Certificate Size Reduction in Abstraction-Carrying Code∗

ELVIRA ALBERT, PURI ARENAS

School of Computer Science, Complutense University of Madrid
E28040-Profesor José García Santesmases, s/n, Madrid, Spain
(*e-mail:* {elvira,puri}@sip.ucm.es)

GERMÁN PUEBLA[1], MANUEL HERMENEGILDO[1,2]

[1] *School of Computer Science, Technical University of Madrid*
E28660-Boadilla del Monte, Madrid, Spain
(*e-mail:* {german,herme}@fi.upm.es)
[2] *Madrid Institute for Advanced Studies*
in Software Development Technology (IMDEA Software)
Madrid,Spain
(*e-mail:* manuel.hermenegildo@imdea.org)

submitted 19 September 2007; revised 16 Octorer 2009; 27 May 2010; accepted 6 October 2010

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 fixpoint analyzer. The advantage of providing a (fixpoint) 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 the information relevant to the checker. Interestingly, the fact that the reduced certificate omits (parts of) the abstraction has implications in the design of the checker. We provide the sufficient conditions which allow us to ensure that 1) if the checker succeeds in validating the certificate, then the certificate is valid for the program (correctness) and 2) the checker will succeed for any reduced certificate which is valid (completeness). Our approach has been implemented and benchmarked within the `CiaoPP` system. The experimental results show that our proposal is able to greatly reduce the size of certificates in practice.
To appear in Theory and Practice of Logic Programming (TPLP).

KEYWORDS: Proof-Carrying Code. Abstraction-Carrying Code. Static Analysis. Reduced Certificates.

∗ A preliminary version of this work appeared in the Proceedings of ICLP'06 (Albert et al. 2006).

1 Introduction

Proof-Carrying Code (PCC) (Necula 1997) 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 which 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 task of the consumer is reduced from the level of proving to the level of checking, a procedure that should be much simpler, efficient, and automatic than generating the original certificate.

Abstraction-Carrying Code (ACC) (Albert et al. 2005; Albert et al. 2008) 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 carried out automatically by a fixpoint analyzer. In this article 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 fixpoint algorithms, we use the *generic* description of (Hermenegildo et al. 2000), 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* (Hermenegildo et al. 2000). 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 combined (by computing their lub) and the result is used 1) to *update* the table, and 2) to launch the recomputation of those calls whose answer depends on the answer currently computed. In the original ACC framework, the final *full* answer table constitutes the certificate. A main idea is that, since this certificate contains the fixpoint, a single pass over the analysis graph is sufficient to validate such certificate on 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 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 limited bandwidth and/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 (among other reasons because pervasive and embedded systems also suffer typically

from limited computing –and power– resources). 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 work is to characterize the smallest 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 main contributions of this article are:

1. The notion of *reduced certificate* which characterizes the subset of the abstraction which, for a given analysis graph traversal strategy, the checker needs in order to validate (and re-construct) the full certificate in a single pass.
2. An instrumentation of the generic abstract interpretation-based analysis algorithm of (Hermenegildo et al. 2000) with the necessary extensions in order to identify relevant information to the checker.
3. 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.
4. 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 for any reduced certificate which is valid.
5. An experimental evaluation of the effect of our approach on the `CiaoPP` system (Hermenegildo et al. 2005), the abstract interpretation-based preprocessor of the `Ciao` multi-paradigm (Constraint) Logic Programming system. The experimental results show that the certificate can be greatly reduced (by a factor of 3.35) with no increase in checking time.

Both the ACC framework and our work here are applied at the *source* level. In contrast, in existing PCC frameworks, the code supplier typically packages the certificate with the *object* code rather than with the *source* code (both are untrusted). Nevertheless, our choice of making our presentation at the source level is without loss of generality because both the original ideas in the ACC approach and those in our current proposal can also be applied directly to bytecode. Indeed, a good number of abstract interpretation-based analyses have been proposed in the literature for bytecode and machine code, most of which compute a fixpoint during analysis which can be reduced using the general principle of our proposal. For instance, in recent work, the concrete CLP verifier used in the original ACC implementation has itself been shown to be applicable without modification also to Java bytecode via a transformational approach, based on partial evaluation (Albert et al. 2007) or via direct transformation (Méndez-Lojo et al. 2007a) using standard tools such as Soot (Vallee-Rai et al. 1999). Furthermore, in (Méndez-Lojo et al. 2007b; Méndez-Lojo et al. 2007a) a fixpoint-based analysis framework has been developed specifically for Java bytecode which is essentially equivalent to that used in the ACC proposal and to the one that we will apply in this work on the producer side to perform the analysis and verification. This supports the direct applicability of our approach to bytecode-based program representations and, in general, to other languages and paradigms.

The rest of the article is organized as follows. The following section presents a

general view of ACC. Section 3 gives a brief overview of our method by means of a simple example. Section 4 recalls the certification process performed by the code supplier and illustrates it with a running example. Section 5 characterizes the notion of reduced certificate and instruments a generic certifier for its generation. Section 6 presents a generic checker for reduced certificates together with correctness and completeness results. Finally, Section 7 discusses some experimental results and related work.

2 A General View of Abstraction-Carrying Code

We assume the reader familiar with abstract interpretation (see (Cousot and Cousot 1977)) and (Constraint) Logic Programming (C)LP (see, e.g., (Marriot and Stuckey 1998) and (Lloyd 1987)).

An abstract interpretation-based certifier is a function $\mathsf{certifier} : Prog \times ADom \times APol \mapsto ACert$ which for a given program $P \in Prog$, an abstract domain $\langle D_\alpha, \sqsubseteq \rangle \in ADom$ and a safety policy $I_\alpha \in APol$ 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 essential idea in the certification process carried out in ACC is that a fixpoint static analyzer is used to automatically infer an abstract model (or simply *abstraction*) about the mobile code which can then be used to prove that the code is safe w.r.t. the given policy in a straightforward way. The basics for defining the abstract interpretation-based certifiers in ACC are summarized in the following four points and equations.

Approximation. We consider a *description (or abstract) domain* $\langle D_\alpha, \sqsubseteq \rangle \in ADom$ and its corresponding *concrete domain* $\langle 2^D, \subseteq \rangle$, both with a complete lattice structure. Description (or abstract) values and sets of concrete values are related by an *abstraction* function $\alpha : 2^D \to D_\alpha$, and a *concretization* function $\gamma : D_\alpha \to 2^D$. The pair $\langle \alpha, \gamma \rangle$ forms a Galois connection. The concrete and abstract domains must be related in such a way that the following condition holds (Cousot and Cousot 1977):

$$\forall x \in 2^D, \forall y \in D_\alpha : \ (\alpha(x) \sqsubseteq y) \Longleftrightarrow (x \subseteq \gamma(y))$$

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.

Abstraction generation. We consider the class of *fixpoint 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., $[\![P]\!]$=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 use an operator S_P^α which works in the abstract domain and which is the abstract counterpart of S_P. This operator induces the abstract meaning of the program, which we refer to as $[\![P]\!]_\alpha$. Now, again, starting from the bottom element of the lattice we can

obtain the least fixpoint of S^{α}_{P}, denoted lfp(S^{α}_{P}), and we define $[\![P]\!]_{\alpha}$=lfp(S^{α}_{P}). Correctness of analysis (Cousot and Cousot 1977) ensures that $[\![P]\!]_{\alpha}$ safely approximates $[\![P]\!]$, i.e., $[\![P]\!] \in \gamma([\![P]\!]_{\alpha})$. In actual analyzers, it is often the case that the analysis computes a post-fixpoint of S^{α}_{P}, which we refer to as $Cert_{\alpha}$, instead of the least fixpoint. The reason for this is that computing the least fixpoint may require a too large (even infinite) number of iterations. An analyzer is a function $\mathsf{analyzer} : Prog \times ADom \mapsto ACert$ such that:

$$\mathsf{analyzer}(P, D_{\alpha}) = Cert_{\alpha} \wedge S^{\alpha}_{P}(Cert_{\alpha}) = Cert_{\alpha} \tag{1}$$

Since $[\![P]\!]_{\alpha} \sqsubseteq Cert_{\alpha}$, $Cert_{\alpha}$ is a safe approximation of $[\![P]\!]$.

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:

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

Certification. Together, Equations (1) and (2) define a certifier which provides program fixpoints, $Cert_{\alpha}$, as certificates which entail a given safety policy, i.e., by taking $Cert_{\alpha} = \mathsf{analyzer}(P, D_{\alpha})$.

The second main idea in ACC is that a simple, easy-to-trust abstract interpretation-based checker verifies the validity of the abstraction on the mobile code. The checker is defined as a specialized abstract interpreter whose key characteristic is that it does not need to iterate in order to reach a fixpoint (in contrast to standard analyzers). The basics for defining the abstract interpretation-based checkers in ACC are summarized in the following two points and equations.

Checking. If a certificate $Cert_{\alpha}$ is a fixpoint of S^{α}_{P}, then $S^{\alpha}_{P}(Cert_{\alpha}) = Cert_{\alpha}$. Thus, 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^{α}_{P} or not:

$$\mathsf{checker}(P, D_{\alpha}, Cert_{\alpha}) \textit{ returns true iff } (S^{\alpha}_{P}(Cert_{\alpha}) = Cert_{\alpha}) \tag{3}$$

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 to the safety policy.

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

Therefore, the general idea in ACC is that, while analysis –Equation (1)– is an iterative process, which may traverse (parts of) the abstraction more than once until the fixpoint is reached, checking –Equation (3)– is guaranteed to be done in a single pass over the abstraction. This characterization of checking ensures that the task performed by the consumers is indeed strictly more efficient than the certification carried out by the producers, as shown in (Albert et al. 2005).

3 An Informal Account of our Method

In this section we provide an informal account of the idea of reduced certificate within the ACC framework by means of a very simple example.

Example 3.1
Consider the following program, the simple abstract domain $\bot \sqsubseteq$ `int` $\sqsubseteq$ `real` $\sqsubseteq$ `term` that we will use in all our examples, and the initial calling pattern $S_\alpha = \{$`q(X)`:$\langle$`term`$\rangle\}$ which indicates that `q` can be called with any term as argument:

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

A (top-down) analyzer for logic programs would start the analysis of `q(term)` which in turn requires the analysis of `p(term)` and, as a result, the following fixpoint can be inferred: $Cert_\alpha = \{$`q(X)`:$\langle$`term`$\rangle \mapsto \langle$`real`$\rangle$, `p(X)`:$\langle$`term`$\rangle \mapsto \langle$`real`$\rangle\}$. This gives us a safe approximation of the result of executing `q(X)`. In particular, it says that we obtain a real number as a result of executing `q(X)`. Observe that the fixpoint is sound but possibly inaccurate since when only the second rule defining `p(X)` is executed, we would obtain an integer number.

Given a safety policy, the next step in any approach to PCC is to verify that $Cert_\alpha$ entails such policy. For instance, if the safety policy specifies that $I_\alpha = \{$`q(X)`:$\langle$`term`$\rangle \mapsto \langle$`term`$\rangle\}$, then clearly $Cert_\alpha \sqsubseteq I_\alpha$ holds and, hence, $Cert_\alpha$ can be used as a certificate. Similarly, a safety policy $I'_\alpha = \{$`q(X)`:$\langle$`term`$\rangle \mapsto \langle$`real`$\rangle\}$ is entailed by the certificate, while $I''_\alpha = \{$`q(X)`:$\langle$`term`$\rangle \mapsto \langle$`int`$\rangle\}$ is not.

The next important idea in ACC is that, given a valid certificate $Cert_\alpha$, a single pass of a static analyzer over it must not change the result and, hence, this way $Cert_\alpha$ can be validated. Observe that when analyzing the second rule of `p(X)` the inferred information `X` $\mapsto$ `int` is lubbed with `X` $\mapsto$ `real` which we have in the certificate and, hence, the fixpoint does not change. Therefore, the checker can be implemented as a non-iterating single-pass analyzer over the certificate. If the result of applying the checker to $Cert_\alpha$ yields a result that is different from $Cert_\alpha$ an error is issued. Once the checker has verified that $Cert_\alpha$ is a fixpoint (and thus it safely approximates the program semantics) the only thing left is to verify that $Cert_\alpha$ entails I_α, thus ensuring that the validated certificate enforces the safety policy, exactly as the certifier does.

We now turn to the key idea of reduced certificates in ACC: the observation that *any information in the certificate that the checker is able to reconstruct by itself in a single-pass does not need to be included in the certificate.* For example, if generation of the certificate does not require iteration, then no information needs to be included in the certificate, since by performing the same steps as the generator the checker will not iterate. If the generator does need to iterate, then the challenge is to find the minimal amount of information that needs to be included in $Cert_\alpha$ to avoid such iteration in the checker.

Whether a generator requires iteration depends on the strategy used when computing the fixpoint as well as on the domain and the program itself (presence of loops and recursions, multivariance, etc.). In fact, much work has been done in

order to devise optimized strategies to reduce as much as possible iterations during analysis. As mentioned before, (Hermenegildo et al. 2000), which will be our starting point, presents a parametric algorithm that allows capturing a large class of such strategies. An important observation is that whether the checker can avoid iteration is controlled by the same factors as in the generator, modified only by the effects of the information included in the (reduced) certificate, that we would like to be minimal.

As an (oversimplied) example in order to explain this idea, let us consider two possible fixpoint strategies, each one used equally in both the analyzer (generator) and the checker:

(1) a strategy which first analyzes the first rule for `p(X)` and then the second one, and
(2) a strategy which analyzes the rules in the opposite order than (1).

Assume also that the analyzer has the simple iteration rule that as soon as an answer changes during analysis then analysis is restarted at the top (these strategies are really too simple and no practical analyzer would really iterate on this example, but they are useful for illustration here –the general issue of strategies will become clear later in the paper).

In (1), the answer $\texttt{X} \mapsto \texttt{real}$ is inferred after the checking of the first rule. Then, the second rule is analyzed which leads to the answer $\texttt{X} \mapsto \texttt{int}$ that is lubbed with the previous one yielding $\texttt{X} \mapsto \texttt{real}$. Hence, in a single pass over the program the fixpoint is reached. Therefore, with this strategy $\texttt{X} \mapsto \texttt{real}$ can be reconstructed by the checker without iterating and should not be included in the certificate.

However, with strategy (2) we first obtain the answer $\texttt{X} \mapsto \texttt{int}$. Then, after the analysis of the first rule, $\texttt{X} \mapsto \texttt{real}$ is inferred. When lubbing it with the previous value, $\texttt{X} \mapsto \texttt{int}$, the answer obtained is $\texttt{X} \mapsto \texttt{real}$. Since the answer has changed the analyzer starts a new iteration in which it reanalyzes the second rule with the new answer $\texttt{X} \mapsto \texttt{real}$. Since now nothing changes in this iteration the fixpoint is reached.

The key idea is that, if strategy (2) is used, then more than one iteration is needed to reach the fixpoint. Hence the certificate cannot be empty and instead it has to include (some of) the analysis information. The conclusion is that the notion of reduced certificate is strongly related to the strategy used during analysis and checking. □

The remainder of the article will formalize and discuss in detail each of the above steps and issues.

4 Generation of Certificates in Abstraction-Carrying Code

This section recalls ACC and the notion of full certificate in the context of (C)LP (Albert et al. 2005). This programming paradigm offers a good number of advantages for ACC, an important one being the maturity and sophistication of the analysis tools available for it. It is also a non-trivial case in many ways, including

the fact that logic variables and incomplete data structures essentially represent respectively pointers and structures containing pointers (see also the arguments and pointers to literature in Section 1 which provide evidence that our approach is applicable essentially directly to programs in other programming paradigms, including their bytecode representations).

Very briefly, *terms* are constructed from variables $x \in \mathcal{V}$, *functors* (e.g., f) and *predicates* (e.g., p). We denote by $\{x_1/t_1, \ldots, x_n/t_n\}$ the *substitution* σ, where $x_i \neq x_j$, if $i \neq j$, and t_i are terms. A *renaming* is a substitution ρ for which there exists the inverse ρ^{-1} such that $\rho\rho^{-1} \equiv \rho^{-1}\rho \equiv id$. A *constraint* is a conjunction of expressions built from predefined predicates (such as inequalities over the reals) whose arguments are constructed using predefined functions (such as real addition). An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and t_i are terms. A *literal* is either an atom or a constraint. A *rule* is of the form H `:-` D where H, the *head*, is an atom and D, the *body*, is a possibly empty finite sequence of literals. A *constraint logic program* $P \in Prog$, or *program*, is a finite set of rules. Program rules are assumed to be normalized: only distinct variables are allowed to occur as arguments to atoms. Furthermore, we require that each rule defining a predicate p has identical sequence of variables $x_{p_1}, \ldots x_{p_n}$ in the head atom, i.e., $p(x_{p_1}, \ldots x_{p_n})$. We call this the *base form* of p. This is not restrictive since programs can always be normalized.

4.1 The Analysis Algorithm

Algorithm 1 has been presented in (Hermenegildo et al. 2000) as a generic description of a fixpoint algorithm which generalizes those used in state-of-the-art analysis engines, such as the one in `CiaoPP` (Hermenegildo et al. 2005), PLAI (Muthukumar and Hermenegildo 1992 de la Banda et al. 1996), GAIA (Le Charlier and Van Hentenryck 1994), and the CLP($\mathcal{R}$) analyzer (Kelly et al. 1998). It has the description domain D_α (and functions on this domain) as parameters. Different domains give analyzers which provide different kinds of information and degrees of accuracy. 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 $S_\alpha \subseteq AAtom$ 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, ANALYZE_F, constructs an *and–or graph* (Bruynooghe 1991) (or analysis graph) for S_α which is an abstraction of the (possibly infinite) set of (possibly infinite) execution paths (and-or trees) explored by the concrete execution of the initial calls described by S_α in P. Let S_P^α be the abstract semantics of the program for the call patterns S_α defined in (Bruynooghe 1991). Following the notation in Section 2, the analysis graph –denoted as $[\![P]\!]_\alpha$– corresponds to (or safely approximates) $\mathrm{lfp}(S_P^\alpha)$.

The program analysis graph is implicitly represented in the algorithm by means

Algorithm 1 Generic Analyzer for Abstraction-Carrying Code

```
    Initialization of global data structures: DAT=AT=∅
 1: function ANALYZE_F(S_α ⊆ AAtom, Ω ∈ QHS)
 2:     for A : CP ∈ S_α 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 AT;

 9: procedure NEW_CALL_PATTERN(A : CP ∈ AAtom, Ω ∈ QHS)
10:     for all rule A_k : −B_{k,1}, ..., B_{k,n_k} do
11:         CP_0 :=Aextend(CP, vars(..., B_{k,i}, ...));
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:     add_answer_table(A : CP↦⊥);

15: procedure PROCESS_ARC(H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2 ∈ Dep, Ω ∈ QHS)
16:     if B_{k,i} is not a constraint then
17:         add H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2 to DAT;
18:     W := vars(H_k, B_{k,1}, ..., B_{k,n_k});
19:     CP_3 := GET_ANSWER(B_{k,i} : CP_2, CP_1, W, Ω);
20:     if CP_3 ≠ ⊥ and i ≠ n_k then
21:         CP_4 := Arestrict(CP_3, vars(B_{k,i+1}));
22:         add_event( arc(H_k : CP_0 ⇒ [CP_3] B_{k,i+1} : CP_4),Ω);
23:     else if CP_3 ≠ ⊥ and i=n_k then
24:         AP_1 := Arestrict(CP_3, vars(H_k)); INSERT_ANSWER_INFO(H : CP_0↦AP_1, Ω);

25: function GET_ANSWER(L : CP_2 ∈ AAtom, CP_1 ∈ D_α, W ⊆ V, Ω ∈ QHS)
26:     if L is a constraint then return Aadd(L, CP_1);
27:     else AP_0 := LOOKUP_ANSWER(L : CP_2, Ω); AP_1 := Aextend(AP_0, W);
28:          return Aconj(CP_1, AP_1);

29: function LOOKUP_ANSWER(A : CP ∈ AAtom, Ω ∈ QHS)
30:     if there exists a renaming σ s.t. σ(A : CP)↦AP in AT then
31:         return σ^{-1}(AP);
32:     else add_event(newcall(σ(A : CP)), Ω) where σ is renaming s.t. σ(A) in base form;
33:         return ⊥;

34: procedure INSERT_ANSWER_INFO(H : CP↦AP ∈ Entry, Ω ∈ QHS)
35:     AP_0 := LOOKUP_ANSWER(H : CP); AP_1 := Alub(AP, AP_0);
36:     if AP_0 ≠ AP_1 then
37:         add_answer_table((H : CP↦AP_1);
38:         add_event(updated(H : CP), Ω);

39: procedure ADD_DEPENDENT_RULES(A : CP ∈ AAtom, Ω ∈ QHS)
40:     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
41:         add_event(arc(H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2), Ω);
```

of two global data structures, the *answer table* AT and the *dependency arc table* DAT, both initially empty as shown at the beginning of Algorithm 1.[1]

[1] Given the information in these, it is straightforward to construct the graph and the associated program-point annotations.

Definition 4.1 (*answer and dependency arc table*)
Let $P \in Prog$ be a program and D_α an abstract domain.

- An *Answer Table* ($AT \subseteq Entry$) for P and D_α is a set of entries of the form $A : CP \mapsto AP \in Entry$ where $A : CP \in AAtom$, A is always in base form and CP and AP are abstract substitutions in D_α.
- *A Dependency Arc Table* ($DAT \subseteq Dep$) for P and D_α is a set of *dependencies* of the form $A_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2 \in Dep$, where A_k :- $B_{k,1}, \ldots, B_{k,n}$ is a program rule in P and CP_0, CP_1, CP_2 are abstract substitutions in D_α.

Informally, an entry $A : CP \mapsto AP$ in AT should be interpreted as "the answer pattern for calls to A satisfying precondition (or call pattern) CP meets post-condition (or answer pattern), AP." Dependencies are used for efficiency. As we will explain later, Algorithm 1 finishes when there are no more events to be processed (function ANALYZE_F). This happens when the answer table AT reaches a fixpoint. Any entry $A : CP \mapsto AP$ in AT is generated by analyzing all rules associated to A (procedure NEW_CALL_PATTERN). Thus, if we have a rule of the form A_k :- $B_{k,1}, \ldots, B_{k,n}$, we know that the answer for A depends on the answers for all literals in the body of the rule. We annotate this fact in DAT by means of the dependencies $A_k : CP \Rightarrow [CP_{k,i-1}]\ B_{k,i} : CP_{k,i}$, $i \in \{1, ..n\}$, which mean that the answer for $A_k : CP$ depends on the answer for $B_{k,i} : CP_{k,i}$, also stored in AT. Then if during the analysis, the answer for $B_{k,i} : CP_{k,i}$ changes, the *arc* $A_k : CP \Rightarrow [CP_{k,i-1}]B_{k,i} : CP_{k,i}$ must be reprocessed in order to compute the "possibly" new answer for $A_k : CP$. This is to say that the rule for A_k has to be processed again starting from atom $B_{k,i}$. Thus, as we will see later, dependency arcs are used for forcing recomputation until a fixpoint is reached. The remaining part $CP_{k,i-1}$ is the program annotation just before $B_{k,i}$ is reached and contains information about all variables in rule k. $CP_{k,i-1}$ is not really necessary, but is included for efficiency.

Intuitively, the analysis algorithm is a graph traversal algorithm which places entries in the answer table AT and dependency arc table DAT as new nodes and arcs in the program analysis graph are encountered. To capture the different graph traversal strategies used in different fixpoint algorithms, a *prioritized event queue* is used. We use $\Omega \in QHS$ to refer to a *Queue 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 abstract substitution 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 to call pattern A with abstract substitution CP has been changed in AT.

The algorithm is defined in terms of five abstract operations on the domain D_α:

- Arestrict(CP, V) performs the abstract restriction of an abstract substitution CP to the set of variables in the set V.

- Aextend(CP, V) extends the abstract substitution CP to the variables in the set V.
- Aadd(C, CP) performs the abstract operation of conjoining the actual constraint C with the abstract substitution CP.
- Aconj(CP_1, CP_2) performs the abstract conjunction of two abstract substitutions.
- Alub(CP_1, CP_2) performs the abstract disjunction of two abstract substitutions.

Apart from the parametric domain-dependent functions, the algorithm has several other undefined functions. 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. When an arc $H_k : CP \Rightarrow [CP'']\ B_{k,i} : CP'$ is added to DAT, it replaces any other arc of the form $H_k : CP \Rightarrow [\ \_\ ]\ B_{k,i} : \_$ (modulo renaming) in the table and the priority queue. Similarly when an entry $H_k : CP \mapsto AP$ is added to the AT (add_answer_table), it replaces any entry of the form $H_k : CP \mapsto \_$ (modulo renaming). Note that the underscore (_) matches any description, and that there is at most one matching entry in DAT or AT at any time.

More details on the algorithm can be found in (Hermenegildo et al. 2000; Puebla and Hermenegildo 1996). 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 $\bot$ (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 DAT (L17).
  - Atoms are processed by function GET_ANSWER:
    - Constraints are simply added to the current description (L26).
    - In the case of literals, the function LOOKUP_ANSWER first looks up an answer for the given call pattern in AT (L30) and if it is not found, it places a *newcall* event (L32). When it finds one, then this answer is extended to the variables in the rule the literal occurs in (L27) and *conjoined* with the current abstract substitution (L28). The resulting answer (L19) 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 (L20); otherwise, the new answer is computed by INSERT_ANSWER_INFO.
- The part of the algorithm that is more relevant to the generation of reduced certificates is within INSERT_ANSWER_INFO. The new answer for the rule is *combined* with the current answer in the table (L35). If the fixpoint for such

call has not been reached, then the corresponding entry in AT is updated with the combined answer (L37) and an updated event is added to the queue (L38).

- The purpose of an updated event is that the function ADD_DEPENDENT_RULES (re)processes those calls which depend on the call pattern $A : CP$ whose answer has been updated (L40). This effect is achieved by adding the arc events for each of its dependencies (L41). The fact that dependency arcs contain information at the level of body literals, identified by a pair k, i, allows reprocessing only those rules for the predicate which depend on the updated pattern. Furthermore, those rules are reprocessed precisely from the body atom whose answer has been updated. If, instead, dependencies were kept at the level of rules, rules would need to be reprocessed always from the leftmost atom. Furthermore, if dependencies were kept at the level of predicates, all rules for a predicate would have to be reprocessed from the leftmost atom as soon as an answer pattern it depended on were updated.

In the following section, we illustrate the algorithm by means of an example.

4.2 Running Example

Our running example is the program `rectoy` taken from (Rose 1998). We will use it to illustrate our algorithms and show that our approach improves on state-of-the-art techniques for reducing the size of certificates. Our approach can deal with the very wide class of properties for which abstract interpretation has been proved useful (for example in the context of LP this includes variable sharing, determinacy, non-failure, termination, term size, etc.). For brevity and concreteness, in all our examples abstract substitutions simply assign an abstract value in the simple domain introduced in Section 3 to each variable in a set V over which each such substitution ranges. 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 4.2
Consider the `Ciao` version of procedure `rectoy` (Rose 1998) and the call pattern `rectoy(N,M)` : $\langle$`int,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 $\Omega \in QHS$:

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

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

The initial answer $E_1 \equiv \texttt{rectoy(N,M)} : \langle \texttt{int}, \texttt{term} \rangle \mapsto \bot$ is inserted in AT.

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

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

Arc $A_{1,2}$ is handled exactly as $A_{1,1}$ and GET_ANSWER simply adds the constraint M=0, returning $\{\texttt{N/int}, \texttt{M/int}\}$. As it is the last atom in the body (L23), procedure INSERT_ANSWER_INFO computes Alub between $\bot$ and the above answer and overwrites E_1 with:

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

Therefore, the event $U_1 \equiv updated(\texttt{rectoy(N,M)} : \langle \texttt{int}, \texttt{term} \rangle)$ 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 by just using the following information provided by the system:

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

where the three values in the abstract substitutions correspond to variables A, B, and C, in this order. 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:

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

is introduced in the queue and the corresponding dependency is stored in DAT. The call to GET_ANSWER returns the current answer E_1'. Then, we use this answer as call pattern to process the last addition by creating a new arc $A_{2,3}$.

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

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

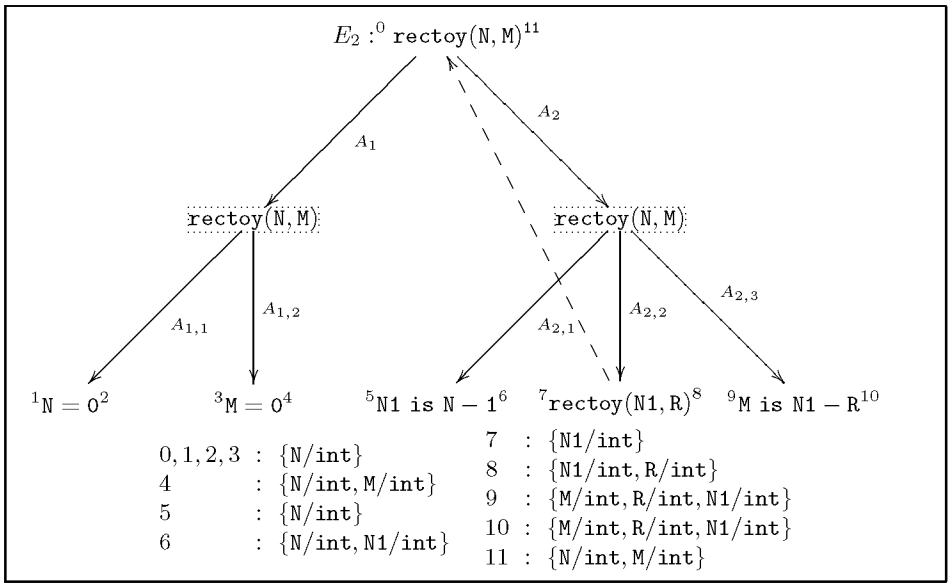

Fig. 1. Analysis Graph for our Running Example

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 the analysis terminates computing as answer table the entry E_1' and as unique dependency arc $A_{2,2}$.

Figure 1 shows the analysis graph for the analysis above. 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 the abstract substitution ⟨int, term⟩, the answer computed is ⟨int, int⟩. As mentioned before, variables whose type is term will often not be shown in what follows. Those nodes which correspond to rules are called "AND-nodes." In Figure 1, 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 there are atoms 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 at hand contains two occurrences of the abstract atom rectoy(N, M) : ⟨int, term⟩ (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 AT contains entries for the different OR-nodes which appear in the graph. In our example AT contains E_1' associated to the (root) OR-node

discussed above. Dependencies in DAT 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 $A_{2,2}$ indicates that the OR-node $\texttt{rectoy}(\texttt{N1},\texttt{R}) : \langle \texttt{int}, \texttt{term} \rangle$ is used in the OR-node $\texttt{rectoy}(\texttt{N},\texttt{M}) : \langle \texttt{int}, \texttt{term} \rangle$. Thus, if the answer pattern for $\texttt{rectoy}(\texttt{N1},\texttt{R}) : \langle \texttt{int}, \texttt{term} \rangle$ is ever updated, then we must reprocess the OR-node $\texttt{rectoy}(\texttt{N},\texttt{M}) : \langle \texttt{int}, \texttt{term} \rangle$. □

4.3 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 4.3 (*full certificate*)
We define function CERTIFIER_F : $Prog \times ADom \times 2^{AAtom} \times APol \times QHS \mapsto ACert$ which takes $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \subseteq AAtom$, $I_\alpha \in APol$, $\Omega \in QHS$ 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 $\mathsf{FCert} \sqsubseteq I_\alpha$.

If the inclusion does not hold, we do not have a certificate. This can happen either because the program does not satisfy the policy or because the analyzer is not precise enough. In the latter case, a solution is to try analyzing with a more precise (and generally more expensive) abstract domain. In the former case (the program does not satisfy the policy), this can be due to two possible reasons. A first one is that we have formalized a policy which is unnecessarily restrictive, in which case the solution is to weaken it. The other possible reason is that the program actually violates the policy, either inadvertently or on purpose. In such a case there is of course no way a certificate can be found for such program and policy.

Example 4.4
Consider the safety policy expressed by the following specification $I_\alpha : \texttt{rectoy}(\texttt{N},\texttt{M}) : \langle \texttt{int}, \texttt{term} \rangle \mapsto \langle \texttt{int}, \texttt{real} \rangle$. The certifier in Definition 4.3 returns as valid certificate the single entry E'_1. Clearly $E'_1 \sqsubseteq I_\alpha$ since $\bot \sqsubseteq \texttt{int} \sqsubseteq \texttt{real} \sqsubseteq \texttt{term}$. □

5 Abstraction-Carrying Code with Reduced Certificates

As already mentioned in Section 1, in the ACC framework, since this certificate contains the fixpoint, a *single pass* over the analysis graph is sufficient to validate such certificate on the consumer side. The key observation in order to reduce the size of certificates within the ACC framework 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 event

updated$(A : CP)$ is generated during the analysis. The above entry is *relevant* in a certificate (under some strategy) when its updates launch 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. Section 5.1 identifies redundant updates which should not be considered. In Section 5.2, we characterize formally the notion of reduced certificate containing only relevant answers. Then, in Section 5.3, we instrument an analysis algorithm to identify relevant answers and define a certifier based on the instrumented analyzer which generates reduced certificates.

5.1 Identifying Redundant Updates

According to the above intuition, we are interested in determining when an entry in the answer table has been "updated" during the analysis and such changes affect other entries. There is a special kind of updated events which can be directly considered irrelevant and correspond to those updates which launch a redundant computation (like the U_1 event generated in step B of Example 4.2). We write $DAT|_{A:CP}$ to denote the set of arcs of the form $H : CP_0 \Rightarrow [CP_1]\ B : CP_2 \in Dep$ in the current dependency arc table which depend on $A : CP$, i.e., such that $A : CP = (B : CP_2)\sigma$ for some renaming σ.

Definition 5.1 (*redundant update*)
Let $P \in Prog$, $S_\alpha \subseteq AAtom$ and $\Omega \in QHS$. We say that an event *updated*$(A : CP)$ which appears in the prioritized event queue during the analysis of P for S_α is *redundant* w.r.t. Ω if, when it is generated, $DAT|_{A:CP} = \emptyset$.

It should be noted that redundant updates can only be generated by updated events for call patterns which belong to S_α, i.e., to the initial set of call patterns. Otherwise, $DAT|_{A:CP}$ cannot be empty. Let us explain the intuition of this. The reason is that whenever an event *updated*$(A : CP)$, $A : CP \notin S_\alpha$, is generated is because a rule for A has been completely analyzed. Hence, a corresponding call to INSERT_ANSWER_INFO for $A : CP$ (L24 in Algorithm 1) has been done. If such a rule has been completely analyzed then all its arcs were introduced in the prioritized event queue. Observe that the first time that an arc is introduced in the queue is because a call to procedure NEW_CALL_PATTERN for $A : CP$ occurred, i.e., a *newcall*$(A : CP)$ event was analyzed. Consider the first event *newcall* for $A : CP$. If $A : CP \notin S_\alpha$, then this event originates from the analysis of some other arc of the form $H : CP_0 \Rightarrow [CP_1]A : CP$ for which $A : CP$ has no entry in the answer table. Thus, the dependency $H : CP_0 \Rightarrow [CP_1]A : CP$ was added to DAT. Since dependencies are never removed from DAT, then any later updated event for $A : CP$ occurs under the condition $DAT|_{A:CP} \neq \emptyset$. Even if it is possible to fix the strategy and define an analysis algorithm which does not introduce redundant updates, we prefer to follow as much as possible the generic one.

Example 5.2

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 recomputation of the second arc $A_{2,2}$ for `rectoy`. □

Note that redundant updates are indeed events which if processed immediately correspond to "nops".

The following proposition ensures the correctness of using a queue handling strategy which assigns the highest priority to redundant updates. This result can be found in (Hermenegildo et al. 2000), where it is stated that ANALYZE_F is correct independently of the order in which events in the prioritized event queue are processed.

Proposition 5.3
Let $\Omega \in QHS$. Let $\Omega' \in QHS$ be a strategy which assigns the highest priority to any updated event which is redundant. Then, $\forall\ P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \subseteq AAtom$, ANALYZE_F(S_α, Ω)=ANALYZE_F(S_α, Ω').

5.2 The Notion of Reduced Certificate

As mentioned above, the notion of reduced certificate is directly related to the idea of recomputation in the program analysis graph. Now, we are interested in finding those entries $A : CP \in Entry$ in the answer table, whose analysis has launched the reprocessing of some arcs and hence recomputation has occurred. Certainly, the reprocessing of an arc may only be caused by a non-redundant updated event for $A : CP$, which inserted (via ADD_DEPENDENT_RULES) all arcs in $DAT|_{A:CP}$ into the prioritized event queue. However some updated events are not dangerous. For instance, if the processing of an arc $H : CP_0 \Rightarrow [CP_1]A : CP$ has been stopped because of the lack of answer for $A : CP$ (L20 and L23 in Algorithm 1), this arc must be considered as "suspended", since its continuation has not been introduced in the queue. In particular, we do not take into account updated events for $A : CP$ which are generated when $DAT|_{A:CP}$ only contains suspended arcs. Note that this case still corresponds to the first traversal of any arc and should not be considered as a reprocessing. The following definition introduces the notion of *suspended arc*, i.e., of an arc suspended during analysis.

Definition 5.4 (*suspended arc*)
Let $P \in Prog$, $S_\alpha \subseteq AAtom$ and $\Omega \in QHS$. We say that an arc $H : CP_0 \Rightarrow [CP_1]\ B : CP_2$ in the dependency arc table is *suspended* w.r.t. Ω during the analysis of P for S_α iff when it is generated, the answer table does not contain any entry for $B : CP_2$ or contains an entry of the form $B : CP_2 \mapsto \bot$.

For the rest of the updated events, their relevance depends strongly on the strategy used to handle the prioritized event queue. For instance, assume that the prioritized event queue contains an event $arc(H : CP_0 \Rightarrow [CP_1]A : CP)$, coming from a suspended arc in DAT. If all updated events for $A : CP$ are processed before this arc (i.e., the fixpoint of $A : CP$ is available before processing the arc), then these

updated events do not launch any recomputation. Let us define now the notion of recomputation.

Definition 5.5 (*multi-traversed arc*)
Let $P \in Prog$, $S_\alpha \subseteq AAtom$ and $\Omega \in QHS$. We say that an arc $H : CP \Rightarrow [CP_0]A : CP_1$ in the dependency arc table has been *multi-traversed* w.r.t. Ω after the analysis of P for S_α iff it has been introduced in the dependency arc table at least twice as a non suspended arc w.r.t. Ω.

Example 5.6
Assume that we use a strategy $\Omega'' \in QHS$ such that step C in Example 4.2 is performed before B (i.e., the second rule is analyzed before the first one). Then, when the answer for $\texttt{rectoy}(\texttt{N1},\texttt{R}):\langle\texttt{int},\texttt{term}\rangle$ 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 L20 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 suspended 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, and again $A_{2,2}$ is introduced in the dependency arc table, but now as not suspended. 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) had not been handled by the algorithm yet. Hence, finally $A_{2,2}$ has not been multi-traversed. □

We define now the notion of *relevant entry*, which will be crucial for defining reduced certificates. The key observation is that those answer patterns whose computation has generated multi-traversed arcs should be available in the certificate.

Definition 5.7 (*relevant entry*)
Let $P \in Prog$, $S_\alpha \subseteq AAtom$ and $\Omega \in QHS$. We say that the entry $A : CP \mapsto AP$ in the answer table is *relevant* w.r.t. Ω after the analysis of P for S_α iff there exists a multi-traversed arc $\_ \Rightarrow [\_]A : CP$ w.r.t. Ω in the dependency arc table.

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 5.8 (*reduced certificate*)
Let $P \in Prog$, $S_\alpha \subseteq AAtom$ and $\Omega \in QHS$. 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 5.9
From now on, in our running example, we assume the strategy $\Omega' \in QHS$ which assigns the highest priority to redundant updates (see Proposition 5.3). For this strategy, the entry $E'_1 \equiv \texttt{rectoy}(\texttt{N},\texttt{M}) : \langle\texttt{int},\texttt{term}\rangle \mapsto \langle\texttt{int},\texttt{int}\rangle$ in Example 4.2 is

not relevant since no arc has been multi-traversed. Therefore, the reduced certificate for our running example is empty. In the following section, we show that our checker is able to reconstruct the fixpoint in a single pass from the empty certificate. It should be noted that, using Ω as in Example 4.2, 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 becomes relevant for Ω. Thus, this arc has been multi-traversed. □

Consider now the Java version of the procedure `rectoy`, borrowed from (Rose 1998):

```
int rectoy(int n) {
  int m; int r;
  m=0;
  if (n > 0) {
    n= n-1;
    r = this.rectoy(n);
    m = n + 4;
  };
  return m;   // Program point 30
}
```

For this program, lightweight bytecode verification (LBV) (Rose 1998) sends, together with the program, the reduced *non-empty* certificate $cert = (\{30 \mapsto (\epsilon, rectoy \cdot int \cdot int \cdot \bot)\}, \epsilon)$, which states that at program point 30 the stack does not contain information (first occurrence of ϵ),[2] and variables n, m and r have type int, int and $\bot$. The need for 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`). In LBV, *cert* has to inform the checker that it is possible for variable r at point 30 to be undefined, if the *if* condition does not hold. However, in our method this is not necessary because the checker is able to reproduce this information itself. 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.

5.3 Generation of Certificates without Irrelevant Entries

In this section, we instrument the analyzer of Algorithm 1 with the extensions necessary for producing reduced certificates, as defined in Definition 5.8. Together with the answer table returned by Algorithm 1, this new algorithm returns also the set RED (initially empty) of call patterns which will form finally the reduced certificate RCert. The resulting analyzer ANALYZE_R is presented in Algorithm 2. Except for procedure PROCESS_ARC and INSERT_ANSWER_INFO, it uses the same procedures as Algorithm 1, adapting them to the new syntax of arcs. Now, arcs will be annotated with an integer value u which counts the number of times that the

[2] The second occurrence of ϵ indicates that there are no backward jumps.

arc has been traversed during the analysis. The first time that an arc is introduced in the prioritized event queue, it is annotated with 0. Thus, L13 in Algorithm 1 must be replaced by:

$$\boxed{13: \quad \mathsf{add\_event}(arc(A_k(0) : CP \Rightarrow [CP_0]\ B_{k,1} : CP_1),\Omega)}$$

Let us see the differences between Algorithm 2 and Algorithm 1:

1. *We detect all multi-traversed arcs.* When a call to PROCESS_ARC is generated, this procedure checks if the arc is suspended (L13) before introducing the corresponding arc in the dependency arc table. If the arc is suspended, then its u value is not modified, since, as explained before, it cannot be considered as a reprocessing. Otherwise, the u-value is incremented by one. Furthermore, if $B_{k,i}$ is not a constraint and u is greater than 1, then $B_{k,i}{:}CP_2$ is added to the RED set, since this means that the arc has been multi-traversed. Note that the RED set will contain in the end those call patterns whose analysis launches the recomputation of some arc.
   Another important issue is how to handle the continuation of the arc which is being currently processed. If the arc is suspended, then no continuation is introduced in the queue (checked by L4 and L9). Otherwise (L4), before introducing the continuation in the queue, we check if the dependency arc table already contains such a continuation (L6). In that case, we add the arc with the same u annotation than that in the queue (L7). Otherwise, we introduce the continuation as an arc initialized with 0 (L8).
2. *We ignore redundant updates.* Only non-redundant updates are processed by procedure INSERT_ANSWER_INFO (L23). Each time an updated event is generated, we check if $DAT|_{H:CP}$ is different from $\emptyset$ (L23). Only then, an updated event for $H{:}CP$ is generated (L24).

Example 5.10
Consider the four steps performed in the analysis of our running example. Step A is identical. In step B the INSERT_ANSWER_INFO procedure detects a redundant updated event (L23). No updated event is generated. Step C remains identical and the arc $A_{2,2}$ (the only one able to contribute to the RED set) is annotated with 1, and step D does not occur. As expected, upon return, the RED set remains empty. □

5.4 Correctness of Certification

This section shows the correctness of the certification process carried out to generate reduced certificates, based on the correctness of the certification with full certificates of (Albert et al. 2008). First, note that, except for the control of relevant entries, ANALYZE_F(S_α, Ω) and ANALYZE_R(S_α, Ω) have the same behavior and thus compute the same answer table.

Algorithm 2 ANALYZE_R: Analyzer instrumented for Certificate Reduction

1: **procedure** PROCESS_ARC($H_k(u) : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2 \in Dep, \Omega \in QHS$)
2: $W := vars(H_k, B_{k,1}, \ldots, B_{k,n_k})$;
3: $CP_3 :=$ GET_ANSWER($B_{k,i} : CP_2, CP_1, W, \Omega$);
4: **if** $CP_3 \neq \bot$ and $i \neq n_k$ **then**
5: $CP_4 :=$ Arestrict($CP_3, vars(B_{k,i+1})$);
6: **if** there exists the arc $H_k(w) : \_ \Rightarrow \_ : B_{k,i+1}$ in the dependency arc table **then**
7: add_event($arc(H_k(w) : CP_0 \Rightarrow [CP_3]\ B_{k,i+1} : CP_4), \Omega$);
8: **else** add_event($arc(H_k(0) : CP_0 \Rightarrow [CP_3]\ B_{k,i+1} : CP_4), \Omega$);
9: **else if** $CP_3 \neq \bot$ and $i = n_k$ **then**
10: $AP_1 :=$ Arestrict($CP_3, vars(H_k)$);
11: INSERT_ANSWER_INFO($H : CP_0 \mapsto AP_1, \Omega$);
12: **if** $B_{k,i}$ is not a constraint **then**
13: **if** $CP_3 = \bot$ **then**
14: add $H_k(u){:}CP_0 \Rightarrow [CP_1]\ B_{k,i}{:}CP_2$) to dependency arc table;
15: **else** % non-suspended arc
16: add $H_k(u+1){:}CP_0 \Rightarrow [CP_1]\ B_{k,i}{:}CP_2$ to dependency arc table;
17: **if** $u{+}1{>}1$ **then** add $B_{k,i}{:}CP_2$ to RED;

18: **procedure** INSERT_ANSWER_INFO($H : CP \mapsto AP \in Entry, \Omega \in QHS$)
19: $AP_0 :=$ LOOKUP_ANSWER($H : CP, \Omega$);
20: $AP_1 :=$ Alub(AP, AP_0);
21: **if** $AP_0 \neq AP_1$ **then** %updated required
22: add_answer_table($H{:}CP \mapsto AP_1$);
23: **if** $DAT|_{H:CP} \neq \emptyset$ **then** % non-redundant updated
24: add_event($updated(H : CP)$);

Proposition 5.11
Let $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \subseteq AAtom$, $\Omega, \Omega' \in QHS$. Let AT be the answer table computed by ANALYZE_R(S_α, Ω'). Then, ANALYZE_F(S_α, Ω) $= AT$.

Proof
First note that except for the u-annotations, the procedures PROCESS_ARC in Algorithms 1 and 2 are similar. In fact, there is a one-to-one correspondence between the definition of both procedures. Concretely, we have the following mapping:

ANALYZE_F	ANALYZE_R
L16-L17	L12-17
L18	L2
L19	L3
L20-L22	L4-L8
L23-L24	L9-L11

The only difference between Algorithms 1 and 2 relies on INSERT_ANSWER_INFO. For the case of Algorithm 2, redundant updates are never introduced in the prioritized event queue (L23). Then, let us choose a new strategy Ω'', identical to Ω' except when dealing with redundant updates. For redundant updates, let us assume that

Ω'' processes them inmediately after being introduced in the event queue. Such processing does not generate any effect since the dependency arc table does not contain arcs to be launched for these updates. Hence it holds that ANALYZE_R(S_α, Ω') generates the same answer table AT than ANALYZE_F(S_α, Ω''). From Proposition 5.3 it holds that ANALYZE_F(S_α, Ω)=ANALYZE_F(S_α, Ω'') and the claim follows. □

The following definition presents the certifier for reduced certificates.

Definition 5.12
We define the function CERTIFIER_R: $Prog \times ADom \times 2^{AAtom} \times APol \times QHS \mapsto ACert$, which takes $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \subseteq AAtom$, $I_\alpha \in APol$, $\Omega \in QHS$. It returns as certificate, $\mathsf{RCert} = \{A : CP \mapsto AP \in \mathsf{FCert} \mid A : CP \in \mathsf{RED}\}$, where $\langle \mathsf{FCert}, \mathsf{RED} \rangle$ =ANALYZE_R(S_α, Ω), iff $\mathsf{FCert} \sqsubseteq I_\alpha$.

Finally, we can establish the correctness of CERTIFIER_R which amounts to say that RCert contains all relevant entries in FCert.

Theorem 5.13
Let $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \subseteq AAtom$, $I_\alpha \in APol$ and $\Omega \in QHS$. Let FCert =ANALYZE_F(S_α, Ω) and RCert= CERTIFIER_R$(P, D_\alpha, S_\alpha, I_\alpha, \Omega)$. Then, an entry $A : CP \mapsto AP \in \mathsf{FCert}$ is relevant w.r.t. Ω iff $A : CP \mapsto AP \in \mathsf{RCert}$.

Proof
According to Definition 5.12, $\mathsf{RCert} = \{A : CP \mapsto AP \in \mathsf{FCert} \mid A : CP \in \mathsf{RED}\}$, where $\langle \mathsf{FCert}, \mathsf{RED} \rangle$ =ANALYZE_R(S_α, Ω). Hence, it is enough to prove that an entry $A : CP \mapsto AP \in \mathsf{FCert}$ is relevant w.r.t. Ω iff $A : CP \in \mathsf{RED}$.

($\Leftarrow$) Assume that $A : CP \in \mathsf{RED}$. Then, from L16 and L17 it holds that there exists an arc $H(u) : CP' \Rightarrow [\_]A : CP$ in the dependency arc table such that $u > 1$. But the u-value of an arc can only be increased in procedure PROCESS_ARC (L16) after checking that CP_3 is different from $\bot$ (L15). But CP_3 is computed by means of GET_ANSWER (L3) which calls LOOKUP_ANSWER (L27). This last function only returns a value different from $\bot$ if $A : CP$ as an entry in the answer table (L30 and L31). Since $u > 1$ then u has been incremented at least twice and as argued before, in both cases the answer table contained an entry for $A : CP$, i.e., by Definition 5.5, the arc $H : CP' \Rightarrow [\_]A : CP$ is multi-traversed w.r.t Ω. Hence, by Definition 5.7, $A : CP \mapsto AP$ is a relevant entry.

($\Rightarrow$) Assume now that the entry $A : CP \mapsto AP$ is relevant w.r.t Ω. Then, by Definition 5.7, there exists an arc $H : CP' \Rightarrow [\_]A : CP$ in the dependency arc table which has been multi-traversed. By Definition 5.5, this arc has been introduced in the dependency arc table at least twice as non-suspended arc. But arcs are introduced in DAT via procedure PROCESS_ARC and each time the arc is non suspended (L15) its u-value is increased by 1 (L16). Hence the u value for $H : CP' \Rightarrow [\_]A : CP$ is at least 2. Now, L17 ensures that $A : CP \in \mathsf{RED}$.

□

6 Checking Reduced Certificates

In the ACC framework for full certificates (Albert et al. 2005) a concrete checking algorithm is used with a specific graph traversal strategy which we will refer to as Ω_C. This checker has been shown to be very efficient (i.e., this particular Ω_C is a good choice) but here we would like to consider a more generic design for the checker in which it is parametric on Ω_C in addition to being parametric on the abstract domain.[3] This lack of parametricity on Ω_C was not an issue in the original formulation of ACC in (Albert et al. 2005) since there full certificates were used. Note that even if the certifier uses a strategy Ω_A which is different from Ω_C, all valid full certificates are guaranteed to be validated in one pass by that specific checker, independently of Ω_C. This result allowed using a particular strategy in the checker without loss of generality. However, the same result does not hold any more 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. In this section, we design a generic checker which is not tied to a particular graph traversal strategy. In practice, upon agreeing on the appropriate parameters, the consumer uses the particular instance of the generic checker resulting from the application of such parameters. In a particular application of our framework, we expect that the graph traversal strategy is agreed a priori between consumer and producer. Alternatively, if necessary (e.g., when the consumer does not implement this strategy), the strategy can be sent along with the certificate in the transmitted package.

It should be noted that the design of generic checkers is also relevant in light of current trends in verified analyzers (e.g., (Klein and Nipkow 2003; Cachera et al. 2004)), 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 transformation techniques (Jones et al. 1993) to specialize a certified (specific) analysis algorithm in order to obtain a certified checker with the same strategy while preserving correctness and completeness.

6.1 The Generic Checking Algorithm

The following definition presents a generic checker for validating reduced certificates. In addition to the genericity issue discussed above, an important difference with the checker for full certificates (Albert et al. 2005) 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

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

RCert, the answer table returned by ANALYZE_F, FCert, in order to test for adherence to the safety policy –Equation (4). Note that reconstructing the answer table does not add any additional cost compared to the checker in (Albert et al. 2005), since the full answer table also has to be created in (Albert et al. 2005).

Algorithm 3 Generic Checker for Reduced Certificates CHECKING_R

```
 1: procedure INSERT_ANSWER_INFO(H:CP↦AP ∈ Entry, Ω ∈ QHS)
 2:     AP_0 := LOOKUP_ANSWER(H:CP, Ω);
 3:     AP_1 := Alub(AP, AP_0);
 4:     (IsIn,AP')=LOOK_FIXPOINT(H:CP,RCert);
 5:     if IsIn and Alub(AP, AP') ≠ AP' then return error;        % error of type a)
 6:     if AP_0≠AP_1 then        % updated required
 7:         if IsIn and AP_0=⊥ then AP_1=AP'
 8:         add_answer_table(H:CP↦AP_1);
 9:         if DAT|_{H:CP} ≠∅ then
10:             add_event(updated(H:CP), Ω);

11: function LOOK_FIXPOINT(A:CP ∈ AAtom, RCert ∈ ACert)
12:     if ∃ a renaming σ such that σ(A:CP↦AP)∈ RCert then
13:         return (true,σ^{-1}(AP));
14:     else return (false,⊥);
```

Definition 6.1 (*checker for reduced certificates*)
Function CHECKING_R is defined as function ANALYZE_R with the following modifications:

1. It receives RCert as an additional input parameter.
2. It does not use the set RED and it replaces L17 of Algorithm 2 with:

   17: If u+1>1 **return** error

3. If it fails to produce an answer table, then it issues an error.
4. Function INSERT_ANSWER_INFO is replaced by the new one in Algorithm 3.

Function CHECKER_R takes $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \subseteq AAtom$, $I_\alpha \in APol$, $\Omega \in QHS$, RCert $\in ACert$ and returns:

1. error if CHECKING_R(S_α, Ω, RCert) for P in D_α returns error.
2. Otherwise it returns FCert=CHECKING_R(S_α, Ω, RCert) for P and D_α iff FCert $\sqsubseteq$ I_α.

Let us briefly explain the differences between Algorithms 2 and 3. First, the checker has to detect (and issue) two sources of errors:

a) The answer in the certificate and the one obtained by the checker differ (L5). This is the traditional error in ACC and means that the certificate and program at hand do not correspond to each other. The call to function LOOK_FIXPOINT($H : CP$,RCert) in L4 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 $\bot$.

b) Recomputation is required. This should not occur during checking, i.e., no arcs must be multi-traversed by the checker (L17). This second type of error corresponds to situations in which some non-redundant update is needed in order to obtain an answer (it cannot be obtained in one pass). This is detected in L17 prior to check that the arc is not suspended (L9) and it has been traversed before, i.e., its u value is greater than 1. Note that we flag this as an error because the checker will have to iterate and the description we provided does not include support for it. In general, however, it is also possible to use a checker that is capable of iterating. In that case of course the certificates transmitted can be even smaller than the reduced ones, at the cost of increased checking time (as well as some additional complexity in the checking code). This allows supporting different tradeoffs between certificate size, checking time, and checker code complexity.

The second difference is that the $A : CP \mapsto AP'$ entries stored in RCert have to be added to the answer table after finding the first partial answer for $A : CP$ (different from $\bot$), in order to detect errors of type a) above. In particular, L7 and L8 add the fixpoint AP' stored in RCert to the answer table.

Example 6.2
All steps given for the analysis of Example 5.10 are identical in CHECKER_R except for the detection of possible errors. Errors of type a) are not possible since RCert is empty. An error of type b) can only be generated because of the u value of arc $A_{2,2}$. However note that in step C, this arc is introduced in the queue with $u = 0$. After processing the arc, the arc goes to the dependency arc table with $u = 1$. But since no updated events are generated, this arc is no longer processed. Hence, the program is validated in a single pass over the graph. □

6.2 Correctness of Checking

In this section we prove the correctness of the checking process, which amounts to saying that if CHECKER_R does not issue an error when validating a certificate, then the reconstructed answer table is a fixpoint verifying the given input safety policy. As a previous step, we prove the following proposition in which we also ensure that the validation of the certificate is done in one pass.

Proposition 6.3
Let $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \subseteq AAtom$, $I_\alpha \in APol$ and $\Omega \in QHS$. Let FCert= CERTIFIER_F$(P, D_\alpha, S_\alpha, I_\alpha, \Omega)$, RCert=CERTIFIER_R$(P, D_\alpha, S_\alpha, I_\alpha, \Omega)$. Then CHECKING_R$(S_\alpha, \Omega,$ RCert$)$ does not issue an error and it returns FCert. Furthermore, the validation of FCert does not generate multi-traversed arcs.

Proof
Let us consider first the call:

$$(*)\ \text{CHECKING\_R}(S_\alpha, \Omega, \text{RCert})$$

For this call, let us prove that (1) it does not issue an error and; (2) it returns FCert as result.

(1) CHECKING_R(S_α, Ω, RCert) does not issue an error.

Errors of type (a) (L5 of Algorithm 3) are not possible since, from Definition 5.12, RCert $\subseteq$ FCert, where FCert is the answer table computed by ANALYZE_F(S_α, Ω). The correctness of Algorithm ANALYZE_F(S_α, Ω) (see (Hermenegildo et al. 2000)) avoids this kind of errors.

Errors of type (b) can only occur in L17 of procedure PROCESS_ARC (Algorithm 3), for some arc $H : CP_0 \Rightarrow [CP_1]B_{k,i} : CP_2$. Since we follow the same strategy Ω in CHECKING_R and ANALYZE_R, then ANALYZE_R(S_α, Ω) introduces $B_{k,i} : CP_2$ in RED (L17 of Algorithm 2), and thus, Definition 5.12 ensures that $B_{k,i} : CP_2 \mapsto AP \in$ RCert. But this is a contradiction since for all entries in RCert, the first time that the arc is processed without answer in AT for $B_{k,i} : CP_2$, Algorithm 3 (L7 and L8) introduces $B_{k,i} : CP_2 \mapsto AP$ in AT together with the corresponding event $updated(B_{k,i} : CP_2)$. So when Ω selects this event, the new event $arc(H : CP_0 \Rightarrow [CP_1]B_{k,i} : CP_2)$ is again introduced in the prioritized event queue. When this arc is selected by Ω, the arc goes again to DAT. But since $B_{k,i} : CP_2 \mapsto AP \in AT$, no more events of the form $updated(B_{k,i} : CP_2)$ may occur (L6 of INSERT_ANSWER_INFO($B_{k,i} : CP_2$) in Algorithm 3 never holds). Hence, no more calls to process arc for $arc(H : CP_0 \Rightarrow [CP_1]B_{k,i} : CP_2)$ occur. Then the u-value for this arc will be at most 1 and no error will be generated.

(2) The call $(*)$ returns FCert.

The only differences between the call $(*)$ and the call ANALYZE_R(S_α, Ω) rely on procedure INSERT_ANSWER_INFO and L17 of procedure PROCESS_ARC. Since (1) ensures that no error is issued by $(*)$, then L5 and L17 of Algorithm 3 are never executed. Then, it is trivial that (1) computes an answer table AT as result. Furthermore, since $(*)$ and ANALYZE_R(S_α, Ω) use the same strategy, the only difference is in the prioritized event queue since for $(*)$ no relevant updates will appear in the queue. Instead of this, the real fixpoints in RCert $\subseteq$ FCert are introduced in AT in L7 and L8 of INSERT_ANSWER_INFO. Except for this fact, Algorithms 2 and 3 behave identically and thus $(*)$ computes FCert as result.

Finally, proving that the validation of RCert does not generate multi-traversed arcs is trivial since, by definition, multi-traversed arcs correspond to arcs in DAT with the u-value greater than 1. Since the call $(*)$ does not issue an error, L17 of Algorithm 3 is never executed, i.e., no arc is multi-traversed.

□

Corollary 6.4
Let $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \subseteq AAtom$, $I_\alpha \in APol$ and $\Omega \in QHS$. Let FCert=CERTIFIER_F($P, D_\alpha, S_\alpha, I_\alpha, \Omega$), and RCert$_\Omega$=CERTIFIER_R($P, D_\alpha, S_\alpha, I_\alpha, \Omega$).

If CHECKING_R($S_\alpha, \Omega, \mathsf{RCert}$), $\mathsf{RCert} \in ACert$, does not issue an error, then it returns FCert and $\mathsf{RCert}_\Omega \subseteq \mathsf{RCert}$. Furthermore, the validation of FCert does not generate multi-traversed arcs.

Proof
Let us prove, by contradiction, that $\mathsf{RCert}_\Omega \subseteq \mathsf{RCert}$. If we assume that $\mathsf{RCert}_\Omega \not\subseteq \mathsf{RCert}$, then there exists an entry $A : CP \mapsto AP \in \mathsf{RCert}_\Omega$ such that $A : CP \mapsto AP \notin \mathsf{RCert}$. By definition of RCert_Ω, $A : CP \in \mathsf{RED}$. Hence, L16 of Algorithm 2 ensures that there exists an arc $H : CP_0(u) \Rightarrow [CP_1]A : CP$ in DAT with $u > 1$. But this is not possible since otherwise the call CHECKING_R($S_\alpha, \Omega, \mathsf{RCert}_\Omega$) would issue an error, what is a contradiction by Proposition 6.5.

Now observe that from Proposition 6.5 it holds that CHECKING_R($S_\alpha, \Omega, \mathsf{RCert}_\Omega$) returns FCert and the validation of RCert_Ω does not generate multi-traversed arcs. But since $\mathsf{RCert}_\Omega \subseteq \mathsf{RCert}$, then it trivially holds that CHECKING_R($S_\alpha, \Omega, \mathsf{RCert}$) also returns FCert exactly in the same way that CHECKING_R($S_\alpha, \Omega, \mathsf{RCert}_\Omega$) does, i.e., without generating multi-traversed arcs. □

Theorem 6.5 (*correctness*)
Let $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \subseteq AAtom$, $I_\alpha \in APol$, $\Omega \in QHS$ and $\mathsf{RCert} \in ACert$. Then, if CHECKER_R($P, D_\alpha, S_\alpha, I_\alpha, \Omega, \mathsf{RCert}$) does not issue an error and returns a certificate $\mathsf{FCert} \in ACert$, then

- FCert is a fixpoint of P.
- $\mathsf{FCert} \sqsubseteq I_\alpha$;

Proof
If CHECKER_R($P, D_\alpha, S_\alpha, I_\alpha, \Omega, \mathsf{RCert}$) does not issue an error then, from Definition 6.1, it holds that FCert =CHECKING_R($S_\alpha, \Omega, \mathsf{RCert}$) does not issue an error and $\mathsf{FCert} \sqsubseteq I_\alpha$. From Corollary 6.4. it follows that FCert =CERTIFIER_F($P, D_\alpha, S_\alpha, I_\alpha, \Omega$). Hence, as Definition 4.3 establishes, FCert is the answer table computed by ANALYZE_F(S_α, Ω'). Finally, by the results in (Hermenegildo et al. 2000), FCert is a fixpoint for P.
□

6.3 Completeness of Checking

The following theorem (completeness) provides sufficient conditions under which a checker is guaranteed to validate reduced certificates which are actually valid. In other words, if a certificate is valid and such conditions hold, then the checker is guaranteed to validate the certificate. Note that it is not always the case when the strategy used to generate it and the one used to check it are different.

Theorem 6.6 (*completeness*)
Let $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \subseteq AAtom$, $I_\alpha \in APol$ and $\Omega_A \in QHS$. Let $\mathsf{FCert}=$ 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 such that $\mathsf{RCert}_{\Omega_C}=$ CERTIFIER_R$(P, D_\alpha, S_\alpha, I_\alpha, \Omega_C)$ and $\mathsf{RCert}_{\Omega_A} \supseteq \mathsf{RCert}_{\Omega_C}$. Then, CHECKER_R$(P, D_\alpha, S_\alpha, \Omega_C, \mathsf{RCert}_{\Omega_A})$ returns FCert and does not issue an error.

Proof
We prove it by contradiction. The only cases in which CHECKER_R$(P, D_\alpha, S_\alpha, \Omega_C, \mathsf{RCert}_{\Omega_A})$ issues an error are the following:

- The partial answer AP computed for some calling pattern $A : CP$ (provided in $\mathsf{RCert}_{\Omega_A}$) leads to $\mathsf{Alub}(AP, AP') \neq AP'$ (L5), where AP' is the answer for $A : CP$, i.e., $A : CP \mapsto AP' \in \mathsf{RCert}_{\Omega_A}$. But, $\mathsf{RCert}_{\Omega_A} \subseteq \mathsf{FCert}$, i.e., FCert would contain an incorrect answer for $A : CP$, which is a contradiction with the assumption that FCert is a valid certificate for P.
- There exists some arc $H : CP \Rightarrow [CP_1]B : CP_2$ which has been traversed more than once, i.e., its u-value is greater than 1 (L17 in Algorithm 3). Since $\mathsf{RCert}_{\Omega_C} \subseteq \mathsf{RCert}_{\Omega_A}$, i.e., $\mathsf{RCert}_{\Omega_C}$ contains possibly less entries than $\mathsf{RCert}_{\Omega_A}$, then the call $(*)$ in Theorem 6.5 fails also because of such a multi-traversed arc. But this is a contradiction with (**1**) in Theorem 6.5.

Consequently, CHECKER_R$(P, D, S, \Omega_C, \mathsf{RCert}_{\Omega_A})$ returns an answer table AT. Finally, by Theorem 6.5, we know that since no error is issued, then CHECKER_R returns FCert. □

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. Note that if $\mathsf{RCert}_{\Omega_A} \not\supseteq \mathsf{RCert}_{\Omega_C}$, CHECKER_R with the strategy Ω_C may fail to validate $\mathsf{RCert}_{\Omega_A}$, which is indeed valid for the program under Ω_A.

Example 6.7
Consider the program of Example 3.1, the same abstract domain D_α than in our running example, and the call pattern $S_\alpha = \{$q(X):⟨term⟩$\}$: The full certificate computed by CERTIFIER_F is $\mathsf{FCert} = \{$q(X):⟨term⟩ $\mapsto$ ⟨real⟩, p(X):⟨term⟩ $\mapsto$ ⟨real⟩$\}$. Let us consider two different queue handling strategies $\Omega_A \neq \Omega_C$. Under both strategies, we start the analysis introducing q(X):⟨term⟩ $\mapsto \bot$ in the answer table and processing the single rule for q. The arc q(X)(0):⟨term⟩⇒[{X/term}] p(X):⟨term⟩ is introduced in the queue and processed afterward. As a result, q(X)(0):⟨term⟩ ⇒[{X/term}] p(X):⟨term⟩ goes to DAT and event *newcall*(p(X):⟨term⟩) is generated. The processing of this last event adds p(X):⟨term⟩ $\mapsto \bot$ to the answer table. Now, using Ω_A, the analyzer processes both rules for p(X) in textual order. None of the arcs introduced in DAT can issue an error. After traversing the first rule, answer p(X):⟨term⟩ $\mapsto$ ⟨real⟩ is inferred and non-redundant updated event *updated*(p(X):⟨term⟩) is generated. The analysis of the second rule produces as

answer $\langle$`int`$\rangle$ and does not update the entry since $\mathsf{Alub}(\{$`X/real`$\},\{$`X/int`$\})$ returns $\{$`X/real`$\}$. We process the non-redundant update for `p` by calling function ADD_DEPENDENT_RULES. The arc for `q` stored in the dependency arc table with 0 is launched. When processing this arc, again the arc is introduced in DAT with $u = 1$, and the answer `q(X)`:$\langle$`term`$\rangle \mapsto \langle$`real`$\rangle$ replaces the old one in the answer table. Since RED is empty, then $\mathsf{RCert}_{\Omega_A}$ is empty.

Assume now that Ω_C assigns a higher priority to the second rule of `p`. In this case, the answer for `p(X)`:$\langle$`term`$\rangle$ changes from $\bot$ to $\{$`X/int`$\}$, producing a non-redundant update. Suppose now that the updated event is processed, which launches the arc for `q` stored in DAT. If we process such an arc, then it will be introduced again in DAT, but now with $u = 1$. Answer `q(X)`:$\langle$`term`$\rangle \mapsto \{$`X/int`$\}$ is inserted in the answer table. When the first arc for `p` is processed, the computed answer is $\{$`X/real`$\}$. Now, a new non-redundant updated event is needed. The processing of this update event launches again the arc for `q` stored in DAT, whose analysis introduces it in DAT with $u = 2$.

Hence $\mathsf{RCert}_{\Omega_A}$ is empty but $\mathsf{RCert}_{\Omega_C}$ contains the single entry `p(X)`:$\langle$`term`$\rangle \mapsto \langle$`real`$\rangle$. Thus, CHECKER_R(P, $D_\alpha, S_\alpha, \Omega_C, \mathsf{RCert}_{\Omega_A}$) will issue an error (L17) when trying to validate the program if provided with the empty certificate $\mathsf{RCert}_{\Omega_A}$. On the contrary, by Theorem 6.6, CHECKER_R(P, $D_\alpha, S_\alpha, \mathsf{RCert}_{\Omega_C}, \Omega_A$) returns FCert and does not issue an error. This justifies the results intuitively shown in Section 3. □

7 Discussion and Experimental Evaluation

As we have illustrated throughout the paper, the reduction in the size of the certificates is directly related to the number of *updates* (or iterations) performed during analysis. Clearly, depending on the "quality" of the graph traversal strategy used, different instances of the generic analyzer will generate reduced certificates of different sizes. Significant and successful efforts have been made during recent years towards improving the efficiency of analysis. The most optimized analyzers actually aim at reducing the number of updates necessary to reach the final fixpoint (Puebla and Hermenegildo 1996). Interestingly, our framework greatly benefits from all these advances, since the more efficient analysis, the smaller the corresponding reduced certificates. We have implemented a generator and a checker of reduced certificates as an extension of the efficient, highly optimized, state-of-the-art analysis system available in `CiaoPP`. Both the analysis and checker use the optimized depth-first new-calling QHS of (Puebla and Hermenegildo 1996).

In our experiments we study two crucial points for the practicality of our proposal: the size of reduced vs. full certificates (Table 7.1) and the relative efficiency of checking reduced vs. full certificates (Table 7.2). As mentioned before, the algorithms are parametric w.r.t. the abstract domain. In all our experiments we use the same implementation of the domain-dependent functions of the *sharing+freeness* (Muthukumar and Hermenegildo 1991) abstract domain. We have selected this domain because it is highly optimized and also because the information it infers is very useful for reasoning about instantiation errors, which is a cru-

Program	Source	ByteC	BC/S	FCert	RCert	F/R	R/S
aiakl	1555	3817	2.455	3090	1616	1.912	1.039
bid	4945	10376	2.098	5939	883	6.726	0.179
browse	2589	8492	3.280	1661	941	1.765	0.363
deriv	957	4221	4.411	288	288	1.000	0.301
grammar	1598	3182	1.991	1259	40	31.475	0.025
hanoiapp	1172	2264	1.932	2325	880	2.642	0.751
occur	1367	6919	5.061	1098	666	1.649	0.487
progeom	1619	3570	2.205	2148	40	53.700	0.025
qsortapp	664	1176	1.771	2355	650	3.623	0.979
query	2090	8818	4.219	531	40	13.275	0.019
rdtok	13704	15423	1.125	6533	2659	2.457	0.194
rectoy	154	140	0.909	167	40	4.175	0.260
serialize	987	3801	3.851	1779	1129	1.576	1.144
zebra	2284	5396	2.363	4058	40	101.450	0.018
Overall			2.17			3.35	0.28

Table 1. Size of Reduced and Full Certificates

cial aspect for the safety of logic programs. Furthermore, as mentioned previously, sharing domains have also been shown to be useful for checking properties of imperative programs, including for example information flow characteristics of Java bytecode (Secci and Spoto 2005; Genaim and Spoto 2005). On the other hand, we have used $\top$ as call patterns in order to get all possible modes of use of predicate calls.

The whole system is written in `Ciao` (Bueno et al. 2009) and the experiments have been run using version 1.13r5499 with compilation to bytecode on a Pentium 4 (Xeon) at 2 Ghz and with 4 Gb of RAM, running GNU Linux Fedora Core-2 2.6.9.

A relatively wide range of programs has been used as benchmarks. They are the same ones used in (Hermenegildo et al. 2000; Albert et al. 2005), where they are described in more detail.

7.1 Size of Reduced Certificates

Table 7.1 shows our experimental results regarding certificate size reduction, coded in compact (*fastread*) format, for the different benchmarks. It compares the size of each reduced certificate to that of the full certificate and to the corresponding source code for the same program.

The column **Source** shows the size of the source code and **ByteC** its corresponding bytecode. To make this comparison fair, in column **BC/S** we subtract 4180 bytes from the size of the bytecode for each program: the size of the bytecode for an empty program in this version of `Ciao`(minimal top-level drivers and

Program	C_F	C_R	C_F/C_R
aiakl	85	86	0.986
bid	46	48	0.959
browse	20	20	0.990
deriv	28	27	1.038
grammar	14	14	1.014
hanoiapp	31	30	1.033
occur	18	20	0.911
progeom	17	16	1.012
qsortapp	24	19	1.290
query	13	14	0.917
rdtok	59	56	1.061
rectoy	8	9	0.909
serialize	27	30	0.875
zebra	125	129	0.969
Overall			0.99

Table 2. Comparison of Checking Times

exception handlers for any executable). The size of the certificates is showed in the following columns. The columns **FCert** and **RCert** contain the size of the full and reduced certificates, respectively, for each benchmark, and they are compared in the next column (**F/R**). Our results show that the reduction in size is quite significant in all cases. It ranges from 101.45 in *zebra* (RCert is indeed empty –the minimum size of an empty certificate is 40 bytes– whereas FCert is 4058) to 1 for *deriv* (both certificates have the same size).

The last column (**R/S**) compares the size of the reduced certificate to the source code (i.e., the size of the final package to be submitted to the consumer). The results show the size of the reduced certificate to be very reasonable. It ranges from 0.018 times the size of the source code (for zebra) to 1.144 (in the case of serialize). Overall, it is 0.28 times the size of the source code. We consider this satisfactory since in general (C)LP programs are quite compact (up to 10 times more compact than equivalent imperative programs).

7.2 Checking Time of Reduced Certificates

Table 7.2 presents our experimental results regarding checking time. Execution times are given in milliseconds and measure *runtime*. They are computed as the arithmetic mean of five runs. For each benchmark, columns $\mathbf{C}_F$ and $\mathbf{C}_R$ are the times for executing CHECKER_F and CHECKER_R, respectively. Column $\mathbf{C}_F/\mathbf{C}_R$ compares both checking times. These times show that the efficiency of CHECKER_R is very similar to that of CHECKER_F in most cases.

The last row (Overall) summarizes the results for the different benchmarks using

a weighted mean which places more importance on those benchmarks with relatively larger certificates and checking times. We use as weight for each program its actual checking time. We believe that this weighted mean is more informative than the arithmetic mean, since, for example, doubling the speed in which a large and complex program is checked is more relevant than achieving this for small, simple programs. As mentioned before, the efficiency of the checker for reduced certificates is very similar to that of CHECKER_F (the overall slowdown is 0.99).

8 Related Work

A detailed comparison of the technique of ACC with related methods can be found in (Albert et al. 2008). In this section, we focus only on work related to certificate size reduction in PCC. The common idea in order to compress a certificate in the PCC scheme is to store only the analysis information which the checker is not able to reproduce by itself (Leroy 2003). In the field of abstract interpretation, this is known as *fixpoint compression* and it is being used in different contexts and tools. For instance, in the Astrée analyzer (Cousot et al. 2005) designed to detect runtime errors in programs written in C, only one abstract element by head of loop is kept for memory usage purposes. Our solution is an improvement in the sense that some of these elements many not need to be included in the certificate (i.e., if they are not relevant). In other words, some loops do not require iteration to reach the fixpoint and our technique detects this.

With our same purpose of reducing the size of certificates, Necula and Lee (Necula and Lee 1998) designed a variant of the Edinburgh Logical Framework LF (Harper et al. 1993), called LF_i, in which certificates (or proofs) discard part of the information that is redundant or that can be easily synthesized. LF_i inherits from LF the possibility of encoding several logics in a natural way but avoiding the high degree of redundancy proper of the LF representation of proofs. In the producer side, the original certificate is an LF proof to which a representation algorithm is applied. On the consumer side, LF_i proofs are validated by using a one pass LF type checker which is able to reconstruct on the fly the missing parts of the proof in one pass. Experimental results for a concrete implementation reveal an important reduction on the size of certificates (w.r.t. LF representation proofs) and on the checking time. Although this work attacks the same problem as ours the underlying techniques used are clearly different. Furthermore, our certificates may be considered minimal, whereas in (Necula and Lee 1998), redundant information is still left in the certificates in order to guarantee a more efficient behaviour of the type checker.

A further step is taken in Oracle-based PCC (Necula and Rahul 2001). This is a variation of the PCC idea that allows the size of proofs accompanying the code to adapt to the complexity of the property being checked such that when PCC is used to verify relatively simple properties such as type safety, the essential information contained in a proof is significantly smaller than the entire proof. The proof as an oracle is implemented as a stream of bits aimed at resolving the nondeterministic interpretation choices. Although the underlying representations and techniques are different from ours, we share with this work the purpose of reducing

the size of certificates by providing the checker with the minimal information it requires to perform a proof and the genericity which allows both techniques to deal with different kinds of properties beyond types.

The general idea of certificate size reduction has also been deployed in lightweight bytecode verification (LBV) (Rose 1998; Rose 2003). LBV is a practical PCC approach to Java Bytecode Verification (Leroy 2003) applied to the KVM (an embedded variant of the JVM). The idea is that the type-based bytecode verification is split in two phases, where the producer first computes the certificate by means of a type-based dataflow analyzer and then the consumer simply checks that the types provided in the code certificate are valid. As in our case, the second phase can be done in a single, linear pass over the bytecode. However, LBV is limited to types while ACC generalizes it to arbitrary domains. Also, ACC deals with multivariance with the associated accuracy gains (while LBV is monovariant). Regarding the reduction of certificate size, our work characterizes precisely the minimal information that can be sent for a generic algorithm not tied to any particular graph traversal strategy. While the original notion of certificate in (Rose 1998) includes the complete entry solution with respect to each basic block, (Rose 2003) reduces certificates by sending information only for "backward" jumps. As we have seen through our running example, (Rose 2003) sends information for all such backward jumps while our proposal 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. We believe that our notion of reduced certificate could also be used within Rose's framework.

As a final remark, the main ideas in ACC showed in Equations 2 and 4 in Section 2 have been the basis to build a PCC architecture based on *certified* abstract interpretation in (Besson et al. 2006). Therefore, this proposal is built on the basics of ACC for certificate generation and checking, but relies on a *certified* checker specified in Coq (Barras et al. 1997) in order to reduce the trusted computing base. In contrast to our framework, this work is restricted to safety properties which hold for all states and, for now, it has only been implemented for a particular abstract domain.

Acknowledgments

The authors would like to gratefully thank the anonymous referees for useful comments on a preliminary version of this article. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 *DOVES* project, the TIN2008-04473-E (Acción Especial) project, the HI2008-0153 (Acción Integrada) project, the UCM-BSCH-GR58/08-910502 Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS* project.

References

Albert, E., Gómez-Zamalloa, M., Hubert, L., and Puebla, G. 2007. Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In *Ninth International Symposium on Practical Aspects of Declarative Languages (PADL 2007).* Number 4354 in LNCS. Springer-Verlag, 124–139.

Albert, E., Puebla, G., and Hermenegildo, M. 2005. Abstraction-Carrying Code. In *11th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2004).* Number 3452 in LNAI. Springer-Verlag, 380–397.

Albert, E., Puebla, G., and Hermenegildo, M. 2008. Abstraction-Carrying Code: A Model for Mobile Code Safety. *New Generation Computing 26,* 2 (March), 171–204.

Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., Parent, C., Paulin-Mohring, C., Saibi, A., and Werner, B. 1997. The Coq Proof Assistant Reference Manual : Version 6.1. Tech. Rep. RT-0203. citeseer.ist.psu.edu/barras97coq.html.

Besson, F., Jensen, T., and Pichardie, D. 2006. A PCC Architecture based on Certified Abstract Interpretation. In *Proc. of first International Workshop on Emerging Applications of Abstract Interpretation (EAAI 2006).* ENTCS.

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

Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., and Puebla-(Eds.), G. 2009. The Ciao System. Ref. Manual (v1.13). Tech. rep., School of Computer Science, T.U. of Madrid (UPM). Available at `http://www.ciaohome.org`.

Cachera, D., Jensen, T., Pichardie, D., and Rusu, V. 2004. Extracting a Data Flow Analyser in Constructive Logic. In *The European Symposium on Programming (ESOP 2004).* Number 2986 in LNCS. Springer-Verlag, 385–400.

Cousot, P. and Cousot, R. 1977. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL'77).* ACM Press, 238–252.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. 2005. The ASTRÉE Analyser. In *The European Symposium on Programming (ESOP 2005).* Number 3444 in LNCS. Springer-Verlag, 21–30.

de la Banda, M. G., Hermenegildo, M., Bruynooghe, M., Dumortier, V., Janssens, G., and Simoens, W. 1996. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems 18,* 5 (September), 564–615.

Genaim, S. and Spoto, F. 2005. Information Flow Analysis for Java Bytecode. In *Sixth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2005).* Number 3385 in LNCS. Springer-Verlag, 346–362.

Harper, R., Honsell, F., and Plotkin, G. 1993. A Framework for Defining Logics. *Journal of the Association for Computing Machinery 40,* 1, 143–184.

Hermenegildo, M., Puebla, G., Bueno, F., and López-García, P. 2005. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming 58,* 1–2 (October), 115–140.

Hermenegildo, M., Puebla, G., Marriott, K., and Stuckey, P. 2000. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems 22,* 2 (March), 187–223.

Jones, N., Gomard, C., and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, New York.

Kelly, A., Marriott, K., Søndergaard, H., and Stuckey, P. 1998. A Practical Object-Oriented Analysis Engine for CLP. *Software: Practice and Experience 28,* 2, 188–224.

Klein, G. and Nipkow, T. 2003. Verified Bytecode Verifiers. *Theoretical Computer Science 3(298)*, 583–626.

Le Charlier, B. and Van Hentenryck, P. 1994. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems 16,* 1, 35–101.

Leroy, X. 2003. Java Bytecode Verification: Algorithms and Formalizations. *Journal of Automated Reasoning 30,* 3-4, 235–269.

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

Marriot, K. and Stuckey, P. 1998. *Programming with Constraints: An Introduction.* The MIT Press.

Méndez-Lojo, M., Navas, J., and Hermenegildo, M. 2007a. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007).* Number 4915 in LNCS. Springer-Verlag, 154–168.

Méndez-Lojo, M., Navas, J., and Hermenegildo, M. 2007b. An Efficient, Parametric Fixpoint Algorithm for Analysis of Java Bytecode. In *ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2007).* Electronic Notes in Theoretical Computer Science. Elsevier - North Holland.

Muthukumar, K. and Hermenegildo, M. 1991. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *International Conference on Logic Programming (ICLP 1991).* MIT Press, 49–63.

Muthukumar, K. and Hermenegildo, M. 1992. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming 13,* 2/3 (July), 315–347.

Necula, G. 1997. Proof-Carrying Code. In *ACM Symposium on Principles of programming languages (POPL 1997).* ACM Press, 106–119.

Necula, G. and Lee, P. 1998. Efficient Representation and Validation of Proofs. In *IEEE Symposium on Logic in Computer Science (LICS 1998).* IEEE Computer Society, 93–104.

Necula, G. and Rahul, S. 2001. Oracle-Based Checking of Untrusted Software. In *Principles of Programming Languages (POPL 2001).* ACM Press, 142–154.

Puebla, G. and Hermenegildo, M. 1996. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium (SAS 1996).* Number 1145 in LNCS. Springer-Verlag, 270–284.

Rose, E. 2003. Lightweight Bytecode Verification. *Journal of Automated Reasoning 31*, 303–334.

Rose, E. Rose, K. 1998. Lightweight Bytecode Verification. In *OOPSLA Workshop on Formal Underpinnings of Java.*

Secci, S. and Spoto, F. 2005. Pair-Sharing Analysis of Object-Oriented Programs. In *Static Analysis Symposium (SAS 2005).* Number 3672 in LNCS. 320–335.

Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., and Co, P. 1999. Soot - a Java Optimization Framework. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON).* 125–135.