_{M'_i}, \{\mathcal{G}', \{\mathscr{L}'_i\}\}, \emptyset)$
        $\mathscr{L}_M := \textsf{flatten}(\{\mathcal{G}''\{\mathscr{L}'_i\}\})$
        $\Delta_M \leftarrow \emptyset$
```

Fig. 10. Enhanced modular deletion strategy.

boundary of the modules, and adding to the queue to reanalyze the dependent predicates and call patterns. The operations performing local incremental analysis are

AnalyzeOutdated: Adds to the analysis queue the entries of modules that changed, that is, those whose diff (Δ) is not empty.

AnalyzeNew: Adds to the analysis queue the entries that have not been analyzed yet.

Imported: Collects the current approximations made about the predicates imported by the module to be analyzed.

IncorrectImported: Collects in I_c the nodes of the $\mathscr{L}_M$ that are incorrect (below the fixpoint), that is, the ones whose approximation in the $\mathscr{L}_M$ was smaller than in the $\mathcal{G}$ to reanalyze them later.

ImpreciseImported: Collects in I_p all the imported nodes that are potentially imprecise, that is, those in which in the abstraction $\mathscr{L}_M$ that are *bigger* than the current stored in $\mathcal{G}$.

DelImprecise: Deletes from $\mathscr{L}_M$ the nodes that relied on assumptions that depend on I_p, because they are potentially imprecise.

Analyze: The IncAnalyze function is called with entries for: the calls scheduled by the modular analyzer (E), the nodes that depended on imported information that may be below the fixpoint (I_c). Note that no entries will be added for the nodes that were imprecise as its information will be removed up to the entries that they were triggered by, which guarantees that the analysis will be correct and precise for those entries.

RemoveUnused Removes the call patterns of the imported predicates that were not reached, that is, if there are no edges in the call graph to them.

For the remaining operations, we refer the reader to the description of Figure 7.

Enhancing the deletion strategy. The proposed deletion strategy is quite pessimistic. Updating imprecise information about imported predicates most of the times means reusing only a few answers that did not depend on the changes per module. However, it may occur that the analysis does not change after these changes occur, or that some nodes/edges are still correct and precise. A solution is to partially reanalyze the program without removing these potentially useful results. Our proposed algorithm allows performing such a partial reanalysis, by partitioning the desired module into smaller partitions, for example, using information on *strongly connected components*. This can be achieved within the algorithm by replacing line **DelImprecise** with Figure 10. This runs the algorithm with a partition of the current module as input program, which is split using the (static) SCCs

M: (unchanged) main module

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

:- use_module(bitops).
main(Msg, P) :-
    par(Msg, 0, P).

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

B_0: initial state of `bitops`

```
:- module(bitops, [xor/3]).
xor(0,0,0).
```

B_1: clauses added to `bitops`

```
:- module(bitops, [xor/3]).
xor(0,0,0). % also in B0
xor(0,1,1).
xor(1,0,1).
xor(1,1,0).
```

B_2: a clause is deleted from `bitops`

```
:- module(bitops, [xor/3]).
xor(0,0,0).
xor(0,1,1).
xor(1,0,1).
% xor(1,1,0). %%% line commented
```

Fig. 11. Different program states.

of the clauses (split-sources-scc). This includes also partitioning the results (split-in-scc) to initialize $\mathcal{G}$ using $\mathscr{L}_M$, and setting as Q_α the initial E of this modular analysis. The reanalysis of this partitioned module will be given in a modular form, so it has to be *flattened* back for it to be compatible with the rest of the analysis results. This process consists in merging all the graphs in which the analysis was performed into one graph that contains all the nodes and edges except the edges in $\mathcal{G}''$.

5.2 Running examples of the algorithm

To show the algorithm in action, we now analyze incrementally different versions of the program that computes the parity (some of which are incomplete). The different states of the sources are shown in Figure 11. Initially, we have the analysis result of $P_0 = \{M, B_0\}$, $\mathscr{A}_0$ in Figure 12. This was the result of running the algorithm from scratch $\mathscr{A}_0 = \textsc{ModIncAnalyze}(P_0, Q_\alpha, \emptyset, (\emptyset, \emptyset))$, with initial query $Q_\alpha = \{\langle \mathtt{main}(M, P), (M/\top, P/\top)\rangle\}$. In this version, it was inferred that if `main(M, P)` succeeded, then `P` is 0 ($\gamma(z)$).

Example 4 (*Adding clauses*)
If some clauses are added to `bitops` resulting in B_1, the program to be (re)analyzed becomes $P_1 = \{M, B_1\}$. Incremental analysis by running $\textsc{ModIncAnalyze}(P, Q_\alpha, \mathscr{A}_0, (\{\mathtt{xor}_2, \mathtt{xor}_3, \mathtt{xor}_4\}, \emptyset))$ proceeds as follows. The entries of `bitops` are added to the queue and it is analyzed with $E = \{\langle \mathtt{xor}(C, P_0, P_1), P_0/z\rangle\}$ and the analysis result changes to $(C/b, P_0/z, P_1/b)$ (shown in $\mathscr{A}'_0$). This change needs to be propagated to module `main`, which is analyzed next in the queue. Following the steps of the algorithm:

AnalyzeOutdated The entries to the module `main` are added.
AnalyzeNew No entries are added because there are no new queries.
Imported $I = \{\langle \mathtt{xor}(C, P_0, P_1), P_0/z\rangle\}$
IncorrectImported $I_c = \{\langle \mathtt{xor}(C, P_0, P_1), P_0/z\rangle\}$
ImpreciseImported $I_p = \emptyset$ since the only imported node was below the fixed point.
Analyze The analyzer is called with $E = \{\langle \mathtt{main}(M, P), (M/\top, P/\top)\rangle\}$ and I_c as described.
RemoveUnused All imported call patterns in $\mathscr{L}_{\mathtt{main}}$ are reached (there is an edge to them) and nothing is removed.

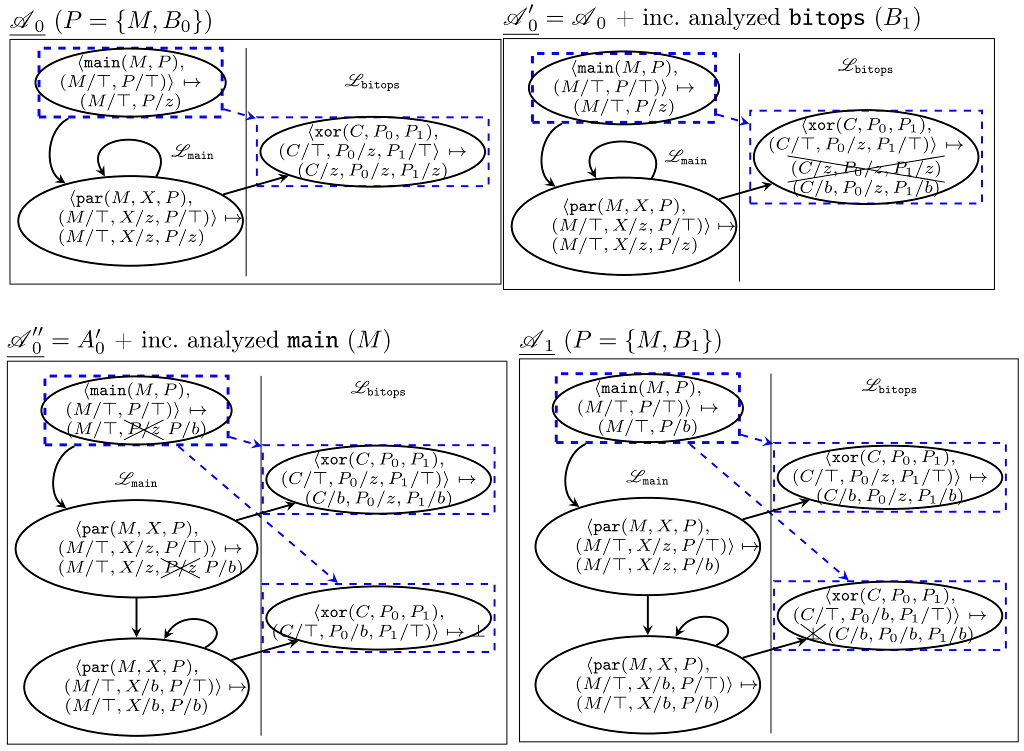

$\mathscr{A}_0$ $(P = \{M, B_0\})$
$\mathscr{A}'_0 = \mathscr{A}_0$ + inc. analyzed bitops (B_1)
$\mathscr{A}''_0 = A'_0$ + inc. analyzed main (M)
$\mathscr{A}_1$ $(P = \{M, B_1\})$
$\mathscr{L}_{\mathtt{main}}$
$\mathscr{L}_{\mathtt{bitops}}$
$\langle \mathtt{main}(M,P), (M/\top, P/\top)\rangle \mapsto (M/\top, P/z)$
$\langle \mathtt{par}(M,X,P), (M/\top, X/z, P/\top)\rangle \mapsto (M/\top, X/z, P/z)$
$\langle \mathtt{xor}(C,P_0,P_1), (C/\top, P_0/z, P_1/\top)\rangle \mapsto (C/z, P_0/z, P_1/z)$
$\langle \mathtt{xor}(C,P_0,P_1), (C/\top, P_0/z, P_1/\top)\rangle \mapsto$ ~~$(C/z, P_0/z, P_1/z)$~~ $(C/b, P_0/z, P_1/b)$
$\langle \mathtt{main}(M,P), (M/\top, P/\top)\rangle \mapsto (M/\top,$ ~~P/z~~ $P/b)$
$\langle \mathtt{par}(M,X,P), (M/\top, X/z, P/\top)\rangle \mapsto (M/\top, X/z,$ ~~P/z~~ $P/b)$
$\langle \mathtt{par}(M,X,P), (M/\top, X/b, P/\top)\rangle \mapsto (M/\top, X/b, P/b)$
$\langle \mathtt{xor}(C,P_0,P_1), (C/\top, P_0/z, P_1/\top)\rangle \mapsto (C/b, P_0/z, P_1/b)$
$\langle \mathtt{xor}(C,P_0,P_1), (C/\top, P_0/b, P_1/\top)\rangle \mapsto \bot$
$\langle \mathtt{main}(M,P), (M/\top, P/\top)\rangle \mapsto (M/\top, P/b)$
$\langle \mathtt{xor}(C,P_0,P_1), (C/\top, P_0/b, P_1/\top)\rangle \mapsto$ ~~$\bot$~~ $(C/b, P_0/b, P_1/b)$

Fig. 12. Analysis results in several reanalysis steps.

StoreAnswers $\langle \mathtt{main}(M,P), (M/\top, P/\top)\rangle \mapsto (M/\top, P/z)\}$ is updated in $\mathcal{G}$, no (parent) entries need to be added to the queue because it is the initial query.

UpdateDependencies All the edges of $\mathcal{G}$ from nodes of `main` to `bitops` are removed.
$R = \{\langle \mathtt{main}(M,P), (M/\top, P/\top)\rangle \to \langle \mathtt{xor}(C,P_0,P_1), (C/\top, P_0/z, P_1/\top)\rangle,$
$\langle \mathtt{main}(M,P), (M/\top, P/\top)\rangle \to \langle \mathtt{xor}(C,P_0,P_1), (C/\top, P_0/b, P_1/\top)\rangle\}$

ScheduleNewCalls A newly encountered call description is added in add-entries, $\langle \mathtt{xor}(C,P_0,P_1), (C/\top, P_0/b, P_1/\top\rangle$, and all the edges in R are added to $\mathcal{G}$.

Next, module `bitops` needs to be analyzed again, only for the pending call description $\langle \mathtt{xor}(C,P_0,P_1), (C/\top, P_0/b, P_1/\top)\rangle$, the new answer $(C/b, P_0/b, P_1/b)$ will be updated in $\mathcal{G}$, adding again an entry for predicate `main`. For the next iteration of the analysis loop, the answer will be updated but it will not imply any changes in the analysis result of the module, therefore the algorithm reached a fixed point ($\mathscr{A}_1$ in Figure 12).

Example 5 (*Deleting clauses*)
The `bitops` module is edited from B_1 to B_2, and the program to be analyzed is $P_2 = \{M, B_2\}$. Incremental analysis by ModIncAnalyze$(P_2, Q_\alpha, \mathscr{A}_1, (\emptyset, \{\mathtt{xor}_4\}))$ proceeds as follows. Module `bitops` was changed, so it is analyzed with $E = \{\langle \mathtt{xor}(C,P_0,P_1), (C/\top, P_0/z, P_1/\top)\rangle, \langle \mathtt{xor}(C,P_0,P_1), (C/\top, P_0/b, P_1/\top)\rangle\}$. The answers are recomputed from scratch; however, the overall result of the module does not change, so nothing needs to be done in $\mathcal{G}$, and it is not necessary to recompute the analysis graph of module `main`, and $\mathscr{A}_2 = \mathscr{A}_1$.

6 Fundamental results of the algorithm

In this section, we provide the correctness and precision guarantees of the proposed algorithm. We first provide some notation that will be instrumental in the proofs of the theorems. We build the domain of analysis results (parametric on D_α) as sets of $(\texttt{pred\_name}, D_\alpha, D_\alpha)$. The set of predicate names may be infinite in general but in each program it is finite. We do not represent the dependencies (edges in the graph), as they are needed only for efficiency. We define the partial order in this domain AG that compares the answer patterns to the call patterns abstracted by the analysis graph. That is:

$$g_1, g_2 \in AG, g_1 \sqsubseteq_{AG} g_2 \text{ if } \forall k \mapsto {\lambda^s}_1 \in g_1 . \exists k \mapsto {\lambda^s}_2 \in g_2 \ \wedge {\lambda^s}_1 \sqsubseteq {\lambda^s}_2 .$$

For simplicity, D_α in the following represents this domain of analysis results, and all domain operators refer to this domain.

The incremental analysis of a module within the algorithm (in the body of the while loop in the pseudocode lines 5 to 12) is denoted with the function:

$$\mathscr{L}_{M'} = \textsc{LocIncAnalyze}(M', E, \mathcal{G}, \mathscr{L}_M, \Delta_M),$$

where M' is a module, $\mathscr{L}_M$ is the analysis result of M for E, Δ_M are the differences between M' and M, to which $\mathscr{L}_M$ corresponds, to get $\mathscr{L}_{M'}$, the analysis result of M', and $\mathcal{G}$ contains the (possibly temporary) information for the predicates imported by M'.

Lastly, we represent performing an iteration of the while loop (lines 5 to 25) as the high-level operation of updating the newly computed information in $\mathcal{G}$:

$$MA(M', E, \mathcal{G}, \mathscr{L}_M, \Delta_M) = \mathsf{upd}(\mathcal{G}, \textsc{LocIncAnalyze}(M', E, \mathcal{G}, \mathscr{L}_M, \Delta_M)).$$

Note that, after a number of (chaotic) iterations, MA is monotonic, and ultimately stationary due to the use of the widening operator.

6.1 Correctness

The following lemma shows that if a module M is analyzed for entries E assuming some $\mathcal{G}$ obtaining $\mathscr{L}_M$, if the assumptions change to $\mathcal{G}'$, incrementally updating these assumptions produces an analysis graph $\mathscr{L}'_M$ that is correct assuming $\mathcal{G}'$.

Lemma 3 (*Correctness updating $\mathscr{L}$ modulo $\mathcal{G}$*)
Let M be a module of program P and E a set of entries. Let $\mathcal{G}$ be a previous state of the global analysis graph, if $\mathscr{L}_M$ is correct for M and $\gamma(E)$ assuming $\mathcal{G}$. If $\mathcal{G}$ changes to $\mathcal{G}'$, the analysis result

$$\mathscr{L}'_M = \textsc{LocIncAnalyze}(M, E, \mathcal{G}', \mathscr{L}_M, \emptyset)$$

is *correct* (see Definition 3) for M and $\gamma(E)$ assuming $\mathcal{G}$.

Proof
To prove this, we need to show that all the answers that differ from $\mathcal{G}$ to $\mathcal{G}'$ for the calls to predicates imported by M are included in E. Since these are the requisites in Theorem 4 to guarantee that the result is correct. The **ImpreciseImported** are collected and removed, therefore it is guaranteed that all the entries in E that depended on these will

be correct (the analysis is empty). When collecting the **IncorrectImported,** only those nodes are added to the entries. However, because $\mathscr{L}_M$ was assumed to be correct, it is guaranteed that adding these entries is enough, because $\mathscr{L}_M$ correctly over-approximates the parts of $[\![P]\!]_Q$ that were already in $\mathscr{L}_M$ and the ones that are missing are guaranteed to be correct because they are missing. □

The following Theorem 9 captures the correctness of the algorithm when starting from an empty analysis result, that is, starting with an empty $\mathcal{G}$ and $\mathscr{L}_{M_i}$. Note that this is not the same as running the traditional modular analysis, as information is reused when iterating between modules, whereas in ModAnalyze every iteration the $\mathscr{L}$'s are clean.

Theorem 9 (*Correctness of* ModIncAnalyze *from scratch*)
Let P be a modular program and Q_α a set of abstract queries. Then, if:

$$\{\mathcal{G}, \{\mathscr{L}_{M_i}\}\} = \textsc{ModIncAnalyze}(P, Q_\alpha, \emptyset, \emptyset)$$

$\mathcal{G}$ is *correct* (see Definition 5) for P and $\gamma(Q_\alpha)$.

Proof
Correctness follows using the same argument as in Theorem 7, with the difference that instead of applying Lemma 1 to IncAnalyze, we apply Lemma 3 to LocIncAnalyze. □

Theorem 10 (*Correctness of* ModIncAnalyze)
Let P and P' be modular programs that differ by Δ, Q_α a set of queries, and $\mathscr{A} = \textsc{ModIncAnalyze}(P, Q_\alpha, \emptyset, (\emptyset, \emptyset))$, then if:

$$\{\mathcal{G}', \{\mathscr{L}'_{M_i}\}\} = \textsc{ModIncAnalyze}(P', Q_\alpha, \mathscr{A}, \Delta)$$

$\mathcal{G}'$ is *correct* (see Definition 3) for P and $\gamma(Q_\alpha)$.

Proof
By induction on the number of modular partitions, it is true for any partition of program P in n modules with no recursive dependencies on predicates between modules. This condition ensures that if removing for some clause in $\mathscr{L}_M$ is needed, *all* the dependent information for recomputing is indeed removed (nothing imprecise is reused from some other $\mathscr{L}_{M'}$).

- If program P has one module, it is the same case as having one module in the modular analysis. So it follows from Theorem 7.
- As in the proof of Theorem 7, if program P is partitioned into n modules, we need to prove that if we finish analyzing $n-1$ modules, then we finish analyzing all n modules. Assuming that the analysis of the first $n-1$ modules finishes and it is correct, this $n-1$ result could be seen as one module, reducing this general case to the case of two modules.
- If program $P = \{M_a, M_b\}$ is partitioned into two modules, let us assume that M_a imports M_b. Let us assume that we reanalyze M_b first. We study the reanalysis cases of $\mathcal{G}' = MA(M_b, E, \mathcal{G}, \mathscr{L}_{M_b}, \Delta_{M_b})$:

1. If $\mathcal{G}' = \mathcal{G}$, the procedure is equivalent if program P has one module.
2. If $\mathcal{G} \sqsubset \mathcal{G}'$, then analysis results need to be propagated to A. Once the results of A are updated, the analysis iterations of A and B will be equivalent as when analyzing from scratch, only new call patterns may appear.
3. If $\mathcal{G}' \sqsubset \mathcal{G}$, these analysis results need to be propagated to the analysis of A, which will be reanalyzed. Once A and B have updated their incompatible information, the further (re)analyses can only become smaller, but since MA is monotonic and D_α is finite, a fixpoint is reached, which is correct for P, since the computation of each of the modules is correct.
4. Else, the information is incompatible. This can only happen if there were additions and deletions. This information needs to be propagated to M_a and the reanalysis of M_a will only lead to cases 1, 2, or 3. □

Note that the correctness of the proposed enhanced deletion strategy follows from Theorem 10.

6.2 Precision

As in the previous sections, we now show the precision properties of the algorithm when analyzing with finite abstract domains, if we remove the generalization via widening as stated in Sections 4.1 and 4.2.2. First note that for the MA function, since the *lfp* is monotonic w.r.t. the initial assumptions and `upd` is monotonic, if generalization is disabled then $\mathcal{G}$ will be the *least program analysis graph*, as the *lfp* of each of the individual modules was computed.

The following lemma shows that if a module M is analyzed for E assuming some $\mathcal{G}$ obtaining $\mathscr{L}_M$, then if the assumptions change to $\mathcal{G}'$, incrementally updating these assumptions will produce an analysis graph $\mathscr{L}'_M$ that is the same as analyzing M with assumptions $\mathcal{G}$ from scratch. That is, the least analysis graph for module M.

Lemma 4 (*Precision updating $\mathscr{L}$ modulo $\mathcal{G}$*)
Let M be a module contained in program P and E a set of entries. Let $\mathcal{G}$ be a previous state of the global analysis graph, if $\mathscr{L}_M = \textsc{LocIncAnalyze}(M, E, \mathcal{G}, \emptyset, \emptyset)$. If $\mathcal{G}$ changes to $\mathcal{G}'$, the analysis result:

$$\textsc{LocIncAnalyze}(M, E, \mathcal{G}', \mathscr{L}_M, \emptyset) = \textsc{LocIncAnalyze}(M, E, \mathcal{G}', \emptyset, \emptyset)$$

is the same as analyzing from scratch, that is, the *lfp* of M and E.

Proof
The proof of this lemma follows from the proof of Lemma 3 and the guarantee that $\mathscr{L}_M$ is the least analysis graph if the generalization is removed from IncAnalyze (Theorem 6). □

Theorem 11 (*Precision of* ModIncAnalyze *from scratch*)
Let P be a modular program and Q_α a set of abstract queries. The analysis result

$$\mathscr{A} = \textsc{ModIncAnalyze}(P, Q_\alpha, \emptyset, \emptyset) = \textsc{ModAnalyze}(P, Q_\alpha)$$

such that $\mathscr{A} = \{\mathcal{G}, \{\mathscr{L}_{M_i}\}\}$, then $\mathcal{G} = \mathcal{G}'$.

Proof
Since the *lfp* is monotonic w.r.t. the initial assumptions and `upd` is monotonic, *MA* is monotonic. Therefore, chaotic iteration of MA with the different modules of a program will reach a fixpoint which is the *least fixed point* because the separated *lfp* of each of the modules is computed. Chaotic iteration is guaranteed in the same way as correctness in Theorem 9. Termination is guaranteed because MA is monotonic and D_α is finite. □

If P is changed to P' by editions Δ and it is reanalyzed incrementally, the algorithm will return a $\mathcal{G}$ that encodes the same *global analysis result* as if P' is analyzed from scratch. That is, the least program analysis graph.

Theorem 12 (*Precision of* ModIncAnalyze)
Let P and P' be modular programs that differ by Δ, Q_α a set of queries, and $\mathscr{A} = \textsc{ModIncAnalyze}(P, Q_\alpha, \emptyset, (\emptyset, \emptyset))$, then

$$\textsc{ModIncAnalyze}(P', Q_\alpha, \emptyset, (\emptyset, \emptyset)) = \textsc{ModIncAnalyze}(P', Q_\alpha, \mathscr{A}, \Delta).$$

Proof
This is proved by following the same strategy as in Theorem 10, replacing the termination condition that relied on the widening operator with the guarantees that the abstract domain is finite and that *MA* is monotonic, and the guarantee of Lemma 4 that no imprecision is introduced analyzing each individual module. □

7 Experiments

We have implemented the proposed algorithm within the `Ciao`/`CiaoPP` system (Hermenegildo *et al.* 2012, 2005), which can be found in `https://github.com/ciao-lang/ciaopp`. We use a selection of well-known benchmarks from previous studies of incremental analysis, for example, `ann` (a parallelizer) and `boyer` (a theorem prover kernel) are programs with a relatively large number of clauses located in a small number of modules. In contrast, for example, `bid` is a more modularized program (see Table 1 for more details, and `https://github.com/ciao-lang/ciaopp_tests/tree/master/tests/incanal` for the source code of the benchmarks). We used the original modular structure as modular partition and evaluated five strategies:

- `mon`: the baseline non-modular, non-incremental algorithm (Muthukumar and Hermenegildo 1992), that is, IncAnalyze described in Section 4.1 with initial results always empty.
- `mon-inc`: the monolithic incremental algorithm (Hermenegildo *et al.* 2000) as described in Section 4.1.
- `mon-scc`: the monolithic incremental algorithm (Hermenegildo *et al.* 2000) as described in Section 4.1 with the bottom-up deletion strategy of Hermenegildo *et al.* (2000).
- `mod`: as a coarse-grained modular algorithm, which consists on our proposed algorithm, without keeping each of the local analysis graphs. Note that this is not the same as the algorithm of Section 4.2, as it did not consider modifying the modules.
- `mod-inc`: our proposed algorithm that is modular and incremental (Section 5).

Table 1. *Benchmark characteristics sorted by lines of code*

Bench	# Modules	# Predicates	# Clauses	LOC
`hanoi`	2	4	6	46
`aiakl`	4	8	15	71
`qsort`	3	8	17	49
`progeom`	2	10	18	73
`bid`	7	21	48	207
`rdtok`	5	15	57	293
`cleandirs`	3	36	81	528
`read`	3	25	94	352
`warplan`	3	37	114	281
`boyer`	4	29	145	279
`peephole`	3	33	169	377
`witt`	4	69	176	618
`ann`	3	69	229	641
`manag_proj`	8	105	143	805
`check_links`	4	220	504	2042

- `mod-scc`: for the experiments that include deleting clauses, the alternative strategy for updating the analysis following the strongly connected components of the program (Section 5.1).

We performed experiments with four different abstract domains: a simple reachability domain (`pdb`), a groundness domain (`gr`), a dependency tracking via propositional clauses domain (Dumortier *et al.* 1993) (`def`), and the sharing and freeness abstract domain (Muthukumar and Hermenegildo 1991) (pointer sharing and uninitialized pointers, `shfr`). We use the exported predicates from the main module (with $\top$ call pattern) as the set of initial queries (i.e., no additional information is provided in the program).

We ran all experiments on a Linux machine (kernel 4.9.0-8-amd64) with Debian 9.0, a Xeon Gold 6154 CPU, and 16 GB of RAM. However, running the test in a standard laptop shows similar performance.

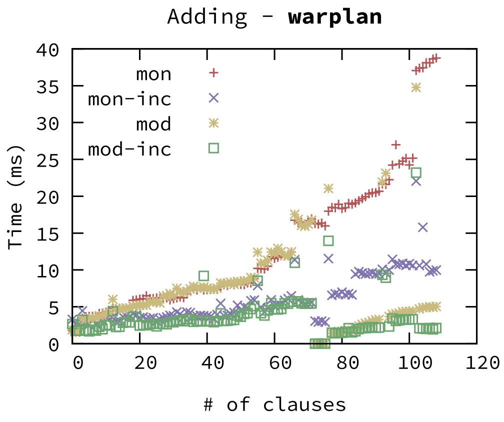
Analyzing from scratch. We first study the analysis from scratch of all the benchmarks for all approaches, to observe the overhead introduced by the bookkeeping of the algorithms. The analysis times in milliseconds are shown in Table 2. For each benchmark, four rows are shown, corresponding to the four analysis algorithms mentioned earlier: monolithic (`mon`), monolithic incremental (`mon-inc`), modular (`mod`), and, lastly, modular incremental (`mod-inc`), that is, the proposed approach. In the monolithic setting, the overhead introduced is negligible. Interestingly, the incremental modular analysis performs better overall than simply modular even in analysis from scratch. This is due to the reuse of local information specially in complex benchmarks such as `ann`, `peephole`, `warplan`, or `witt`. In the best cases (e.g., `witt`, `cleandirs`, or `check_links` analyzed with `shfr`), the performance of incremental modular competes with monolithic thanks to the incremental updates, dropping from 20 to 3 s, from 1.2 to 0.8 s, and from 2.5 to 1.2 s, respectively.

Table 2. *Analysis times from scratch (ms)*

Benchmark		pdb	gr	def	shfr
hanoi	mon	5.2	2.9	2.2	10.7
	mon-inc	5.3	3.0	2.2	10.2
	mod	12.3	7.2	5.8	22.1
	mod-inc	10.0	6.2	4.6	18.3
aiakl		5.9	6.4	4.6	7.9
		7.6	7.3	5.8	8.4
		15.0	18.9	14.0	18.0
		16.0	15.5	13.0	16.4
qsort		7.8	8.3	4.0	9.5
		8.0	8.5	4.3	10.5
		21.7	21.6	13.5	24.9
		19.5	20.3	10.0	20.1
progeom		5.4	5.4	5.1	6.4
		6.1	5.4	5.4	7.1
		24.1	23.6	21.6	28.7
		18.4	18.7	14.8	20.3
bid		18.8	14.8	9.9	22.9
		17.7	15.4	10.2	26.8
		61.0	55.1	39.1	68.3
		42.4	42.1	32.4	55.7
rdtok		33.5	44.0	15.7	63.3
		51.3	29.2	17.8	66.0
		85.1	61.3	40.2	122.0
		52.3	53.5	36.6	90.9
cleandirs		33.2	27.6	26.2	384.1
		31.7	29.1	27.7	389.0
		145.5	123.8	140.3	1189.2
		93.8	77.2	80.5	778.3
read		217.5	116.5	47.8	399.0
		172.3	105.0	35.0	400.7
		192.0	118.4	45.1	422.4
		189.9	126.9	45.5	472.4
warplan		46.0	24.5	20.1	63.3
		41.0	26.6	16.6	64.3
		71.7	52.7	35.8	180.7
		57.0	37.1	24.1	102.9
boyer		38.3	24.1	14.9	50.0
		37.0	31.5	17.4	51.5
		48.3	39.3	21.5	68.2
		44.9	37.3	19.1	65.4
peephole		67.0	43.2	19.2	157.6
		64.1	45.6	21.2	156.4
		155.8	75.6	43.6	392.8
		115.1	62.4	40.2	267.0
witt		183.4	11.6	33.5	2490.4
		186.0	16.4	38.8	2491.2
		1134.6	6.7	120.8	20550.3
		414.8	9.8	71.1	3222.9
ann		84.5	58.4	35.0	120.5
		85.4	64.1	38.5	123.7
		264.1	174.5	89.7	296.6
		145.3	127.0	60.6	241.5
manag_proj		111.0	24.1	51.3	18049.2
		98.3	28.3	48.8	17967.3
		291.3	54.7	150.3	37184.9
		221.8	44.4	104.7	34595.0
check_links		701.7	301.6	167.5	803.3
		678.9	251.5	178.5	819.2
		1292.6	680.8	600.5	2530.3
		776.1	360.8	267.2	1162.5

Note that a smaller program does not necessarily imply that the analyzer will run faster, and it depends on the structure of the code and the kind of data that the program operates with. Also, the cost of performing a modular analysis highly depends on the module scheduling policy, and whether the modular partitions were correctly produced, in this case, if the programmer divided the program in a reasonable manner. For example, analyzing `boyer` (with any domain) modularly comes at no cost, while in the case of `cleandirs` it is three times slower than doing it monolithically.

Clause addition/deletion experiment. As a stress test for the proposed algorithm, we measured the cost of re-analyzing the program incrementally adding (or removing) one clause at a time, until the program is completed (or empty). That is, for the addition experiment, the analysis was first run for the first clause only. Then the next clause was added and the resulting program (re)analyzed. This process was repeated until all the clauses in all the modules were added. For the deletion experiment, starting from



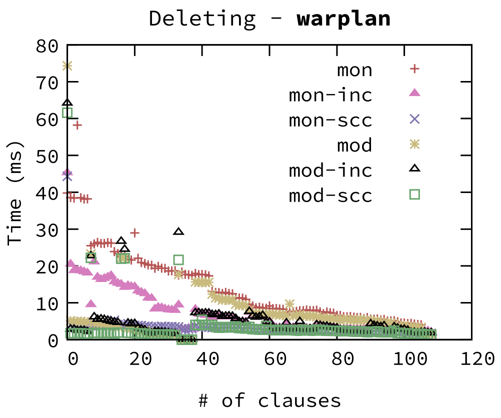


Fig. 13. Analysis time (ms) for **warplan** with **def** for both experiments.

an already analyzed program, the last clause was deleted and the resulting program (re)analyzed. This process was repeated until no clauses were left. The experiment was performed for all the approaches using the initial (top-down) deletion strategy (**mod-inc**) and the SCC partition deletion strategy of Section 5.1 (**mod-scc**).

7.1 Discussion

In our experiments, we observed that the analyses performed with the **gr** and **def** domains were the most relevant to evaluate the usefulness of the algorithm. This is due to the domain operations being fairly simple (when compared with the cost of executing the fixpoint algorithm), so that the complexity of the algorithm is not completely hidden by the complexity of the domain operations (e.g., **shfr**). At the same time, they are complex enough that there is some fixpoint iteration (which does not occur in, e.g., **pdb**, since $\top$ is assumed for every call pattern). Therefore, in this section, we focus mainly in the analysis with the **def** domain but include as well some discussion about **gr**. Nevertheless, for the results for the remaining domains, we refer the reader to the Appendix A pp. (2–11).

Figure 13 shows the addition and deletion experiments for the **warplan** benchmark analyzed with **def**. Each point represents the time taken to reanalyze the program after incrementally adding/deleting one clause. The horizontal axis denotes the number of clauses added/deleted at that point of the experiment. We observe that the proposed incremental algorithm outperforms overall the non-incremental settings when the time needed to reanalyze is large. We find that for smaller benchmarks, our algorithm performs up to eight times faster than the traditional monolithic, non-incremental algorithm, and, in the worst cases performs as fast as the traditional modular algorithm. The detailed analysis times per iteration for the remaining benchmarks are available in Figures A1, A2, and A3 pp. (2–4).

We observe that, even when analyzing takes less time, that is, when the program has fewer clauses, the analysis time of the algorithm proposed is faster overall. Moreover, as the analysis grows in complexity, the cost of our approach grows significantly slower than that of the traditional algorithm. In the case of the deletion experiments, we observe also clear advantages, specially when using the strategy of partitions in SCC presented in Section 5.1.

Table 3. *Analysis times (ms) per action of the clause* addition *experiment with* **def**

	mon			mon-inc			mod			mod-inc		
bench	mean	max	min	mean	max	min	mean	max	min	mean	max	min
aiakl	2.2	5.6	1.0	1.8	5.9	1.3	1.8	14.9	0.5	1.7	14.1	0.5
ann	7.2	76.8	1.1	3.3	51.8	1.6	2.5	156.8	0.3	2.3	133.9	0.6
bid	3.4	18.1	1.5	2.6	11.9	1.9	2.5	32.7	0.3	2.3	26.6	0.5
boyer	21.5	46.4	1.5	3.2	13.0	1.4	6.0	15.1	0.4	1.9	14.8	0.5
check_links	29.3	608.6	2.6	14.5	571.5	5.5	21.6	889.4	0.4	7.1	664.1	0.8
cleandirs	9.1	28.9	1.5	4.5	18.6	1.9	6.2	96.5	0.9	3.7	63.4	0.9
hanoi	2.4	4.7	0.7	1.8	5.3	1.1	2.9	11.5	0.4	2.1	10.0	0.4
manag_proj	19.7	100.8	4.4	8.3	35.2	4.8	6.9	97.7	0.3	4.6	69.5	0.4
peephole	9.6	64.8	1.5	2.7	30.2	1.6	2.1	95.8	0.4	1.9	76.5	0.6
progeom	2.3	5.4	1.1	1.6	3.4	0.7	2.3	10.2	0.8	2.2	9.7	0.8
read	38.9	177.4	1.1	13.4	151.3	1.3	18.5	214.2	0.8	9.9	165.2	0.6
qsort	2.8	11.0	1.1	1.8	4.4	1.1	2.8	10.5	0.4	2.4	9.5	0.5
rdtok	8.2	31.1	1.2	6.2	57.6	1.3	6.0	27.2	0.3	6.0	59.3	0.4
warplan	10.2	35.0	0.8	4.2	18.5	1.5	5.4	30.7	0.7	2.5	21.3	0.9
witt	15.2	177.7	1.5	5.5	142.0	2.2	11.1	653.2	0.3	4.8	323.4	0.4

Analysis time per action. In order to get an overall idea of the cost in terms of the time taken by analysis, we have included Tables 3 and 4 for the addition and deletion experiment, respectively. They show, split by benchmark and analysis configuration, the **mean**, **max**imum, and **min**imum analysis times after each modification made in the experiment, for each program. The objective is to provide intuition for the "response times" of the analyses after each such modification. We center our attention on the costlier *instances* of the benchmarks, that is, the (re)analysis runs which take the longest after a modification is performed in the program for the traditional, *monolithic* analysis. In absolute terms, these are **check_links**, with the largest analysis time (608.6 ms), followed by **witt** (177.7 ms), **read** (177.4 ms), and **manag_proj** (100.8 ms). In terms of overall cost (*mean*) of the reanalysis, we have **read** (38.5 ms), **check_links** (29.3 ms), and **boyer** (21.5 ms). These high differences between the mean and maximum analysis times are due to the very small values for the first additions, in which the program is very small and there are no iterations. The analysis times for the remaining experiments that are available in detailed analysis times for each step are provided in Section A.2 pp. (5–6). These should be in principle the benchmarks that we should focus on incrementalizing, as more time is saved. This applies not only to the monolithic analysis but also to the modular analysis. To observe the increase in performance obtained, Table 5 shows the speedup of our algorithm with respect to: **mon-inc**, in the case of the addition experiments, and **mod-scc**, in the case of the deletion experiment. The analysis times for the remaining speedups are provided in Section A.3 pp. (7–8).

Accumulated analysis time. To observe in a more detailed manner how the analyzers behave, we present in Figures 14 and 15 the accumulated analysis times, that is, the analysis time of all the experiments aggregated by benchmark, *divided by how much time was spent in the different parts of the algorithm.* The results are for the **gr** and **def**

Table 4. *Analysis times (ms) per action of the clause* deletion *experiment with* **def**.

	mon			mon-inc			mon-scc			mod			mod-inc			mod-scc		
bench	mean	max	min	mean	max	min	mean	max	min	mean	max	min	mean	max	min	mean	max	min
aiakl	2.6	7.2	1.3	1.9	6.9	1.3	1.4	6.1	0.9	2.0	15.2	0.5	1.8	13.4	0.4	1.5	13.7	0.3
ann	7.9	85.8	1.2	4.0	80.4	1.8	3.3	80.3	1.5	8.8	197.1	0.3	3.4	153.6	0.6	2.8	153.1	0.6
bid	3.0	17.1	1.6	2.5	19.4	1.6	1.9	18.3	1.4	3.4	59.8	0.5	2.9	53.1	0.4	2.6	50.3	0.4
boyer	21.1	51.5	1.3	11.4	35.3	1.3	2.2	36.2	1.1	6.3	44.9	0.4	5.3	45.3	0.6	1.6	45.1	0.3
check_links	29.0	635.6	1.5	28.3	674.8	6.4	20.9	598.2	5.5	22.6	946.0	0.3	17.0	720.1	0.7	13.3	716.0	0.5
cleandirs	9.2	27.5	1.6	5.6	32.1	1.7	3.3	30.1	1.4	6.2	128.9	0.9	4.4	90.8	0.7	3.5	86.5	0.6
hanoi	2.6	5.3	0.8	1.8	5.6	0.9	1.5	5.9	0.7	3.5	13.8	0.5	2.7	12.4	0.7	2.5	12.6	0.5
manag_proj	20.9	103.6	4.5	9.3	98.3	4.5	7.7	90.1	4.1	7.9	259.3	0.3	5.7	212.1	0.3	5.8	217.7	0.3
peephole	10.3	100.6	1.7	3.4	67.4	1.5	2.3	65.4	1.3	5.6	138.6	1.1	3.7	113.4	1.0	2.7	116.2	0.9
progeom	2.3	5.3	1.1	1.8	5.5	1.0	1.2	6.3	0.9	2.7	22.0	0.7	2.5	19.3	0.7	2.1	18.6	0.6
read	39.0	184.7	1.1	33.8	189.1	1.1	3.9	171.9	1.0	17.9	185.7	0.7	19.4	191.3	0.7	5.8	187.2	0.6
qsort	2.5	7.5	1.1	2.2	8.5	1.2	1.4	9.3	0.8	3.6	22.3	0.4	3.4	20.4	0.6	3.1	20.8	0.5
rdtok	8.2	29.7	1.5	7.3	31.8	1.3	2.2	31.8	1.1	6.6	61.8	0.3	8.2	54.1	0.6	4.8	54.8	0.3
warplan	11.1	52.7	1.0	6.2	40.2	1.3	2.0	39.3	1.0	5.5	63.2	0.8	4.1	56.0	1.0	2.7	53.7	0.7
witt	14.6	174.8	1.2	7.1	179.3	2.1	4.5	185.2	1.9	15.0	1,021.7	0.3	7.9	461.3	0.4	7.1	423.9	0.4

Table 5. *Speedups of the clause* addition *(left) and* deletion *(right) experiments with* `def`

	mod-inc			mod-scc				
	vs.	vs.	vs.	vs.	vs.	vs.	vs.	vs.
bench	mon	mon-inc	mod	mon	mon-inc	mon-scc	mod	mod-inc
aiakl	1.3	1.1	1.0	1.7	1.2	0.9	1.3	1.2
ann	3.1	1.4	1.1	2.9	1.4	1.2	3.2	1.2
bid	1.5	1.1	1.1	1.2	1.0	0.7	1.3	1.1
boyer	11.6	1.7	3.2	13.0	7.1	1.4	3.9	3.3
check_links	4.1	2.1	3.1	2.2	2.1	1.6	1.7	1.3
cleandirs	2.5	1.2	1.7	2.6	1.6	1.0	1.8	1.3
hanoi	1.2	0.9	1.4	1.1	0.7	0.6	1.4	1.1
manag_proj	4.3	1.8	1.5	3.6	1.6	1.3	1.4	1.0
peephole	5.0	1.4	1.1	3.8	1.3	0.9	2.1	1.4
progeom	1.1	0.7	1.1	1.1	0.9	0.6	1.3	1.2
read	3.9	1.4	1.9	6.8	5.9	0.7	3.1	3.4
qsort	1.2	0.7	1.2	0.8	0.7	0.5	1.2	1.1
rdtok	1.4	1.0	1.0	1.7	1.5	0.5	1.4	1.7
warplan	4.1	1.7	2.2	4.2	2.3	0.8	2.1	1.6
witt	3.1	1.1	2.3	2.1	1.0	0.6	2.1	1.1

abstract domains, and each of the bars shows of the full set of addition and deletion experiments.

Figure 14 shows the accumulated analysis time for the addition experiments. As mentioned before, the bars are split to show the time taken in each operation: `analyze` is the time spent in the module analyzer, `incAct` is the time spent updating the local analysis results, `preProc` is the time spent processing clause relations (e.g., calculating the SCCs), `updG` is the time spent updating $\mathcal{G}$, and `procDiff` is the time spent applying the changes to the analysis. This last parameter only appears in the incremental settings. The bars are normalized with respect to the monolithic non-incremental (`mon`) algorithm, which always takes "1" to execute. For example, if analyzing `rdtok` with the `gr` domain for the monolithic non-incremental setting is taken as 1, the modular incremental (`mod-inc`) setting takes approx. 0.6, so it is approx. 1.67 times faster.

As before, the benchmarks are sorted by number of LOC. Because of this, it can be observed that the incremental analysis does tend to be more useful as program size grows. Overall, the incremental settings (`mon-inc`, `mod-inc`) are always faster than the corresponding non-incremental settings (`mon`, `mod`). Furthermore, while the traditional modular analysis is sometimes slower than the monolithic one (for the small benchmarks: `hanoi` and `qsort`), our modular incremental algorithm always outperforms both, obtaining 10× overall speedup over monolithic in the best cases (`boyer` analyzed with `def` or `peephole` analyzed with `shfr`). Furthermore, in the larger benchmarks, modular incremental outperforms even the monolithic incremental approach.

Figure 15 shows the results of the deletion experiment. The analysis performance of the incremental approaches is in general better than the non-incremental approaches, except some cases for small programs. Again, our proposed algorithm shows very good

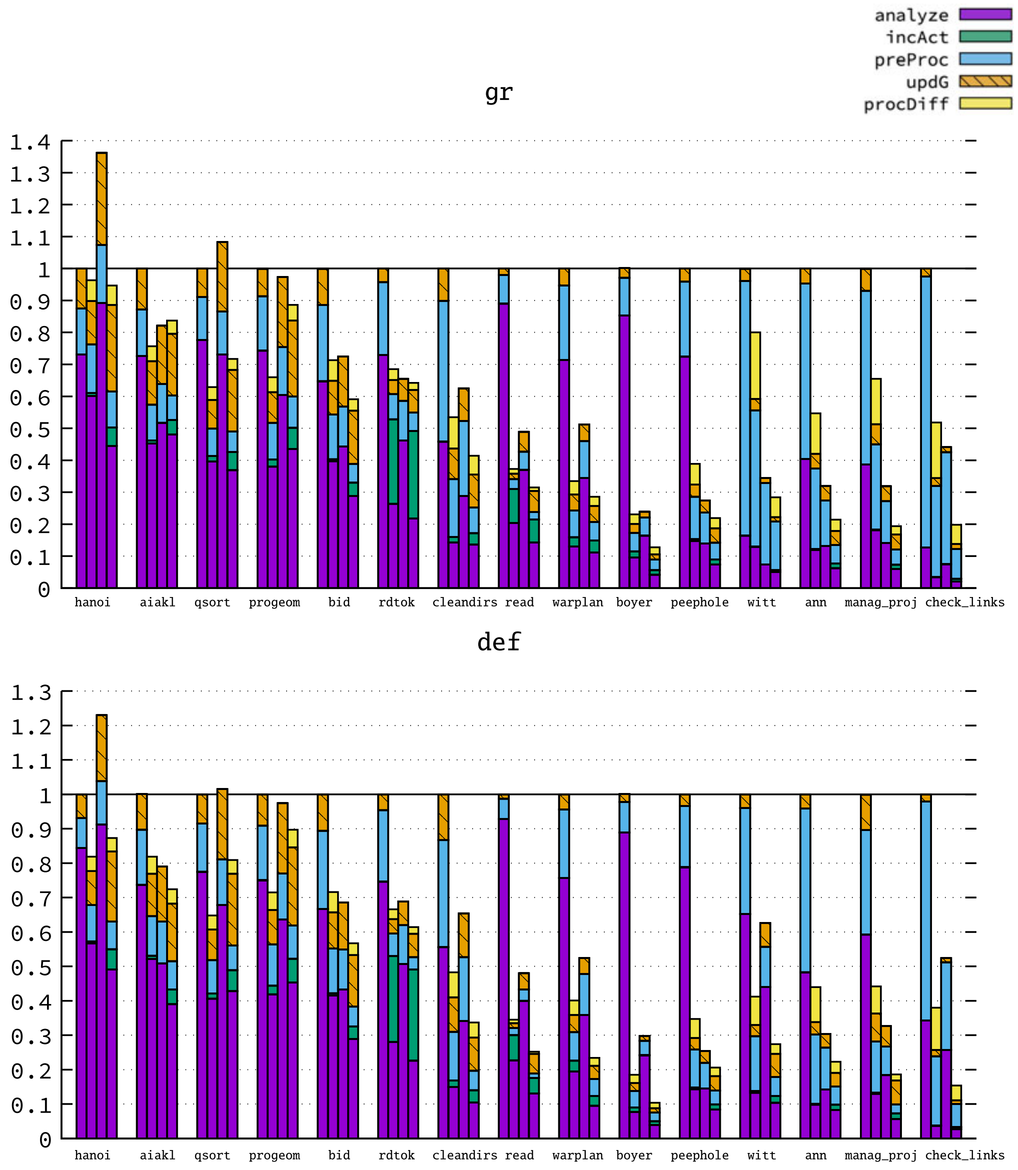


Fig. 14. Accumulated analysis time (normalized w.r.t `mon`) adding clauses. The order inside each set of bars is |`mon`|`mon-inc`|`mod`|`mod-inc`|.

performance, in the best case 10× speedup (`read` analyzed with `shfr`), and overall 5× speedup (`ann`, `peephole`, and `witt`), competing with monolithic incremental `scc` and outperforming in general monolithic incremental `td`. The SCC-guided deletion strategy seems to be more efficient than the top-down deletion strategy. This confirms that the top-down deletion strategy tends to be quite pessimistic when deleting information, and modular partitions limit the scope of deletion. For the accumulated analysis time of the remaining domains, please see Figures A4 and A5 pp. (9–10).

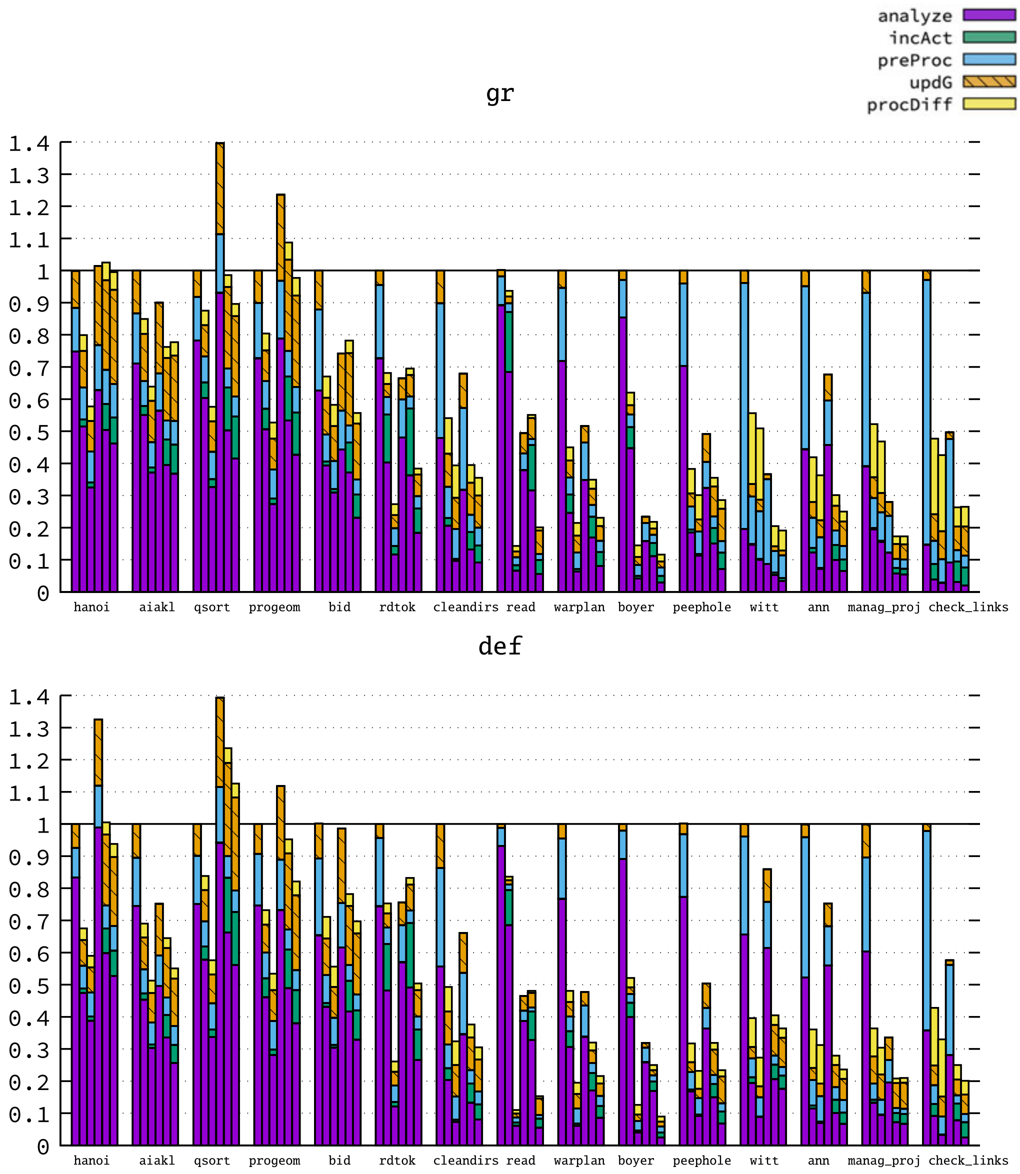


Fig. 15. Accumulated analysis time (normalized w.r.t `mon`) deleting clauses. The order inside each set of bars is |`mon`|`mon-inc`|`mon-scc`|`mod`|`mod-inc`|`mod-scc`|.

Distribution of analysis times. Next, we study how the analysis time of the experiments is distributed. Figures 16 and 17 show histograms that illustrate the number of analyzed instances of the experiments with respect to the analysis time, regardless of the order in which the experiments were performed, and for each configuration. In the vertical axis, we plot the number of tests, that is, *how many different instances of the addition or deletion experiment that were performed could be analyzed in that time or less.* In the horizontal axis, we represent the analysis time. For example, on the left-hand side of Figure 16, for 5 ms in the vertical axis, starting from the bottom of the graph, we first find the red line corresponding to the monolithic analysis (`mon`). This means that approx.

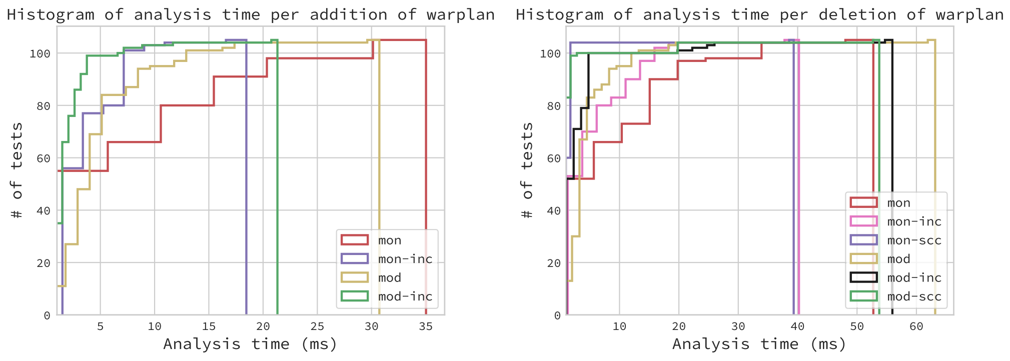


Fig. 16. Distribution over time of instances of the addition (left) and deletion (right) experiments for **warplan** with **def**.

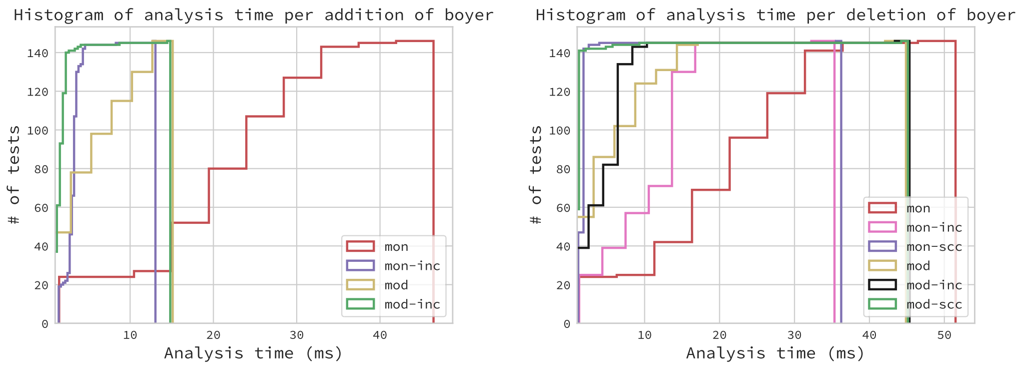


Fig. 17. Distribution over time of instances of the addition (left) and deletion (right) experiments for **boyer** with **def**.

55 of the analyses performed in **warplan** finished in 5 ms or less. Then we find the yellow line (**mod** analysis): for this setting, 70 of the instances of the addition experiment were analyzed in 5 ms or less. The next line that we find is the purple line, corresponding to the **mon-inc** configuration. In this case, 78 instances were analyzed in 5 ms or less. Finally, we have our configuration, **mod-inc**, that was able to analyze 99 instances of the addition experiment in 5 ms. Figures 16 and 17 show that, overall, the analysis time of the proposed algorithm is faster than that of the previous configurations.

Correlations to benchmark and analysis graph characteristics. We also looked for correlations between benchmark characteristics and the speedups observed. While this topic would require a study of its own, we have observed some correlations with benchmark-related analysis characteristics. Figures 18 and 19 show scatter plots of the speedup obtained with respect to two such characteristics: the number of nodes in the analysis graph and the number of calls to $\top$. The plots show that there is some correlation between the size of the analysis graph and the speedup obtained: we observed that the incremental and modular analysis proposed is beneficial for larger analysis graphs. The sizes of the analysis graphs depend themselves on the complexity of the abstract domain (due to the

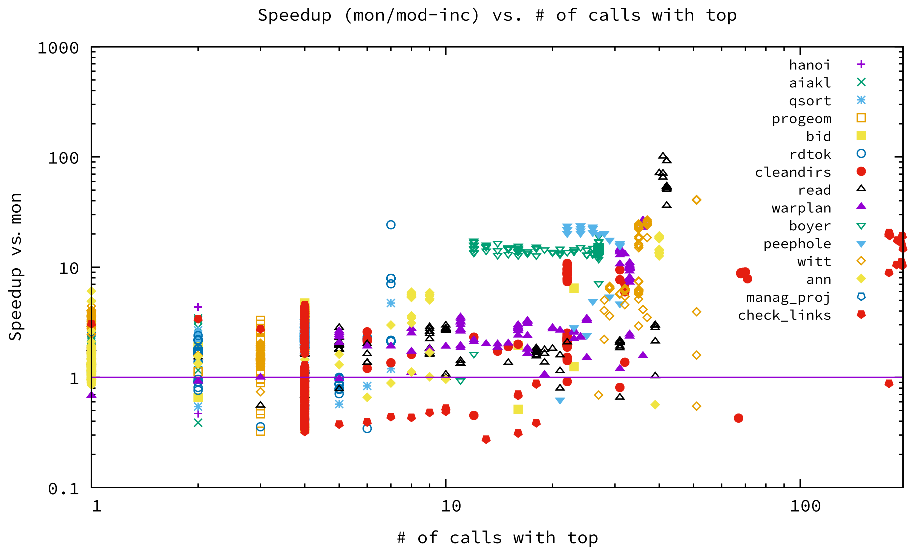


Fig. 18. Speedup versus monolithic depending on the number of nodes in the analysis graph.

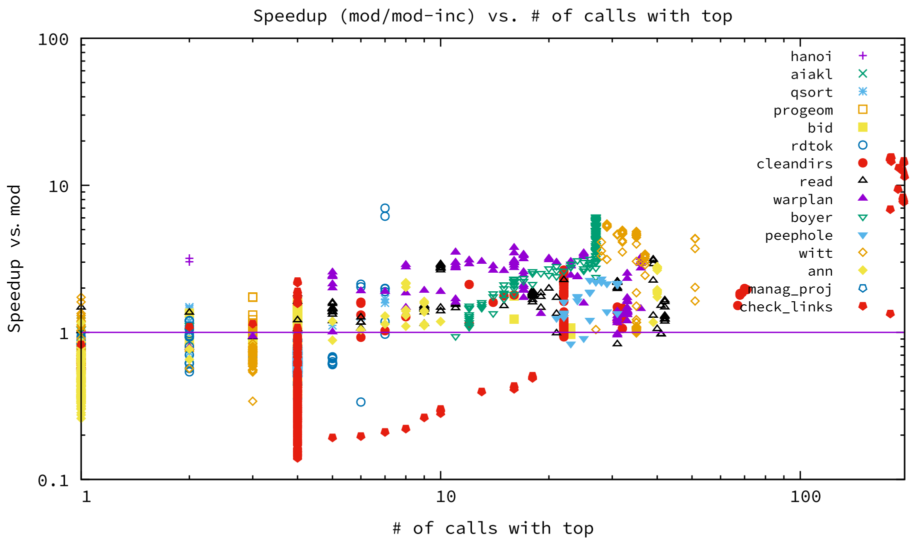


Fig. 19. Speedup versus modular depending on the number of calls to ⊤.

algorithm being multivariant), the size of the program, and the size of the strongly connected components of the program. Also, we observed that the slowdowns encountered correspond to very small runtimes of the algorithms, for example, for smaller programs, which are likely to be due to the overhead of the additional bookeeping required by the algorithm, which is not very concerning, as they are small.

Memory Usage. We also studied the memory usage for the structures needed for analysis, that is, the analysis graphs, and the other structures needed for memoizing. Table 6 contains the maximum memory needed for these structures for any of the modifications

Table 6. *Maximum memory usage for the edition experiments with* `def` *in bytes*

bench	mon	mon-inc	mod	mod-inc	reduction (mon-inc vs mod-inc)
hanoi	16K	16K	12K	12K	0.75 (25%)
aiakl	28K	28K	8K	16K	0.57 (43%)
qsort	28K	32K	12K	16K	0.50 (50%)
progeom	24K	32K	20K	20K	0.63 (37%)
bid	80K	80K	36K	40K	0.50 (50%)
rdtok	100K	112K	68K	80K	0.71 (29%)
cleandirs	200K	204K	144K	152K	0.75 (25%)
read	304K	308K	260K	268K	0.87 (13%)
warplan	144K	156K	116K	128K	0.82 (18%)
boyer	140K	144K	76K	80K	0.55 (44%)
peephole	200K	208K	108K	116K	0.56 (44%)
witt	504K	524K	352K	364K	0.69 (30%)
ann	316K	324K	120K	132K	0.41 (59%)
manag_proj	464K	460K	248K	268K	0.58 (42%)
check_links	2.3M	2.3M	1.8M	1.8M	0.78 (22%)

analyzed for each benchmark, that is, the Memory High-Water Mark. For the monolithic case, this is the maximum memory necessary to keep the analysis results, and for the modular case, the maximum size of the analysis results of a module and the intermodular information. We do not show any distinction between the different deletion strategies of the incremental algorithm as the necessary bookkeeping of both is the same.

First, note that, since the incremental algorithms (`mon-inc` and `mod-inc`) need to perform additional bookkeeping, they always need more memory than the corresponding non-incremental ones (`mon` and `mod`). However, this difference is small and arguably a very reasonable price to pay for the significant reductions in analysis times. Also note that the modular analyses (`mod` and `mod-inc`) always bring a reduction in the memory required to be able to complete every analysis instance. This is of course important because, while it is always possible to wait a bit longer for an analysis result, if the analysis does not fit in the available memory, either the performance will be much worse, due to swapping, or the analysis simply cannot be completed, if virtual memory is depleted.

More importantly, we observe that we obtain a reduction in the memory use of the proposed modular incremental algorithm, `mod-inc`, with respect to the original monolithic incremental algorithm, `mon-inc`. This is shown in the last column of Table 6. The memory usage reduction obtained ranges between 59% for the `ann` benchmark and 13% for the `read` benchmark. Ideally, we would like to achieve a reduction of memory proportional to the number of modules in which the program is distributed, but on one hand there is overhead due to the fact that each module needs to keep information for the calls to predicates imported from other modules, and in addition a very large reduction in maximum memory usage requires the partitions to be of similar size, quite independent, and with similarly sized analysis graphs. Our benchmarks instead typically contain a module with the main functionality and some libraries with simpler code, so that the distribution of code among the modules is not even, and so the correlation between memory usage reduction and number of modules in the program is not direct. However, we expect that in actual applications, which tend to have a much larger number of modules and use a good number of libraries, the memory usage reduction will be much larger.

8 Related work

Classical data flow analysis: Since the first algorithm for incremental analysis was proposed by Rosen (1981), there has been considerable research and proposals in this topic (see the bibliography of Ramalingam and Reps 1993). Depending on how data flow equations are solved, these algorithms can be separated into those based on variable elimination, which include Burke (1990), Carroll and Ryder (1988), and Ryder (1988); and those based on iteration methods which include Cooper and Kennedy (1984) and Pollock and Soffa (1989). A hybrid approach is described in Marlowe and Ryder (1990). Our algorithms are most closely related to those using iteration. Early incremental approaches such as Cooper and Kennedy (1984) were based on *restarting iteration*. That is, the fixpoint of the new program's data flow equations is found by starting iteration from the fixpoint of the old program's data flow equations. This is always safe but may lead to unnecessary imprecision if the old fixpoint is not below the *lfp* of the new equations (Ryder *et al*. 1988). *Reinitialization approaches* such as Pollock and Soffa (1989) improve the accuracy of this technique by reinitializing nodes in the data flow graph to bottom, if they are potentially affected by the program change. Thus, they are as precise as if the new equations had been analyzed from scratch. These algorithms are generally not based on abstract interpretation. Reviser (Arzt and Bodden 2014) extends the more generic IFDS (Reps *et al*. 1995) framework to support incremental program changes. However IFDS is limited to distributive flow functions (related to *condensing* domains), while our approach does not impose any restriction on the domains.

Other work on CHCs and CLPs: Apart from the work that we extend (Hermenegildo *et al*. 1995, 2000; Puebla and Hermenegildo 1996), incremental analysis was proposed (just for incremental addition) in the Vienna abstract machine model (Krall and Berger 1995a,b). It was studied also in compositional analysis of modules in (C)LP (Codish *et al*. 1993; Bossi *et al*. 1994), but it did not consider incremental analysis at the level of clauses. More recently, FLIX (Madsen *et al*. 2016) uses a bottom-up semi-naïve strategy to solve *Datalog programs* extended with lattices and monotone transfer functions. This approach is similar to CLP analysis via bottom-up abstract interpretation. However, it has not been extended to support incremental updates. *Incremental tabling* (Swift 2014) offers a straightforward method to design incremental analyses (Eichberg *et al*. 2007), when they can be expressed as tabled logic programs. However, while these methods are much closer to our incremental algorithm, they may suffer similar problems than generic incremental computation due to the lack of fine-grained control.

Other work on modular analysis: Cousot and Cousot (2002) is based on splitting large programs into smaller parts (e.g., based on the source code structure). Exploiting modularity has proved essential in industrial-scale analyzers (Cousot *et al*. 2009; Fähndrich and Logozzo 2011; Calcagno and Distefano 2011). Despite the fact that separate analysis provides only coarse-grained incrementality, there have been surprisingly few results studying its combination with fine-grained incremental analysis.

Verification. Incremental algorithms have also been proposed to perform formal verification of programs. They reuse results of prior verification, given some properties to be verified (Conway *et al.* 2005; Fedyukovich *et al.* 2016; Rothenberg *et al.* 2018), which do not take advantage of modular structures of programs. Sery *et al.* (2012) use a compositional approach to obtain properties for each procedure, checking whether each property holds for the successive versions of the program. These approaches are not directly comparable with the work that we have presented, since we do not rely on any specifications of the program behavior to run the analysis.

Generic incremental computation frameworks: Obviously, the possibility exists of using a general incrementalized execution algorithm. Incremental algorithms compute an updated output from a previous output and a difference on the input data, which the hope that the process is (computationally) cheaper than computing from scratch a new output for the new input. The approach of Szabó *et al.* (2016) takes advantage of an underlying incremental evaluator, IncQuery, and implements modules via the monolithic approach. There exist other frameworks such as self-adjusting computation (Acar 2009) that provides libraries for incrementalizing non-incremental algorithms. Although in some cases (like *tree contraction*) it can reproduce the performance of specialized incremental algorithms, experiments show that in general there is a significant overhead (between 4 and 10) over non-incremental algorithms. We believe that it is a promising approach but not yet ready for replacing incremental algorithms designed, proved, and implemented from scratch.

9 Conclusions

We have described, implemented, and evaluated a context-sensitive, fixpoint analysis algorithm that performs efficient context-sensitive analysis incrementally on modular partitions of programs. We provided a unified view of the algorithms that we built upon, providing a formal description of their correctness and precision guarantees that also covers widening. Our algorithm takes care of propagating the fine-grain change information across module boundaries and implements all the actions required to recompute the analysis fixpoint incrementally after additions and deletions in the program. We have shown that the algorithm is correct and computes the most precise analysis for finite abstract domains, while supporting widening for dealing with infinite domains. We have also provided some new results for the baseline algorithms. We have also implemented and benchmarked the proposed approach within the Ciao/CiaoPP system. Our preliminary results show promising speedups for programs of medium and larger size when compared with existing non-modular, fine-grain incremental analysis techniques, as well as improvements in memory consumption. In addition, the finer granularity of the proposed modular incremental fixpoint algorithm also brings improvements with respect to modular analysis alone (which only preserved analysis results at the module boundaries), producing better results even in the limit case of analyzing the whole program from scratch. Finally, we have also observed some correlations between obtainable speedups and certain benchmark-related analysis characteristics, such as the number of nodes in the analysis graph and the number of calls to ⊤ abstractions. Going deeper in this direction is a clear avenue for future work.

Conflict of interest

The authors declare none.

Supplementary material

To view supplementary material for this article, please visit https://doi.org/10.1017/S1471068420000496.

References

Acar, U. A. 2009. Self-adjusting computation: (an overview). In *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, 19–20 January 2009*, G. Puebla and G. Vidal, Eds. ACM, 1–6.

Albert, E., Arenas, P., Genaim, S., Puebla, G. and Zanardini, D. 2012. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science (Special Issue on Quantitative Aspects of Programming Languages) 413,* 1, 142–159.

Albert, E., Correas, J., Puebla, G. and Román-Díez, G. 2012. Incremental resource usage analysis. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, 23–24 January 2012.* ACM Press, 25–34.

Albert, E., Gómez-Zamalloa, M., Hubert, L. and Puebla, G. 2007. Verification of Java bytecode using analysis and transformation of logic programs. In *Ninth International Symposium on Practical Aspects of Declarative Languages (PADL 2007)*, LNCS, vol. 4354. Springer-Verlag, 124–139.

Apt, K. R. 1990. Introduction to logic programming. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Elsevier, 493–576.

Arzt, S. and Bodden, E. 2014. Reviser: Efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *36th International Conference on Software Engineering, ICSE'14, Hyderabad, India - May 31–June 07, 2014*, P. Jalote, L. C. Briand and A. van der Hoek, Eds. ACM, 288–298.

Banda, G. and Gallagher, J. P. 2009. Analysis of linear hybrid systems in CLP. In *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008, Valencia, Spain, 17–18 July 2008*, M. Hanus, Ed. Lecture Notes in Computer Science, vol. 5438. Springer, 55–70.

Bjørner, N., Gurfinkel, A., McMillan, K. L. and Rybalchenko, A. 2015. Horn clause solvers for program verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner and W. Schulte, Eds. Lecture Notes in Computer Science, vol. 9300. Springer, 24–51.

Bjørner, N., McMillan, K. L. and Rybalchenko, A. 2013. On solving universally quantified Horn clauses. In *SAS*, F. Logozzo and M. Fähndrich, Eds. LNCS, vol. 7935. Springer, 105–125.

Bossi, A., Gabbrieli, M., Levi, G. and Meo, M. 1994. A compositional semantics for logic programs. *Theoretical Computer Science 122,* 1, 2, 3–47.

Braem, C., Charlier, B. L., Modart, S. and Hentenryck, P. V. 1994. Cardinality analysis of Prolog. In *Proceedings of International Symposium on Logic Programming*. MIT Press, Ithaca, NY, 457–471.

Bruynooghe, M. 1991. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming 10*, 91–124.

Bueno, F., de la Banda, M. G., Hermenegildo, M. V., Marriott, K., Puebla, G. and Stuckey, P. 2001. A model for inter-module analysis and optimizing compilation. In *Logic-based Program Synthesis and Transformation*. LNCS, vol. 2042. Springer-Verlag, 86–102.

Burke, M. 1990. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems 12,* 3, 341–395.

Calcagno, C. and Distefano, D. 2011. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, 18–20 April 2011. Proceedings*, M. G. Bobaru, K. Havelund, G. J. Holzmann and R. Joshi, Eds. Lecture Notes in Computer Science, vol. 6617. Springer, 459–465.

Carroll, M. and Ryder, B. 1988. Incremental data flow analysis via dominator and attribute updates. In *15th ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 274–284.

Codish, M., Debray, S. and Giacobazzi, R. 1993. Compositional analysis of modular logic programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93*. ACM, Charleston, South Carolina, 451–464.

Conway, C. L., Namjoshi, K. S., Dams, D. and Edwards, S. A. 2005. Incremental algorithms for inter-procedural analysis of safety properties. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, 6–10 July 2005*, K. Etessami and S. K. Rajamani, Eds. Lecture Notes in Computer Science, vol. 3576. Springer, 449–461.

Cooper, K. and Kennedy, K. 1984. Efficient computation of flow insensitive interprocedural summary information. In *ACM SIGPLAN Symposium on Compiler Construction (SIGPLAN Notices*, vol. 19(6)). ACM Press, 247–258.

Correas, J., Puebla, G., Hermenegildo, M. V. and Bueno, F. 2006. Experiments in context-sensitive analysis of modular programs. In *15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*. LNCS, vol. 3901. Springer-Verlag, 163–178.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL'77)*. ACM Press, 238–252.

Cousot, P. and Cousot, R. 2002. Modular static program analysis, invited paper. In *Eleventh International Conference on Compiler Construction, CC 2002*. LNCS, vol. 2304. Springer, 159–178.

Cousot, P., Cousot, R., Feret, J., Miné, A., Mauborgne, L. and Rival, X. 2009. Why does Astrée scale up? *Formal Methods in System Design (FMSD) 35,* 3, 229–264.

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2014. VeriMAP: A tool for verifying programs through transformations. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, 5–13 April 2014. Proceedings*, E. Ábrahám and K. Havelund, Eds. Lecture Notes in Computer Science, vol. 8413. Springer, 568–574.

de Moura, L. M. and Bjørner, N. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, C. R. Ramakrishnan and J. Rehof, Eds. Lecture Notes in Computer Science, vol. 4963. Springer, 337–340.

Debray, S., Lopez-Garcia, P. and Hermenegildo, M. V. 1997. Non-failure analysis for logic programs. In *1997 International Conference on Logic Programming*. MIT Press, Cambridge, MA, Cambridge, MA, 48–62.

Dumortier, V., Janssens, G., Simoens, W. and García de la Banda, M. 1993. Combining a definiteness and a freeness abstraction for CLP languages. In *Workshop on Logic Program Synthesis and Transformation*.

Eichberg, M., Kahl, M., Saha, D., Mezini, M. and Ostermann, K. 2007. *Automatic Incrementalization of Prolog Based Static Analyses*. Springer, Berlin, Heidelberg, 109–123.

Fähndrich, M. and Logozzo, F. 2011. Static contract checking with abstract interpretation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software, FoVeOOS'10*. Lecture Notes in Computer Science, vol. 6528. Springer-Verlag, Berlin, Heidelberg, 10–30.

Fedyukovich, G., Gurfinkel, A. and Sharygina, N. 2016. Property directed equivalence via abstract simulation. In *International Conference on Computer Aided Verification*. Springer, 433–453.

Gallagher, J., Hermenegildo, M. V., Kafle, B., Klemen, M., Lopez-Garcia, P. and Morales, J. 2020. From big-step to small-step semantics and back with interpreter specialization (invited paper). In *Proceedings of the Eighth International Workshop on Verification and Program Transformation (VPT 2020)*. Electronic Proceedings in Theoretical Computer Science (EPTCS). Open Publishing Association (OPA), 50–65. Co-located with ETAPS 2020.

Grebenshchikov, S., Gupta, A., Lopes, N. P., Popeea, C. and Rybalchenko, A. 2012. HSF(C): A software verifier based on Horn clauses - (competition contribution). In *TACAS*, C. Flanagan and B. König, Eds. LNCS, vol. 7214. Springer, 549–551.

Gurfinkel, A., Kahsai, T., Komuravelli, A. and Navas, J. A. 2015. The SeaHorn verification framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, 18–24 July 2015, Proceedings, Part I*. LNCS, vol. 9206. Springer, 343–361.

Henriksen, K. S. and Gallagher, J. P. 2006. Abstract interpretation of PIC programs through logic programming. In *SCAM'06, Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society, 184–196.

Hermenegildo, M. V., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J. and Puebla, G. 2012. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming 12,* 1–2, 219–252.

Hermenegildo, M. V., Puebla, G., Bueno, F. and Lopez-Garcia, P. 2005. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Science of Computer Programming 58,* 1–2, 115–140.

Hermenegildo, M. V., Puebla, G., Marriott, K. and Stuckey, P. 1995. Incremental analysis of logic programs. In *International Conference on Logic Programming*. MIT Press, 797–811.

Hermenegildo, M. V., Puebla, G., Marriott, K. and Stuckey, P. 2000. Incremental analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems 22,* 2, 187–223.

Jaffar, J. and Lassez, J.-L. 1987. Constraint logic programming. In *ACM Symposium on Principles of Programming Languages*. ACM, 111–119.

Jaffar, J., Murali, V., Navas, J. A. and Santosa, A. E. 2012. TRACER: A symbolic execution tool for verification. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, 7–13 July 2012 Proceedings*, P. Madhusudan and S. A. Seshia, Eds. Lecture Notes in Computer Science, vol. 7358. Springer, 758–766.

Kafle, B., Gallagher, J. P. and Morales, J. F. 2016. RAHFT: A tool for verifying Horn clauses using abstract interpretation and finite tree automata. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, 17–23 July 2016,* *Proceedings, Part I*, S. Chaudhuri and A. Farzan, Eds. Lecture Notes in Computer Science, vol. 9779. Springer, 261–268.

Kahn, G. 1987. Natural semantics. In F. Brandenburg, G. Vidal-Naque and M. Wirsing, Eds. Lecture Notes in Computer Science, vol. 247. Springer, 22–39.

Kelly, A., Marriott, K., Søndergaard, H. and Stuckey, P. 1997. A generic object oriented incremental analyser for constraint logic programs. In *Proceedings of the 20th Australasian Computer Science Conference*, 92–101.

Khedker, U. P. and Karkare, B. 2008. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In *Compiler Construction, 17th International Conference, CC 2008, Budapest, Hungary, March 29–April 6, 2008*, L. J. Hendren, Ed. Lecture Notes in Computer Science, vol. 4959. Springer, 213–228.

King, A., Lu, L. and Genaim, S. 2006. Detecting determinacy in Prolog programs. In *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, 17–20 August 2006, Proceedings*, S. Etalle and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 4079. Springer, 132–147.

Krall, A. and Berger, T. 1995a. Incremental global compilation of Prolog with the vienna abstract machine. In *International Conference on Logic Programming.* MIT Press.

Krall, A. and Berger, T. 1995b. The VAM$_{\mathrm{AI}}$ - An abstract machine for incremental global dataflow analysis of Prolog. In *ICLP'95 Post-Conference Workshop on Abstract Interpretation of Logic Languages*, M. G. de la Banda, G. Janssens, and P. Stuckey, Eds. Science University of Tokyo, Tokyo, 80–91.

Liqat, U., Georgiou, K., Kerrison, S., Lopez-Garcia, P., Hermenegildo, M. V., Gallagher, J. P. and Eder, K. 2016. Inferring parametric energy consumption functions at different software levels: ISA vs. LLVM IR. In *Foundational and Practical Aspects of Resource Analysis: 4th International Workshop, FOPARA 2015, London, UK, 11 April 2015. Revised Selected Papers*, M. V. Eekelen and U. D. Lago, Eds. Lecture Notes in Computer Science, vol. 9964. Springer, 81–100.

Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., Lopez-Garcia, P., Grech, N., Hermenegildo, M. V. and Eder, K. 2014. Energy consumption analysis of programs based on XMOS ISA-level models. In *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Revised Selected Papers*, G. Gupta and R. Peña, Eds. Lecture Notes in Computer Science, vol. 8901. Springer, 72–90.

Lloyd, J. 1987. *Foundations of Logic Programming*, 2nd extended edition. Springer.

Lopez-Garcia, P., Bueno, F. and Hermenegildo, M. V. 2010. Automatic inference of determinacy and mutual exclusion for logic programs using mode and type analyses. *New Generation Computing 28,* 2, 117–206.

Lopez-Garcia, P., Darmawan, L., Klemen, M., Liqat, U., Bueno, F. and Hermenegildo, M. V. 2018. Interval-based resource usage verification by translation into Horn clauses and an application to energy consumption. *Theory and Practice of Logic Programming, Special Issue on Computational Logic for Verification 18,* 2, 167–223. arXiv:1803.04451.

Madsen, M., Yee, M. and Lhoták, O. 2016. From Datalog to FLIX: A declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, 13–17 June 2016*, C. Krintz and E. Berger, Eds. ACM, 194–208.

Marlowe, T. and Ryder, B. 1990. An efficient hybrid algorithm for incremental data flow analysis. In *17th ACM Symposium on Principles of Programming Languages (POPL).* ACM Press, 184–196.

Marriott, K. and Stuckey, P. J. 1998. *Programming with Constraints: an Introduction.* MIT Press.

Méndez-Lojo, M., Navas, J. and Hermenegildo, M. 2007. A flexible (C)LP-based approach to the analysis of object-oriented programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*. Lecture Notes in Computer Science, vol. 4915. Springer-Verlag, 154–168.

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

Muthukumar, K. and Hermenegildo, M. 1991. Combined determination of sharing and freeness of program variables through abstract interpretation. In *International Conference on Logic Programming (ICLP 1991)*. MIT Press, 49–63.

Muthukumar, K. and Hermenegildo, M. 1992. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming 13,* 2/3, 315–347.

Navas, J., Méndez-Lojo, M. and Hermenegildo, M. 2008. Safe upper-bounds inference of energy consumption for Java bytecode applications. In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, 29–32. Extended Abstract.

Navas, J., Méndez-Lojo, M. and Hermenegildo, M. V. 2007. An efficient, context and path sensitive analysis framework for Java programs. In *9th Workshop on Formal Techniques for Java-like Programs FTfJP 2007*.

Navas, J., Méndez-Lojo, M. and Hermenegildo, M. V. 2009. User-definable resource usage bounds analysis for Java bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*. Electronic Notes in Theoretical Computer Science, vol. 253. Elsevier - North Holland, 65–82.

Perez-Carrasco, V., Klemen, M., Lopez-Garcia, P., Morales, J. and Hermenegildo, M. V. 2020. Cost analysis of smart contracts via parametric resource analysis. In *Proceedings of the 27th Static Analysis Symposium (SAS 2020)*. LNCS. Springer-Verlag.

Plotkin, G. 1981. *A Structural Approach to Operational Semantics*. Technical report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark.

Plotkin, G. D. 2004. A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming 60–61*, 17–139.

Pollock, L. and Soffa, M. 1989. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering 15,* 12, 1537–1549.

Puebla, G., Correas, J., Hermenegildo, M. V., Bueno, F., García de la Banda, M., Marriott, K. and Stuckey, P. J. 2004. A generic framework for context-sensitive analysis of modular programs. In *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, M. Bruynooghe and K. Lau, Eds. LNCS, vol. 3049. Springer-Verlag, Heidelberg, Germany, 234–261.

Puebla, G. and Hermenegildo, M. V. 1996. Optimized algorithms for the incremental analysis of logic programs. In *International Static Analysis Symposium (SAS 1996)*. Lecture Notes in Computer Science, vol. 1145. Springer-Verlag, 270–284.

Ramalingam, G. and Reps, T. 1993. categorized bibliography on incremental computation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93*. ACM, Charleston, South Carolina.

Reps, T. W., Horwitz, S. and Sagiv, S. 1995. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 49–61.

Robinson, J. A. 1965. machine oriented logic based on the resolution principle. *Journal of the ACM 12,* 23, 23–41.

Rosen, B. 1981. Linear cost is sometimes quadratic. In *Eighth ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 117–124.

Rothenberg, B., Dietsch, D. and Heizmann, M. 2018. Incremental verification using trace abstraction. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, 29–31 August 2018*, A. Podelski, Ed. Lecture Notes in Computer Science, vol. 11002. Springer, 364–382.

Ryder, B. 1988. Incremental data-flow analysis algorithms. *ACM Transactions on Programming Languages and Systems 10,* 1, 1–50.

Ryder, B., Marlowe, T. and Paull, M. 1988. Conditions for incremental iteration: Examples and counterexamples. *Science of Computer Programming 11,* 1, 1–15.

Sery, O., Fedyukovich, G. and Sharygina, N. 2012. Incremental upgrade checking by means of interpolation-based function summaries. In *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, 22–25 October 2012*, G. Cabodi and S. Singh, Eds. IEEE, 114–121.

Sharir, M. and Pnueli, A. 1978. *Two Approaches to Interprocedural Data Flow Analysis.* New York University. Courant Institute of Mathematical Sciences.

Swift, T. 2014. Incremental tabling in support of knowledge representation and reasoning. *Theory and Practice of Logic Programming 14,* 4–5, 553–567.

Szabó, T., Erdweg, S. and Voelter, M. 2016. Inca: A DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, 3–7 September 2016*, D. Lo, S. Apel and S. Khurshid, Eds. ACM, 320–331.

Thakur, M. and Nandivada, V. K. 2020. Mix your contexts well: Opportunities unleashed by recent advances in scaling context-sensitivity. In *Proceedings of the 29th International Conference on Compiler Construction.* CC 2020. Association for Computing Machinery, New York, NY, USA, 27–38.