Reduced Certificates for Abstraction-Carrying Code

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

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

Abstract. *Abstraction-Carrying Code* (ACC) has recently been proposed as 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. The intuitive idea is to only include in the certificate information that 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 instrument a generic analysis algorithm with the necessary extensions in order to identify information which can be reconstructed by the single-pass checker. Finally, we study what the effects of reduced certificates are on the correctness and completeness of the checking process. We provide a correct checking algorithm together with sufficient conditions for ensuring its completeness. Our ideas are illustrated through a running example, implemented in the context of constraint logic programs, which shows that our approach improves state-of-the-art techniques for reducing the size of certificates.

1 Introduction

Proof-Carrying Code (PCC) [15] 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) [1, 8] 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 [9], 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* [9]. In the original ACC framework, the *full* answer table constitutes the certificate. The key idea 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.

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 for at least the following reasons. First, 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 network connectivity conditions. Second, the certificate has to be stored on the consumer end in order to validate it and, hence, reducing its size is important in the realm of pervasive and embedded systems which typically have limited storage (and computing) resources. 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 a subset of the abstraction which must be sent within a certificate –and which still guarantees a single pass checking process– and to design an ACC scheme which generates and validates such reduced certificates.

The basic idea in order to reduce the size of the certificate in ACC checkers is to store only the analysis information which the checker is not able to reproduce by itself [12]. For instance, this general idea has been deployed in lightweight bytecode verification [18] where the certificate, rather than being the whole set of frame types (FT) associated to each program point as obtained by standard bytecode verification [12], is reduced by omitting those (local) program point 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. The main contributions of this paper are:

1. We introduce the notion of *reduced certificate* which characterizes a subset of the abstraction which the checker needs in order to validate (and re-construct) the full certificate in a single pass.
2. We instrument the generic analysis algorithm of [9] with the necessary extensions in order to identify the information which can be computed in one pass.
3. 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 for the program, no matter what the graph traversal strategy used is.
4. Finally, we 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.

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 instrument a generic certifier for its generation. Section 5 presents a correct and complete generic checker for reduced certificates. Finally, Section 6 discusses the work presented in this paper and some future work.

2 A General View of Abstraction-Carrying Code

We assume the reader is familiar with abstract interpretation (see [6]) and (Constraint) Logic Programming (C)LP (see, e.g., [14] and [13]).

A certifier is a function $\mathsf{certifier} : \mathit{Prog} \times \mathit{ADom} \times \mathit{AInt} \mapsto \mathit{ACert}$ which for a given program $P \in \mathit{Prog}$, an abstract domain $D_\alpha \in \mathit{ADom}$ and a safety policy $I_\alpha \in \mathit{AInt}$ generates a 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 by using the same subscript 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 [6] $\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall y \in D_\alpha : \ \alpha(\gamma(y)) = y$. 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 $[\![P]\!]_\alpha = \mathrm{lfp}(S^\alpha_P)$. The operator S^α_P is the abstract counterpart of S_P.

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

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

Verification Condition. 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:

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

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 $\mathit{Cert}_\alpha = [\![P]\!]_\alpha$.

Checking. A checker is a function $\mathsf{checker} : \mathit{Prog} \times \mathit{ADom} \times \mathit{ACert} \mapsto \mathit{bool}$ which for a program $P \in \mathit{Prog}$, an abstract domain $D_\alpha \in \mathit{ADom}$ and certificate $\mathit{Cert}_\alpha \in \mathit{ACert}$ checks whether Cert_α is a fixpoint of S^α_P or not:

$$\mathsf{checker}(P, D_\alpha, \mathit{Cert}_\alpha) \textit{ returns true iff } (S^\alpha_P(\mathit{Cert}_\alpha) \equiv \mathit{Cert}_\alpha) \tag{3}$$

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 Generation of Certificates in Abstraction-Carrying Code

This section recalls ACC in the context of (C)LP [1]. For concreteness, we build on the algorithms of `CiaoPP` [10], the abstract interpretation-based preprocessor of the `Ciao` multi-paradigm CLP system.

3.1 The Analysis Algorithm

Algorithm 1 has been presented in [9] as a generic description of a fixed-point algorithm which generalizes those used in generic analysis engines, such as the one in `CiaoPP` [10]. 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 *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 from S_α in the algorithms. The analyzer constructs an *and–or graph* [4] (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 of [9] by means of two data structures, the *answer table* and the *dependency arc table*. 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. The answer table contains entries of the form $A : CP \mapsto AP$ where A is always a base form. 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. As we will see below, dependency arcs are used for forcing recomputation until a fixed-point is reached.

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, we use a priority queue. Thus, the third, and final, structure used in the algorithm is a *prioritized event queue*. In the following, we use $\Omega \in QHS$ to refer to a *Queue Handling Strategy* which a particular instance of the generic algorithm may use. Clearly, Ω determines the graph traversal strategy and it is a parameter of the algorithm. 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.

Algorithm 1 Generic Analyzer for Abstraction-Carrying Code

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

– *updated*$(A : CP)$ which indicates that the answer description to call pattern A with description CP has been changed.

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.

Apart from the parametric description domain-dependent functions, the algorithm has several other undefined functions. The functions add_event and next_event respectively add an event to the priority queue and return (and delete) the event of highest priority, according to Ω. The function initial_guess returns an initial guess for the answer to a new call pattern. The default value is $\bot$ but if the call pattern is more general than an already computed call then the answer value for the more particular call can be returned.[4]

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 the rules in the definition of the internal literal A, by adding arc events for each of the first literals of these rules (L13), and determines an initial answer for the call pattern (L14) and places this in the table (L16). The function 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 (L21), the arc is added to the dependency arc table (L22). Constraints are simply added to the current description (L32). Literals are processed by function get_answer. In this function, the current answer to that literal for the current description is looked up (L33). The function lookup_answer first looks up an answer for the given call pattern in the answer table (L37) 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 (L34) and conjoined with the current description (L35). The resulting answer (L24) 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 literal (L25); otherwise the new answer for the rule is combined with the current answer in insert_answer_info (L30). Finally, insert_answer_info, updates the answer table entry when a new answer is found (L46). The function add_dependent_rules adds arc events for each dependency arc which depends on the call pattern $A : CP$ for which the answer has been updated. More details on the algorithm can be found in [9, 17].[5]

3.2 Running Example

Our running example is the program `rectoy` taken from [19]. 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* [7] 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 $p(x_1, \ldots, x_n)$ is shown as a tuple $\langle t_1, \ldots, t_n \rangle$, 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 $\bot$ assigns the empty set of values to each variable.

Example 1. Consider the `Ciao` version of procedure `rectoy` [19] and the call pattern `rectoy(N,M)` : $\langle$`int`, `term`$\rangle$ which indicates that external calls to `rectoy` are performed with an integer value in the first argument.

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

We now distinguish four main steps carried out in the analysis using some $\Omega \in QHS$:

[4] This case is only possible for calls to ANALYZE_F with the answer table not empty.
[5] It is also illustrated in Appendix A through an example for reviewing purposes.

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

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

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

B. Assume that Ω has assigned higher priority to $A_{1,1}$. The procedure get_answer simply adds the constraint $\texttt{N} = \texttt{0}$ to the description $\{\texttt{N}/\texttt{int}\}$. Upon return, as it is not the last body atom (L25), the following arc event is generated:

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

Arc $A_{1,2}$ is handled exactly as $A_{1,1}$ and get_answer simply adds the constraint $\texttt{M} = \texttt{0}$, returning $\{\texttt{N}/\texttt{int}, \texttt{M}/\texttt{int}\}$. As it is the last atom in the body (L28), procedure insert_answer_info computes Alub between $\bot$ and the above answer and overwrites E_1 with $\boxed{E_1' \mid \texttt{rectoy}(\texttt{N},\texttt{M}) : \langle \texttt{int}, \texttt{term}\rangle \mapsto \langle \texttt{int}, \texttt{int}\rangle}$. Therefore, the event U_1 : $\texttt{updated}(\texttt{rectoy}(\texttt{N},\texttt{M}) : \langle \texttt{int}, \texttt{term}\rangle)$ is introduced in the queue. Note that no dependency has been originated during the processing of this rule (as both body atoms are constraints).

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 have to assume that predefined functions "−", "+" and "is" are dealt by the algorithm as standard constraints (see [2] for further details) by just using the following information provided by the system:

E_2	$\texttt{C is A} + \texttt{B} : \ \langle \texttt{int}, \texttt{int}, \texttt{term}\rangle \mapsto \langle \texttt{int}, \texttt{int}, \texttt{int}\rangle$
E_3	$\texttt{C is A} - \texttt{B} : \ \langle \texttt{int}, \texttt{int}, \texttt{term}\rangle \mapsto \langle \texttt{int}, \texttt{int}, \texttt{int}\rangle$

In particular, after analyzing the subtraction with the initial call pattern, we infer that N1 is of type int and no dependency is asserted. Next, the arc:

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

is introduced in the queue and the corresponding dependency is stored in the dependency arc table. The call to get_answer returns the current answer for E_1'. Using this answer we get the arc $A_{2,3}$:

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

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 it in the queue. This relaunches an arc identical to $A_{2,2}$. This 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 is 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. 2. □

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 1 (full certificate). *We define function* CERTIFIER_F*:*$Prog \times ADom \times AAtom \times AInt \times QHS \mapsto ACert$ *which takes* $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $I_\alpha \in AInt$, $\Omega \in QHS$ *and returns as* full certificate, $\mathsf{FCert} \in ACert$, *the answer table computed by* ANALYZE_F(S_α, Ω) *for* P *in* D_α *iff* $\mathsf{FCert} \sqsubseteq I_\alpha$.

Example 2. Consider the safety policy expressed by the following specification I_α : $\texttt{rectoy(N,M)} : \langle \texttt{int}, \texttt{term} \rangle \mapsto \langle \texttt{int}, \texttt{real} \rangle$. The certifier in Def. 1 returns as valid certificate the single entry E_1'. Clearly $E_1' \sqsubseteq I_\alpha$. We assume that predefined information is available for the consumer, otherwise entries E_2 and E_3 should be included in the certificate. □

4 Abstraction-Carrying Code with Reduced Certificates

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 $\mathsf{updated}(A : CP)$ 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 $A : CP$. 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.

4.1 The Notion of Reduced Certificate

According to the above intuition, we are interested in determining when an entry in the answer table has been "updated" during analysis. 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 are similar to event U_1 generated in step B of Ex. 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 2 (redundant update). *Let* $P \in Prog$, $S_\alpha \in AAtom$ *and* $\Omega \in QHS$. *We say that an event* $\mathsf{updated}(A : CP)$ *which appears in the event queue during the analysis of* P *for* S_α *is* redundant *w.r.t.* Ω *iff* $DAT|_{A:CP} = \emptyset$.

It should be noted that redundant updates can only be generated by updated events for call patterns which belong to S_α. Otherwise, $DAT|_{A:CP}$ cannot be empty.

Example 3. 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. There is a matching dependency when U_1 is actually handled by procedure add_dependent_rules. This causes the unnecessary re-computation of the second arc for `rectoy` ($A_{2,2}$). In the following section, we propose a slight modification to the analysis algorithm so that redundant updates are executed as soon as they appear. □

Proposition 1. *Let $\Omega \in QHS$. Let $\Omega' \in QHS$ be a strategy which assigns the highest priority to any updated event which is redundant. Then, $\forall\ P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$,* $\textsc{Analyze\_f}(S_\alpha, \Omega) = \textsc{Analyze\_f}(S_\alpha, \Omega')$.

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 3 (initial update). *In the conditions of Def. 2, 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 [17]. 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. Suppose that we use a strategy $\Omega'' \in QHS$ such that step C in Ex. 1 is performed before B. Then, when the answer for $\mathtt{rectoy(N1,R)} : \langle \mathtt{int}, \mathtt{term} \rangle$ is looked up, procedure get_answer returns $\bot$ and thus the processing of arc $A_{2,2}$ is *suspended* at this point (see L25 in Algorithm 1). Next, we proceed with the remaining arc $A_{1,1}$ which is processed exactly as in step B. Now, 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. □

Relevant updates which are neither redundant nor initial force (re)computation.

Definition 4 (relevant update). *In the conditions of Def. 2, we say that an event* $\mathsf{updated}(A : CP)$ *which appears in the event queue during the analysis of P for S_α is* relevant *for Ω if it is not initial nor relevant for Ω.*

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

Definition 5 (relevant entry). *In the conditions of Def. 2, 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 6 (reduced certificate). *In the conditions of Def. 2 and let* $\mathsf{FCert} = \textsc{Analyze\_f}(S_\alpha, \Omega)$ *for P and S_α. We define the* reduced certificate, RCert, *as the set of relevant entries in* FCert *w.r.t. Ω.*

Example 5. 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 $\boxed{E_1'}\boxed{\mathtt{rectoy(N,M)} : \langle \mathtt{int}, \mathtt{term} \rangle \mapsto \langle \mathtt{int}, \mathtt{int} \rangle}$ in Example 1 is not relevant for Ω' 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. In the next section, we show

that our checker is able to reconstruct the fixpoint in a single pass from the empty certificate.[6] □

For function `rectoy` in Example 1, lightweight bytecode verification [19] 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 program point 30, the stack does not contain information (first occurrence of ϵ),[7] 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 point 30 variable R is undefined, if the *if* condition does not hold. As showed in the above example, our approach improves on state-of-the-art PCC techniques by reducing the certificate even further while still keeping the checking process one-pass.

4.2 Generation of Certificates without Irrelevant Entries

In Algorithm 2, we instrument the analyzer of Algorithm 1 with the extensions necessary for producing reduced certificates, according to Def. 6. Let us briefly explain the analysis algorithm for reduced certificates, ANALYZE_R. Essentially, we associate to each entry in the answer table a new field with the boolean u whose purpose is to indicate whether the entry is relevant. Now, an entry in the answer table is of the form $A(u) : CP \mapsto AP$. In the algorithm, we still use the previous notation $A : CP \mapsto AP$ while we access the field u by using function get_from_answer_table and procedure set_in_answer_table. A call u=get_from_answer_table$(A : CP)$ looks up in the answer table the entry for $A : CP$ and returns its u-value. A call set_in_answer_table$(A(u) : CP \mapsto AP)$ replaces the entry for $A : CP$ with the new one $A(u) : CP \mapsto AP$. If such entry does not exist, then it simply adds it. All entries initially have the value False for u, i.e., L16 of ANALYZE_F is replaced by "16: set_in_answer_table$(A(false) : CP \mapsto AP)$" in Algorithm 2. The remaining procedures remain identical except for insert_answer_info whose new definition appears in Algorithm 2. The characterization of relevant update is performed in this procedure as follows. L5 is in charge of removing redundant updates. The case of initial updates is captured in L10. L13 allows us to identify relevant updates.

Algorithm 2 Analyzer ANALYZE_R

```
 1: procedure INSERT_ANSWER_INFO(H : CP ↦ AP, Ω)
 2:     AP_0 := lookup_answer(H : CP, Ω)
 3:     AP_1 := Alub(AP, AP_0)
 4:     if AP_0 ≠ AP_1 then
 5:         if DAT|_{H:CP} = ∅ then   % redundant updates
 6:             u=get_from_answer_table(H : CP)
 7:             set_in_answer_table(H(u) : CP ↦ AP_1)
 8:         else
 9:             add_event(updated(H : CP), Ω)
10:             if AP_0 = ⊥ then   % initial updates
11:                 u=get_from_answer_table(H : CP)
12:                 set_in_answer_table(H(u) : CP ↦ AP_1)
13:             else set_in_answer_table(H(true) : CP ↦ AP_1)   % relevant updates
```

[6] It should be noted that, using Ω as in Example 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.

[7] The second occurrence of ϵ indicates that there are no backwards jumps.

Example 6. Consider the four steps performed in the analysis of our running example. In step A, the answer E_1 is initialized with u equal to *false* in L16. Then, in step B, the procedure insert_answer_info checks that the condition in L5 holds. Therefore, the entry is updated with the new answer but its u status does not change and it is considered non relevant (in fact the update is redundant). Both steps C and D do not satisfy the condition in L4. Hence, upon return, the status of u for E_1' is still False.□

Proposition 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 is true.*

Note that, except for the control of relevant entries, Analyze_f(S_α, Ω) and Analyze_r(S_α, Ω) have the same behaviour, they compute the same answer table (see Proposition 3 in Appendix B). When the analysis terminates, in order to obtain the reduced certificate, we use function remove_irrelevant_answers which takes a set of answers of the form $A(u) : CP \mapsto AP \in$ FCert and returns, RCert, the set of answers $A : CP \mapsto AP$ such that u is *true*.

Definition 7. *We define the function* Certifier_r*: $Prog \times ADom \times AAtom \times AInt \times QHS \mapsto ACert$, which takes $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $I_\alpha \in AInt$, $\Omega \in QHS$. It returns as certificate,* RCert=remove_irrelevant_answers(FCert)*, where* FCert=Analyze_r*(S_α, Ω), iff* FCert $\sqsubseteq I_\alpha$.

5 Checking Reduced Certificates

In the ACC framework for full certificates, the checking algorithm [1] 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).[8] 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. Unfortunately, this 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 receiving the appropriate parameters from the supplier, the consumer uses the particular instance of the generic checker resulting from application of such parameters.

It should be noted that the design of generic checkers is also relevant in light of current trends in verified analyzers (e.g., [11, 5]), which could be transferred directly to the checking end. In particular, since the design of the checking process is generic, it becomes feasible in ACC to use automatic program transformers to specialize a certified (specific) analysis algorithm in order to obtain a certified checker with the same strategy while preserving correctness and completeness.

5.1 The Generic Checking Algorithm

The following definition presents a generic checker for validating reduced certificates. In addition to the genericity issue discussed above, a important difference with the

[8] Note, however, 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 already available domains, being one of the fundamental advantages of ACC.

checker for full certificates [1] is that there are certain entries which are not available in the certificate and that we want to reconstruct and output in checking. The reason for this is that the safety policy has to be tested w.r.t. the full answer table –Equation (2). Therefore, the checker must reconstruct, from RCert, the answer table returned by ANALYZE_F, FCert, in order to test for adherence to the safety policy –Equation (4). The checker can identify two sources of errors: a) a relevant update is needed to obtain an answer (cannot be obtained in one pass), b) the answer in the certificate is more precise than the one obtained by the checker (the certificate and program at hand do not correspond to each other). In both cases, the checker has to issue Error. Note that reconstructing the answer table does not add any additional cost compared to the checker in [1], since the full answer table also has to be created in [1]. In the definition below, we start from ANALYZE_F rather than from ANALYZE_R because the instrumentation for classifying updated events is not needed for the purpose of checking.

Definition 8 (checker for reduced certificates). *Function* CHECKING_R *is defined as function* ANALYZE_F *with the following modifications:*

1. *It receives* RCert *as an additional input parameter.*
2. *It mail fail to produce an answer table. In that case it issues an* Error.
3. *It uses the trivial* initial_guess *function which returns* $\bot$ *for any call pattern.*
4. *Function* insert_answer_info *is replaced by the new one in Fig 3.*

Algorithm 3 Generic Checker for Reduced Certificates CHECKING_R

```
1: procedure INSERT_ANSWER_INFO(H : CP ↦ AP, Ω, RCert)
2:     AP_0 := lookup_answer(H : CP, Ω)
3:     (IsIn,AP')=look_fixpoint(H : CP,RCert)
4:     if AP_0 = ⊥ then process_initial_update(IsIn,AP',H : CP ↦ AP,Ω)
5:     else if IsIn then          % case 1.2
6:         if Alub(AP, AP_0) ≠ AP' then return Error     % error type b) second update
7:     else          % case 2.2
8:         AP_1 := Alub(AP, AP_0)
9:         if DAT|_{H:CP} = ∅ then  add H : CP ↦ AP_1 to answer table
10:        else if AP_0 ≠ AP_1 then  return Error        % relevant update, error type a)

11: function LOOK_FIXPOINT(A : CP,RCert)
12:     if ∃ a renaming σ such that σ(A : CP ↦ AP) ∈ RCert then returns (True,σ^{-1}(AP))
13:     else return (False,⊥)

14: function PROCESS_INITIAL_UPDATE(IsIn,AP',H : CP ↦ AP, Ω)
15:     if IsIn then          % case 1.1
16:         if Alub(AP, AP') ≠ AP' then return Error      % error type b) for first update
17:         else  add H : CP ↦ AP' to answer table
18:     else  add H : CP ↦ AP to answer table          % case 2.1
19:     if DAT|_{H:CP} ≠ ∅ and AP ≠ ⊥ then add_event(updated(H : CP), Ω)
```

Function CHECKER_R *takes* $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $\Omega \in QHS$, RCert $\in ACert$ *and returns the result of* CHECKING_R$(S_\alpha, \Omega,$ RCert*) for* P *in* D_α.

Let us briefly explain Algorithm 3. We distinguish four main cases in the algorithm by combining: 1) whether there is an entry for the call pattern in the certificate or not, 2) if it is the first time the call pattern is processed (initial update) or not. We use function look_fixpoint for finding out 1). A call look_fixpoint($H : CP$,RCert) returns a tuple $(IsIn, AP')$ such that: if $H : CP$ is in RCert, then $IsIn$ is equal to True and AP' returns the fixpoint stored in RCert. Otherwise, $IsIn$ is equal to False and AP' is $\bot$.

IsIn=true. There are two cases which are distinguished in L4:

1.1. *Initial update.* This corresponds to the case when $AP_0 = \bot$ in L4. Now process_initial_update is executed. L16 checks if the first computed solution AP for $H : CP$ is more general than AP' (answer in RCert). This case corresponds to an Error of type b). Otherwise, L17 introduces in the answer table the fixpoint for $H : CP$. Finally, if the initial update is non redundant (L19), it is launched in order to awake suspended arcs in the dependency arc table.

1.2. *Next updates.* Only L6 is executed to check whether the fixpoint of $H : CP$ (already stored in the answer table) is more general or equal than the partial computed solution AP. Otherwise, an Error type b) is issued.

IsIn=false. The same cases as above are distinguished in L4:

2.1. *Initial update.* In this case, process_initial_update stores AP as first answer for $H : CP$ (L18), and the corresponding initial update event is launched (L19).

2.2. *Next updates.* After this first one, the next partial solutions cannot generate relevant updates or the algorithm issues an Error (L10). This ensures that the checking process is done in a single pass. If no relevant updates are generated, a new answer for $H : CP$ is computed (L8-9).

Note that, although CHECKER_R has information in RCert about certain answers, function initial_guess does not consult it and instead introduces $\bot$ initially. The reason is that initial updates are needed in order to awake the possibly suspended computations which depend on the answers not available in RCert. This does not occur in the checking of [1] because certificates are full and suspension never occurs.

Example 7. Following Example 1, we assume that an empty certificate is sent along with the untrusted code. Let us compare the fours steps performed in the analysis (see Example 6) with the corresponding CHECKER_R which uses the same QHS:

A. This step is executed identically by the checker. E_1 is inserted.
B. The difference in this step is that the redundant update U_1 is not generated because in L10 the checker realizes that there are no dependencies for its call.
C. Exactly the same events and the same answer as computed in step C of analysis occur here.
D. This step is not performed because U_1 is not in the queue.

Notice that, as expected, the reprocessing carried out in step D of analysis is not performed during checking and, hence, the program is validated in a single pass over the analysis graph. □

5.2 Completeness and Correctness Results

The following theorem ensures that if CHECKER_R validates a certificate (i.e., it does not return Error), then the re-constructed answer table is a fixpoint. This implies that any certificate which gets validated by the checker is indeed a valid one.

Theorem 1 (correctness). *Let $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $I_\alpha \in AInt$ and $\Omega_A, \Omega_C \in QHS$. Let* FCert= CERTIFIER_F*$(P, D_\alpha, S_\alpha, I_\alpha, \Omega_A)$ and* RCert= CERTIFIER_R*$(P, D_\alpha, S_\alpha, I_\alpha, \Omega_A)$. If* CHECKER_R*$(P, D_\alpha, S_\alpha, I_\alpha,$* RCert*$, \Omega_C)$ does not issue an* Error*, then it returns* FCert.

As already mentioned, using a CHECKER_R with a strategy Ω_C, we may fail to validate a reduced certificate which is indeed valid for the program, though reduced w.r.t. another strategy $\Omega_A \neq \Omega_C$. This is illustrated in the example below.

Example 8. Consider the analysis of the following simple program w.r.t. the call pattern q(X) : $\top$ using the same domain as for the running example.

```
q(X) :- p(X).          p(X) :- X = 1.0.
                       p(X) :- X = 1.
```

We will consider two different strategies $\Omega_1 \neq \Omega_2$. Under both strategies, we start by processing the single rule for `q`. As a result, the event $\mathsf{newcall}(\mathtt{p(X)} : \langle term \rangle)$ is generated, and $\mathtt{q(X)} : \langle \mathtt{term} \rangle \mapsto \bot$ is added to the answer table. Also, an entry for `q` is introduced in the dependency arc table. Now, using Ω_1, the analyzer processes the two arcs for `p(X)` in the order in which the clauses are written. After traversing the first arc, the answer $\mathtt{p(X)} : \langle \mathtt{term} \rangle \mapsto \langle \mathtt{real} \rangle$ is inferred and an (initial) updated event is generated. The analysis of the second arc produces the answer $\langle \mathtt{int} \rangle$ and does not update the entry since $\mathsf{Alub}(\{\mathtt{X/real}\}, \{\mathtt{X/int}\})$ returns $\{\mathtt{X/real}\}$. After processing the initial update for `p`, the answer $\mathtt{q(X)} : \langle \mathtt{term} \rangle \mapsto \langle \mathtt{real} \rangle$ replaces the old one in the answer table, and the fixpoint is reached without any iteration (i.e., no relevant update for the call). Assume now that Ω_2 assigns a higher priority to the second arc of `p`. In this case, the answer for $\mathtt{p(X)} : \langle \mathtt{term} \rangle$ changes from $\bot$ to $\{\mathtt{X/int}\}$ (producing an initial update). When the first arc is processed the computed answer is $\{\mathtt{X/real}\}$. Now, a relevant update is needed since there is a dependency $\mathtt{q(X)} : \_ \Rightarrow [\_]\mathtt{p(X)} : \_$ in the dependency arc table.

Thus, the certificate reduced w.r.t. Ω_1 is empty, whereas the one reduced w.r.t. Ω_2 contains the single entry $\mathtt{p(X)} : \langle \mathtt{term} \rangle \mapsto \langle \mathtt{real} \rangle$. Since, as seen above, the strategy Ω_2 performs a relevant update for the call pattern $p(X) : \langle term \rangle$, a checker using Ω_2 will issue an error when trying to validate the program if provided with the empty certificate. □

The following theorem (completeness) provides sufficient conditions under which a checker is guaranteed to validate reduced certificates which are actually valid.

Theorem 2 (completeness). *Let $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $I_\alpha \in AInt$ and $\Omega_a \in QHS$. Let* $\mathsf{FCert} = \textsc{Certifier\_f}(P, D_\alpha, S_\alpha, I_\alpha, \Omega_a)$ *and* $\mathsf{RCert}_{\Omega_a} = \textsc{Certifier\_r}(P, D_\alpha, S_\alpha, I_\alpha, \Omega_a)$*. Let $\Omega_c \in QHS$ be such that* $\mathsf{RCert}_{\Omega_c} = \textsc{Certifier\_r}(P, D_\alpha, S_\alpha, I_\alpha, \Omega_c)$ *and* $\mathsf{RCert}_{\Omega_a} \supseteq \mathsf{RCert}_{\Omega_c}$*. Then,* $\textsc{Checker\_r}(P, D_\alpha, S_\alpha, I_\alpha, \mathsf{RCert}_{\Omega_a}, \Omega_c)$ *returns* FCert *and does not issue an* Error*.*

Obviously, if $\Omega_c = \Omega_a$ then the checker is guaranteed to be complete. Additionally, a checker using a different strategy Ω_c is also guaranteed to be complete as long as the certificate reduced w.r.t Ω_c is equal to or smaller than the certificate reduced w.r.t Ω_a. Furthermore, if the certificate used is full, the checker is complete for any strategy.

6 Conclusions

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 without increasing checking time. As we have illustrated throughout the paper, the size of the certificate is directly related to the amount of *updates* (or iterations) performed during analysis. Clearly, depending on the the 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 [17]. Interestingly, our framework greatly benefits from all these advances, since *the more efficient analysis is, the smaller the corresponding reduced certificates are.* In 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 plan to incorporate our new framework for generating and checking reduced certificates in the `CiaoPP` preprocessor and to study different certificate size vs. checking time trade-offs. We also want to investigate ways of reducing the trusted base code (see, e.g., [3, 16]) in ACC.

References

1. 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.
2. E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code: A Model for Mobile Code Safety. Technical Report CLIP7/2005.0, Technical University of Madrid, School of Computer Science, UPM, July 2005.
3. A. Appel and A. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. of POPL'00*. ACM Press, 2000.
4. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
5. D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a Data Flow Analyser in Constructive Logic. In *Proc. of ESOP 2004*, volume LNCS 2986, pages 385 – 400, 2004.
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. T. Früwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. LICS'91*, pages 300–309, 1991.
8. M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction Carrying Code and Resource-Awareness. In *Proc. of PPDP'05*. ACM Press, July 2005.
9. 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.
10. Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2), 2005.
11. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 3(298):583–626, 2003.
12. Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.
13. J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.
14. Kim Marriot and Peter Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.
15. G. Necula. Proof-Carrying Code. In *Proc. of POPL'97*, pages 106–119. ACM Press, 1997.
16. G. Necula and R. Schneck. A Sound Framework for Untrustred Verification-Condition Generators. In IEEE Computer Society, editor, *Proc. of LICS03*, pages 248–260, 2003.
17. 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.
18. E. Rose and K. Rose. Java access protection through typing. *Concurrency and Computation: Practice and Experience*, 13(13):1125–1132, 2001.
19. K. Rose, E. Rose. Lightweight bytecode verification. In *OOPSALA Workshop on Formal Underpinnings of Java*, 1998.

A Analysis Graph for the Running Example

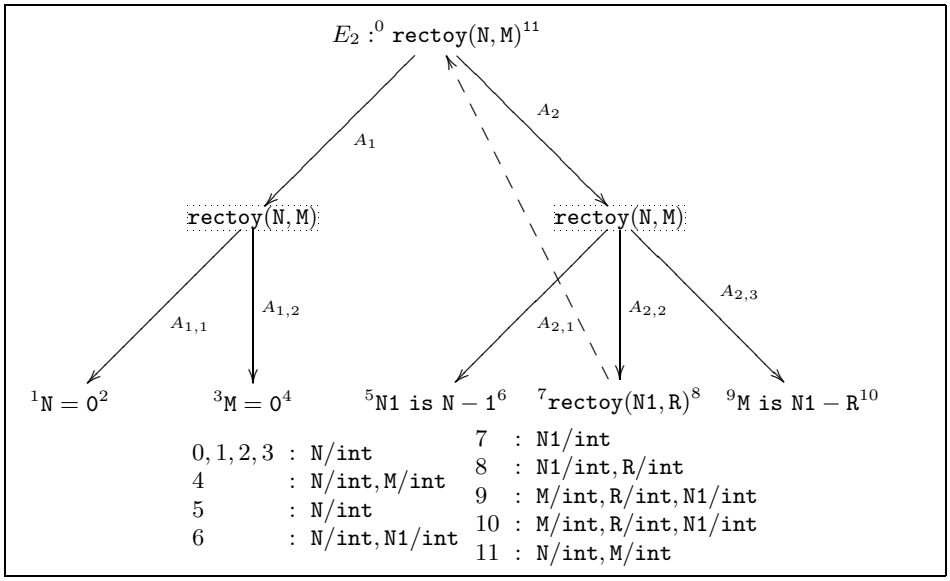

Fig. 1. Analysis Graph for our Running Example

Figure 1 shows the analysis graph for our running example. The graph has two sorts of nodes. Those which correspond to atoms are called "OR-nodes." An OR-node of the form ${}^{CP}A^{AP}$ is interpreted as: the answer for the call pattern $A : CP$ is AP. For instance, the OR-node

$$^{\{\texttt{N1/int}\}}\texttt{rectoy(N1,R)}^{\{\texttt{N1/int,R/int}\}}$$

indicates that, when the atom rectoy(N1,R) is called with description $\langle$int, term$\rangle$, the answer computed is $\langle$int, int$\rangle$. Those nodes which correspond to rules are called "AND-nodes." In the figure above, they appear within a dotted box and contain the head of the corresponding clause. Each AND-node has as children as many OR-nodes as atoms there are in the body. If a child OR-node is already in the tree, it is not expanded any further and the currently available answer is used. For instance, the analysis graph in the figure above contains two occurrences of the abstract atom rectoy(N,M) : $\langle$int, term$\rangle$ (modulo renaming), but only one of them (the root) has been expanded. This is depicted by a dashed arrow from the non-expanded occurrence to the expanded one.

The answer table contains entries for the different OR-nodes which appear in the graph. For instance, there exists an entry of the form

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

associated to the (root) OR-node discussed above, which is in fact the fixpoint. Dependencies in the dependency arc table indicate direct relations among OR-nodes. An OR-node $A_F : CP_F$ *depends on* another OR-node $A_T : CP_T$ iff the OR-node

$A_T : CP_T$ appears in the body of some clause for $A_F : CP_F$. For instance, the dependency set for the abstract atom $\texttt{rectoy(N,M)} : \langle \texttt{int}, \texttt{term} \rangle$ is

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

It indicates that the OR-node $\texttt{rectoy(N1,R)} : \langle \texttt{int}, \texttt{term} \rangle$ is only used in the OR-node $\texttt{rectoy(N,M)} : \langle \texttt{int}, \texttt{term} \rangle$. Thus, if the answer pattern for $\texttt{rectoy(N1,R)} : \langle \texttt{int}, \texttt{term} \rangle$ is ever updated, then we must reprocess the OR-node $\texttt{rectoy(N,M)} : \langle \texttt{int}, \texttt{term} \rangle$.

B Proofs of Correctness and Completeness

B.1 Results for Analysis

Proposition 1. *Let $\Omega \in QHS$. Let $\Omega' \in QHS$ be a strategy which assigns the highest priority to any updated event which is redundant. Then, $\forall\ P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$,* Analyze_f*(S_α, Ω) =* Analyze_f*(S_α, Ω').*

Proof. Analyze_f is correct independently of the order in which events in the priority queue are processed [9]. Thus, any redundant event $\mathsf{updated}(A : CP)$ can be processed as soon as it is inserted in the queue. In such a case, there are no arcs ($DAT|_{A:CP} = \emptyset$) in the dependency arc table applicable to the event. Therefore, no action is taken. Consequently, redundant updates are not necessary for constructing the result. □

Proposition 2. *Let $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $\Omega \in QHS$. Let AT be the answer table computed by* Analyze_r*(S_α, Ω) for P in D_α. Then, an entry $A(u) : CP_A \mapsto AP \in AT$ is relevant iff u is true.*

Proof. From the definition of function insert_answer_info, it holds trivially that $A(u) : CP_A \mapsto AP$ verifies that u is *true* iff some event $\mathsf{updated}(B : CP_B)$ (generated by an update with an answer different from $\bot$) occurred with an arc $A : CP_A \Rightarrow \_\ B : CP_B$ (modulo renaming) in the dependency arc table. Therefore, by Def. 2, it is not a redundant update as its DAT is not empty. Similarly, by Def. 3, it is not an initial update since the answer is not $\bot$. Hence, by Def. 4, it is relevant. As a consequence, by Def. 5, the entries for $A(u) : CP_A : CP \mapsto AP$ with $u \equiv true$ are relevant. □

Proposition 3. *Let $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $\Omega, \Omega' \in QHS$. Then,* Analyze_f*(S_α, Ω)=*Analyze_r*(S_α, Ω').*

Proof. By the correctness of Analyze_f [9], we know that Analyze_f(S_α, Ω) computes the same answer table than Analyze_f(S_α, Ω'). Then, Analyze_f and Analyze_r differ only on the instrumentation for detecting relevant solutions and the absence of redundant updates which, by Proposition 1, does not affect the computation of the final answer table. Hence, Analyze_f(S_α, Ω') computes the same answer table than Analyze_r(S_α, Ω') and the claim follows. □

B.2 Results for Checking

Theorem 1 (Correctness). *Let $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $I_\alpha \in AInt$ and $\Omega_A, \Omega_C \in QHS$. Let* FCert= Certifier_f*$(P, D_\alpha, S_\alpha, I_\alpha, \Omega_A)$ and* RCert= Certifier_r*$(P, D_\alpha, S_\alpha, I_\alpha, \Omega_A)$. If* Checker_r*$(P, D_\alpha, S_\alpha, I_\alpha, \mathsf{RCert}, \Omega_C)$ does not issue an* Error*, then it returns* FCert*.*

Proof. If CHECKER_R$(P, D_\alpha, S_\alpha, I_\alpha, \mathsf{RCert}, \Omega)$ does not issue an Error, then it computes an answer table AT. Let us Assume the following three claims:

(1) If $A : CP \mapsto AP \in AT$ and FCert contains an entry for $A : CP$, then it holds that $A : CP \mapsto AP \in \mathsf{FCert}$.
(2) If there is an entry for $A_0 : CP_0$ in AT, then there is one for $A_0 : CP_0$ in FCert.
(3) If there is an entry for $A_0 : CP_0$ in FCert, then there is one for $A_0 : CP_0$ in AT.

Then, from (1) and (2) it holds $AT \subseteq \mathsf{FCert}$. From (3), it holds that $AT =$FCert. We now prove the three claims separately.

Claim (1) We assume that $A_0 : CP_0 \mapsto AP_0 \in AT$. We distinguish two cases. 1) If $A_0 : CP_0 \mapsto AP_0 \in \mathsf{RCert}$, since RCert$\subseteq$ FCert and RCert$\in AT$, then the result holds. 2) We now consider that $A_0 : CP_0 \mapsto AP_0$ belongs to $AT -$ RCert. We want to prove that $A_0 : CP_0 \mapsto AP_0 \in$ FCert by contradiction.

The assumption is that $A_0 : CP_0 \mapsto AP_0' \in$ FCert and $AP_0 \neq AP_0'$. This means that in some step of the checking process, we have evaluated an arc of the form:

$$A_0 : CP_0 \Rightarrow \ [\_]A_1 : CP_1$$

such that the final computed answer for $A_1 : CP_1$ in AT is p_1^c and FCert contains p_1^a as fixpoint for $A_1 : CP_1$, with $p_1^c \neq p_1^a$. Necessarily, $A_1 : CP_1 \notin$ RCert, and $A_1 : CP_1$ does not suffer updated event which is not initial (otherwise $A_1 : CP_1$ has to belong to RCert). The key is that, for this kind of call patterns, CHECKER_R behaves exactly equal to CERTIFIER_F$(P, D_\alpha, S_\alpha, I_\alpha, \Omega)$ if the answer table is initially with the contents of RCert. Moreover, by [9]:

$$\text{CERTIFIER\_F}(P, D_\alpha, S_\alpha, I_\alpha, \Omega') = \text{CERTIFIER\_F}(P, D_\alpha, S_\alpha, I_\alpha, \Omega)$$

Thus $p_1^c = p_1^a$, and hence $AP_0 = AP_0'$, which contradicts our assumption.

Claim (2) If $A_0 : CP_0 \in$ RCert or $A_0 : CP_0 \in S_\alpha$, then the result holds trivially. Let us assume that $A_0 : CP_0$ does not belong to these sets. We again reason by contradiction. The assumption is that $A_0 : CP_0$ does not have an answer in FCert. Consider $A_1 : CP_1 \in S_\alpha$ whose evaluation after newcall$(A_1 : CP_1)$ generates the event newcall$(A_0 : CP_0)$ in the checking process. Necessarily, during the processing of the event newcall$(A_1 : CP_1)$, the following two arcs must have existed:

$$A_2 : CP_2 \Rightarrow \_\ A_3 : CP_3$$
$$A_2 : CP_2 \Rightarrow \_\ A_0 : CP_0$$

corresponding to a rule A_2:- $\ldots, A_3, A_0, \ldots$, where either $A_3 : CP_3$ has an answer in FCert (different from the existing one in AT) or $A_3 : CP_3$ has no entry in FCert. The former case is impossible for (1). The latter case, reasoning identically, would lead to a similar situation. By iterating on the process, we obtain that $A_1 : CP_1$ has no entry in FCert. This is impossible because $A_1 : CP_1 \in S_\alpha$.

Claim (3) If $A_0 : CP_0$ has an entry in RCert, or belongs to S_α, then the result holds trivially. Otherwise, the claim follows by contradiction, similarly to case (2).

□

Theorem 2 (Completeness).

Let $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $I_\alpha \in AInt$ and $\Omega_a \in QHS$. Let FCert$=$ CERTIFIER_F$(P, D_\alpha, S_\alpha, I_\alpha, \Omega_a)$ and $\mathsf{RCert}_{\Omega_a} =$ CERTIFIER_R$(P, D_\alpha, S_\alpha, I_\alpha, \Omega_a)$. Let $\Omega_c \in QHS$ be s.t. $\mathsf{RCert}_{\Omega_c} =$ CERTIFIER_R$(P, D_\alpha, S_\alpha, I_\alpha, \Omega_c)$ and $\mathsf{RCert}_{\Omega_a} \supseteq \mathsf{RCert}_{\Omega_c}$. Then, the execution of CHECKER_R$(P, D_\alpha, S_\alpha, I_\alpha, \mathsf{RCert}_{\Omega_a}, \Omega_c)$ returns FCert and does not issue an Error.

Proof. The only cases in which CHECKER_R returns Error are the following:

- The partial answer AP computed for some calling pattern $A : CP$ (provided in $\mathsf{RCert}_{\Omega_a}$) causes that $\mathsf{Alub}(AP, AP') \neq AP'$, where AP' is the answer for $A : CP$, i.e., $A : CP \mapsto AP' \in \mathsf{RCert}_{\Omega_a}$. But, this is in contradiction with the assumption that $\mathsf{RCert}_{\Omega_a}$ is a valid reduced certificate for P.
- A calling pattern $A : CP$ not provided in $\mathsf{RCert}_{\Omega_a}$ produces the reprocessing of its dependent arcs due to some relevant update (L10 in Algorithm 3). But this is in contradiction with the assumption that $\mathsf{RCert}_{\Omega_a} \supseteq \mathsf{RCert}_{\Omega_c}$, since $\mathsf{RCert}_{\Omega_c}$ contains all relevant entries generated by CERTIFIER_R by using the strategy Ω_c.

Thus, CHECKER_R$(P, D, S, \mathcal{I}, \mathsf{RCert}_{\Omega_a}, \Omega_c)$ returns an answer table AT. Finally, by Theorem 1, we know that since no Error is issued, CHECKER_R returns FCert. □