On Abstraction-Carrying Code and Certificate-Size Reduction

Germán Puebla[1] Elvira Albert[2]
Puri Arenas[2] Manuel Hermenegildo[1,3]

[1] *Technical University of Madrid,* {german,herme}@fi.upm.es
[2] *Complutense University of Madrid,* {elvira,puri}@sip.ucm.es
[3] *University of New Mexico,* herme@unm.edu

Abstract
Abstraction-Carrying Code (ACC) is a framework for mobile code safety 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 is carried out automatically by a fixed-point analyzer. The advantage of providing a (fixed-point) abstraction to the code consumer is that its validity is checked in a *single pass* (i.e., one iteration) of an abstract interpretation-based checker. A main challenge to make ACC useful in practice is to reduce the size of certificates as much as possible, while at the same time not increasing checking time. Intuitively, we only include in the certificate the information which the checker is unable to reproduce without iterating. We introduce the notion of *reduced certificate* which characterizes the subset of the abstraction which a checker needs in order to validate (and re-construct) the full certificate in a single pass. Based on this notion, we show how to instrument a generic analysis algorithm with the necessary extensions in order to identify the information relevant to the checker.

1 Introduction

Proof-Carrying Code (PCC) [?] is a general framework 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 that should be much simpler, efficient, and automatic than generating the original certificate.

Abstraction-carrying code (ACC) [?,?] has been recently proposed as an enabling technology for PCC in which an *abstraction* (or 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 is automatically carried out by a fixed-point analyzer. 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. To capture the different graph traversal strategies used in different fixed-point algorithms, we use the *generic* description of [?], which generalizes the algorithms used in state-of-the-art analysis engines. Essentially, the certification/analysis carried out by the supplier is an iterative process which repeatedly traverses the analysis graph until a fixpoint is reached. The analysis information inferred for each call which appears during the (multiple) graph traversals is stored in the *answer table* [?]. After each iteration (or graph traversal), if the answer computed for a certain call is different from the one previously stored in the answer table, both answers are lubbed and the result is used 1) to *update* the table, and 2) to enforce recomputation of those calls whose answer depends on it. In the original ACC framework, the final *full* answer table constitutes the certificate. The key idea is that, since the certificate contains the fixpoint, a single pass over the analysis graph is sufficient to validate the certificate in the consumer side. It should be noted that the ACC framework and our work here are defined at the source-level, whereas in existing PCC frameworks the code supplier typically packages the certificate with the *object* code rather than with the *source* code (both are untrusted). The reasons and motivations for our approach can be found in [?].

One of the main challenges for the practical uptake of ACC (and related methods) is to produce certificates which are reasonably small. This is important since the certificate is transmitted together with the untrusted code and, hence, reducing its size will presumably contribute to a smaller transmission time –very relevant for instance under scarce (or expensive) network connectivity conditions. Also, this reduces the storage cost for the certificate. Nevertheless, a main concern when reducing the size of the certificate is that checking time is not increased as a consequence. In principle, the consumer could use an analyzer for the purpose of generating the whole fixpoint from scratch, which is still feasible as analysis is automatic. However, this would defeat one of the main purposes of ACC, which is to reduce checking time. The objective of this paper is to characterize the subset of the abstraction which must be sent within a certificate and which still guarantees a single pass checking process.

Fixpoint compression is being used in different contexts and tools. For instance, in the Astrée analyzer [?], only one abstract element by head of loop is kept for memory usage purposes. In the PCC scheme, the basic idea in order to compress a certificate is to store only the analysis information which the checker is not able to reproduce by itself [?]. For instance, this general idea has also been deployed in lightweight bytecode verification [?] where the certificate, rather than being the whole set of frame types (FT) associated to each program point as obtained by standard bytecode verification [?], is reduced by omitting those (local) program points FTs which correspond to instructions without branching *and* which are lesser than the final FT (fixpoint). Our proposal for ACC is at the same time more general (because of the parametricity of the ACC approach) and carries the reduction further because it includes only the analysis information of those calls in the analysis graph whose answers have been *updated*, including both branching and non branching instructions. The intuition is that, when there is at most one (initial) update during the computation of an entry in the answer table, the part of the analysis graph associated to it has been computed in one traversal, i.e., its fixpoint has been reached in a single pass. Hence, we can safely extract such information from the certificate and the checker should still be able to re-generate it in a single pass. In this work, we introduce the notion of *reduced certificate* which characterizes the subset of the abstraction which the checker needs in order to validate (and re-construct) the full certificate in a single pass. Then, we show how to instrument the generic analysis algorithm of [?] with the necessary extensions in order to identify relevant information to the checker.

The rest of the paper is organized as follows. The following section presents a general view of ACC. Section 3 recalls the certification process performed by the code supplier and illustrates it through our running example. In Section 4, we characterize the notion of reduced certificate and in Section 5, we instrument a generic certifier for its generation. Finally, Section 6 concludes and discusses the work presented in this paper and future work.

2 A General View of Abstraction-Carrying Code

We assume the reader is familiar with abstract interpretation (see [?]) and (Constraint) Logic Programming (C)LP (see, e.g., [?] and [?]).

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

The basics for defining such certifiers (and their corresponding checkers) in ACC are summarized in the following six 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 [?] $\forall x \in 2^D : \gamma(\alpha(x)) \supseteq x$ 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., $[\![P]\!] = \text{lfp}(S_P)$. 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 $[\![P]\!]_\alpha = \text{lfp}(S_P^\alpha)$. The operator S_P^α is the abstract counterpart of S_P.

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

Correctness of analysis ensures that $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$, i.e., $[\![P]\!] \in \gamma([\![P]\!]_\alpha)$.

Verification Condition. Let $Cert_\alpha$ be a safe approximation of P. If an abstract safety specification I_α can be proved w.r.t. $Cert_\alpha$, then P satisfies the safety policy and $Cert_\alpha$ is a valid certificate:

$$Cert_\alpha \text{ is } \textit{a valid certificate} \text{ for } P \text{ w.r.t. } I_\alpha \text{ if } Cert_\alpha \sqsubseteq I_\alpha \quad (2)$$

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

$$\mathsf{checker}(P, D_\alpha, Cert_\alpha) \textit{ returns true iff } (S_P^\alpha(Cert_\alpha) \equiv Cert_\alpha) \quad (3)$$

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.

$$P \textit{ is trusted iff } Cert_\alpha \sqsubseteq I_\alpha \quad (4)$$

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 Generation of Certificates in ACC

This section recalls ACC and the notion of full certificate in the context of (C)LP [?]. For concreteness, we build on the algorithms of `CiaoPP` [?].

3.1 The Analysis Algorithm

Algorithm 1 has been presented in [?] as a generic description of a fixed-point algorithm which generalizes those used in state-of-the-art analysis engines, such as the one in `CiaoPP` [?]. In order to analyze a program, traditional (goal dependent) abstract interpreters for (C)LP programs receive as input, in addition to the program P and the abstract domain D_α, a set $S_\alpha \in AAtom$ of Abstract Atoms (or *call patterns*). Such call patterns 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 $CP \in D_\alpha$. For brevity, we sometimes omit the subscript α in the algorithms. The analyzer of Algorithm 1 constructs an *and–or graph* [?] (or analysis graph) for S_α which is an abstraction of the (possibly infinite) set of (possibly infinite) and-or trees explored by the concrete execution of initial calls described by S_α in P. The program analysis graph is implicitly represented in the algorithm by means of two data structures, the *answer table* and the *dependency arc table.*

- The answer table contains entries of the form $A : CP \mapsto AP$ where A is always a base form.[1] Informally, its entries should be interpreted as "the answer pattern for calls to A satisfying precondition (or call pattern), CP, accomplishes postcondition (or answer pattern), AP."
- A dependency arc is of the form $H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$. This is interpreted as follows: if the rule with H_k as head is called with description CP_0 then this causes that the i-th literal $B_{k,i}$ to be called with description CP_2. The remaining part CP_1 is the program annotation just before $B_{k,i}$ is reached and contains information about all variables in rule k.

Intuitively, the analysis algorithm is a graph traversal algorithm which places entries in the answer table and dependency arc table as new nodes and arcs in the program analysis graph are encountered. To capture the different graph traversal strategies used in different fixed-point algorithms, a *prioritized event queue* is used. We use $\Omega \in QHS$ to refer to a *Queue*

[1] 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.

Algorithm 1 Generic Analyzer for Abstraction-Carrying Code

```
 1: function ANALYZE_F($S, \Omega$)
 2:     for $A : CP \in S$ do
 3:         add_event($newcall(A : CP), \Omega$)
 4:     while $E :=$ next_event($\Omega$) do
 5:         if $E := newcall(A : CP)$ then new_call_pattern($A : CP, \Omega$)
 6:         else if $E := updated(A : CP)$ then add_dependent_rules($A : CP, \Omega$)
 7:         else if $E := arc(R)$ then process_arc($R, \Omega$)
 8:     return answer table
 9: procedure NEW_CALL_PATTERN($A : CP, \Omega$)
10:     for all rule $A_k : -B_{k,1}, \ldots, B_{k,n_k}$ do
11:         $CP_0 :=$Aextend($CP, vars(B_{k,1}, \ldots, B_{k,n_k})$)
12:         $CP_1 :=$ Arestrict($CP_0, vars(B_{k,1})$)
13:         add_event($arc(A_k : CP \Rightarrow [CP_0]\ B_{k,1} : CP_1)$,$\Omega$)
14:     add $A : CP \mapsto \bot$ to answer table
15: procedure PROCESS_ARC($H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$,$\Omega$)
16:     if $B_{k,i}$ is not a constraint then
17:         add $H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$ to dependency arc table
18:     $W := vars(A_k, B_{k,1}, \ldots, B_{k,n_k})$; $CP_3 :=$ get_answer($B_{k,i} : CP_2, CP_1, W, \Omega$)
19:     if $CP_3 \neq \bot$ and $i \neq n_k$ then
20:         $CP_4 :=$ Arestrict($CP_3, vars(B_{k,i+1})$);
21:         add_event( $arc(H_k : CP_0 \Rightarrow [CP_3]\ B_{k,i+1} : CP_4)$,$\Omega$)
22:     else if $CP_3 \neq \bot$ and $i = n_k$ then
23:         $AP_1 :=$ Arestrict($CP_3, vars(H_k)$); insert_answer_info($H : CP_0 \mapsto AP_1, \Omega$)
24: function GET_ANSWER($L : CP_2, CP_1, W, \Omega$)
25:     if $L$ is a constraint then return Aadd($L, CP_1$)
26:     else $AP_0 :=$ lookup_answer($L : CP_2, \Omega$); $AP_1 :=$ Aextend($AP_0, W$)
27:         return Aconj($CP_1, AP_1$)
28: function LOOKUP_ANSWER($A : CP, \Omega$)
29:     if there exists a renaming $\sigma$ s.t.$\sigma(A : CP) \mapsto AP$ in answer table then
30:         return $\sigma^{-1}(AP)$
31:     else add_event($newcall(\sigma(A : CP)), \Omega$) where $\sigma$ is renaming s.t. $\sigma(A)$ in base
            form; return $\bot$
32: procedure INSERT_ANSWER_INFO($H : CP \mapsto AP, \Omega$)
33:     $AP_0 :=$ lookup_answer($H : CP$); $AP_1 :=$ Alub($AP, AP_0$)
34:     if $AP_0 \neq AP_1$ then
35:         add ($H : CP \mapsto AP_1$) to answer table ;
36:         add_event($updated(H : CP), \Omega$)
37: procedure ADD_DEPENDENT_RULES($A : CP, \Omega$)
38:     for all arc of the form $H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$ in graph where there
            exists renaming $\sigma$ s.t. $A : CP = (B_{k,i} : CP_2)\sigma$ do
39:         add_event($arc(H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2), \Omega$)
```

Handling Strategy which a particular instance of the generic algorithm may use. Events are of three forms:

- newcall($A : CP$) which indicates that a new call pattern for literal A with description CP has been encountered.
- arc(H_k : _ $\Rightarrow$ [_] $B_{k,i}$: _) which indicates that the rule with H_k as head needs to be (re)computed from the position k, i.
- updated($A : CP$) which indicates that the answer description to call pat-

tern A with description CP has been changed.

The functions add_event and next_event respectively push an event to the priority queue and pop the event of highest priority, according to Ω. The algorithm is defined in terms of four abstract operations on the domain D_α:

- Arestrict(CP, V) performs the abstract restriction of a description CP to the set of variables in the set V, denoted $vars(V)$;
- Aextend(CP, V) extends the description CP to the variables in the set V;
- Aconj(CP_1, CP_2) performs the abstract conjunction of two descriptions;
- Alub(CP_1, CP_2) performs the abstract disjunction of two descriptions.

More details on the algorithm can be found in [?,?]. Let us briefly explain its main procedures. The algorithm centers around the processing of events on the priority queue, which repeatedly removes the highest priority event (Line 4) and calls the appropriate event-handling function (L5-7). The function new_call_pattern initiates processing of all the rules for the definition of the internal literal A, by adding arc events for each of the first literals of these rules (L13). Initially, the answer for the call pattern is set to $\perp$ (L14). The procedure process_arc performs the core of the analysis. It performs a single step of the left-to-right traversal of a rule body. If the literal $B_{k,i}$ is not a constraint (L16), the arc is added to the dependency arc table (L17). Atoms are processed by function get_answer. Constraints are simply added to the current description (L25). In the case of literals, the function lookup_answer first looks up an answer for the given call pattern in the answer table (L29) and if it is not found, it places a *newcall* event (L39). When it finds one, then this answer is extended to the variables in the rule the literal occurs in (L26) and *conjoined* with the current description (L27). The resulting answer (L18) is either used to generate a new arc event to process the next literal in the rule, if $B_{k,i}$ is not the last one (L19); otherwise, the new answer is computed by insert_answer_info. This is the part of the algorithm more relevant to the generation of reduced certificates. The new answer for the rule is *combined* with the current answer in the table (L33). If the fixpoint for such call has not been reached, then the answer table entry is updated with the combined answer (L35) and an updated event is added to the queue (L36). The purpose of such an update is that the function add_dependent_rules (re)processes those calls which depend on the call pattern $A : CP$ whose answer has been updated (L38). This effect is achieved by adding the arc events for each of its dependencies (L39). Note that dependency arcs are used for efficiency: they allow us to start the reprocessing of a rule from the body atom which actually needs to be recomputed due to an update rather than from scratch.

3.2 Running Example

Our running example is the program `rectoy` taken from [?]. We will use it to illustrate our algorithms and show that our approach improves state-of-the-art techniques for reducing the size of certificates. In all our examples, abstract substitutions over a set of variables V, assign a *regular type* [?] to each variable in V. We use `term` as the most general type (i.e., `term` corresponds to all possible terms). For brevity, variables whose regular type is `term` are often not shown in abstract substitutions. Also, when it is clear from the context, an abstract substitution for an atom $p(x_1, \dots, x_n)$ is shown as a tuple $\langle t_1, \dots, t_n \rangle$, such that each value t_i indicates the type of x_i. The most general substitution $\top$ assigns `term` to all variables in V. The least general substitution $\perp$ assigns the empty set of values to each variable.

Example 3.1 Consider the `Ciao` version of procedure `rectoy` [?] and the call pattern $\texttt{rectoy(N,M)} : \langle \texttt{int}, \texttt{term} \rangle$ which indicates that external calls to `rectoy` are performed with an integer value, `int`, in the first argument `N`.

```
rectoy(N,M) :- N = 0, M = 0.
rectoy(N,M) :- N1 is N-1, rectoy(N1,R), M is N1+R.
```

We now briefly describe four main steps carried out in the analysis using some $\Omega \in QHS$ (the detailed steps and analysis graph can be found in the technical report [?]):

A. The initial event $\texttt{newcall(rectoy(N,M)} : \langle \texttt{int}, \texttt{term} \rangle)$ introduces the arcs $A_{1,1}$ and $A_{2,1}$ in the queue, each one corresponds to the rules in the order above:

   $A_{1,1} \equiv \texttt{arc(rectoy(N,M)} : \langle \texttt{int}, \texttt{term} \rangle \Rightarrow [\{\texttt{N/int}\}]\ \texttt{N} = \texttt{0} : \{\texttt{N/int}\})$
   $A_{2,1} \equiv \texttt{arc(rectoy(N,M)} : \langle \texttt{int}, \texttt{term} \rangle \Rightarrow [\{\texttt{N/int}\}]\ \texttt{N1 is N} - \texttt{1} : \{\texttt{N/int}\})$

   The initial answer $\boxed{E_1 \mid \texttt{rectoy(N,M)} : \langle \texttt{int}, \texttt{term} \rangle \mapsto \perp}$ is inserted in the answer table. Label E_1 is introduced for future reference.

B. Assume that Ω assigns higher priority to $A_{1,1}$. The next arc is generated:

   $A_{1,2} \equiv \texttt{arc(rectoy(N,M)} : \langle \texttt{int}, \texttt{term} \rangle \Rightarrow [\{\texttt{N/int}\}]\ \texttt{M} = \texttt{0} : \{\texttt{M/term}\})$

   As it is the last atom in the body (L22), procedure insert_answer_info computes $\mathsf{Alub}(\perp, \{\texttt{N/int}, \texttt{M/int}\})$ and overwrites E_1 with

   $$\boxed{E_1' \mid \texttt{rectoy(N,M)} : \langle \texttt{int}, \texttt{term} \rangle \mapsto \langle \texttt{int}, \texttt{int} \rangle}$$

   Consequently, $U_1 : \texttt{updated(rectoy(N,M)} : \langle \texttt{int}, \texttt{term} \rangle)$ is introduced in the queue.

C. Now, Ω can choose between the processing of U_1 or $A_{2,1}$. Let us assume that $A_{2,1}$ has higher priority. For its processing, we consider that predefined functions "$-$", "$+$" and "`is`" are dealt by the algorithm as standard constraints (see [?] for further details). Next, the arc:

$$A_{2,2} \equiv \texttt{arc(rectoy(N,M)} : \langle\texttt{int},\texttt{term}\rangle \Rightarrow$$
$$[\{\texttt{N/int},\texttt{N1/int}\}]\ \texttt{rectoy(N1,R)} : \langle\texttt{int},\texttt{term}\rangle)$$

is introduced in the queue and the corresponding dependency stored. By using the current answer E'_1, we get the arc $A_{2,3}$:

$$A_{2,3} \equiv \texttt{arc(rectoy(N,M)} : \langle\texttt{int},\texttt{term}\rangle \Rightarrow$$
$$[\{\texttt{N/int},\texttt{N1/int},\texttt{R/int}\}]\ \texttt{M is N1} + \texttt{R} : \{\texttt{N1/int},\texttt{R/int}\})$$

Clearly, the processing of $A_{2,3}$ does not change the final answer E'_1. Hence, no more updates are introduced in the queue.

D. Finally, we have to process the event U_1 introduced in step B to which Ω has assigned lowest priority. The procedure add_dependent_rules finds the dependency corresponding to arc $A_{2,2}$ and inserts an arc for it in the queue. This relaunches an arc identical to $A_{2,2}$, which in turn launches an arc identical to $A_{2,3}$. However, the reprocessing does not change the fixpoint result E'_1 and analysis terminates.

A fundamental issue here is that if we use some $\Omega' \in QHS$ which assigns a priority to U_1 higher than to $A_{2,1}$, the whole reprocessing of $A_{2,2}$ and $A_{2,3}$ in step D will not be performed. The reason is that the dependency arc table would be empty prior to processing $A_{2,2}$. Hence add_dependent_rules would not introduce any arc. This corresponds to the notion of redundant update which we will introduce in Def. 4.1. □

3.3 Full Certificate

The following definition corresponds to the essential idea in the ACC framework –equations (1) and (2)– of using a static analyzer to generate the certificates. The analyzer corresponds to Algorithm 1 and the certificate is the *full* answer table.

Definition 3.2 [full certificate] We define function CERTIFIER_F:*Prog* × *ADom* ×*AAtom* × *AInt* × *QHS* ↦ *ACert* which takes $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $I_\alpha \in AInt$, $\Omega \in QHS$ and returns as *full certificate*, FCert $\in ACert$, the answer table computed by ANALYZE_F(S_α, Ω) for P in D_α iff FCert $\sqsubseteq I_\alpha$.

Example 3.3 Consider the safety policy expressed by the following specification I_α : `rectoy(N,M)` : $\langle$`int`,`term`$\rangle \mapsto \langle$`int`,`real`$\rangle$. The certifier in Def. 3.2 returns as valid certificate the single entry E'_1. Clearly $E'_1 \sqsubseteq I_\alpha$. □

4 The Notion of Reduced Certificate

The key observation in order to reduce the size of certificates is that certain entries in a certificate may be *irrelevant*, in the sense that the checker is able to reproduce them by itself in a single pass. The notion of *relevance* is directly related to the idea of recomputation in the program analysis graph. Intuitively, given an entry in the answer table $A : CP \mapsto AP$, its fixpoint may have been computed in several iterations from $\bot$, AP_0, $AP_1, \ldots$ until AP. For each change in the answer, an updated event updated($A : CP$) is generated during analysis. The above entry is relevant in a certificate (under some strategy) when its updates force the recomputation of other arcs in the graph which *depend* on $A : CP$ (i.e., there is a dependency from it in the table). Thus, unless $A : CP \mapsto AP$ is included in the (reduced) certificate, a single-pass checker which uses the same strategy as the code producer will not be able to validate the certificate.

According to the above intuition, we are interested in determining when an entry in the answer table has been "updated" during analysis and such changes affect other entries. However, there are two special types of updated events which can be considered "irrelevant". The first one is called *redundant update* and corresponds to the kind of updates which force a redundant computation (like the event U_1 generated in step B of Ex. 3.1). We write $DAT|_{A:CP}$ to denote the set of arcs of the form $H : CP_0 \Rightarrow [CP_1]B : CP_2$ in the current dependency arc table such that they depend on $A : CP$, i.e., $A : CP = (B : CP_2)\sigma$, for some renaming σ.

Definition 4.1 [redundant update] Let $P \in Prog$, $S_\alpha \in AAtom$ and $\Omega \in QHS$. We say that an event updated($A : CP$) which appears in the event queue during the analysis of P for S_α is *redundant* w.r.t. Ω iff, when it is generated, $DAT|_{A:CP} = \emptyset$.

In the following section, we propose a slight modification to the analysis algorithm so that redundant updates are executed as soon as they appear, so that they never enforce redundant recomputation. Correctness of this modification can be found in the technical report [?].

Example 4.2 In our running example, U_1 is redundant for Ω at the moment it is generated. However, since the event has been given low priority, its processing is delayed until the end and, in the meantime, a dependency from it has been added. This causes the unnecessary redundant re-computation of the second arc for `rectoy` ($A_{2,2}$). □

The second type of updates which can be considered irrelevant are *initial updates* which, under certain circumstances, are generated in the first pass over an arc. In particular, we do not take into account updated events generated when the answer table contains $\bot$ for the updated entry. Note that this case still corresponds to the first traversal of any arc and should not be considered as a reprocessing.

Definition 4.3 [initial update] In the conditions of Def. 4.1, we say that an event $\mathsf{updated}(A : CP)$ which appears in the event queue during the analysis of P for S_α is *initial* for Ω if, when it is generated, the answer table contains $A : CP \mapsto \bot$.

Initial updates do not occur in certain very optimized algorithms, like the one in [?]. However, they are necessary to model generic graph traversal strategies. In particular, they are intended to *awake* arcs whose evaluation has been *suspended*.

Example 4.4 Suppose that we use a strategy $\Omega'' \in QHS$ such that step C in Ex. 3.1 is performed before B (i.e., the second rule is analyzed before the first one). Then, when the answer for $\texttt{rectoy(N1,R)} : \langle\texttt{int},\texttt{term}\rangle$ is looked up, procedure get_answer returns $\bot$ and thus the processing of arc $A_{2,2}$ is *suspended* at this point in the sense that its continuation $A_{2,3}$ is not inserted in the queue (see L19 in Algorithm 1). Indeed, we can proceed with the remaining arc $A_{1,1}$ which is processed exactly as in step B. In this case, the updated event U_1 is not redundant for Ω'', as there is a dependency introduced by the former processing of arc $A_{2,2}$ in the table. Therefore, the processing of U_1 introduces the suspended arc $A_{2,2}$ again in the queue. The important point is that the fact that U_1 inserts $A_{2,2}$ must not be considered as a reprocessing, since $A_{2,2}$ had been suspended and its continuation ($A_{2,3}$ in this case) has not been handled by the algorithm yet. $\square$

Definition 4.5 [relevant update] In the conditions of Def. 4.1, we say that an event $\mathsf{updated}(A : CP)$ is *relevant* iff it is not initial nor redundant.

The key idea is that those answer patterns whose computation has introduced relevant updates should be available in the certificate.

Definition 4.6 [relevant entry] In the conditions of Def. 4.1, we say that the entry $A : CP \mapsto AP$ in the answer table is *relevant* for Ω iff there has been at least one relevant event $\mathsf{updated}(A : CP)$ during the analysis of P for S_α.

The notion of *reduced certificate* allows us to remove irrelevant entries from the answer table and produce a smaller certificate which can still be validated in one pass.

Definition 4.7 [reduced certificate] In the conditions of Def. 4.1, let FCert= $\textsc{Analyze\_f}(S_\alpha, \Omega)$ for P and S_α. We define the *reduced certificate*, RCert, as the set of relevant entries in FCert for Ω.

Example 4.8 From now on, in our running example, we assume the strategy $\Omega' \in QHS$ which assigns the highest priority to redundant updates. For this strategy, the entry $\boxed{E_1' \,|\, \texttt{rectoy(N,M)} : \langle\texttt{int},\texttt{term}\rangle \mapsto \langle\texttt{int},\texttt{int}\rangle}$ in Example 3.1 is not relevant as there has been no relevant updated event in the queue (U_1 is redundant). Therefore, the reduced certificate for our running example is empty. [2] □

For function `rectoy` in Example 3.1, lightweight bytecode verification [?] sends, together with the program, the reduced *non-empty* certificate $cert = (\{30 \mapsto (\epsilon, rectoy \cdot int \cdot int \cdot int \cdot \bot)\}, \epsilon)$, which states that in the program point 30, the stack does not contain information (first occurrence of ϵ), [3] and variables N, M and R have type *int*, *int* and $\bot$. The need of sending this information is because `rectoy`, implemented in Java, contains an *if*-branch (equivalent to the branching for selecting one of our two clauses for *rectoy*). And *cert* has to inform the checker that it is possible that in the point 30, variable R is undefined, if the *if* condition does not hold. Therefore, the above example shows that our approach improves on state-of-the-art PCC techniques by reducing the certificate even further while still keeping the checking process one-pass.

5 Generation of Certificates without Irrelevant Entries

In this section, we proceed to instrument the analyzer of Algorithm 1 with the extensions necessary for producing reduced certificates, as defined in Def. 4.7. The resulting analyzer ANALYZE_R is presented in Algorithm 2. It uses the same procedures of Algorithm 1 except for the new definitions of add_dependent_rules and insert_answer_info. They differ from the original definitions in that:

(i) *We count the number of relevant updates for each call pattern.* To do this, we associate to each entry in the answer table a new field "u" whose purpose is to identify relevant entries. Concretely, u indicates the number of updated events processed for the entry. u is initialized when the (unique and first) initial updated event occurs for a call pattern. The initialization of u is different for redundant and initial updates as explained in the next point. When the analysis finishes, if $u > 1$, we know that at least one reprocessing has occurred and the entry is thus relevant. The essential point to note is that u has to be increased when the event is actually *extracted* from the queue (L3) and not when it is *introduced* in it (L14). The reason for this is that when a non redundant updated event is introduced, if the priority queue contains an identical event, then the processing is performed only once. Therefore, our counter must not be increased.

[2] It should be noted that, using Ω as in Example 3.1, the answer is obtained by performing two analysis iterations over the arc associated to the second rule of `rectoy(N,M)` (steps C and D) due to the fact that U_1 has been delayed and become relevant for Ω.
[3] The second occurrence of ϵ indicates that there are no backwards jumps.

(ii) *We do not generate redundant updates.* Our algorithm does not introduce redundant updated events (L14). However, if they are initial (and redundant), they have to be counted as if they had been introduced and processed and, thus, the next update over them has to be considered always relevant. This effect is achieved by initializing the u-value with a higher value ("1" in L12) than for initial updates ("0" in L11). Indeed, the value "0" just indicates that the initial updated event has been introduced in the priority queue but not yet processed. It will be increased to "1" once it is extracted from the queue. Therefore, in both cases, the next updated event over the call pattern will increase the counter to "2" and will be relevant.

In Algorithm 2, a call (u, AP)=get_from_answer_table$(A : CP)$ looks up in the answer table the entry for $A : CP$ and returns its u-value and its answer AP. A call set_in_answer_table$(A(u) : CP \mapsto AP)$ replaces the entry for $A : CP$ with the new one $A(u) : CP \mapsto AP$.

Algorithm 2 ANALYZE_R: Analyzer instrumented for Certificate Reduction

```
1: procedure ADD_DEPENDENT_RULES($A : CP, \Omega$)
2:     $(AP, u)$ =get_from_answer_table($A : CP$)
3:     set_in_answer_table($A(u+1) : CP \mapsto AP$)
4:     for all arc of the form $H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$ in graph where there
           exists renaming $\sigma$ s.t. $A : CP = (B_{k,i} : CP_2)\sigma$ do
5:         add_event($arc(H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2), \Omega$)
6: procedure INSERT_ANSWER_INFO($H : CP \mapsto AP, \Omega$)
7:     $AP_0$ := lookup_answer($H : CP, \Omega$)
8:     $AP_1$ := Alub($AP, AP_0$)
9:     if $AP_0 \neq AP_1$ then
10:        if $AP_0 = \bot$ then          % initial update
11:            if $DAT|_{H:CP} \neq \emptyset$ then $u = 0$
12:            else $u = 1$
13:        else $(u, \_)$=get_from_answer_table($H : CP$)          % not initial update
14:        if $DAT|_{H:CP} \neq \emptyset$ then add_event(updated($H : CP$))
15:        set_in_answer_table($H(u) : CP \mapsto AP_1$)
```

Example 5.1 Consider the four steps performed in the analysis of our running example. Step A is identical. In step B, the procedure insert_answer_info detects an initial and redundant updated event (L12) and initializes the u-value of E'_1 to 1. No updated event is generated (L14). Step C remains identical and step D does not occur. As expected, upon return, the value of u for E'_1 is 1. □

Proposition 5.2 *Let $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $\Omega \in QHS$. Let* FCert *be the answer table computed by* ANALYZE_R*(S_α, Ω) for P in D_α. Then, an entry $A(u) : CP_A \mapsto AP \in$* FCert *is relevant iff $u > 1$.*

Note that, except for the control of relevant entries, ANALYZE_F(S_α, Ω) and ANALYZE_R(S_α, Ω) have the same behaviour and they compute the same answer table (see our technical report [?]). We use function remove_irrelevant-

_answers which takes a set of answers of the form $A(u) : CP \mapsto AP \in$ FCert and returns, in RCert, the set of answers $A : CP \mapsto AP$ such that $u > 1$.

Definition 5.3 We define the function CERTIFIER_R: $Prog \times ADom \times AAtom \times AInt \times QHS \mapsto ACert$, which takes $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $I_\alpha \in AInt$, $\Omega \in QHS$, and returns as certificate, RCert=remove_irrelevant_answers(FCert) iff FCert $\sqsubseteq I_\alpha$, where FCert=ANALYZE_R(S_α, Ω).

6 Discussion

In this paper we have proposed an extension of the ACC framework which generates (and checks) *reduced certificates* by eliminating from certificates the information which the checker can reproduce in a single pass. This allows reducing transmission and storage costs for certificates without increasing checking time.

6.1 Some Issues on Checking Reduced Certificates

In the ACC framework for full certificates, the checking algorithm [?] uses a specific graph traversal strategy Ω_C. This checker has been shown to be very efficient but in turn its design is not generic with respect to this issue (in contrast to the analysis design).[4] This is not problematic in the context of full certificates as, even if the certifier uses a strategy Ω_A different from Ω_C, it is ensured that all valid certificates get validated in one pass by such specific checker. This result does not hold anymore in the case of reduced certificates. In particular, *completeness* of checking is not guaranteed if $\Omega_A \neq \Omega_C$. This occurs because though the answer table is identical for all strategies, the subset of redundant entries depends on the particular strategy used. The problem is that, if there is an entry $A : CP \mapsto AP$ in FCert such that it is relevant w.r.t. Ω_C but it is not w.r.t. Ω_A, then a single pass checker will fail to validate the RCert generated using Ω_A. Therefore, it is essential in this context to design generic checkers which are not tied to a particular graph traversal strategy. Upon agreeing on the appropriate parameters[5], the consumer uses the particular instance of the generic checker resulting from application of such parameters. In our technical report [?], we design a checker for reduced certificates which is *correct*, i.e., if the checker succeeds in validating the certificate, then the certificate is valid

[4] Note that both the analysis and checking algorithms are always parametric on the abstract domain, with the resulting genericity, which allows proving a wide variety of properties by using the large set of available domains, being one of the fundamental advantages of ACC.

[5] In a particular application of our framework, we expect that the graph traversal strategy is agreed a priori between consumer and producer. But, if necessary, (e.g., the consumer does not implement this strategy), it could be sent along with the transmitted package.

for the program, no matter what the graph traversal strategy used is. We also provide sufficient conditions for ensuring *completeness* of the checking process. Concretely, if the checker uses the same strategy as the analyzer then our proposed checker will succeed in validating any reduced certificate which is valid.

6.2 On the Experimental Evaluation

As we have illustrated throughout the paper, the reduction achieved is directly related to the amount of *updates* (or iterations) performed during analysis. Clearly, depending on the graph traversal strategy used, different instances of the generic analyzer will generate reduced certificates of different sizes. Important efforts have been made during the last years in order to improve the efficiency of analysis. The most optimized analyzers aim at reducing the number of updates necessary to reach the final fixpoint [?]. Interestingly, our framework greatly benefits from all these advances, since the more efficient analysis is, the smaller the corresponding reduced certificates are. We have implemented the generator of reduced certificates as an extension of the efficient, highly optimized, state-of-the-art analysis system available in `CiaoPP` and which is part of a working compiler. Both, the analysis and checker use the optimized depth-first new-calling QHS of [?]. We are now in the process of experimentally evaluating our approach. Preliminary results are very encouraging. They show reductions in certificate size of around 70% on average (see [?]).

6.3 Future Work

We plan to assess the influence that different strategies have on certificate reduction. Also, we will consider and compare with the case of using the fixed-point analyzers also on the checking side. In this case, since the certificate can be recreated at the receiving end as much as needed, there is clearly a wide range of trade-offs between the size of the certificate and the checking time. We also want to investigate ways of reducing the trusted base code (see, e.g., [?,?]) in ACC. Additionally, we are studying the application of incremental analysis algorithms in order to reduce both certificate size and checking time in the context of modifications to a program for which a certificate has already been checked at the consumer side [?].