An Incremental Approach to Abstraction-Carrying Code

Elvira Albert[1], Puri Arenas[1], and Germán Puebla[2]

[1] Complutense University of Madrid, {elvira,puri}@sip.ucm.es
[2] Technical University of Madrid, german@fi.upm.es

Abstract. *Abstraction-Carrying Code* (ACC) has recently been proposed as a framework for proof-carrying code (PCC) in which the code supplier provides a program together with an *abstraction* (or abstract model of the program) whose validity entails compliance with a predefined safety policy. The abstraction plays thus the role of safety certificate and its generation (and validation) is carried out automatically by a fixed-point analyzer. Existing approaches for PCC are developed under the assumption that the consumer reads and validates the entire program w.r.t. the *full* certificate at once, in a non incremental way. In the context of ACC, we propose an *incremental* approach to PCC for the generation of certificates and the checking of untrusted *update*s of a (trusted) program, i.e., when a producer provides a modified version of a previously validated program. By update, we refer to any arbitrary change on a program, i.e., the extension of the program with new procedures, the deletion of existing procedures and the replacement of existing procedures by new versions for them. Our proposal is that, if the consumer keeps the original (fixed-point) abstraction, it is possible to provide only the program updates and the incremental certificate (i.e., the *difference* of abstractions). The first obvious advantage is that the size of the transmitted certificate can be considerably reduced. Furthermore, it is now possible to define an *incremental checking* algorithm which, given the new updates and its incremental certificate, only re-checks the fixpoint for each procedure affected by the updates and the propagation of the effect of these fixpoint changes. As a consequence, both certificate transmission time and checking time can be reduced significantly. To the best of our knowledge, this is the first incremental approach to PCC.

1 Introduction

Proof-Carrying Code (PCC) [12] is a general technique for mobile code safety which proposes to associate safety information in the form of a *certificate* to programs. The certificate (or proof) is created at compile time by the *certifier* on the code supplier side, and it is packaged along with the code. The consumer who receives or downloads the (untrusted) code+certificate package can then run a *checker* which by an efficient inspection of the code and the certificate can verify the validity of the certificate and thus compliance with the safety policy. The key benefit of this "certificate-based" approach to mobile code safety is that the consumer's task is reduced from the level of proving to the level of checking, a task which should be much simpler, efficient, and automatic than generating the original certificate.

Abstraction-carrying code (ACC) [3] has been recently proposed as an enabling technology for PCC in which an *abstraction* (i.e., an abstract model of the program) plays the role of certificate. An important feature of ACC is that not only the checking, but also the generation of the abstraction (or fixpoint) is *automatically* carried

out by a fixed-point analyzer. Lightweight bytecode verification [14] is another PCC method which relies on analysis techniques (namely on type analyses in the style of those used for Java bytecode verification [9]) to generate and check certificates in the context of the Java Card language. In this paper, we will consider analyzers which construct a program *analysis graph* which is interpreted as an abstraction of the (possibly infinite) set of states explored by the concrete execution. Essentially, the certification/analysis carried out by the supplier is an iterative process which repeatedly traverses the analysis graph until a fixpoint is reached. A key idea in ACC is that, since the certificate is a fixpoint, a single pass over the analysis graph is sufficient to validate the certificate in the consumer side.

Existing models for PCC (ACC among them) are based on checkers which receive a "certificate+program" package and read and validate the entire program w.r.t. its certificate at once, in a non incremental way. However, there are situations which are not well suited to this simple model and which instead require only rechecking certain parts of the analysis graph which has already been validated. In particular, we consider possible untrusted *updates* of a validated (trusted) code, i.e., a code producer can (periodically) send to its consumers new updates of a previously submitted package. By updates, we mean any modification over a program including: 1) the *addition* of new data/procedures and the extension of already existing procedures with new functionalities, 2) the *deletion* of procedures or parts of them and 3) the *replacement* of certain (parts of) procedures by new versions for them. In such a context of frequent software updates, it appears inefficient to submit a full certificate (superseding the original one) and to perform the checking of the entire updated program from scratch, as needs to be done with current systems. In the context of ACC, we investigate an *incremental* approach to PCC, both for the certificate generation as well as for the checking process.

Regarding the first issue, when a program is updated, a new fixpoint has to be computed for the updated program. Such fixpoint differs from the original fixpoint stored in the certificate in a) the new fixpoint for each procedure affected by the changes and b) the update of certain (existing) fixpoints affected by the propagation of the effect of a). However, certain parts of the original certificate may not have affected by the changes. Our proposal is that, if the consumer still keeps the original abstraction, it is possible to provide, along with the program updates, only the *difference* of both abstractions, i.e., the *incremental certificate*. Essentially, the incremental certificate will contain the subset of the new fixpoint which is different from the original fixpoint. The first obvious advantage of our incremental approach is that the size of the certificate may be substantially reduced by submitting only the increment.

The second issue in incremental PCC is that the task performed by the checker can also be further reduced. In principle, a non-incremental checker (like the one in [3]) requires a whole traversal of the analysis graph where the entire program + updates is checked against the (full) certificate. However, it is now possible to define an *incremental checking* algorithm which, given the updates and its incremental certificate, only rechecks the part of the analysis graph for the procedures which have been affected by the updates and, also, propagates and rechecks the effect of these changes. In order to perform such propagation of changes, the *dependencies* between the nodes of the original analysis graph have to be stored by the consumers, together with the original certificate. With this, the checking process is carried out in a single pass over the part of the abstraction affected by the updates. Thus, the second advantage of our incremental approach is that checking time is further reduced. We

believe that our incremental approach contributes to the practical uptake of PCC systems since it can significantly reduce certificate size and checking time in the context of program updates.

The paper is organized as follows. Section 2 introduces briefly some notation and preliminary notions on abstract interpretation and ACC. In Section 3, we present an overview of incremental ACC. Section 4 recalls the notion of full certificate and Section 5 introduces the concepts of incremental certificate and incremental certifier. In Section 6, we instrument a non incremental checking algorithm with dependencies in order to support incrementally. In Section 7, we present an incremental abstract interpretation-based checker. Finally, Section 8 concludes.

2 Abstraction-Carrying Code

We assume some familiarity with abstract interpretation (see [6]), (Constraint) Logic Programming (C)LP (see, e.g., [11, 10]) and PCC [12].

An abstract interpretation-based certifier is a function CERTIFIER: $\mathit{Prog} \times \mathit{ADom} \times \mathit{AInt} \mapsto \mathit{ACert}$ which for a given program $P \in \mathit{Prog}$, an abstract domain $D_\alpha \in \mathit{ADom}$ and an abstract safety policy $I_\alpha \in \mathit{AInt}$ generates an abstract certificate $\mathit{Cert}_\alpha \in \mathit{ACert}$, by using an abstract interpreter for D_α, which entails that P satisfies I_α. In the following, we denote that I_α and Cert_α are specifications given as abstract semantic values of D_α by using the same subscript α.

The basics for defining such certifiers (and their corresponding checkers) in ACC are summarized in the following five points:

Approximation. We consider an *abstract domain* $\langle D_\alpha, \sqsubseteq \rangle$ and its corresponding *concrete domain* $\langle 2^D, \subseteq \rangle$, both with a complete lattice structure. Abstract values and sets of concrete values are related by an *abstraction* function $\alpha : 2^D \to D_\alpha$, and a *concretization* function $\gamma : D_\alpha \to 2^D$. An abstract value $y \in D_\alpha$ is a *safe approximation* of a concrete value $x \in D$ iff $x \in \gamma(y)$. The concrete and abstract domains must be related in such a way that the following holds [6] $\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x$ and $\forall y \in D_\alpha : \ \alpha(\gamma(y)) = y$. In general $\sqsubseteq$ is induced by $\subseteq$ and α. Similarly, the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of 2^D in a precise sense.

Analysis. We consider the class of *fixed-point semantics* in which a (monotonic) semantic operator, S_P, is associated to each program P. The meaning of the program, $[\![P]\!]$, is defined as the least fixed point of the S_P operator, i.e., $[\![P]\!] = \mathrm{lfp}(S_P)$. If S_P is continuous, the least fixed point is the limit of an iterative process involving at most ω applications of S_P starting from the bottom element of the lattice. Using abstract interpretation, we can usually only compute $[\![P]\!]_\alpha$, as $[\![P]\!]_\alpha = \mathrm{lfp}(S_P^\alpha)$. The operator S_P^α is the abstract counterpart of S_P.

$$\mathsf{analyzer}(P, D_\alpha) = \mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha \tag{1}$$

Correctness of analysis ensures that $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$, i.e., $[\![P]\!] \in \gamma([\![P]\!]_\alpha)$. Thus, such *abstraction* can be used as certificate.

Certificate. Let Cert_α be a safe approximation of P. If an abstract safety specification I_α can be proved w.r.t. Cert_α, then P satisfies the safety policy and Cert_α is a valid certificate:

$$\mathit{Cert}_\alpha \text{ is } \textit{a valid certificate} \text{ for } P \text{ w.r.t. } I_\alpha \text{ iff } \mathit{Cert}_\alpha \sqsubseteq I_\alpha \tag{2}$$

Together, Equations (1) and (2) define a certifier which provides program fixpoints, $[\![P]\!]_\alpha$, as certificates which entail a given safety policy, i.e., by taking $Cert_\alpha = [\![P]\!]_\alpha$.

Checking. A checker is a function CHECKER: $Prog \times ADom \times ACert \mapsto bool$ which for a program $P \in Prog$, an abstract domain $D_\alpha \in ADom$ and certificate $Cert_\alpha \in ACert$ checks whether $Cert_\alpha$ is a fixpoint of S_P^α or not:

$$\text{CHECKER}(P, D_\alpha, Cert_\alpha) \textit{ returns true iff } (S_P^\alpha(Cert_\alpha) \equiv Cert_\alpha) \tag{3}$$

Verification Condition Regeneration. To retain the safety guarantees, the consumer must regenerate a trustworthy verification condition –Equation (2)– and use the incoming certificate to test for adherence of the safety policy.

$$P \textit{ is trusted iff } Cert_\alpha \sqsubseteq I_\alpha \tag{4}$$

A fundamental idea in ACC is that, while analysis –Equation (1)– is an iterative process, checking –Equation (3)– is guaranteed to be done in a *single pass* over the abstraction.

3 An Overview of Incremental ACC

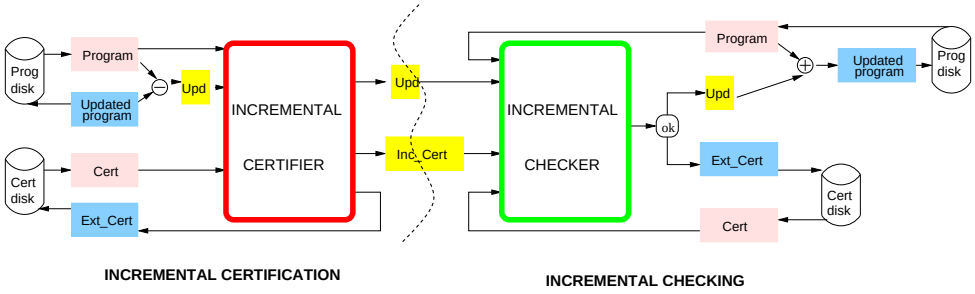


Fig. 1. Overview of Incremental Abstraction-Carrying Code

Figure 1 presents an overview of incremental ACC. On the left side, the producer starts from an Updated Program, U_P, w.r.t. a previously certified Program, P. It first retrieves from disk P and its certificate, Cert, computed in the previous certification phase. Next, the process "$\ominus$" compares both programs and returns the differences between them, $Upd(P)$, i.e, the program Updates which applied to P result in U_P, written as $Upd(P) = U_P \ominus P$. Note that, from an implementation perspective, a program update should contain both the new updates to be applied to the program and instructions on where to place and remove such new code. This can be easily done by using the traditional Unix *diff* format for coding program updates. An Incremental Certifier generates from Cert, P and $Upd(P)$ an incremental certificate, Inc_Cert, which can be used by the consumer to validate the new updates. The package "$Upd(P)$+Inc_Cert" is submitted to the code consumer. Finally, in order to

have a compositional incremental approach, the producer has to update the original certificate and program with the new updates. Thus, the resulting Ext_Cert and U_P are stored in disk replacing Cert and P, respectively.

On the right side of the figure, the consumer receives the untrusted package. In order to validate the incoming update w.r.t. the provided (incremental) certificate, it first retrieves from disk P and Cert. Next, it reconstructs the updated program by using an operator "$\oplus$" which applies the update to P and generates $U_P = P \oplus Upd(P)$. This can implemented by using a program in the spirit of the traditional Unix *patch* command as $\oplus$ operator. An Incremental Checker now efficiently validates the new modification by using the stored data and the incoming incremental certificate. If the validation succeeds (returns ok), the checker will have reconstructed the full certificate. As before, the updated program and extended certificate are stored in disk (superseding the previous versions) for future (incremental) updates. In order to simplify our scheme, we assume that the safety policy and the generation of the verification condition [12] are embedded within the certifier and checker. However, in an incremental approach, producer and consumer could perfectly agree on a new safety policy to be implied by the modification. It should be noted that this does not affect our incremental approach and the verification condition would be generated exactly as in non incremental PCC.

Let us now characterize the types of updates we consider and how they can be dealt within the ACC scheme. Given a program P, we define an *update* of P, written as $Upd(P)$, as a set of tuples of the form $\langle A, Add(A), Del(A)\rangle$, where $A = p(x_1, \ldots, x_n)$ is an atom in base form, and:

- $Add(A)$ is the set of rules which are to be added to P for predicate p. This includes both the case of addition of new procedures, when p did not exist in P, as well as the extension of additional rules (or functionality) for p, if it existed.
- $Del(A)$ is the set of rules which are to be removed from P for predicate p.

Note that, for the sake of simplicity, we do not include the instructions on where to place and remove such code in the formalization of our method. We distinguish three classes of updates: the *addition* of procedures occurs when $\forall A,\ Del(A) = \emptyset$, the *deletion* of procedures occurs if $\forall A,\ Add(A) = \emptyset$ and the remaining cases are considered *arbitrary changes*.

Addition of Procedures. When a program P is extended with new procedures or new clauses for existing procedures, the original certificate $Cert_\alpha$ is not guaranteed to be a correct fixpoint any longer, because the contribution of the new rules can lead to a more general answer. Consider P^{add} the program after applying some additions and $Cert_\alpha^{add}$ the certificate computed from scratch for P^{add}. Then, $Cert_\alpha \sqsubseteq Cert_\alpha^{add}$. This means that $Cert_\alpha$ is no longer valid in order to entail safety. Therefore, we need to lub the contribution of the new rules and submit, together with the extension, the new certificate $Cert_\alpha^{add}$ (or the difference of both certificates). The consumer will thus test the safety policy w.r.t. $Cert_\alpha^{add}$.

Deletion of Procedures. The first thing to note is that in order to entail the safety policy, unlike extensions over the program, we need not change the certificate at all when some procedures are deleted. Consider P^{del} the program after applying some deletions and $Cert_\alpha^{del}$ the certificate computed from scratch for P^{del}. The original certificate $Cert_\alpha$ is trivially guaranteed to be a correct fixpoint (hence a correct certificate), because the contribution of the rules were lubbed to give $Cert_\alpha$ and so it still correctly describes the contribution of each remaining rule. By applying

Equation 2, $Cert_\alpha$ is still valid for P^{del} w.r.t. I_α since $Cert_\alpha \sqsubseteq I_\alpha$. Therefore, more accuracy is not needed to ensure compliance with the safety policy. This suggests that the incremental certificate can be empty and the checking process does not have to check any procedure. However, it can happen that a new, more precise, safety policy is agreed by the consumer and producer. Also, this accuracy could be required in a later modification. Although $Cert_\alpha$ is a correct certificate, it is possibly less *accurate* than $Cert_\alpha^{del}$, i.e., $Cert_\alpha^{del} \sqsubseteq Cert_\alpha$. It is therefore interesting to define the corresponding incremental algorithm for reconstructing $Cert_\alpha^{del}$ and checking the deletions and the propagation of their effects.

Arbitrary Changes. The case of arbitrary changes considers that rules can both be deleted from and added to an already validated program. In this case, the new certificate for the modified program can be either equal, more or less precise than the original one, or even not comparable. Imagine that an arbitrary change replaces a rule R_a, which contributes to a fixpoint $Cert_\alpha^a$, with a new one R_b which contributes to a fixpoint $Cert_\alpha^b$ such that $Cert_\alpha^{ab} = \mathsf{Alub}(Cert_\alpha^a, Cert_\alpha^b)$ and $Cert_\alpha^a \sqsubset Cert_\alpha^{ab}$ and $Cert_\alpha^b \sqsubset Cert_\alpha^{ab}$. The point is that we cannot just abstract the new rule and add it to the previous fixpoint, i.e., we cannot use $Cert_\alpha^{ab}$ as certificate and have to provide the more accurate $Cert_\alpha^b$. The reason is that it might be possible to attest the safety policy by independently using $Cert_\alpha^a$ and $Cert_\alpha^b$ while it cannot be implied by using their lub $Cert_\alpha^{ab}$. This happens for certain safety policies which contain disjunctions, i.e., $Cert_\alpha^a \vee Cert_\alpha^b$ does not correspond to their lub $Cert_\alpha^{ab}$. Therefore, arbitrary changes require a precise recomputation of the new fixpoint and its proper checking.

4 The Full Certificate

Although ACC and Incremental ACC, as overviewed above, are general proposals not tied to any particular programming paradigm, our developments for incremental ACC (as well as for the original ACC framework [3]) are formalized in the context of (C)LP. Very briefly, a *constraint* is essentially a conjunction of expressions built from predefined predicates (such as term equations or inequalities over the reals) whose arguments are constructed using predefined functions (such as real addition). An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and t_i are terms. A *literal* is either an atom or a constraint. A *goal* is a finite sequence of literals. A *rule* is of the form H `:-` D where H, the *head*, is an atom and D, the *body*, is a possibly empty finite sequence of literals. A *constraint logic program*, or *program*, is a finite set of rules. Program rules are assumed to be normalized: only distinct variables are allowed to occur as arguments to atoms. Furthermore, we require that each rule defining a predicate p has identical sequence of variables $x_{p_1}, \ldots x_{p_n}$ in the head atom, i.e., $p(x_{p_1}, \ldots x_{p_n})$. We call this the *base form* of p. This is not restrictive since programs can always be normalized, and it will facilitate the presentation of the checking algorithms.

For concreteness, we rely on an abstract interpretation-based analysis algorithm in the style of the generic analyzer of [7]. This goal-dependent analysis algorithm, which we refer to as ANALYZE, given a program P and abstract domain D_α, receives a set $S_\alpha \in AAtom$ of Abstract Atoms (or *call patterns*) and constructs an *analysis graph* [5] for S_α. The elements of S_α are pairs of the form $A : CP$ where A is a procedure descriptor and CP is an abstract substitution (i.e., a condition of the run-time bindings) of A expressed as $CP \in D_\alpha$.[1] Then, the analysis graph is an

[1] We sometimes omit the subscript α from S_α when it is clear from the context.

abstraction of the (possibly infinite) set of (possibly infinite) trees explored by the concrete execution of initial calls described by S_α in P. The program analysis graph computed by ANALYZE(S_α) for P in D_α can be implicitly represented by means of two data structures, the *answer table* and the *dependency arc table* (which are in fact the result of the analysis algorithm).

- *Answer Table.* Its entries correspond to the *nodes* in the analysis graph. They are of the form $A : CP \mapsto AP$, where A is always an atom in base form. They should be interpreted as "the answer pattern for calls to A satisfying precondition (or call pattern), CP, accomplishes postcondition (or answer pattern), AP."
- *Dependency Arc Table.* The dependencies correspond to the *arcs* in the analysis graph. The intended meaning of a dependency $A_k : CP \Rightarrow B_{k,i} : CP_1$ associated to a program rule A_k:-$B_{k,1}, \ldots, B_{k,n}$ with $i \in \{1, ..n\}$ is that the answer for $A_k : CP$ depends on the answer for $B_{k,i} : CP_1$, say AP_1. Thus, if AP_1 changes with the update of some rule for $B_{k,i}$ then, the arc $A_k : CP \Rightarrow B_{k,i} : CP_1$ must be reprocessed in order to compute the new answer for $A_k : CP$. In particular, the rule for A_k has to be processed again starting from its body atom $B_{k,i}$.

All the details and the formalization of the algorithm can be found in [7]. Certification in ACC [3] consists in using the *complete* set of entries stored in the answer table as certificate. Dependencies are not needed for certificate generation neither for non incremental checking though they will be needed later for incremental certificate checking.

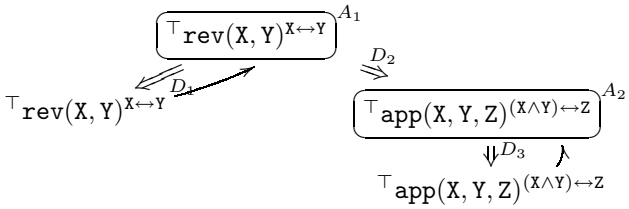
Definition 1 (certificate [3]). *Let $P \in Prog$, $D_\alpha \in ADom$ and $S_\alpha \in AAtom$. We define $\mathsf{Cert} \in ACert$, the* certificate *for P and S_α, as the set of entries stored in the answer table computed by* ANALYZE(S_α) *[7] for P in D_α.*

Example 1. The next example shows a piece of a module which contains the following (normalized) program for the naive reversal of a list and uses an implementation of `app` with several base cases (e.g., added automatically by a partial evaluator [8] for efficiency purposes).

($\mathtt{rev}_1$) $\mathtt{rev(X,Y)} :- \mathtt{X} = [\,],\ \mathtt{Y} = [\,].$
($\mathtt{rev}_2$) $\mathtt{rev(X,Y)} :- \mathtt{X} = [\mathtt{U}|\mathtt{V}],\ \mathtt{rev(V,W)},\ \mathtt{T} = [\mathtt{U}],\ \mathtt{app(W,T,Y)}.$
($\mathtt{app}_1$) $\mathtt{app(X,Y,Z)} :- \mathtt{X} = [\,],\ \mathtt{Y} = \mathtt{Z}.$
($\mathtt{app}_2$) $\mathtt{app(X,Y,Z)} :- \mathtt{X} = [\mathtt{U}],\ \mathtt{Z} = [\mathtt{U}|\mathtt{Y}].$
($\mathtt{app}_3$) $\mathtt{app(X,Y,Z)} :- \mathtt{X} = [\mathtt{U},\mathtt{V}],\ \mathtt{Z} = [\mathtt{U},\mathtt{V}|\mathtt{Y}].$
($\mathtt{app}_4$) $\mathtt{app(X,Y,Z)} :- \mathtt{X} = [\mathtt{U}|\mathtt{V}],\ \mathtt{Z} = [\mathtt{U}|\mathtt{W}],\ \mathtt{app(V,Y,W)}.$

The description domain that we use in our examples is the *definite Boolean functions* [4], denoted *Def*. The key idea in this description is to use implication to capture groundness dependencies. The reading of the function $x \rightarrow y$ is "if the program variable x is (becomes) ground, so is (does) program variable y." For example, the best description of the constraint $\mathtt{f(X,Y)} = \mathtt{f(a,g(U,V))}$ is $\mathtt{X} \wedge (\mathtt{Y} \leftrightarrow (\mathtt{U} \wedge \mathtt{V}))$. Groundness information is useful for many program optimizations and is also of great importance as a safety property, in order to verify that (C)LP programs are "well moded". The most general description $\top$ does not provide information about any variable. The least general substitution $\bot$ assigns the empty set of values to each variable.

For the analysis of our running example, we consider the calling pattern $\mathtt{rev(X,Y)} : \top$, i.e., no entry information is provided on `X` nor `Y`. The analysis algorithm of [7] computes the following analysis graph (we have notably simplified the graph by not showing the constraints in order to facilitate the understanding):



The nodes in the graph are labeled with their call pattern (left superscript) and answer pattern (right superscript). When a node is already in the graph, it is not expanded any further and the currently available answer is used. This is illustrated in the graph with backwards arrows. In order to compute A_1 and A_2, the analysis algorithm has to iterate twice over each of the dependencies D_1, D_2 and D_3 in the graph until the fixpoint is reached. Intuitively, D_2 denotes that the answer for $\mathtt{rev}(\mathtt{X},\mathtt{Y}) : \top$ may change if the answer for $\mathtt{app}(\mathtt{W},\mathtt{T},\mathtt{Y}) : \top$ changes. In such a case, the second rule for `rev` must be processed again starting from atom $\mathtt{app}(\mathtt{W},\mathtt{T},\mathtt{Y})$ in order to recompute the fixpoint for $\mathtt{rev}(\mathtt{X},\mathtt{Y}) : \top$. D_1 and D_3 reflect the recursivity of $\mathtt{rev}(\mathtt{X},\mathtt{Y}) : \top$ and $\mathtt{app}(\mathtt{W},\ \mathtt{T},\mathtt{Y}) : \top$, respectively, since they depend on themselves. The detailed steps performed by the algorithm can be found in [7] for the same program without the rules $\mathtt{app}_2$ and $\mathtt{app}_3$. However these rules do not add any further information to the fixpoint computation and the steps performed there still apply for our example.

(State 0) The above graph is returned by ANALYZE by means of the following answers (nodes in the graph) and dependencies (arrows in the graph):

(A_1) $\mathtt{rev}(\mathtt{X},\mathtt{Y}) : \top \mapsto \mathtt{X} \leftrightarrow \mathtt{Y}$
(A_2) $\mathtt{app}(\mathtt{X},\mathtt{Y},\mathtt{Z}) : \top \mapsto (\mathtt{X} \wedge \mathtt{Y}) \leftrightarrow \mathtt{Z}$

(D_1) $\mathtt{rev}(\mathtt{X},\mathtt{Y}) : \top \Rightarrow \mathtt{rev}(\mathtt{V},\mathtt{W}) : \top$
(D_2) $\mathtt{rev}(\mathtt{X},\mathtt{Y}) : \top \Rightarrow \mathtt{app}(\mathtt{W},\mathtt{T},\mathtt{Y}) : \top$
(D_3) $\mathtt{app}(\mathtt{X},\mathtt{Y},\mathtt{Z}) : \top \Rightarrow \mathtt{app}(\mathtt{V},\mathtt{Y},\mathtt{W}) : \top$

According to Definition 1, the certificate Cert for this example is composed of all entries in the answer table, i.e., A_1 and A_2.

5 Incremental Certificate and Incremental Certifier

When a program is updated, it appears inefficient to generate, transmit, and check the full certificate Ext_Cert for the updated program U_P defined as $U_P = P \oplus Upd(P)$. Our proposal is that it is possible to submit only the new program update $Upd(P)$ together with the *incremental certificate* Inc_Cert, i.e., the *difference* of Ext_Cert w.r.t. the original certificate, Cert. The bottom line is that the full fixpoint Ext_Cert differs from Cert in 1) the new fixpoint for each procedure *directly* affected by the updates which may be equal, more, less precise or incomparable depending on the kind of update (see Section 3), and 2) the update of other fixpoints possibly *indirectly* affected by the propagation of the effect of 1. However, there may be large parts of Cert which have not been affected by the changes and which do not need to be submitted nor checked again. In order to materialize this idea and achieve a compositional approach to incremental PCC, it is necessary that the consumer is able to reconstruct Ext_Cert. To do this, the consumer has to store Cert and then properly extend it with the upcoming incremental certificate. Some storage vs time trade-offs are discussed in Sect. 8.

Definition 2 (incremental certificate). *In the conditions of Def. 1, we consider* $Upd(P) \in UProg$*, the update of* P*. Let* Cert *be the certificate for* P *and* S_α*. Let* Ext_Cert *be the certificate for* $P \oplus Upd(P)$ *and* S_α*. We define* Inc_Cert*, the* incremental certificate *for* $Upd(P)$ *w.r.t.* Cert*, as the difference of certificates* Ext_Cert − Cert*.*

The difference *of two certificates,* Ext_Cert − Cert*, is defined as the set of entries* $B : CP_B \mapsto AP_B \in$ Ext_Cert *such that:*

1. $B : CP_B \mapsto \_ \notin$ Cert *or,*
2. $A : CP_A \mapsto AP_A \in$ Cert*,* $A : CP_A = B : CP_B$ *and* $AP_A \neq AP_B$ *(modulo renaming).*

Intuitively, Inc_Cert contains the subset of Ext_Cert which corresponds to the extensions and modifications w.r.t. Cert. The first obvious advantage of the incremental approach is that Inc_Cert can be much smaller than Ext_Cert. On the other, the following example illustrates that updating a program can require the change in the analysis information previously computed for other procedures whose fixpoint is indirectly affected by the updates, although their definitions have not been directly changed.

Example 2. Consider the following new definition for predicate app which is a specialization of the previous app to concatenate lists of a's of the same length :

($\texttt{Napp}_1$) $\texttt{app}(\texttt{X},\texttt{Y},\texttt{Z}) :- \texttt{X} = [\,], \texttt{Y} = [\,], \texttt{Z} = [\,].$
($\texttt{Napp}_2$) $\texttt{app}(\texttt{X},\texttt{Y},\texttt{Z}) :- \texttt{X} = [\texttt{a}|\texttt{V}], \texttt{Y} = [\texttt{a}|\texttt{U}], \texttt{Z} = [\texttt{a},\texttt{a}|\texttt{W}], \texttt{app}(\texttt{V},\texttt{U},\texttt{W}).$

The update consists in deleting all rules for predicate app in Example 1, and replacing them by $\texttt{Napp}_1$ and $\texttt{Napp}_2$. $Upd(P)$ is composed of the following information:

$Add(\texttt{app}(\texttt{X},\texttt{Y},\texttt{Z})) = \{\texttt{Napp}_1, \texttt{Napp}_2\}$
$Del(\texttt{app}(\texttt{X},\texttt{Y},\texttt{Z})) = \{\texttt{app}_1, \texttt{app}_2, \texttt{app}_3, \texttt{app}_4\}$

After running the (incremental) analysis algorithm in [7], the following answer table and dependencies are computed **(State 1)**:

(NA_1) $\texttt{rev}(\texttt{X},\texttt{Y}) : \top \mapsto \texttt{X} \wedge \texttt{Y}$
(NA_2) $\texttt{app}(\texttt{X},\texttt{Y},\texttt{Z}) : \top \mapsto \texttt{X} \wedge \texttt{Y} \wedge \texttt{Z}$
(NA_3) $\texttt{app}(\texttt{X},\texttt{Y},\texttt{Z}) : \texttt{X} \mapsto \texttt{X} \wedge \texttt{Y} \wedge \texttt{Z}$

(ND_1) $\texttt{rev}(\texttt{X},\texttt{Y}) : \top \Rightarrow \texttt{rev}(\texttt{V},\texttt{W}) : \top$
(ND_2) $\texttt{rev}(\texttt{X},\texttt{Y}) : \top \Rightarrow \texttt{app}(\texttt{W},\texttt{T},\texttt{Y}) : \texttt{W}$
(ND_3) $\texttt{app}(\texttt{X},\texttt{Y},\texttt{Z}) : \texttt{X} \Rightarrow \texttt{app}(\texttt{V},\texttt{U},\texttt{W}) : \texttt{V}$

Note that the analysis information has changed because the new definition of app allows inferring that all its arguments are ground upon success (NA_2 and NA_3). This change propagates to the answer of rev and allows inferring that, regardless of the calling pattern, both arguments of rev will be ground on the exit (NA_1).
According to Definition 2, the incremental certificate Inc_Cert contains NA_3 as it corresponds to a new calling pattern (by point 1) and contains also NA_1 and NA_2 since their answers have changed w.r.t. the ones in certificate of **State 0** (by point 2).

The next definition introduces the notion of incremental certifier which, given the original certificate and program and an update of the program, returns the incremental certificate iff the safety policy can still be entailed from the extended certificate.

Definition 3 (incremental certifier). *We define function* INC_CERTIFIER*:* $Prog \times UProg \times ADom \times AAtom \times AInt \times ACert \mapsto ACert$ *which takes* $P \in Prog$*,* $Upd(P) \in UProg$*,* $D_\alpha \in ADom$*,* $S_\alpha \in AAtom$*,* $I_\alpha \in AInt$*, the certificate* $\mathsf{Cert} \in ACert$ *for* P *and* S_α*. Let* $\mathsf{Ext\_Cert} \in ACert$ *be the full certificate for* $P \oplus Upd(P)$*. Then, it returns* $\mathsf{Inc\_Cert}$*, the* incremental certificate *for* $Upd(P)$ *w.r.t.* Cert*, iff* $\mathsf{Ext\_Cert} \sqsubseteq I_\alpha$*.*

Note that the above definition does not depend on the particular analysis algorithm used to generate Ext_Cert. Incremental analysis algorithms (like the ones in [16, 7, 13, 15]) are very well suited to do this task. They reanalyze only the part of the analysis graph affected by the increment. Thus, the time required to generate the certificate on the producer side can be reduced. Although this optimization on the producer side is always desirable, it is not as critical within the PCC scheme as the reduction of the package transmission time or the checking time, which take place on the consumer side and that we discuss in the next section.

Following the scheme in Figure 1, the producer has started by retrieving P and Cert from disk. In its last phase, if certification for the new updated program has succeeded, then P and Cert will be superseded in disk by the updated program and the extended certificate, respectively.

6 A Checking Algorithm with Support for Incrementallity

In this section, we present a checking algorithm for full certificates which is instrumented with a *Dependency Arc Table* (*DAT* in the following). The DAT stores the dependencies between the atoms in the analysis graph (see Section 4). This structure is not required by non incremental checkers [3] but it is fundamental to support an incremental design. Intuitively, our abstract interpretation-based checking algorithm receives a certificate Cert and constructs a program analysis graph in a single iteration by assuming the fixpoint information in Cert. The original dependencies are not required because they can be reconstructed from the program. While the graph is being constructed, the obtained answers are stored in AT_{mem} and compared with the corresponding fixpoints stored in Cert. If any of the computed answers is not consistent with the certificate (i.e., it is greater than the fixpoint), the certificate is considered invalid and the program is rejected. Otherwise, Cert gets checked. The checker returns the reconstructed answer table AT_{mem} and the set of dependencies DAT_{mem} which have been traversed.

Algorithm 2 presents our checker which is parametric w.r.t. the abstract domain of interest. It is hence defined in terms of five abstract operations on a selected domain D_α:

- $\mathsf{Arestrict}(CP, V)$ performs the abstract restriction of a description CP to the set of variables in the set V, denoted $vars(V)$;
- $\mathsf{Aextend}(CP, V)$ extends the description CP to the variables in the set V;
- $\mathsf{Aadd}(C, CP)$ performs the abstract operation of conjoining the constraint C with the description CP;
- $\mathsf{Aconj}(CP_1, CP_2)$ performs the abstract conjunction of two descriptions;
- $\mathsf{Alub}(CP_1, CP_2)$ performs the abstract disjunction of two descriptions.

Example 3. For the domain *Def* in Example 1, these abstract operations are defined as follows:

```
1: procedure checking(P, S, Cert, AT_mem, DAT_mem)
2:     AT_mem := ∅;  DAT_mem := ∅;  CP_checked := ∅;
3:     for all A : CP ∈ S do process_node(P, A : CP, Cert, AT_mem, DAT_mem, CP_checked);
4:     return Valid;
5: procedure process_node(P, A : CP, Cert, AT_mem, DAT_mem, CP_checked)
6:     if (∃ a renaming σ s.t. σ(A : CP ↦ AP) in Cert) then
7:         add A : CP ↦ σ^{-1}(AP) to AT_mem ;
8:         CP_checked := CP_checked ∪ {A : CP};
9:         Cert = Cert − {σ(A : CP ↦ AP)};
10:    else return Error;
11:    process_set_of_rules(P, P|_A, A : CP ↦ σ^{-1}(AP), Cert, AT_mem, DAT_mem, CP_checked);
12: procedure process_set_of_rules(P, R, A : CP ↦ AP, Cert, AT_mem, DAT_mem, CP_checked)
13:    for all rule A_k ← B_{k,1}, ..., B_{k,n_k} in R do
14:        W := vars(A_k, B_{k,1}, ..., B_{k,n_k});  CP_b := Aextend(CP, vars(B_{k,1}, ..., B_{k,n_k}));
15:        CPR_b := Arestrict(CP_b, B_{k,1});
16:        CP_a := process_rule(P, A : CP, A_k ← B_{k,1}, ..., B_{k,n_k}, W, CP_b, CPR_b, Cert, AT_mem, DAT_mem, CP_checked);
17:        AP_1 := Arestrict(CP_a, vars(A_k)) ;
18:        AP_2 := Alub(AP_1, AP);
19:        if (AP <> AP_2) then return Error;
20: procedure process_rule(P, A : CP, A_k ← B_{k,j}, ..., B_{k,n_k}, W, CP_b, CPR_b, Cert, AT_mem, DAT_mem, CP_checked)
21:    for all B_{k,i} in the rule body i = j, ..., n_k do
22:        CP_a := process_arc(P, A : CP, B_{k,i} : CPR_b, CP_b, W, Cert, AT_mem, DAT_mem, CP_checked);
23:        if (i <> n_k) then  CPR_a := Arestrict(CP_a, var(B_{k,i+1}));
24:        CP_b := CP_a;  CPR_b := CPR_a;
25:    return CP_a;
26: procedure process_arc(P, A : CP, B_{k,i} : CPR_b, CP_b, W, Cert, AT_mem, DAT_mem, CP_checked)
27:    if (B_{k,i} is a constraint) then  CP_a := Aadd(B_{k,i}, CP_b);
28:    else
29:        if (∄ a renaming σ s.t. σ(B_{k,i} : CPR_b ↦ AP') in AT_mem) then
30:            process_node (P, B_{k,i} : CPR_b, Cert, AT_mem, DAT_mem, CP_checked);
31:        AP_1 := Aextend (ρ^{-1}(AP), W); where ρ is a renaming s.t. ρ(B_{k,i} : CPR_b ↦ AP) in AT_mem
32:        CP_a := Aconj (CP_b, AP_1);
33:        add A : CP ⇒ B_{k,i} to DAT_mem;
34:    return CP_a;
```

Fig. 2. Checking Algorithm 2 with Support for Incrementallity

$\mathsf{Arestrict}(CP, V) = \exists_{-V} CP$ $\quad$ $\mathsf{Aconj}(CP_1, CP_2) = CP_1 \wedge CP_2$
$\mathsf{Alub}(CP_1, CP_2) = CP_1 \sqcup CP_2$ $\quad$ $\mathsf{Aextend}(CP, V) = CP$
$\mathsf{Aadd}(C, CP) = \alpha_{Def}(C) \wedge CP$
$\alpha_{Def}(X = t) = (X \leftrightarrow \bigwedge\{Y \in vars(t)\})$

where $\exists_{-V} F$ represents $\exists v_1, \ldots, v_n F$, where $\{v_1, \ldots, v_n\} = vars(F) - V$, and $\sqcup$ is the least upper bound (lub) operation over the *Def* lattice. For instance, $\mathsf{Aconj}(X, Y \leftrightarrow (X \wedge Z)) = X \wedge (Y \leftrightarrow Z)$. $\mathsf{Aadd}(X = [U|V], Y) = (X \leftrightarrow (U \wedge V)) \wedge Y$. $\mathsf{Alub}(X, Y) = X \vee Y$. □

Let us briefly explain the main functions of Algorithm 2.

1. Function checking(P,S,Cert,AT_{mem},DAT_{mem}), receives as parameters a program P, a set S of call patterns, the certificate Cert returned by ANALYZE, and two output variables AT_{mem} and DAT_{mem} which are initially empty (Line 2 of procedure checking). If it succeeds (L4), AT_{mem} coincides with Cert, and DAT_{mem} stores all dependencies generated during the checking process.

2. For each abstract atom $A : CP \in S$, checking calls process_node (L3). This corresponds to the creation of a node in the analysis graph. Firstly, process_node checks if there exists an answer for $A : CP$ in Cert and stores it in AT_{mem} (L6-7), removing it from Cert (L9). All calls to be checked must have its answer in Cert or the certificate is not valid and an error is issued (L10). It then proceeds to compute an answer for $A : CP$ by processing all rules (L11) defining A in a depth-first, left-to-right fashion.
3. This is done in procedure process_set_of_rules where the answers obtained for each rule in (L16) are lubbed (L18) with those stored in AT_{mem} (the fixpoint) to check that they are less or equal than the fixpoint. Otherwise an error is issued (L19).[2] As notation, given the set of rules (or arcs) R, $R|_A$ denotes the set of rules (or arcs) in R applicable to atom A.
4. Each particular rule is handled by procedure process_rule, which traverses the rule body (L21) and processes its corresponding atoms from left-to-right by using the answer obtained for $B_{k,i} : CPR_b$ as calling pattern for processing $B_{k,i+1}$ (L23). When all body atoms have been processed, the last answer is returned as final answer (L25).
5. Finally, procedure process_arc creates the dependencies which correspond to the arcs in the analysis graph. Constraints are simply added to the current answer and no dependency is created for them (L27). As for each literal, procedure process_arc looks up for an answer for it (L29) in AT_{mem} (i.e., the answer copied from Cert). This avoids rechecking the same literal several times and ensures termination. If it does not exist, then a (recursive) call to process_node (L30) computes a solution for the atom.

The main difference of the checking Algorithm 2 and the one in [3] is that our checker stores the dependencies in the analysis graph. This is done in procedure process_arc (L33). During analysis, dependencies are used for achieving efficient implementations of fixed-point re-computations. Naturally, they have not been used in non incremental checking algorithms since re-computation should never happen when one assumes the fixpoint. In contrast, DAT_{mem} will be fundamental for the design of our incremental checker, as we will see in the next section. We use the output parameter $CP_{checked}$ (L8) to store the calling patterns which have been already checked during the validation process. This will allow the incremental checking to avoid useless rechecking later. Also, we remove from the certificate the entries which are going to be checked (L9). In the incremental checking, this will be instrumental to detect the entries unrelated to the updates (see Section 7.1).

Example 4. As an example, we illustrate the steps carried out by the checker to validate the rules $\mathtt{app}_1$ and $\mathtt{app}_4$ of Example 1 w.r.t. a certificate Cert composed of the entry A_2. We take as call pattern $\mathtt{app}(\mathtt{X},\mathtt{Y},\mathtt{Z}) : \top$. Consider the call to procedure process_node for $\mathtt{app}(\mathtt{X},\mathtt{Y},\mathtt{Z}) : \top$. The entry A_2 is added (L7) to AT_{mem} (initially empty), and $\mathtt{app}(\mathtt{X},\mathtt{Y},\mathtt{Z}) : \top$ is marked as checked (by inserting it in $CP_{checked}$) and removed from Cert. A call to process_set_of_rules is generated for the call pattern at hand w.r.t. rules $\mathtt{app}_1$ and $\mathtt{app}_4$ (L11). Let us consider the processing of the two rules.

[2] This is the main difference with an analyzer. The latter needs to iterate if the new lub is different from the previously stored one (i.e., the fixpoint has not been reached). To do this iteration efficiently, dependencies detect the parts of the analysis graph which need to be reprocessed.

1. The call to process_rule for app_1 (L16) executes process_arc (L22) for each of the two constraints in the body. The final answer $CP_a \equiv \mathtt{X} \wedge (\mathtt{Y} \leftrightarrow \mathtt{Z})$ (L16) for app_1 is built up from the abstract conjunction (L32) between $\mathtt{X}$ (partial answer from first constraint) and $\mathtt{Y} \leftrightarrow \mathtt{Z}$ (from second constraint). Since the least upper bound (L18) between CP_a and the answer A_2 is A_2, then no Error is issued (L19) and the first rule app_1 gets successfully checked.
2. As before, the call to process_rule for app_4 executes process_arc for the first two constraints and computes as (partial) solution $CP_a \equiv (\mathtt{X} \leftrightarrow (\mathtt{U} \wedge \mathtt{V})) \wedge (\mathtt{Z} \leftrightarrow (\mathtt{U} \wedge \mathtt{W}))$ (L22). Since we are not in the last atom of the rule (L23), CP_a is restricted to the variables in $\mathsf{app}(\mathtt{V}, \mathtt{Y}, \mathtt{W})$, giving as result $CPR_a \equiv \top$. Now, the next call to process_arc for the rightmost body atom $\mathsf{app}(\mathtt{V}, \mathtt{Y}, \mathtt{W}) : \top$ computes as final solution $(\mathtt{X} \leftrightarrow (\mathtt{U} \wedge \mathtt{V})) \wedge (\mathtt{Z} \leftrightarrow (\mathtt{U} \wedge \mathtt{W})) \wedge (\mathtt{V} \wedge (\mathtt{Y} \leftrightarrow \mathtt{W}))$, which is simplified to A_2. The corresponding dependency is stored in the DAT. Thus, the call to process_rule for app_4 computes as solution A_2 (L16), the same answer stored in Cert, and no Error is issued (L19).

Therefore, both rules have been successfully checked in one pass over them and the checker returns VALID. □

In order to support Incrementallity, the final values of the data structures AT_{mem}, DAT_{mem} and P must be available after the end of the execution of the checker. Thus, we denote by $AT_{persist}$, $DAT_{persist}$ and $P_{persist}$ the copy in persistent memory (i.e., in disk) of such structures.

Definition 4 (checker). *We define function* CHECKER*:*$Prog \times ACert \times AAtom \times ADom \mapsto$ *boolean which takes a program* $P \in Prog$ *and its certificate* $\mathsf{Cert} \in ACert$ *for* $S_\alpha \in AAtom$ *in* $D_\alpha \in ADom$ *and*

1. *It returns the result of* $\mathsf{checking}(P, S_\alpha, \mathsf{Cert}, AT_{mem}, DAT_{mem})$.
2. *If it does not issue an* Error*, then it stores in memory* $AT_{persist} := AT_{mem}$, $DAT_{persist} := DAT_{mem}$ *and* $P_{persist} := P$.

7 Incremental Checking

In this section, we propose an incremental checking algorithm which deals with all possible updates over a program in a unified form. The basic idea is that the task performed by an incremental checker has to be optimized such that it only: a) rechecks the part of the abstraction for the procedures which have been directly affected by an update and, b) propagates and rechecks the indirect effect of these changes. In order to do this, we will take as starting point the checker in Algorithm 2. Its DAT will allow the incremental algorithm to propagate the changes and carry out the process in a single pass over the subgraph affected by the updates. Algorithm 3 presents our implementation of this intuition. The intuitive idea is that we start by removing all (possibly incorrect or inaccurate) information *directly* affected by the updates from the answer table and DAT (i.e., the information for the updated procedures) and, then, we check it from scratch against the answers provided in the incremental certificate. If the "direct" checking succeeds, we proceed to check the information *indirectly* affected by such changes in a similar way (i.e., delete the information for them from answer and DAT and recheck it from scratch). This iterative process successfully finishes when all direct and indirect affected information gets checked. Otherwise, an Error is issued.

```
1: procedure incremental_checking(P, Upd(P), Inc_Cert, AT_mem, DAT_mem)
2:     P_mem := P ⊕ Upd(P);
3:     update_answer_table(AT_mem, Inc_Cert);
4:     call_patterns_to_check(Upd(P), AT_mem, CP_tocheck);
5:     CP_checked := ∅;        % call patterns already checked
6:     check_affected_entries(P_mem, Inc_Cert, AT_mem, DAT_mem, CP_tocheck, CP_checked);
7:     unrelated_entries(P_mem, Inc_Cert, AT_mem, DAT_mem, CP_tocheck, CP_checked);
8:     return Valid;
9: procedure update_answer_table(AT_mem, Inc_Cert)
10:    for all entry A : CP ↦ AP in AT_mem do
11:        if (∃ A : CP ↦ AP_A in Inc_Cert and AP ≠ AP_A (modulo renaming)) then
12:            replace entry for A : CP ↦ AP in AT_mem by A : CP ↦ AP_A;
13: procedure call_patterns_to_check(Upd(P), AT_mem, CP_tocheck)
14:    CP_tocheck := ∅;        % call patterns required to be checked
15:    for all entry A : CP ↦ _ ∈ AT_mem do
16:        if A is updated in Upd(P) then CP_tocheck := CP_tocheck ∪ {A : CP};
17: procedure check_affected_entries(P_mem, Inc_Cert, AT_mem, DAT_mem, CP_tocheck, CP_checked)
18:    while CP_tocheck ! = ∅ do
19:        select A : CP from CP_tocheck;
20:        remove_previous_info(A : CP, AT_mem, DAT_mem);
21:        if A : CP ∉ Inc_Cert then
22:            let A : CP ↦ AP the entry for A : CP in AT_mem;
23:            Inc_Cert = Inc_Cert ∪ {A : CP ↦ AP};
24:            propagate := false;
25:        else propagate := true;
26:        process_node(P_mem, A : CP, Inc_Cert, AT_mem, DAT_mem, CP_checked);
27:        CP_tocheck := CP_tocheck − CP_checked;
28:        if propagate then propagate_effects(A : CP, DAT_mem, CP_tocheck, CP_checked);
29: procedure remove_previous_info(A : CP, AT_mem, DAT_mem)
30:    remove entry for A : CP from AT_mem;
31:    remove from DAT_mem all dependencies of the form A : CP ⇒ _ ;
32: procedure propagate_effects(A : CP, DAT_mem, CP_tocheck, CP_checked)
33:    for all B : CP_B ⇒ A : CP ∈ DAT_mem do
34:        if B : CP_B ∉ CP_checked ∪ CP_tocheck then CP_tocheck := CP_tocheck ∪ {B : CP_B};
```

Fig. 3. Incremental Checking Algorithm 3

The incremental checker is defined as follows: replace the procedure checking by the new procedure incremental_checking in Algorithm 3 and use the remaining procedures defined in Algorithm 2. Essentially, the additional tasks that an incremental checker has to perform w.r.t. the non incremental one in Algorithm 2 are the following:

1. *Retrieve stored data.* After checking the original package, the structures $AT_{persist}$, $DAT_{persist}$ and the program $P_{persist}$ have been stored in persistent memory (see Definition 4). Our checker retrieves such stored data and initializes, respectively, the parameters AT_{mem}, DAT_{mem} and P with them.
2. *Update program and answer table.* Prior to proceeding with the proper checking, the incoming updates $Upd(P)$ are applied (by means of the operator $\oplus$) to P in order to generate P_{mem} (L2). Also, the procedure update_answer_table updates the answer table by updating the answers for those call patterns in AT_{mem} which have a different answer in Inc_Cert (L10-12). The new entries not yet present in AT_{mem} will be asserted in it upon request, as it is done in the usual checking process (L7 of Algorithm 2).
3. *Initialize call patterns to check.* The procedure call_patterns_to_check initializes the set $CP_{tocheck}$ with those call patterns with an entry in AT_{mem} which corre-

spond to a rule directly affected by an update (L15-16). During the execution of the checker, the set $CP_{tocheck}$ will be dynamically extended to include the additional call patterns whose checking is indirectly affected by the propagation of changes (L34).

4. *Check affected procedures.* Procedure check_affected_entries is encharged of launching the checking of all procedures affected by the updates, i.e., the call patterns in $CP_{tocheck} - CP_{checked}$. As mentioned in Section 4, the set $CP_{checked}$ is used to avoid rechecking the same call pattern more than once, if it appears several times in the analysis subgraph to be checked. Three actions are taken in order to check a call pattern:[3]
   - remove its analysis information (L20),
   - proceed to check it by calling process_node of Algorithm 2 (L26) and,
   - propagate the effects of type b) if needed (L28).

   We only propagate effects if the answer provided in Inc_Cert for the call pattern at hand is different from that originally stored in $AT_{persist}$ (L25). As a technical detail, in L23, we add to Inc_Cert the information which, although has not changed w.r.t. AT_{mem}, needs to be checked and, therefore, it must be available in Inc_Cert (or process_node would issue an error in L10 of Algorithm 2).
5. *Remove previous analysis information.* Before proceeding with the checking, we need to get rid of previous (possibly incorrect or inaccurate) analysis information. Procedure remove_previous_info eliminates the entry to be checked from AT_{mem} (L30) and all its dependencies from DAT_{mem} (L31).
6. *Propagate effects.* After processing the updated rules, the procedure propagate_effects introduces in the set $CP_{tocheck}$ (L34) the calling patterns whose answer depends on the updated one, i.e., those which are indirectly affected by the updates. Their checking will be later required in L18.
7. *Store data.* Upon return, the checker has to store the computed AT_{mem}, DAT_{mem} and P_{mem}, respectively, in $AT_{persist}$, $DAT_{persist}$, and $P_{persist}$ for achieving a compositional design of our incremental approach.

Definition 5 (incremental checker). *We define function* INCR_CHECKER*:* $UProg \times ACert \times \mapsto$ *boolean which takes* $Upd(P) \in UProg$ *and its incremental certificate* Inc_Cert $\in ACert$ *and*

1. *It retrieves from memory* $AT_{mem} := AT_{persist}$, $DAT_{mem} := DAT_{persist}$ *and* $P := P_{persist}$.
2. *It returns the result of* incremental_checking(P, $Upd(P)$, Inc_Cert, AT_{mem}, DAT_{mem}) *for* P.
3. *If it does not issue an* Error*, then it stores* $AT_{persist} := AT_{mem}$, $DAT_{persist} := DAT_{mem}$ *and* $P_{persist} := P_{mem}$.

An important point to note is that the safety policy has to be tested w.r.t. the answer table for the extended program. Therefore, the checker has reconstructed, from Inc_Cert, the answer table returned by ANALYZE for the extended program, Ext_Cert, in order to test for adherence to the safety policy –Equation (4), i.e., $AT_{persist} \equiv$ Ext_Cert.

It should be noted that the design of our incremental checking algorithm is notably different from the design of an incremental analyzer (like the ones in [7, 13]). In

[3] Note that an updated rule which does not match any entry in AT_{mem} does not need to be processed by now. Its processing may be required by some other new rule or they can simply not be affected by the checking process.

particular, the treatment of deletions and arbitrary changes is completely different. In our case, we can take advantage of the information provided in the certificate in order to avoid the need to compute the strongly connected components (see [7]). This was necessary in the analyzer in order to ensure the correctness of the incremental algorithm. We additionally have had to include the detection of no valid certificates. Unlike [7, 13], we have integrated in a single algorithm all incremental updates over a program in a seamless way. In Section 7.3, we will identify the particular optimizations of our unified algorithm for certain types of updates.

Our first example is intended to illustrate a situation in which the task performed by the incremental checker is optimized such that it only checks a part of the analysis graph.

Example 5. Consider the deletion of rules $\mathtt{app}_2$ and $\mathtt{app}_3$ of Example 1. The analysis algorithm of [7] returns the same state **(State 0)** since the eliminated rules do not affect the fixpoint result, i.e., they do not add any further information. Thus, the incremental certificate Inc_Cert associated to such an update is empty. The checking algorithm proceeds as follows. Initially, AT_{mem} and DAT_{mem} are initialized with the values in **State 0**. P_{mem} is composed of the rules $\mathtt{rev}_1$, $\mathtt{rev}_2$, $\mathtt{app}_1$ and $\mathtt{app}_4$. Procedure update_answer_table (L3) does not modify AT_{mem}. The execution of procedure call_patterns_to_check (L4) adds $E_1 \equiv \mathsf{app}(\mathtt{X},\mathtt{Y},\mathtt{Z}) : \top$ to $CP_{tocheck}$. Procedure check_affected_entries selects E_1 from $CP_{tocheck}$. The next call to remove_previous_info (L20) removes A_2 from AT_{mem} and D_3 from DAT_{mem}. It then inserts A_2 in Inc_Cert. The variable "*propagate*" takes the value *false*. We now jump to the non incremental checking with a call to procedure process_node (L26). This process corresponds exactly to the checking illustrated in Example 4. Upon return from process_node, since the variable "*propagate*" is *false*, no effects have to be propagated.

The important point to note is that the incremental checker has not had to recheck the rules for `rev` since its answer is not affected by the deletion. Once Inc_Cert has been validated, the consumer mems the answer table AT_{mem}, the dependency arc table DAT_{mem} (which are those of **State 0**) and the program P_{mem} in disk. □

Our second example is intended to show how to propagate the effect of a change to the part of the analysis graph affected by such update.

Example 6. Let us illustrate the checking process carried out to validate the update proposed in Example 2 with an incremental certificate, Inc_Cert, which contains the entries NA_1, NA_2 and NA_3. The incremental checker retrieves **State 0** from disk. Next, procedure update_answer_table returns as new AT_{mem} the entries NA_1 and NA_2 which replace the old entries A_1 and A_2, respectively. Then, the set $CP_{tocheck}$ is initialized with $E_1 \equiv \mathsf{app}(\mathtt{X},\mathtt{Y},\mathtt{Z}) : \top$. Procedure check_affected_entries first executes remove_previous_info, which eliminates E_1 from AT_{mem} and dependency D_3 from DAT_{mem}. Moreover, the variable "*propagate*" is initialized to *true*. This annotates that effects have to be propagated later. The execution of process_node for E_1 succeeds and adds the dependency D_3 to DAT_{mem} and the set $CP_{checked}$ is returned with E_1 marked as checked. Upon return, since the variable "*propagate*" is *true*, a call to propagate_effects is generated which forces the checking of `rev`. After inspecting D_2 and D_3 (the two dependencies for E_1), only the entry $E_2 \equiv \mathtt{rev}(\mathtt{X},\mathtt{Y}) : \top$ is added to $CP_{tocheck}$. The dependency for D_3 will not be checked because E_1 has been already processed (hence, it belongs to $CP_{checked}$). Now, procedure check_affected_entries takes E_2 from $CP_{tocheck}$, and similarly to the previous case, successfully executes process_node, and replaces D_2 by ND_2. During the checking of rule $\mathtt{rev}_2$, a new call to process_node is generated for

$E_3 \equiv \mathsf{app}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}) : \mathtt{X}$ which introduces E_3 in $CP_{checked}$, and replaces the dependency D_3 in DAT_{mem} by the new one ND_3 of Example 2. Upon return, since the variable "*propagate*" is *true*, a call to propagate_effects is generated from it. But the affected dependency D_1 is not processed because E_1 was processed already and belongs to $CP_{checked}$.

The conclusion is that a single pass has been performed on the three provided entries in order to validate the certificate. □

The correctness of the checking algorithm amounts to saying that, if our checker does not issue an error, then it returns as computed answer table the extended certificate for the updated program. Moreover, we ensure that it does not iterate during the reconstruction of any answer.

Theorem 1 (correctness). *Let $P \in Prog$, $Upd(P) \in UProg$, $D_\alpha \in ADom$ and $S_\alpha \in AAtom$. Consider:*

- Cert*: the certificate for P and S_α.*
- Ext_Cert*: the certificate for $P \oplus Upd(P)$ and S_α.*
- Inc_Cert*: the incremental certificate for $Upd(P)$ w.r.t.* Cert.

If INCR_CHECKER*($Upd(P)$,* Inc_Cert*) does not issue an* Error*, then the validation of* Inc_Cert *is done in a single pass over* Inc_Cert *and:*

- $AT_{persist} \equiv AT_{mem}$
- $DAT_{persist} \equiv DAT_{mem}$

where AT_{mem} and DAT_{mem} are, respectively, the answer table and DAT returned by checking*($P \oplus Upd(p)$, S_α,* Ext_Cert*, AT_{mem}, DAT_{mem}).*

The proof of this theorem can be found in the appendix.

7.1 Entries unrelated to updates

The incremental setting has to take into account a new type of Error which does not occur in non incremental PCC. This corresponds to a situation in which the producer includes in the incremental certificate certain entries for calling patterns which are *unrelated* to the updates, i.e., they do not correspond to the updated procedures nor to the propagation of their effects. However, they are related with procedures stored in $P_{persist}$ in some previous iteration. These entries may have been included in the certificate by the producer in order to provide a new, more precise fixpoint for some procedure which, for instance, has been obtained by using a higher quality analyzer. In principle, this information may seem not useful as the previous fixpoint in $AT_{persist}$ was already valid to entail the safety policy. Nevertheless, it could be the case that the safety policy is later changed and one needs the more precise information received in this incremental certificate in order to imply such policy. Also, this accuracy could be required in a later modification.

Since the procedures of Algorithm 3 described so far only check the new updates and their demanded entries, we check in a posterior phase that the entries unrelated to the updates are valid fixpoints or, otherwise, issue an error. This is done by the following procedure unrelated_entries which is executed from L7 in Algorithm 3.

procedure unrelated_entries(P_{mem}, Inc_Cert,AT_{mem},DAT_{mem},$CP_{tocheck}$,$CP_{checked}$)
  $CP_{tocheck} := \emptyset$;

for all $A : CP \in$ Inc_Cert **do**
$CP_{tocheck} := CP_{tocheck} \cup \{A : CP\}$;
check_affected_entries (P_{mem},Inc_Cert, AT_{mem}, DAT_{mem},$CP_{tocheck}$,$CP_{checked}$)

Note that we can easily identify the entries unrelated to the updates because they are the only ones which remain in Inc_Cert after having checked the affected procedures. This happens because in L9 of Algorithm 2 we remove from Inc_Cert those entries which are being checked.

By checking unrelated answers as well, we can now ensure that any invalid fixpoint will be rejected by the consumer. The proof can be found in [1].

Theorem 2. *In the conditions of Theorem 1, if an answer in* Inc_Cert *is not a* VALID *fixpoint, then* INCR_CHECKER*(Upd(*P*),* Inc_Cert*) issues an* Error.

It should be noted that in order to ensure that all valid certificates get checked with the incremental checker, we have to provide all the information required by the checker within the incremental certificate. This can be ensured by using an incremental analyzer which uses the same graph traversal strategy than the checker in order to generate the incremental certificates. More details can be found in the appendix.

7.2 Cleaning up the Certificate

Those entries for $A : CP$ in AT_{mem} (and which are not in S) corresponding to deleted rules can be removed from the final answer table. We can identify them by exploring the new dependencies. If there is no entry which depends on it, then we can remove such entry. This is a rather standard procedure performed in analysis algorithms to eliminate useless entries.

procedure remove_useless_entries(AT_{mem},DAT_{mem})
for all $A : CP \in AT_{mem}$ and $A : CP \notin S$ **do**
if $\nexists B : CP_B \Rightarrow A : CP \in DAT_{mem}$ **then**
remove $A : CP$ from AT_{mem}

As an example, consider the following program and $S = \{\mathtt{p(X,Y)} : \top\}$:

$$\begin{array}{l} \mathtt{p(X,Y)} \text{ :- } \mathtt{q(X)},\ \mathtt{h(Y)}. \\ \mathtt{q(X)} \text{ :- } \mathtt{X} = \mathtt{a}. \\ \mathtt{h(Y)} \text{ :- } \mathtt{Y} = \mathtt{b}. \end{array}$$

By using the analysis algorithm in [7], we have that Cert is made up of

$$\begin{array}{l} \{\ \mathtt{p(X,Y)} : \top \mapsto \mathtt{X} \wedge \mathtt{Y}, \\ \ \ \mathtt{q(X)} : \top \mapsto \mathtt{X}, \\ \ \ \mathtt{h(Y)} : \top \mapsto \mathtt{Y} \qquad\quad \} \end{array}$$

The CHECKER stores in disk memory $AT_{persist} \equiv$ Cert and $DAT_{persist}$ is:

$$\begin{array}{l} \{\ \mathtt{p(X,Y)} : \top \Rightarrow \mathtt{q(X)} : \top \\ \ \ \mathtt{p(X,Y)} : \top \Rightarrow \mathtt{h(Y)} : \top\ \} \end{array}$$

Suppose that the unique rule for `h` is deleted and the unique rule for `p` is replaced by the new one:

$$\mathtt{p(X,Y)} \text{ :- } \mathtt{q(X)}.$$

The producer sends to the consumer such an update along with Inc_Cert=$\{\mathtt{p(X,Y)} : \top \mapsto \mathtt{X}\}$. Then INCR_CHECKER, starting from $AT_{persist}$ and $DAT_{persist}$, rechecks

$\mathtt{p}(\mathtt{X},\mathtt{Y}) : \top$, removing its old answer from AT_{mem} and its dependencies from DAT_{mem}. The checking process introduces in AT_{mem} the new answer for $\mathtt{p}$, and in DAT_{mem} only the dependency related to $\mathtt{q}$. Now, procedure remove_useless_entries compares AT_{mem} with the new computed dependency. Since $\mathtt{h}$ does not belong to S and does not occur in the right hand side of the dependency, then the entry $\mathtt{h}(\mathtt{Y}) : \top \mapsto \mathtt{Y}$ is removed from AT_{mem}.

The purpose of the above procedure is clearly to reduce the size of the persistent certificate. Its execution can be switched off when the higher priority is to reduce the checking time w.r.t. optimize storage resources.

7.3 Optimizations to the generic algorithm

Our generic incremental checking algorithm admits an optimization which avoids rechecking some useless call patterns in certain cases. When an entry $A : CP \mapsto AP$ in the new certificate is affected by the update of $B : CP_B$, the call pattern $A : CP$ is fully checked with a call to process_node which traverses all rules for A from their leftmost positions. This is done because if one considers any possible update, the old answer for $A : CP \mapsto AP'$ may be inaccurate in the case of deletion or arbitrary change when $AP' \not\sqsubseteq AP$. Therefore, we cannot assume AP and the checking for $A : CP$ has to delete all its previous information and start from scratch.

However, when we know that the old answer AP' is lesser than AP, i.e., $AP' \sqsubseteq AP$, we can process only the part of the subgraph associated to checking $A : CP$ which depends on the update of $B : CP_B$. This always happens in the addition of predicates and it can eventually happen in the deletion and arbitrary changes (see Section 3). We can easily find out this point in the algorithm through the dependency $A_k : CP \Rightarrow B_{k,i} : CP_B$. In particular, we will replace in such a case the call to process_node for $A : CP$ by a call to process_rule for the rule for A_k starting from its body atom $B_{k,i}$. As an example, consider the program given in Example 5, where Inc_Cert is empty because, after updating the program, all entries have remained the same. In such a case, the only call pattern $\mathtt{app}(\mathtt{X},\mathtt{Y},\mathtt{Z}) : \top$ affected by the update does not need to be checked, since its answer is still correctly described by the remaining rules.

The implementation of this idea in the algorithm, although conceptually clear, gets technically more involved due to the need of ensuring that the remaining arcs from A_k to its body atoms to the right of $B_{k,i}$ are checked only once. For the case of the addition of rules only, an optimized incremental checker can be found in [2]. For an arbitrary update, we do not integrate the same optimization here by lack of space (but can be found in our technical report [1]).

8 Conclusions

Our approach to incremental ACC aims at reducing the size of certificates and the checking time when a supplier provides an untrusted update of a (previously) validated package. Essentially, when a program is subject to an update, the incremental certificate we propose contains only the *difference* between the original certificate for the initial program and the new certificate for the updated one. Checking time is reduced by traversing only those parts of the abstraction which are affected by the changes rather than the whole abstraction. An important point to note is that our incremental approach requires the original certificate and the dependency arc table to be stored by the consumer side for upcoming updates. The appropriateness

of using the incremental approach will therefore depend on the particular features of the consumer system and the frequency of software updates. In general, our approach seems to be more suitable when the consumer prefers to minimize as much as possible the waiting time for receiving and validating the certificate while storage requirements are not scarce. We believe that, in everyday practice, time-consuming safety tests would be avoided by many users, while they would probably accept to store the safety certificate and dependencies associated to the package. Nevertheless, there can sometimes be situations where storage resources can be very limited, while runtime resources for performing upcoming checkings could still be sufficient. We are now in the process of extending the ACC implementation already available in the `CiaoPP` system to support incrementallity. Our preliminary results in certificate reduction are very promising. We expect optimizations in the checking time similar to those achieved in the case of incremental analysis (see, e.g., [7]).

References

1. E. Albert, P. Arenas, and G. Puebla. An Incremental Approach to Abstraction-Carrying Code. Technical Report CLIP3/2006, Technical University of Madrid (UPM), School of Computer Science, UPM, March 2006.
2. E. Albert, P. Arenas, and G. Puebla. Incremental Certificates and Checkers for Abstraction-Carrying Code. In *Sixth Workshop on Issues in the Theory of Security*, March 2006.
3. E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, number 3452 in LNAI, pages 380–397. Springer-Verlag, 2005.
4. T. Armstrong, K. Marriott, P. Schachte, and g H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In Springer-Verlag, editor, *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 266–280, Namur, Belgium, September 1994.
5. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
6. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
7. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.
8. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
9. Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.
10. J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.
11. Kim Marriot and Peter Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.
12. G. Necula. Proof-Carrying Code. In *Proc. of POPL'97*, pages 106–119. ACM Press, 1997.
13. G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *Proc. of SAS'96*, pages 270–284. Springer LNCS 1145, 1996.
14. K. Rose, E. Rose. Lightweight bytecode verification. In *OOPSLA Workshop on Formal Underpinnings of Java*, 1998.
15. B. Ryder. Incremental data-flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, 1988.
16. Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–43, 1997.

A Correctness of the Incremental Checking

Definition 6. *Let $P \in Prog$, $Upd(P) \in UProg$, $D_\alpha \in ADom$ and $S_\alpha \in AAtom$. Consider:*

- Cert*: the certificate for P and S_α.*
- Ext_Cert*: the certificate for $P \oplus Upd(P)$ and S_α.*
- Inc_Cert*: the incremental certificate for $Upd(P)$ w.r.t.* Cert.

We define the set of unchanged *entries in* Cert *as:*

$$unchanged(A : CP) = \{A : CP \in \mathsf{Cert} \mid A : CP \in \mathsf{Ext\_Cert}, A : CP \notin \mathsf{Inc\_Cert}\}$$

Note that $\mathsf{Cert} \not\subseteq \mathsf{Ext\_Cert}$ as shown in Section 7.1. Clearly, $unchanged(\mathsf{Cert})$ is the set of entries in Cert which do not change their answers after an update of the original program.

Definition 7. *Let $P \in Prog$, $Upd(P) \in UProg$, $D_\alpha \in ADom$ and $S_\alpha \in AAtom$. Consider:*

- Cert*: the certificate for P and S_α.*
- Ext_Cert*: the certificate for $P \oplus Upd(P)$ and S_α.*
- Inc_Cert*: the incremental certificate for $Upd(P)$ w.r.t.* Cert.

Let DAT_{mem} the dependency arc table computed by the call checking*($P \oplus Upd(P)$,S,* Ext_Cert*, AT_{mem},DAT_{mem}). We say that an entry $A : CP \in$* Ext_Cert *is* safe *if:*

- $A : CP \in$ Cert.
- $A : CP \notin$ Inc_Cert.
- *A has not an entry in $Upd(P)$.*
- *For all dependency $A : CP \Rightarrow B : CP_B \in DAT_{mem}$, $B : CP_B \in unchanged($*Cert*$)$.*

The following proposition ensures that the checking process for P and Cert behaves similarly to that for $P \oplus Upd(P)$ and Ext_Cert, for those call patterns being safe.

Proposition 1. *Let $P \in Prog$, $Upd(P) \in UProg$, $D_\alpha \in ADom$ and $S_\alpha \in AAtom$. Consider:*

- Cert*: the certificate for P and S_α.*
- Ext_Cert*: the certificate for $P \oplus Upd(P)$ and S_α.*
- Inc_Cert*: the incremental certificate for $Upd(P)$ w.r.t.* Cert.

Consider the calls:

(a) checking*(P,S,*Cert*,AT_{mem},DAT_{mem})*
(b) checking*($P \oplus Upd(P)$,S,*Ext_Cert*,AT'_{mem},DAT'_{mem})*

Then, for all $A : CP \in$ Ext_Cert *being safe, an for all rule $r \equiv A$:- $B_1, \ldots, B_n \in P$ for A, the calls:*

(a') process_rule*(P,$A : CP$,r,W,CP_b, CPR_b,*Cert*,AT_{mem},DAT_{mem},$CP_{checked}$)*
(b') process_rule*($P \oplus Upd(P)$,r,W,CP_b, CPR_b,*Ext_Cert*,AT'_{mem},DAT'_{mem},$CP_{checked}$')*

where $W = vars(r)$, $CP_b = \mathsf{Aextend}(CP, vars(B_1, \ldots, B_n))$, $CPR_b = \mathsf{Arestrict}(CP_b, B_1)$, *compute the same answer* CP_a, *and introduces the same dependencies for* A *in* DAT_{mem} *and* $,DAT'_{mem}$ *respectively.*

Proof. Let $r \equiv A$:- $B_1, \ldots, B_n \in P$ a rule for A. Calls (a') and (b') require n calls to procedure process_arc (L22 of Algorithm 2) in order to compute the final solution. Let us prove that, for all $1 \leq k \leq n$, the k-th call to process_arc in (a') and (b') is done with the same parameters

$$(*) \quad B_k : CPR_k, \ CP_k$$

and returns the same solution CP_a, adding the same dependencies for A in DAT_{mem} and DAT'_{mem}. We reason by induction on k.

Base case ($k = 1$). Trivially, both calls verify (*). We distinguish two cases:

1. B_1 is a constraint. Then, both calls to process_arc computes the same solution (L27).
2. B_1 is an atom. Since $A : CP$ is safe, then $B_1 : CPR_1 \in unchanged(\mathsf{Cert})$. Hence, by Definition 6, $B_1 : CPR_1$ has the same solution in Cert and Ext_Cert. But, the calls (a) and (b) do not issue an Error, then both of them store $B_1 : CPR_1 \mapsto AP_1$ in AT_{mem} and AT'_{mem} respectively, after calling to process_node for $B_1 : CPR_1$. Hence both calls to process_arc compute the same solution AP_1 and store the same dependency in DAT_{mem} and DAT'_{mem} respectively (L33).

Inductive case ($1 \leq k < n$). The inductive hypothesis is that the result holds for the first $1 \leq k < n$ calls to function process_arc. Hence, the $k + 1$ call to process_arc verifies trivially (*). Reasoning similarly to the base case, and considering that $B_{k+1} : CPR_{k+1} \in unchanged(\mathsf{Cert})$, it holds that both calls to process_arc computes the same answer and store the same dependencies for A.

Hence, since all calls to process_arc for (a') and (b') computes the same solution and store the same dependencies for A, then, the calls (a) and (b) also compute the same solution and store the same dependencies for A. □

Note that the above proposition ensures that the calls (a) and (b) behave similarly in presence of safe call patterns. Thus, safe call patterns does not require to be rechecked when a program is updated.

In the following, we will call *original checking* to the checking process executed for P w.r.t. Cert, and *full checking* to the checking process executed for $P \oplus Upd(P)$ w.r.t. Ext_Cert. We will denote by AT^o_{mem} and DAT^o_{mem} to the answer table and the dependency arc table computed by the original checking, and by AT^f_{mem} and DAT^f_{mem} to the corresponding ones computed by the full checking.

Proposition 2. *Let* $P \in Prog$, $Upd(P) \in UProg$, $D_\alpha \in ADom$ *and* $S_\alpha \in AAtom$. *Consider:*

- Cert*: the certificate for* P *and* S_α.
- Ext_Cert*: the certificate for* $P \oplus Upd(P)$ *and* S_α.
- Inc_Cert*: the incremental certificate for* $Upd(P)$ *w.r.t.* Cert.

If incremental_checking*(*P*,*$Upd(P)$*,*Inc_Cert*,* AT_{mem}*,* DAT_{mem}*), where* $AT_{mem} =$ Cert*, and* $DAT_{mem} = DAT^o_{mem}$*, does not issue an* Error *then, for all non safe* $A_0 : CP_0 \in$ Ext_Cert*, it holds that* $A_0 : CP_0 \in CP_{checked}$.

Proof. By Definition 7, if $A_0 : CP_0$ is not safe, then $A_0 : CP_0$ verifies at least one of the following conditions:

(A) $A_0 : CP_0 \notin \mathsf{Cert}$.
(B) $A_0 : CP_0 \in \mathsf{Cert} \cap \mathsf{Inc\_Cert}$.
(C) A_0 has an entry in $Upd(p)$.
(D) There exists a dependency $A_0 : CP_0 \Rightarrow B : CP_B$ in the dependency arc table computed by checking for $P \oplus Upd(P)$ and Ext_Cert (full checking), such that $B : CP_B \notin unchanged(\mathsf{Cert})$.

Let us analyze all above cases:

• **(Case B)** If A_0 has an entry $Upd(P)$ then, since $A_0 : CP_0 \in AT_{mem}$, then $A_0 : CP_0$ was initially introduced in $CP_{tocheck}$ (L16 in Algorithm 3) by procedure calls_patterns_to_check. Hence, a call to process_node (L**??**) was generated, introducing $A_0 : CP_0$ in $CP_{checked}$ (L8 in Algorithm 2).

Suppose that A_0 has no entry in $Upd(P)$. Since $A_0 : CP_0 \in \mathsf{Cert} \cap \mathsf{Inc\_Cert}$, then there exists at least a rule $r \in P$ for A_0 such that, the calls to process_rule for r in the original checking and in the full checking, computed a different answer. Furthermore, at least one of such rules r_0 verifies that there exists at least an atom B_{j_0} in r_0 such that:

- The original and full checking process generated a call to process_arc for $B_{j_0} : CPR_{j_0}$ and they computed different answers.
- $B_{j_0} : CPR_{j_0} \in \mathsf{Cert} \cap \mathsf{Inc\_Cert}$.
- $B_{j_0} : CPR_{j_0} \neq A_0 : CP_0$.

Note that if r_0 does not exists, then $A_0 : CP_0$ only could change its answer (from the original to the full checking) because of a modification of its rules, but we are assuming that A_0 has no entry in $Upd(P)$. Now, We distinguish two cases:

- If B_{j_0} has an entry in $Upd(P)$ then, since $B_{j_0} \in AT_{mem}$, it holds that $B_{j_0} : CPR_{j_0}$ was initially introduced in $CP_{tocheck}$ (L16 in Algorithm 3) by procedure calls_patterns_to_check. But $B_{j_0} : CPR_{j_0} \in \mathsf{Inc\_Cert}$, then procedure check_affected_entries (L25) initializes variable *propagate* to *true*, calling procedure propagate_effects (L28) after calling to process_node (L26) for $B_{j_0} : CPR_{j_0}$. But $A_0 : CP_0 \in \mathsf{Cert} \cap \mathsf{Ext\_Cert}$ and A_0 has no entry in $Upd(P)$, then the dependency $A_0 : CP_0 \Rightarrow B_{j_0} : CPR_{j_0} \in DAT_{mem}$. Hence, $A_0 : CP_0 \in CP_{checked}$(L34).
- If B_{j_0} has not an entry in $Upd(P)$, then reasoning similarly, there exists a rule $r_1 \in P$ and an atom B_{j_1} in r_1 such that, the original and full checking generated a call to process_arc for $B_{j_1} : CPR_{j_1}$ but computed different answers. On the other hand, $B_{j_1} : CPR_{j_1} \in \mathsf{Cert} \cap \mathsf{Inc\_Cert}$ and it is different from $A_0 : CP_0$ and $B_{j_0} : CPR_{j_0}$.
  - If B_{j_1} has an entry in $Upd(P)$, then $B_{j_1} : CPR_{j_1}$ was initially introduced in $CP_{tocheck}$. But $B_{j_0} : CPR_{j_0} \Rightarrow B_{j_1} : CPR_{j_1} \in DAT_{mem}$. Hence, as in the above case, propagate_effects introduces $B_{j_0} : CPR_{j_0}$ in $CP_{tocheck}$ (unless $B_{j_0} : CPR_{j_0} \in CP_{checked}$). But then, check_affected_entries process $B_{j_0} : CPR_{j_0}$ and propagates its effects. Since the dependency $A_0 : CP_0 \Rightarrow B_{j_0} : CPR_{j_0} \in DAT_{mem}$, then, again progagate_effects introduces $A_0 : CP_0$ in $CP_{tocheck}$ (unless $A_0 : CP_0 \in CP_{checked}$). Thus, finally $A_0 : CP_0 \in CP_{checked}$.

- If B_{j_1} has no an entry $Upd(P)$, then reasoning similarly and considering that the process in finite, we obtain a sequence of arcs:

$$\begin{array}{lll} A_0 : CP_0 & \Rightarrow & B_{j_0} : CPR_{j_0} \\ B_{j_0} : CPR_{j_0} & \Rightarrow & B_{j_1} : CPR_{j_1} \\ B_{j_1} : CPR_{j_1} & \Rightarrow & B_{j_2} : CPR_{j_2} \\ \ldots & & \ldots\ldots \\ B_{j_k} : CPR_{j_k} & \Rightarrow & B_{j_{k+1}} : CPR_{j_{k+1}} \end{array}$$

  such that $B_{j_i} : CPR_{j_i} \in \mathsf{Cert} \cap \mathsf{Inc\_Cert}$, $0 \leq i \leq k+1$ and $B_{j_{k+1}}$ has an entry in $Upd(P)$. Thus, $B_{j_{k+1}} : CPR_{j_{k+1}} \in CP_{checked}$, and propagating dependencies (by procedure propagate_effects), we obtain that $A_0 : CP_0 \in CP_{checked}$.

• **(Case A)** If $A_0 : CP_0 \notin \mathsf{Cert}$, then $A_0 : CP_0$ comes from the checking of some rule, and the dependency:

$$B_1 : CP_1 \Rightarrow A_0 : CP_0 \in DAT^f_{mem}$$

We distinguish two cases:

– If $B_1 : CP_1 \in \mathsf{Cert}$ then, the rule associated to the dependency has necessarily the form:

$$r \equiv B_1 : CP_1 \text{ :- } \ldots, A, \ldots, A_0, \ldots$$

and its original and full checking process verify that a call to process_arc for $A : CP \in \mathsf{Cert} \cap \mathsf{Inc\_Cert}$ was generated computing different answers for $A : CP$. But then, $A : CP$ corresponds to **(case B)** previously proved, and hence $A : CP \in CP_{checked}$. Hence, propagate_effects introduces $B_1 : CP_1$ in $CP_{tocheck}$. But then, a call process_node for $B_1 : CP_1$ is generated during the incremental checking, which forces the checking of rule r. Since $A_0 : CP_0 \notin \mathsf{Cert}$, then a call to process_node for $A_0 : CP_0$ is generated (L30 in Algorithm 2), introducing $A_0 : CP_0$ in $CP_{checked}$ (L8). Note that it is possible that $A_0 : CP_0 \in AT_{mem}$, when processing r. But in particular, since $A_0 : CP_0$ does not initially belong to AT_{mem}, this means that $A_0 : CP_0$ suffered a previous call to process_node, i.e., $A_0 : CP_0 \in CP_{checked}$.

– If $B_1 : CP_1 \notin \mathsf{Cert}$. Then reasoning as in the above case, there exists a sequence of arcs in DAT_{mem}:

$$\begin{array}{lll} r_0 & \equiv & B_1 : CP_1 \Rightarrow A_0 : CP_0 \\ r_1 & \equiv & B_2 : CP_2 \Rightarrow B_1 : CP_1 \\ r_2 & \equiv & B_3 : CP_3 \Rightarrow B_2 : CP_2 \\ \ldots & \ldots\ldots & \\ r_{n-1} & \equiv & B_n : CP_n \Rightarrow B_{n-1} : CP_{j_{n-1}} \end{array}$$

such that $B_i : CP_i \notin \mathsf{Cert}$, $1 \leq i < n$, $B_n : CP_n \in \mathsf{Cert}$. Then, the rule associated to the dependency r_{n-1} has necessarily the form:

$$r \equiv B_n : CP_n \text{ :- } \ldots, A, \ldots, B_{n-1}, \ldots$$

and its original and full checking process verify that a call to process_arc for $A : CP \in \mathsf{Cert} \cap \mathsf{Inc\_Cert}$ was generated computing different answers for $A : CP$. But then, $A : CP$ corresponds to **(case B)** previously proved, and hence $A : CP \in CP_{checked}$. Reasoning similarly to the above case, we obtain that $B_i : CP_i \in CP_{checked}$, $1 \leq i \leq n$, and thus $A_0 : CP_0 \in CP_{checked}$.

• **(Case C)** A_0 has an entry in $Upd(P)$. We have two possibilities.

- If $A_0 : CP_0 \in \mathsf{Cert}$, then $A_0 : CP_0 \in AT_{mem}$. Hence, procedure call_patterns_to_check introduces $A_0 : CP_0$ in $CP_{tocheck}$. Thus, finally $A_0 : CP_0$ will belong to $CP_{checked}$ (after the corresponding call to process_node for $A_0 : CP_0$).
- If $A_0 : CP_0 \notin \mathsf{Cert}$, then we are in **(case A)**. Hence, the execution of procedure incremental_checking will introduce $A_0 : CP_0$ in $CP_{checked}$.

• **(Case D)** There exists a dependency $A_0 : CP_0 \Rightarrow B : CP_B$ in the dependency arc table computed by the full checking such that $B : CP_B \notin unchanged(\mathsf{Cert})$. We distinguish several possibilities:

- If $A_0 : CP_0 \notin \mathsf{Cert}$, then the result holds by **(case A)**.
- If $A_0 : CP_0 \in \mathsf{Cert}$, then:
  - If $A_0 : CP_0 \in \mathsf{Inc\_Cert}$, then the result holds by **(case B)**.
  - If $A_0 : CP_0 \notin \mathsf{Inc\_Cert}$, then:
    - If $B : CP_B \notin \mathsf{Cert}$, then by **(case A)**, $B : CP_B \in CP_{checked}$. Hence, the incremental checking generates a call to process_node for $B : CP_B$ and also to propagate_effects. The last call introduces $A_0 : CP_0$ in $CP_{tocheck}$(if $A_0 : CP_0$ was not previously introduced).
    - If $B : CP_B \in \mathsf{Cert} \cap \mathsf{Inc\_Cert}$. Then, by **(case B)**, $B : CP_B \in CP_{checked}$, and reasoning as in the above case, it holds finally that $A_0 : CP_0 \in CP_{checked}$. □

Theorem 1 (correctness) *Let $P \in Prog$, $Upd(P) \in UProg$, $D_\alpha \in ADom$ and $S_\alpha \in AAtom$. Consider:*

- Cert*: the certificate for P and S_α.*
- Ext_Cert*: the certificate for $P \oplus Upd(P)$ and S_α.*
- Inc_Cert*: the incremental certificate for $Upd(P)$ w.r.t.* Cert*.*

If Incr_Checker*($Upd(P)$,* Inc_Cert*) does not issue an* Error*, then the validation of* Inc_Cert *is done in a single pass over* Inc_Cert *and:*

- $AT_{persist} \equiv AT_{mem}$
- $DAT_{persist} \equiv DAT_{mem}$

where AT_{mem} and DAT_{mem} are, respectively, the answer table and DAT returned by checking*($P \oplus Upd(p), S_\alpha$,* Ext_Cert*, AT_{mem}, DAT_{mem}).*

Proof. First note that the validation of Inc_Cert is done in one pass because of the use of the sets $CP_{tocheck}$ and $CP_{checked}$. Firstly $CP_{tocheck}$ stores those call patterns $A : CP$ in Cert such that A has suffered some modification of its rules. The incremental checking only adds call patterns to $CP_{tocheck}$ in procedure propagate_effects but firstly ensuring that such call patterns was not previously checked (they do not belong to the set $CP_{checked}$).

Incr_Checker($Upd(P)$, Inc_Cert) retrieves from memory $AT_{persist} \equiv$ Cert and $DAT_{persist} \equiv DAT^o_{mem}$, and initializes AT_{mem} and DAT_{mem} to such values, respectively, when calling to procedure incremental_checking(P, $Upd(P)$, Inc_Cert, AT_{mem}, DAT_{mem}). For all $A : CP \in$ Ext_Cert, $A : CP$ is of one of the following forms:

– $A : CP$ is safe. Then $A : CP \in \mathsf{Cert} \cap \mathsf{Ext\_Cert}$ and $A : CP \notin \mathsf{Inc\_Cert}$. Hence, $A : CP \mapsto AP \in AT_{mem} \cap \mathsf{Ext\_Cert}$. By definition, if $A : CP$ is safe, then $A : CP \notin CP_{checked}$. Note that this condition holds since A has no entries in $Upd(P)$ (then calls_patterns_to_check can not add $A : CP$ to $CP_{tocheck}$), and all its dependencies contains atoms $B : CP_B$ not belonging to Inc_Cert (i.e., if $B : CP_B$ is checked, variable *propagate* is always *false*, and no call to propagate_effects is generated). From Proposition 1, DAT^o_{mem} contains the same dependencies for $A : CP$ than DAT^f_{mem}. But, since $A : CP \notin CP_{checked}$, DAT_{mem} contains also the same dependencies for $A : CP$ than DAT^o_{mem}. Also Proposition 1 ensures that safe call patterns does not require to be checked after an update of a program, since the full and the original checking compute the same solutions. Hence the result holds for safe call patterns.
– If $A : CP$ is not safe. First note that procedure update_answer_table updates AT_{mem} with the information stored in Inc_Cert. This ensures that all fixpoints used in the checking process will be correct, i.e., will be the same used by the full checking. On the other hand, from Proposition 2, it holds that after executing procedure incremental_checking, $A : CP \in CP_{checked}$. This means that a call to process_node is generated for all non safe $A : CP$, and its fixpoint is introduced in AT_{mem} from Inc_Cert. But, before calling to process_node, the procedure remove_previous_info (L20) removes the entry for $A : CP$ from AT_{mem} and the corresponding dependencies from DAT_{mem}. If $A : CP \notin \mathsf{Inc\_Cert}$ (case in which $A : CP$ has the same fixpoint in the full and original checking process but its rules has been modified), then the fixpoint stored in AT_{mem} is introduced in Inc_Cert (L23) in order to avoid an Error in procedure process_node. Trivially, the incremental checking and the full checking compute the same dependencies for non safe call patterns, and considering now that after updating AT_{mem}, it holds that $AT_{mem} \cup \mathsf{Inc\_Cert} = \mathsf{Ext\_Cert}$, then the result holds trivially. □

B Security of the Incremental Checking

Theorem 2 In the conditions of Theorem 1, if an answer in Inc_Cert is not a VALID fixpoint, then INCR_CHECKER($Upd(P)$, Inc_Cert) issues an Error.

Proof. Assume that $A : CP \mapsto AP \in \mathsf{Inc\_Cert}$ but AP is not a fixpoint for $A : CP$. If A has an entry in $Upd(P)$, then call to process_node for $A : CP$ will issue an Error when processing some of its rules (from the correctness Algorithm 2 [3]). If all rules for A remains the same but $A : CP$ is introduced in $CP_{tocheck}$ because of the propagation of some change (procedure propagate_effects), then again the corresponding call to process_node will issue an Error. If none of the both previous cases occurs, procedure unrelated_entries (L7) will introduce $A : CP$ in $CP_{tocheck}$, calling to procedure check_demanded_entries, which will detect the Error when calling to process_node. □