Abstraction-Carrying Code: a Model for Mobile Code Safety[*]

Elvira Albert[‡] Germán Puebla[§] Manuel Hermenegildo[§¶]

Abstract

Proof-Carrying Code (PCC) is a general approach to mobile code safety in which programs are augmented with a certificate (or proof). The intended benefit is that the program consumer can locally validate the certificate w.r.t. the "untrusted" program by means of a certificate checker—a process which should be much simpler, efficient, and automatic than generating the original proof. The practical uptake of PCC greatly depends on the existence of a variety of enabling technologies which allow both to prove programs correct and to replace a costly verification process by an efficient checking procedure on the consumer side. In this work we propose *Abstraction-Carrying Code* (ACC), a novel approach which uses abstract interpretation as enabling technology. We argue that the large body of applications of abstract interpretation to program verification is amenable to the overall PCC scheme. In particular, we rely on an expressive class of safety policies which can be defined over different abstract domains. We use an *abstraction* (or abstract model) of the program computed by standard static analyzers as a certificate. The validity of the abstraction on the consumer side is checked in a single-pass by a very efficient and specialized abstract-interpreter. We believe that ACC brings the expressiveness, flexibility and automation which is inherent in abstract interpretation techniques to the area of mobile code safety. While our approach is general, we develop it for concreteness in the context of constraint logic programming. We have implemented and benchmarked ACC within the `Ciao` system preprocessor. The experimental results show that the checking phase is indeed faster than the proof generation phase, and that the sizes of certificates are reasonable. Moreover, the preprocessor is based on compile-time (and run-time) tools for the certification of CLP programs with resource consumption assurances. Indeed, as an application of ACC we illustrate that, thanks to the fact that abstract interpretation techniques allow inferring very rich information, our technique can be used to generate certificates which specify complex program properties including traditional safety issues but also *resource*-related properties like, for example, the kind of load the code is going to pose.

1 Introduction

One of the most important challenges which computing research faces today is the development of security techniques for verifying that the execution of a program (possibly) supplied by an untrusted source is *safe*, i.e., that it meets certain properties according to a predefined *safety policy.* Proof-Carrying Code (PCC) [33] 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, and packaged along with the code. The consumer who receives

[*]Preliminary versions of this work appeared in the Proceedings of LPAR'04 [2] and COCV'04 [1]. This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-38059 *ASAP* project and by the Spanish Ministry of Science and Education under the MCYT TIC 2002-0055 *CUBICO* project. Part of this work was performed during a research stay of Elvira Albert and Germán Puebla at UNM supported by respective grants from the Secretaría de Estado de Educación y Universidades, Spanish Ministry of Science and Education. Manuel Hermenegildo is also supported by the Prince of Asturias Chair in Information Science and Technology at UNM.

[‡]DSIP, Universidad Complutense de Madrid, Avda. Complutense s/n, E-28040 Madrid, Spain. Email: `elvira@sip.ucm.es`

[§]Facultad de Informática, Technical University of Madrid, Spain. Email: `{german,herme}@fi.upm.es`

[¶]Depts. of Computer Science and Electrical and Computer Eng., U. of New Mexico. Email: `herme@unm.edu` or downloads the code+certificate package can then run a *checker* which by an efficient inspection of the code and the certificate, can verify the validity of the certificate and thus compliance with the safety policy. The key benefit of this "certificate-based" approach to mobile code safety is that the consumer's task is reduced from the level of proving to the level of checking, a task that should be much simpler, efficient, and automatic than generating the original certificate.

The practical uptake of PCC greatly depends on the existence of a variety of enabling technologies which allow:

1. defining *expressive safety policies* covering a wide range of properties,
2. solving the problem of how to *automatically generate the certificates* (i.e., automatically proving the programs correct), and
3. replacing a costly verification process by an efficient checking procedure on the consumer side.

The main approaches applied up to now are based on either theorem proving or type analysis. For instance, in PCC the certificate is originally [33] a proof in first-order logic of certain *verification conditions* and the checking process involves ensuring that the certificate is indeed a valid first-order proof. λProlog is used in [3] to define a representation of lemmas and definitions which helps keep the proofs small. A recent proposal [5] uses temporal logic to specify security policies in PCC. In Typed Assembly Languages [29], the certificate is a type annotation of the assembly language program and the checking process involves a form of type checking. Each of the different approaches possess their own set of stronger and weaker points. Depending on the particular safety property and the available computing resources in the consumer, some approaches are more suitable than others. In some cases the priority is to reduce the size of the certificate as much as possible in order to fit in small devices or to cope with scarce network access (as in, e.g., Oracle-based PCC [35] or Tactic-based PCC [4]), whereas in other cases the priority is to reduce the checking time (as in, e.g., standard PCC [33] or lightweight bytecode verification [26]). As a result of all this, a successful certificate infrastructure should have a wide set of enabling technologies available for the different requirements.

In this work we propose *Abstraction-Carrying Code* (ACC), a novel approach which uses *abstract interpretation* [14] as enabling technology to handle the above practical (and difficult) challenges. Abstract interpretation is now a well established technique which has allowed the development of very sophisticated global static program analyses that are at the same time automatic, provably correct, and practical. The basic idea of abstract interpretation is to infer information on programs by interpreting ("running") them using abstract values rather than concrete ones, thus obtaining safe approximations of the behavior of the program. The technique allows inferring a wide range of properties about programs. This includes data structure shape (with pointer sharing), bounds on data structure sizes, and other operational variable instantiation properties, as well as procedure-level properties such as determinacy, termination, non-failure, and bounds on resource consumption (time or space cost). Our proposal, ACC, opens the door to the applicability of the above domains as enabling technology for PCC. Figure 1 presents an overview of ACC. The certification process carried out by the code producer is depicted to the left of the figure while the checking process performed by the code consumer appears to the right. In particular, ACC has the following fundamental elements which can handle the three aforementioned challenges:

1. The first element, common to both producer and consumers, is the Safety Policy. We rely on an expressive class of safety policies based on "abstract"—i.e. symbolic—properties over different abstract domains. Our framework is parametric w.r.t. the abstract domain(s) of interest, which gives us generality and expressiveness.
2. The next element at the producer's side is a fixed point-based static Analyzer which automatically infers an abstract model (or simply *abstraction*) of the mobile code which can then be used to prove that this code is safe w.r.t. the given policy in a straightforward way. In particular, we identify the specific *subset* of the analysis results which is sufficient for this purpose.

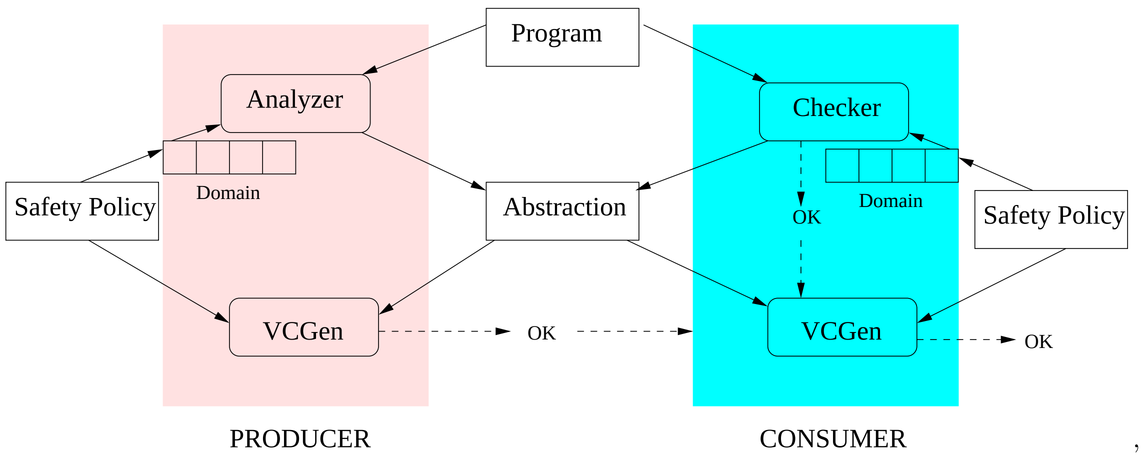


Figure 1: Abstraction-Carrying Code

3. The verification condition generator, VCGen in the figure, generates, from the initial safety policy and the abstraction, a *Verification Condition* (VC), which can be proved only if the execution of the code does not violate the safety policy. As in standard PCC methods, this process is performed also by the consumers in order to have a trustworthy VC.

4. Finally, a simple, easy-to-trust (analysis) checker at the consumer's side verifies the validity of the information on the mobile code. It is indeed a specialized abstract interpreter whose key characteristic is that it does not need to iterate in order to reach a fixed point (in contrast to standard analyzers).

While ACC is a general approach, for concreteness we develop herein an incarnation of it in the context of (Constraint) Logic Programming, (C)LP, because this paradigm offers a good number of advantages, an important one being the maturity and sophistication of the analysis tools available for it. Also for concreteness, we build on the algorithms of (and report on an implementation on) `CiaoPP` [21], the abstract interpretation-based preprocessor of the `Ciao` multi-paradigm CLP system. `CiaoPP` uses modular, incremental abstract interpretation as a fundamental tool to obtain information about programs. In `CiaoPP`, the semantic approximations thus produced have been applied to perform high- and low-level optimizations during program compilation, including transformations such as multiple abstract specialization, parallelization, partial evaluation, resource usage control, and program verification. More recently, novel and promising applications of such semantic approximations are being applied in the more general context of program development. We report on our extension of `CiaoPP` to incorporate ACC and on how this instantiation of ACC already shows promising results.

The article is organized as follows. Section 2 introduces some notation and preliminary notions on CLP and abstract interpretation. In Section 3, we review our abstract interpretation-based approach to program verification. Section 4 describes the assertion language which is used to define our safety policy. Section 5 discusses the generation of program abstractions. In Section 6, we present the verification condition generator which attests compliance of the abstraction with respect to the safety policy. In Section 7, we introduce an abstract interpretation-based checker which validates the safety certificate in the consumer. Section 8 reports some experiments performed in the `CiaoPP`-based implementation. In Section 9, we sketch a promising application of our framework based on safety certificates with resource consumption assurances. Finally, Section 10 discusses the work presented in this article and related work.

2 Preliminaries

We assume familiarity with constraint logic programming [24] (CLP) and the concepts of abstract interpretation [14] which underlie most analyses in CLP. The remaining of this section introduces some notation and recalls preliminary concepts on these topics.

2.1 Constraint Logic Programming

Terms are constructed from variables (e.g., X) and functors (e.g., f). We denote by $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ the *substitution* σ with $\sigma(X_i) = t_i$ for all $i = 1, \ldots, n$ (with $X_i \neq X_j$ if $i \neq j$) and $\sigma(X) = X$ for any other variable X, where t_i are terms. The *identity* substitution, which we denote by id is such that $\forall X\ id(X) = X$. A *renaming* is a substitution ρ for which there exists the inverse ρ^{-1} such that $\rho\rho^{-1} \equiv \rho^{-1}\rho \equiv id$. We say that a renaming ρ is a *renaming substitution* of term t_1 w.r.t. term t_2 if $t_2 = \rho(t_1)$.

A *literal* has the form $p(t_1, ..., t_n)$ where p/n is a procedure name (predicate symbol) and the t_i are terms. Most real-life (C)LP programs use procedures which are not defined in the program (module) being developed. Thus, procedures are classified into *internal* and *external*. Internal procedures are defined in the current program (module), whereas external procedures are not. Examples of external procedures include the traditional “built-in” (predefined) procedures, such as constraints, basic input/output facilities (e.g., `open`). We will also consider as external procedures those defined in a different module, procedures written in another language, etc. We assume the existence of a boolean function external s.t. external($p(t_1, ..., t_n)$) succeeds iff the procedure p/n is external. A *goal* is a finite sequence of literals. A *rule* is of the form H`:-`B where H, the *head*, is a literal and B, the *body*, is a possibly empty finite sequence of literals. A *CLP program*, or *program*, is a finite set of rules.

Example 2.1 (Running Example) *The main procedure,* `create_streams/2`*, of the following CLP program receives a list of numbers which correspond to certain file names, and returns in the second argument the list of file handlers (*streams*) associated to the (opened) files:*

```
create_streams([],[]).
create_streams([N|NL],[F|FL]):-
        number_codes(N,ChInN), app("/tmp/",ChInN,Fname),
        safe_open(Fname,write,F), create_streams(NL,FL).

app([],L,L).
app([X|Xs],L,[X|Y]):-
        app(Xs,L,Y).

safe_open(Fname,Mode,F):-
        atom_codes(File,Fname), open(File,Mode,F).
```

The call `number_codes(N,ChInN)` receives the number `N` and returns in `ChInN` the list of the ASCII codes of the characters comprising the representation of `N` as a decimal number. Then, it uses the well-known list concatenation procedure `app/3`. The call `atom_codes(File,Fname)` receives in `Fname` a list of ASCII codes and returns in `File` the atom (constant) made up of the corresponding characters. Also, a call such as `open(File,Mode,F)` opens the file named `File` and returns in `F` the stream associated to the file. The argument `Mode` can have any of the values: `read`, `write`, or `append`. Procedures `number_codes/2`, `atom_codes/2`, and `open/3` are ISO-standard Prolog procedures, and thus they are available in `CiaoPP` (in the `iso-prolog` library).

In the following, we assume that all rule heads are normalized, i.e., H is of the form $p(X_1, ..., X_n)$ where $X_1, ..., X_n$ are distinct free variables. This is not restrictive since programs can always be normalized, and it will facilitate the presentation of the algorithms later. For instance, the procedure `create_streams` of Example 2.1 in normalized form is as follows.

```
create_streams(X,Y):- X=[],Y=[].
create_streams(X,Y):- X=[N|NL], Y=[F|FL],
        number_codes(N,ChInN), T="/tmp/",
        app(T,ChInN,Fname),Mode=write,
        safe_open(Fname,Mode,F), create_streams(NL,FL).
```

2.2 Abstract Interpretation

A distinguishing feature of our approach is that a class of safety policies can be defined for the different *abstract domains* available in the system. In particular, safety properties are expressed as *substitutions* in the context of an abstract domain (D_α) which is simpler than the corresponding *concrete domain* (D). An abstract value is a finite representation of a, possibly infinite, set of actual values in the concrete domain. Our approach relies on the abstract interpretation theory [14], where the set of all possible abstract semantic values which represents D_α is usually a complete lattice or cpo which is ascending chain finite. However, for this study, abstract interpretation is restricted to complete lattices over sets, both for the concrete $\langle 2^D, \subseteq \rangle$ and abstract $\langle D_\alpha, \sqsubseteq \rangle$ domains. Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : 2^D \rightarrow D_\alpha$, and *concretization* $\gamma : D_\alpha \rightarrow 2^D$, such that $\forall x \in 2^D : \gamma(\alpha(x)) \supseteq x$ and $\forall y \in D_\alpha : \alpha(\gamma(y)) = y$. In general $\sqsubseteq$ is induced by $\subseteq$ and α. Similarly, the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of 2^D in a precise sense. In this framework an *abstract property* is defined as an abstract substitution which allows us to express properties, in terms of an abstract domain, that the execution of a program must satisfy. The description domain we use in our examples is the following *regular type* domain [15].

Example 2.2 (*regular type domain*) *We refer to the* regular type *domain as* eterms*, since it is the name it has in* `CiaoPP`*. Abstract substitutions in* eterms *[45], over a set of variables V, assign a* regular type *to each variable in V. We use in our examples* `term` *as the most general type (i.e.,* `term` $\equiv \top$ *corresponds to all possible terms). We also allow parametric types such as* `list(T)` *which denotes lists whose elements are all of type T. Type* `list` *is equivalent to* `list(term)`*. Also,* `list(T)` $\sqsubseteq$ `list` $\sqsubseteq$ `term` *for any type T. The least general substitution $\bot$ assigns the empty set of values to each variable and indicates that the corresponding program point is unreachable.*

Apart from predefined types, in the eterms domain, the user can define regular types by means of Regular Unary Logic *programs [20].*[1] *For instance, in the context of mobile code, it is a safety issue whether the code tries to access files which are not related to the application in the machine consuming the code. A very simple safety policy can be to enforce that the mobile code only accesses temporary files. In a UNIX system this can be controlled (under some assumptions) by ensuring that the file resides in the directory* `/tmp/`*. The following regular type* `safe_name` *defines this notion of safety, where the* `regtype` *declarations are used in* `CiaoPP` *to define new regular types:*

```
:- regtype safe_name/1.
safe_name("/tmp/"||L) :- list(L,alphanum_code).

:- regtype alphanum_code/1.
alphanum_code(X):- alpha_code(X).
alphanum_code(X):- num_code(X).

:- regtype alpha_code/1.
alpha_code(X):- member(X,"abcdefghijklmnopqrstuvwzyz").
alpha_code(X):- member(X,"ABCDEFGHIJKLMNOPQRSTUVWXYZ").
```

The regular type `num_code(X):- member(X,"0123456789.eE+-")` *is predefined in the system. The abstract property made up of substitution* $\{X \mapsto$ `safe_name`$\}$ *expresses that* `X` *is bound to a string*

[1]Additional such types are inferred by the system independently of the pre- or user-defined types.

which starts with "`/tmp/`" followed by a list of alpha-numerical characters (we use "`||`" to denote list concatenation). In the following, we write simply `safe_name(X)` to represent it. The crucial point here is that `safe_names` cannot contain (back-)slashes. As a result, there is no way `safe_names` can access files outside the `/tmp/` directory.

3 Abstract Interpretation-based Verification

In this section, we briefly describe an abstract interpretation-based approach to program verification [38, 9, 36] which constitutes the basis for the certification process carried out in ACC.

We consider the important class of semantics referred to as *fixed-point semantics.* In this setting, a (monotonic) semantic operator (which we refer to as S_P) is associated to each program P. This S_P function operates on a semantic domain which is generally assumed to be a complete lattice or, more generally, a chain-complete partial order. The meaning of the program (which we refer to as $[\![P]\!]$) is defined as the least fixed point of the S_P operator, i.e., $[\![P]\!] = \mathrm{lfp}(S_P)$. A well-known result is that if S_P is continuous, the least fixed point is the limit of an iterative process involving at most ω applications of S_P and starting from the bottom element of the lattice.

Both program verification and debugging compare the *actual semantics* of the program, i.e., $[\![P]\!]$, with an *intended semantics* for the same program, which we denote by $\mathcal{I}$. This intended semantics embodies the user's requirements, i.e., it is an expression of the user's expectations. The classical verification problem of verifying that P is *partially correct* w.r.t. $\mathcal{I}$ can be formulated as follows:

$$\boxed{P \text{ is \textit{partially correct} w.r.t. } \mathcal{I} \text{ if } [\![P]\!] \subseteq \mathcal{I}}$$

However, using the exact either actual or intended semantics for automatic verification and debugging is in general not realistic, since the exact semantics can be infinite, too expensive to compute, only partially known, etc. An alternative approach is to work with approximations of the semantics. This is interesting, among other reasons, because the technique of abstract interpretation can provide *safe* approximations of the program semantics. For now, we assume that the program specification is given as a semantic value $\mathcal{I}_\alpha \in D_\alpha$. Comparison between actual and intended semantics of the program is most easily done in the same domain, since then the operators on the abstract lattice, that are typically already defined in the analyzer, can be used to perform this comparison. Thus, for comparison we need in principle $\alpha([\![P]\!])$ and we proceed as follows:

$$\boxed{P \text{ is \textit{partially correct} w.r.t. } \mathcal{I}_\alpha \text{ if } \alpha([\![P]\!]) \sqsubseteq \mathcal{I}_\alpha}$$

However, using abstract interpretation, we can usually only compute $[\![P]\!]_\alpha$, which is an approximation of $\alpha([\![P]\!])$ and it is computed by the analyzer as $[\![P]\!]_\alpha = \mathrm{lfp}(S_P^\alpha)$. The operator S_P^α is the abstract counterpart of S_P. A key idea in abstract interpretation-based verification is to use $[\![P]\!]_\alpha$ directly in debugging and verification tasks. The possible loss of accuracy due to approximation prevents full verification in general. However, and interestingly, it turns out that in many cases useful verification and debugging conclusions can still be derived by comparing the approximations of the actual semantics of a program to the (also possibly approximated) abstract intended semantics. Thus, we are interested in studying the implications of comparing $\mathcal{I}_\alpha$ and $[\![P]\!]_\alpha$. Analyses which over-approximate the actual semantics (which we denote $[\![P]\!]_{\alpha^+}$), are specially suited for proving partial correctness and incompleteness with respect to the abstract specification $\mathcal{I}_\alpha$. In particular, a sufficient condition for demonstrating that P is partially correct is as follows:

$$\boxed{P \text{ is \textit{partially correct} w.r.t. } \mathcal{I}_\alpha \text{ if } [\![P]\!]_{\alpha^+} \sqsubseteq \mathcal{I}_\alpha}$$

In our approach, we compare $[\![P]\!]_\alpha$ directly to the (also approximate) intention which is given in terms of *assertions* [37]. Such assertions are linguistic constructions which allow expressing properties of programs, as we will see in the next section.

4 An Assertion Language to Specify the Safety Policy

The purpose of a *safety policy* is to specify precisely the conditions under which the execution of a program is considered safe. Assertions are syntactic objects which allow expressing a wide variety of high-level properties of (in our case CLP-) programs. Examples are assertions which state information on *entry points* to a program module, assertions which describe properties of predefined procedures (built-ins), assertions which provide some type declarations, cost bounds, etc. They will allow us to have an expressive class of safety policies in the context of (constraint) logic programs. Intuitively, we assume that a program will be accepted at the receiving end, provided all properties stated within assertions can be checked, i.e., the intended semantics $\mathcal{I}_\alpha$ expressed in the assertions determines the safety policy. This can be a policy agreed a priori or exchanged dynamically.

The original assertion language [37] available in `CiaoPP` is composed of several assertion schemes. Among them, we only consider the following two schemes for the purpose of this article, which intuitively correspond to the traditional pre- and postconditions on procedures:

$calls(B, \{\lambda^1_{Pre}; \ldots ; \lambda^n_{Pre}\})$**:** This assertion scheme is used to express properties which should hold in *any* call to a given procedure, similarly to the traditional *precondition.* B is a *procedure descriptor*, i.e., it has a procedure name (predicate symbol) as main functor and all arguments are distinct free variables, and λ^i_{Pre}, $i = 1, \ldots, n$, are abstract properties of execution states. The resulting assertion should be interpreted as "in all activations of B *at least* one property λ^i_{Pre} should hold in the calling state."

$success(B, [\lambda_{Pre},]\lambda_{Post})$**:** This assertion scheme is used to describe a *postcondition* which must hold on each success state for a given procedure. B is a procedure descriptor, and λ_{Pre} and λ_{Post} are abstract properties about execution states. λ_{Pre} is optional and must be evaluated w.r.t. the description at the calling state to the procedure while condition λ_{Post} is evaluated at the success state. If the optional λ_{Pre} is present, then λ_{Post} is only required to hold in those success states which correspond to call states satisfying λ_{Pre}. Note that several success assertions for the same procedure and with different λ_{Pre} may be given.

Therefore, abstract properties λ_{Pre} and λ_{Post} in assertions allow us to express conditions, in terms of an *abstract domain*, that the execution of a program must satisfy. Each condition is an abstract substitution corresponding to the variables in some literal.

In existing approaches, safety policies usually correspond to some variants of type safety (which may also control the correct access of memory or array bounds [34]). In our system, the co-existence of several domains allows expressing a wide range of properties using the assertion language. They include several classes of safety policies based on modes, types, non-failure, termination, determinacy, non-suspension, non-floundering, cost bounds, and their combinations. In Section 9 we will show by means of an example, how the combination of such properties can be useful in a particular application of the ACC framework to resource-aware certification/verification.

In general, it is the task of the compiler designer to define the safety policies associated with the predefined system procedures. In addition to these assertions, the user can optionally provide further assertions manually for user-defined procedures. As depicted in Figure 1, given an initial program P, we first define its Safety Policy by means of a set of assertions AS in the context of an abstract domain D_α. The domain is appropriately chosen among a repertoire of Domains available in the system. The assertions are obtained from the assertions for system procedures and those provided by the user. Let us illustrate this process by means of an example.

Example 4.1 (Safety Policy) *Figure 2 shows the assertions which are relevant to the program in our running example. The first four rows correspond to* `calls` *assertions, whereas the last three are* `success` *assertions. Out of the four* `calls`*, the first three are predefined in the system. For example, the first one states that calls to* `number_codes/2` *have to be performed with the first argument bound to a number or the second argument bound to a list of* `num_code`*, which is a predefined type that includes the ASCII characters required for representing floating point (and*

calls(number_codes(X,Y), {(num(X);list(Y,num_code))})
calls(atom_codes(X,Y), {(constant(X);string(Y))})
calls(open(X,Y,_Z), {constant(X),io_mode(Y)})
calls(safe_open(Fname,_,_), {safe_name(Fname)})
success(number_codes(X,Y), $\top$, {num(X),list(Y,num_code)})
success(atom_codes(X,Y),$\top$, {constant(X),string(Y)})
success(open(X,Y,Z), $\top$, {constant(X),io_mode(Y),stream(Z)})

Figure 2: Assertions for the example

integer) numbers as strings. The last `calls` assertion is for procedure `safe_open` and provides a simple way to guarantee that all subsequent calls to `open` are safe. It can be read as "the calling conventions for procedure `safe_open` *require that the first argument be a `safe_name`." The success assertion for `open` is predefined in our system and requires, upon success, the first variable to be of type `constant`, the second a proper `io_mode` and the last one of type `stream`.*

In contrast to traditional approaches, assertions are not compulsory for every procedure. Thus, the user can decide how much effort to put into writing assertions: the more of them there are, the more complete the intended semantics (and thus the partial correctness) of the program is described and more possibilities to detect problems. Indeed, pre- and post-conditions are frequently provided by programmers since they are often easy to write and very useful for generating program documentation. Nevertheless, the analysis algorithm is able to obtain safe approximations of the program behavior even if no assertions are given. This is not always the case in other approaches such as classical program verification, in which loop invariants are actually required. Such invariants are hard to find and existing automated techniques are not always sufficient to infer them, so that often they have to be provided by hand.

Note that the three `success` assertions shown in the example correspond to library procedures. Such assertions can be verified beforehand, and indeed they are verified in our implementation assuming that calls to such procedures satisfy the corresponding `calls` assertions. Unfortunately, calls assertions for library procedures cannot be verified beforehand, since programs which use such procedures may do so incorrectly. Thus, in order to guarantee that the safety policy holds we only need to prove that the four *calls* assertions in Figure 2 hold.

5 Generation of Program Abstractions

This section introduces (part of) the *certification* process, represented to the left of Figure 1, carried out by the producer, namely the generation of an *abstraction* which safely approximates the behaviour of the program. The generation of the verification condition from this certificate is discussed in the next section.

5.1 Using Analysis Results as Certificates

Consider a program P. Once the safety policy is specified as a set of assertions, we have available the program specification $\mathcal{I}_\alpha$ as a semantic value of D_α. Now, we use abstract interpretation in order to compute $[\![P]\!]_\alpha$, which is an approximation of $\alpha([\![P]\!])$. To do this, a standard Analyzer is run. Then, the certification method is based on the following idea:

> *An abstraction of the program, $[\![P]\!]_\alpha$, computed by abstract interpretation-based analyzers can play the role of certificate for attesting program safety if $[\![P]\!]_\alpha \sqsubseteq \mathcal{I}_\alpha$.*

Global program analysis is becoming a practical tool in constraint logic program compilation in which information about calls, answers, and the effect of the constraint store on variables at different program points is computed statically [23, 44, 32, 42, 8]. Essentially, an analyzer returns an *abstraction* of P's execution in terms of the abstract domain D_α. The underlying theory, formalized in terms of abstract interpretation [14], and the related implementation techniques are well understood for several general types of analysis and, in particular, for top-down analysis of Prolog [17, 16, 6, 32, 28, 12]. Several generic analysis engines, such as the one implemented in the CiaoPP system [22], PLAI [32, 30], GAIA [12], and the CLP($\mathcal{R}$) analyzer [25], facilitate construction of such top-down analyzers. As shown in Figure 1 in principle the analyzer is domain–independent. This allows plugging in different abstract Domains provided suitable interfacing functions are defined. From the user's point of view, it is sufficient to specify the particular abstract domain desired during the generation of the safety assertions. Different domains yield analyzers which provide different kinds of information, degrees of accuracy, and efficiency. The core of each generic abstract interpretation-based engine is an algorithm for efficient fixed-point computation [30, 32, 11, 40].

5.2 The Analysis Graph

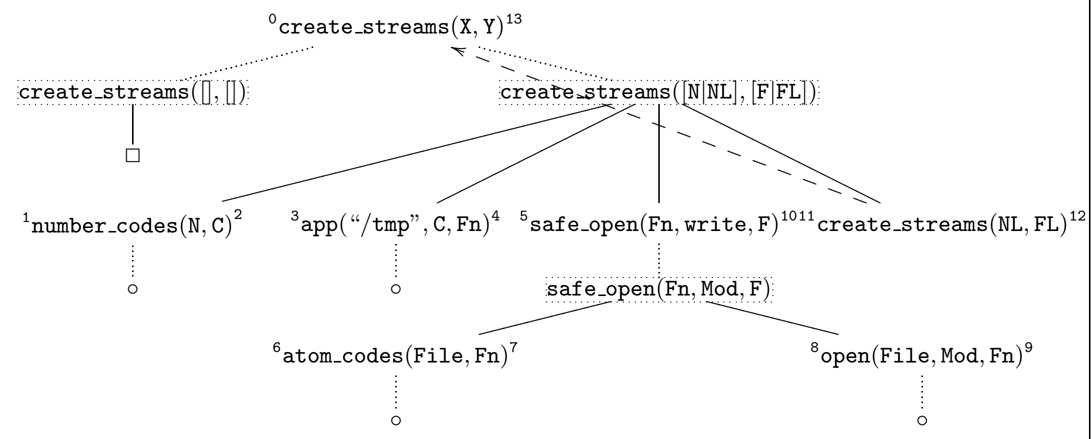
In order to analyze a program, traditional (goal dependent) abstract interpreters for (C)LP programs receive as input, in addition to the program and the abstract domain, a set S of *calling patterns*. Such calling patterns are pairs of the form $A : CP$ where A is a procedure descriptor and CP is an abstract substitution (i.e., a condition of the run-time bindings) of A expressed as $CP \in D_\alpha$.[2] Given a program P and a set S of calling patterns in the context of an abstract domain D_α, the analyzer constructs an *and–or graph* (or analysis graph) for S which can be viewed as an abstraction, i.e., a finite representation of the (possibly infinite) set of (possibly infinite) AND-OR trees explored by the concrete execution of initial calls described by S in P [6]. Finiteness of the program analysis graph (and thus termination of analysis) is achieved by considering abstract domains with certain characteristics (such as being finite, or of finite height, or without infinite ascending chains) or by the use of a *widening* operator [14].

Example 5.1 *Consider our running example and assume that we are interested in analyzing it for the call* `create_streams(X,Y)` *with initial description* $list(X, num)$*, indicating that we wish to analyze it for any call to* `create_streams/2` *with the first argument being a list of numbers. In essence the analyzer must produce the* program analysis graph *given in Figure 3.*

The graph has two sorts of nodes. Those which correspond to literals are called "OR-nodes". For instance, the OR-node $\langle$`create_streams(X,Y)` $:list(X, num) \mapsto \{list(X, num), list(Y, stream)\}\rangle$ *indicates that when the literal* `create_streams(X,Y)` *is called with description* $list(X, num)$ *the answer (or success) substitution computed is* $\{$`list(X,num)`, `list(Y,stream)`$\}$.

Those nodes which correspond to rules are called "AND-nodes". In Fig. 3, they appear within a dashed box and contain the head of the corresponding clause. Each AND-node has as children as many OR-nodes as literals there are in the body. We indicate with the symbol $\square$ *that the rule is a fact with no literals in the body. And the symbol* $\circ$ *denotes that the code for this procedure is not available (i.e., it is an external procedure). These rules are annotated by abstract descriptions (referenced by numbers* $0, \ldots, 13$ *whose corresponding description appears to the bottom of the figure) at each program point when the rule is executed from the calling pattern of the node connected to the rules. The program points are at the entry to the rule, the point between each two literals, and at the return from the call. If a child OR-node is already in the tree, it is not further expanded and the currently available answer is used. For instance, the analysis graph in Figure 3 contains two occurrences of the abstract literal* `create_streams(X,Y)` $:$ $list(X, num)$ *(modulo renaming), but only one of them has been expanded. This is depicted by an arrow from the non-expanded occurrence of* `create_streams(X,Y)` $:$ $list(X, num)$ *to the expanded one. How this program analysis graph is constructed is detailed in Example 5.3 below.*

[2]In principle, calling patterns are only required for exported procedures. The analysis algorithm is able to generate them automatically for the remaining internal procedures. Nevertheless, they can still be automatically generated by assuming $\top$ (i.e., no initial data) for all exported procedures (although the idea is to improve this information in the initial calling patterns).



^{0}create_streams(X, Y)^{13}
create_streams([], [])
create_streams([N|NL], [F|FL])
^{1}number_codes(N, C)^{2}
^{3}app("/tmp", C, Fn)^{4}
^{5}safe_open(Fn, write, F)^{10}
^{11}create_streams(NL, FL)^{12}
safe_open(Fn, Mod, F)
^{6}atom_codes(File, Fn)^{7}
^{8}open(File, Mod, Fn)^{9}

0	:	list(X, num)	7	:	constant(File), sf(Fn)
1	:	num(N)	8	:	constant(File), Mod = write
2	:	num(N), list(C, num_code)	9	:	constant(File), Mod = write, stream(F)
3	:	list(C, num_code)	10	:	sf(FN), stream(F)
4	:	list(C, num_code), sf(Fn)	11	:	list(NL, num)
5	:	sf(Fn)	12	:	list(NL, num), list(FL, stream)
6	:	sf(Fn)	13	:	list(X, num), list(Y, stream)

Figure 3: Analysis Graph

For a given program and set of calling patterns there may be many different analysis graphs. However, for a given set of initial calling patterns, program, and abstract operations on the abstract domain, there is a unique *least analysis graph* which gives the most precise information possible.

5.3 The Analysis Algorithm

In this section we on one hand introduce an analysis algorithm which computes an analysis graph and, on the other hand, identify the fragment of the information stored in such graph which is sufficient in order to play the role of safety certificate. The analysis algorithm presented is an extension of the generic analysis algorithm in [22] in order to handle (constraint) logic programs with *external* (including imported) procedures.

In order to deal with real programs that include external procedures, our analyzers rely again on assertion-based techniques [37]. Intuitively, our analyzer *trusts* the information stated in the *success* assertions for external procedures by considering the answer pattern in them a safe approximation of their concrete answer patterns. However, presenting all details regarding analysis of modular programs is out of the scope of this article (details can be found in [39]). In our algorithm, the analysis of external procedures is handled by the function Atrust which is introduced below.

The program analysis graph is implicitly represented in the algorithm [22] by means of two data structures, the *answer table* and the *dependency arc table*. Given the information in these, it is straightforward to construct the graph and the associated program point annotations. The answer table contains entries of the form $A : CP \mapsto AP$ where A is always a base form. This corresponds to OR-nodes in the analysis graph of the form $\langle A : CP \mapsto AP \rangle$. A dependency arc is of the form $H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$. This is interpreted as follows: if the rule with H_k as head is called with description CP_0 then this causes literal $B_{k,i}$ to be called with description CP_2. The remaining part CP_1 is the program annotation just before $B_{k,i}$ is reached and contains information about all variables in rule k. As we will see below, dependency arcs are used for forcing recomputation until a fixed-point is reached.

Intuitively, the analysis algorithm is a graph traversal algorithm which places entries in the answer table and dependency arc table as new nodes and arcs in the program analysis graph are encountered. To capture the different graph traversal strategies used in different fixed-point algorithms, we use a priority queue. Thus, the third, and final, structure used in the algorithm is a *prioritized event queue*. Events are of three forms:

- $newcall(A : CP)$ which indicates that a new calling pattern for literal A with description CP has been encountered.
- $arc(R)$ which indicates that the rule referred to in R needs to be (re)computed from the position indicated.
- $updated(A : CP)$ which indicates that the answer description to calling pattern A with description CP has been changed.

Our analysis algorithm is given in Figure 4. It is defined in terms of five abstract operations on the abstract domain D_α of interest:

- $\mathsf{Arestrict}(CP, \mathsf{V})$ performs the abstract restriction of a description CP to the set of variables in the set V, denoted $vars(V)$;
- $\mathsf{Aextend}(CP, \mathsf{V})$ extends the description CP to the variables in the set V;
- $\mathsf{Atrust}(A, CP)$ returns an answer pattern for an external procedure A which is a safe approximation of the concrete answer patterns for all call patterns described by CP.
- $\mathsf{Aconj}(CP_1, CP_2)$ performs the abstract conjunction of two descriptions;
- $\mathsf{Alub}(CP_1, CP_2)$ performs the abstract disjunction of two descriptions.

Apart from the parametric description domain-dependent functions, the algorithm has several other undefined functions. The functions add_event and next_event respectively add an event to the priority queue and return (and delete) the event of highest priority.

When an event being added to the priority queue is already in the priority queue, a single event with the maximum of the priorities is kept in the queue. When an arc $H_k : CP \Rightarrow [CP'']B_{k,i} : CP'$ is added to the dependency arc table, it replaces any other arc of the form $H_k : CP \Rightarrow [\_]B_{k,i} : \_$ in the table and the priority queue. Similarly when an entry $H_k : CP \mapsto AP$ is added to the answer table, it replaces any entry of the form $H_k : CP \mapsto \_$. Note that the underscore (_) matches any description, and that there is at most one matching entry in the dependency arc table or answer table at any time. The function initial_guess returns an initial guess for the answer to a new calling pattern. The default value is $\perp$ but if the calling pattern is more general than an already computed call then its current value may be returned.

The algorithm centers around the processing of events on the priority queue in main_loop, which repeatedly removes the highest priority event and calls the appropriate event-handling function. When all events are processed it calls remove_useless_calls. This procedure traverses the dependency graph given by the dependency arcs from the initial calling patterns S and marks those entries in the dependency arc and answer table which are reachable. Those remaining are removed.

The function new_calling_pattern initiates processing of the rules in the definition of the internal literal A, by adding arc events for each of the first literals of these rules, and determines an initial answer for the calling pattern and places this in the table. The function add_dependent_rules adds arc events for each dependency arc which depends on the calling pattern $A : CP$ for which the answer has been updated. The function process_arc performs the core of the analysis. It performs a single step of the left-to-right traversal of a rule body. If the literal $B_{k,i}$ is not for an external procedure, the arc is added to the dependency arc table. The current answer for the call $B_{k,i} : CP_2$ is conjoined with the description CP_1 from the program point immediately before $B_{k,i}$ to obtain the description for the program point after $B_{k,i}$. This is either used to generate a new arc event to process the next literal in the rule if $B_{k,i}$ is not the last literal; otherwise the new answer for the rule is combined with the current answer in insert_answer_info. The function get_answer processes a literal. The current answer to that literal for the current description is looked up; then this answer is extended to the variables in the rule the literal occurs in and conjoined with the current description. The function lookup_answer first looks up an answer for the given calling pattern in the answer table and if it is not found and the procedure is local, it places a *newcall* event. Otherwise (the procedure is external), it uses Atrust to obtain a safe answer pattern. Finally, insert_answer_info, updates the answer table entry when a new answer is found.

```
analyze($S$)
    foreach $A : CP \in S$
        add_event($newcall(A : CP)$)
    main_loop()

main_loop()
    while $E :=$ next_event()
        if ($E = newcall(A : CP)$)
            new_calling_pattern($A : CP$)
        elseif ($E = updated(A : CP)$)
            add_dependent_rules($A : CP$)
        elseif ($E = arc(R)$)
            process_arc($R$)
    endwhile
    remove_useless_calls($S$)

new_calling_pattern($A : CP$)
    foreach rule $A_k$ :- $B_{k,1}, \ldots, B_{k,n_k}$
        $CP_0 :=$
            Aextend($CP, vars(B_{k,1}, \ldots, B_{k,n_k})$)
        $CP_1 :=$ Arestrict($CP_0, vars(B_{k,1})$)
        add_event($arc($
            $A_k : CP \Rightarrow [CP_0]\ B_{k,1} : CP_1)$)
    $AP :=$ initial_guess($A : CP$)
    if ($AP \neq \bot$)
        add_event($updated(A : CP)$)
    add $A : CP \mapsto AP$ to answer table

add_dependent_rules($A : CP$)
    foreach arc of the form
            $H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$
            in graph
        where there exists renaming $\sigma$
            s.t. $A : CP = (B_{k,i} : CP_2)\sigma$
        add_event($arc($
            $H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2)$)

process_arc($H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$)
    if (not external($B_{k,i}$))
        add $H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$
        to dependency arc table
    $W := vars(A_k$ :- $B_{k,1}, \ldots, B_{k,n_k})$
    $CP_3 :=$ get_answer($B_{k,i} : CP_2, CP_1, W$)
    if ($CP_3 \neq \bot$ and $i \neq n_k$)
        $CP_4 :=$ Arestrict($CP_3, vars(B_{k,i+1})$)
        add_event( $arc($
            $H_k : CP_0 \Rightarrow [CP_3]\ B_{k,i+1} : CP_4)$)
    elseif ($CP_3 \neq \bot$ and $i = n_k$)
        $AP_1 :=$ Arestrict($CP_3, vars(H_k)$)
        insert_answer_info($H : CP_0 \mapsto AP_1$)

get_answer($L : CP_2, CP_1, W$)
    $AP_0 :=$ lookup_answer($L : CP_2$)
    $AP_1 :=$ Aextend($AP_0, W$)
    return Aconj($CP_1, AP_1$)

lookup_answer($A : CP$)
    if (there exists a renaming $\sigma$ s.t.
        $\sigma(A : CP) \mapsto AP$ in answer table)
        return $\sigma^{-1}(AP)$
    elsif (not external($A$))
        add_event($newcall(\sigma(A : CP))$)
        where $\sigma$ is a renaming s.t.
        $\sigma(A)$ is in base form
        return $\bot$
    else % external($A$)
        $AP :=$ Atrust($A, CP$)
        add ($A : CP \mapsto AP$) to answer table
        return $AP$

insert_answer_info($H : CP \mapsto AP$)
    $AP_0 :=$ lookup_answer($H : CP$)
    $AP_1 :=$ Alub($AP, AP_0$)
    if ($AP_0 \neq AP_1$)
        add ($H : CP \mapsto AP_1$) to answer table
        add_event($updated(H : CP)$)
```

Figure 4: Fixed-point Analyzer

Theorem 5.2 (correctness [22]) *For a program P and calling patterns S, the analysis algorithm of Figure 4 returns an answer table and dependency arc table which represents the least program analysis graph of P and S.*

A central idea in ACC is that, for certifying program safety, it suffices to send the information stored in the analysis answer table. The theory of abstract interpretation guarantees that the answer table is a safe approximation of the runtime behavior (see [6, 22, 40] for details). In contrast to this analysis algorithm, a simple checker can be designed for validating the answer table without requiring the use of the arc dependency table at all (as we show in Sect. 7).

5.4 An Example

The following example briefly illustrates the operation of the fixed-point algorithm. It shows how the `create_streams` program is analyzed, to obtain the program analysis graph shown in Figure 3. We use in our example well-known abstract operations for a regular type domain, in particular, the operations formalized in [45] for the *eterms* domain described in Example 2.2. For the analysis of library procedures, we assume that the parametric routine Atrust returns the abstract descriptions which appear in the three `success` assertions shown in Figure 2 for the corresponding library procedures. This can be done safely because, as already mentioned, such `success` assertions have been verified beforehand.

Example 5.3 *Analysis begins from an initial set S of calling patterns. In our example S contains the single calling pattern* `create_streams(X,Y)`:$list(X, num)$. *For brevity, variables which do not appear in abstract substitutions are assumed to be "`term`". Also $nil(X)$ indicates that X is the empty list. The first step in the algorithm is to add the initial calling patterns as a newcall event to the priority queue. After this the priority queue contains*

  $newcall($`create_streams(X,Y)`:$list(X, num))$

and the answer and dependency arc tables are empty. The newcall event is taken from the event queue and processed as follows: for each rule defining `create_streams`, *an arc is added to the priority queue which indicates the rule body which must be processed from the initial literal. An entry for the new calling pattern is added to the answer table with an initial guess of $\perp$ as the answer. The data structures are now:*

priority queue:
  $arc($`create_streams(X,Y)`:$list(X, num) \Rightarrow [list(X, num)]$ `X=[]`:$list(X, num))$
  $arc($`create_streams(X,Y)`:$list(X, num) \Rightarrow [list(X, num)]$ `X=[N|NL]`:$list(X, num))$
answer table:
  `create_streams(X,Y)`:$list(X, num) \mapsto \perp$
dependency arc table:
  no entries

An arc on the event queue is now selected for processing, say the first. The routine get_answer *is called to find the answer pattern to the literal* `X=[]` *with description $list(X, num)$. As the literal is an external constraint, the parametric routine* Atrust *is used. It returns the answer pattern $\{list(X, num), nil(X)\}$. A new arc is added to the priority queue which indicates that the second literal in the rule body must be processed. The priority queue is now*

$arc($`create_streams(X,Y)`:$list(X, num) \Rightarrow [list(X, num), nil(X)]$ `Y=[]`:$\{\ \})$
$arc($`create_streams(X,Y)`:$list(X, num) \Rightarrow [list(X, num)]$ `X=[N|NL]`:$list(X, num))$

The answer and dependency arc table remain the same.

Again, an arc on the event queue is selected for processing, say the first. As before, get_answer *and* Atrust *are called to get the next annotation $\{list(X, num), nil(X), nil(Y)\}$. This time, as there are no more literals in the body, the answer table entry for* `create_streams(X,Y)`:$list(X, num)$ *is updated.* Alub *is used to find the least upper bound of the new answer $\{list(X, num), nil(X), nil(Y)\}$ with the old answer $\perp$. This gives $\{list(X, num), nil(X), nil(Y)\}$. The entry in the answer table is updated, and an updated event is placed on the priority queue. The data structures are now:*

priority queue:
  $updated($`create_streams(X,Y)`:$list(X, num))$
  $arc($`create_streams(X,Y)`:$list(X, num) \Rightarrow [list(X, num)]$ `X=[N|NL]`:$list(X, num))$
answer table:
  `create_streams(X,Y)`:$list(X, num) \mapsto \{list(X, num), nil(X), nil(Y)\}$
dependency arc table:
  no entries

The updated event can now be processed. As there are no entries in the dependency arc table, nothing in the current program analysis graph depends on the answer to this call, so nothing needs to be recomputed. The priority queue now contains

$arc($`create_streams(X,Y)`:$list(X, num) \Rightarrow [list(X, num)]$ `X=[N|NL]`:$list(X, num))$

The answer and dependency arc table remain the same. Similarly to before we process the arc, giving rise to the new priority queue

$arc($`create_streams(X,Y)`$:list(X,num) \Rightarrow [list(X,num), num(N), list(NL,num)]$ `Y=[F|FL]:{ }`$)$.

The arc is processed to give the priority queue

$arc($`create_streams(X,Y)`$:list(X,num) \Rightarrow [list(X,num), num(N), list(NL,num), Y=[F|FL]]$
`number_codes(N,ChInN)`$:num(N))$

Note that ***CiaoPP*** *creates the regular type* `rt2` *to represent a term whose top-level functor is a list constructed with* `F` *as head and* `FL` *as tail. For simplicity, we just write this description as* `Y=[F|FL]` *in the following.*

This time, because `number_codes(N,ChInN)` *is an external literal, the parametric routine Atrust is used and no dependency is stored (as success patterns for external procedures are never updated). As a result, the data structures are now:*

priority queue:
$arc($`create_streams(X,Y)`$:list(X,num) \Rightarrow [list(X,num), num(N),$
$list(NL,num), Y=[F|FL], list(ChInN, num\_code)]$ `T="/tmp/":{ }`$)$
answer table:
`create_streams(X,Y)`$:list(X,num) \mapsto list(X,num), nil(X), nil(Y)$
`number_codes(N,ChInN)`$:num(N) \mapsto num(N), list(ChInN, num\_code)$
dependency arc table:
no entries

Following the analysis, we process the unique arc in the priority queue, obtaining the new priority queue:

priority queue:
$arc($`create_streams(X,Y)`$:list(X,num) \Rightarrow [list(X,num), num(N),$
$list(NL,num), Y=[F|FL], list(ChInN, num\_code), T=\text{"}/tmp/\text{"}]$
`app(T,ChInN,Fname)`$:T=\text{"}/tmp/\text{"}, list(ChInN, num\_code))$

Similarly as done so far, and skipping the intermediate steps, we obtain finally the following data structures in which, the dependency arc table contains a different arc for each one of the literals in the second rule of `create_streams` *which are not external.*

priority queue:
$arc($`create_streams(X,Y)`$:list(X,num) \Rightarrow CP$ `create_streams(NL,FL)`$:list(NL,num))$
answer table:
`create_streams(X,Y)`$:list(X,num) \mapsto \{list(X,num), nil(X), nil(Y)\}$
`number_codes(N,ChInN)`$:num(N) \mapsto \{num(N), list(ChInN, num\_code)\}$
`app(T,ChInN,Fname)`$:\{list(ChInN, num\_code), T=\text{"}/tmp/\text{"}\} \mapsto$
$\{list(ChInN, num\_code), T=\text{"}/tmp/\text{"}, sf(Fname)\}$
`safe_open(Fname,Mode,F)`$:\{sf(Fname), Mode=write\} \mapsto$
$\{sf(Fname), Mode=write, stream(F)\}$
`atom_codes(File,Fname)`$:sf(Fname) \mapsto \{constant(File), sf(Fname)\}$
`open(File,Mode,F)`$:\{constant(File), Mode=write\} \mapsto$
$\{constant(File), Mode=write, stream(F)\}$

where

$$\begin{array}{rcll} CP & = & [& list(X,num), num(N), list(NL,num),\ Y=[F|FL], list(ChInN, num\_code), \\ & & & sf(Fname), constant(File), Mode=write, stream(F),\ T=\text{"}/tmp/\text{"} \quad] \end{array}$$

It is interesting to note that `CiaoPP` creates the auxiliary type:

```
sf("/tmp/"||A):-list(A,num_code).
```

to represent strings which start with the prefix `"/tmp/"` and continue with a list of type `num_code`. Since all `num_codes` are also `alphanum_codes`, it is clear that `sf` $\sqsubseteq$ `safe_name`. This will allow our system to infer that calls to `open` *performed within this program satisfy the simple safety policy discussed in Example 4.1. Therefore, the information stored in the answer table is sufficient to attest the safety policy. Also, we use the notation Var = constant to denote that the system generates a new type whose only element is this constant, as it happens: for `write`, in the entries for `safe_open` and `open` and, for `"/tmp/"`, in the entry for `app`.*

Now, the call `get_answer` *for the recursive call* `create_streams(NL,FL)`:$list(NL, num)$ *is made. The answer table is looked up to find the answer and, appropriately renamed and restricted to the variables in the call, gives* $AP_0 = \{nil(NL), nil(FL), list(NL, num)\}$*. This description is extended to all variables (no change) and then conjoined with CP to give the next annotation* $\{nil(X), nil(Y), list(X, num), list(Y, stream)\}$*. We take the least upper bound of this answer with the old answer in the table, giving* $\{list(X, num), list(Y, stream)\}$*. The answer table will replace the current annotation for* `create_streams(X,Y)`: $list(X, num)$ *by* `create_streams(X,Y)`: $list(X, num) \mapsto \{list(X, num), list(Y, stream)\}$*, adding the processed arc to the dependency arc table.*

As the answer has changed, an updated event is added to the priority queue. The priority queue contains:

$updated($`create_streams(X,Y)`:$list(X, num))$

The updated event is processed by looking in the dependency arc table for all arcs which have a body literal which is a variant of `create_streams(X,Y)`:$list(X, num)$ *and adding these arcs to the priority queue to be reprocessed. There is only one (the last processed arc). After reprocessing this arc we obtain as answer* $\{list(X, num), list(Y, stream)\}$*. Taking the least upper bound of this with the old answer, the result is identical to the old answer, hence no updated event is added to the priority queue. As there are no events on the priority queue, the analysis terminates.*

As a result of the whole analysis, the answer table computed by `CiaoPP` contains (among others) these entries:

Procedure	*Calling Pattern*	*Success Pattern*
`create_streams(A,B)`	`list(A,num)`	`list(A,num),list(B,stream)`
`number_codes(A,B)`	`num(A)`	`num(A),list(B,num_code)`
`app(A,B,C)`	`A="/tmp/", list(B,num_code)`	`A="/tmp/", list(B,num_code),sf(C)`
`safe_open(A,B,C)`	`sf(A),B=write`	`sf(A),B=write,stream(C)`
`atom_codes(A,B)`	`sf(B)`	`constant(A),sf(B)`
`open(A,B,C)`	`constant(A),B=write`	`constant(A),B=write,stream(C)`

We show in the next section that the information stored in the above table is sufficient to certify that the mobile code is safe according to the policy defined in Example 4.1.

In order to increase accuracy, analyzers are usually *multivariant* on calls (see, e.g., [22]). Indeed, though not visible in this example, `CiaoPP` incorporates a multivariant analysis, i.e., more than one triple

$$\langle A : CP_1 \mapsto AP_1\rangle, \ldots, \langle A : CP_n \mapsto AP_n\rangle$$

$n > 1$ with $CP_i \neq AP_i$ for some i, j may be computed for the same procedure descriptor A.

It is important to note that our approach would work directly in other programming paradigms, such as imperative or functional programming (the latter already covered in our current system), as long as a static analyzer/checker is available. Note that the fundamental components of the approach (fixed-point semantics and abstract interpretation) have both been widely applied also in these paradigms.

6 The Verification Condition

As part of the certification process carried out by the code producer, the verification condition generator (VCGen in Fig. 1) extracts, from the initial assertions and the abstraction, a *Verification Condition* (VC) which can be proved only if the execution of the code does not violate the safety policy. In particular, we are interested in studying the implications of comparing the intended program specification described in Section 4, denoted $\mathcal{I}_\alpha$, with the program abstraction described in Section 5, denoted $[\![P]\!]_{\alpha^+}$. Therefore, VCGen generates a VC which encodes the comparison $[\![P]\!]_{\alpha^+} \subseteq \mathcal{I}_\alpha$. If VC can be proved (marked as OK in Fig. 1), then the certificate (i.e., the abstraction) is sent together with the program P to the code consumer.

Definition 6.1 (VC – verification condition) *Let AT be an analysis answer table computed for a program P and a set of calling patterns S in the abstract domain D_α. Let S be an assertion. Then, the verification condition, $VC(S, AT)$, for S w.r.t. AT is defined as follows:*

$$VC(S,AT) ::= \begin{cases} \displaystyle\bigwedge_{\langle A:CP \mapsto AP\rangle \in AT} (\rho(CP) \sqsubseteq \lambda^1_{Prec} \vee \ldots \vee \rho(CP) \sqsubseteq \lambda^n_{Prec}) \\ \qquad\qquad \textit{if } S = \textit{calls}(B, \{\lambda^1_{Prec}; \ldots; \lambda^n_{Prec}\}) \\ \displaystyle\bigwedge_{\langle A:CP \mapsto AP\rangle \in AT} \rho(CP) \sqcap \lambda_{Prec} = \bot \vee \rho(AP) \sqsubseteq \lambda_{Post} \\ \qquad\qquad \textit{if } S = \textit{success}(B, \lambda_{Prec}, \lambda_{Post}) \end{cases}$$

where ρ is a variable renaming substitution of A w.r.t. B.

If AS is a finite set of assertions, then its verification condition, $V(AS, AT)$, is the conjunction of the verification conditions of the elements of AS.

Roughly speaking, the VC generated according to Def. 6.1 is a conjunction of boolean expressions (possibly containing disjunctions) whose validity ensures the consistency of a set of assertions w.r.t. the answer table computed by *Analysis*. It distinguishes two different cases depending on the kind of assertion. For *calls* assertions, the VC requires that at least one precondition λ^i_{Prec} be a safe approximation of each existing abstract calling patterns for the literal B. In the case of *success* assertions, there are two cases for them to hold. The first one indicates that the precondition is never satisfied and, thus, the assertion trivially holds (and the postcondition does not need to be tested). The second corresponds to the case in which the success substitutions computed by analysis for the procedure are equal or more particular than the one required by the assertion.

Example 6.2 (Verification Condition) *Consider the answer table generated in Example 5.3 and the calls and success assertions of Figure 2. According to Def. 6.1, the VC is:*

$$\begin{array}{rcl} (\texttt{num(X)} & \sqsubseteq & (\texttt{num(X)};\texttt{list(Y,num\_code)})\wedge \\ \texttt{sf(Y)} & \sqsubseteq & (\texttt{constant(X)};\texttt{string(Y)})\wedge \\ \texttt{constant(X)},\texttt{Y}=\texttt{write} & \sqsubseteq & \texttt{constant(X)},\texttt{io\_mode(Y)}\wedge \\ \texttt{sf(X)} & \sqsubseteq & \texttt{safe\_name(X)} \end{array}$$

Each conjunct corresponds to a calls assertion in Fig. 2 in the same order they appear there. As already mentioned, success assertions for predefined procedures are verified beforehand.

The validity of the whole conjunction can be easily proved by taking into account the following (trivial) relations between the elements in the domain:

$$\begin{array}{rcl} \texttt{sf(X)} & \sqsubseteq & \texttt{string(X)} \\ \texttt{X}=\texttt{write} & \sqsubseteq & \texttt{io\_mode(X)} \end{array}$$

Note that the first two conjuncts contain a disjunction in the right hand condition. In the second one, the condition $\texttt{sf(Y)} \sqsubseteq (\texttt{constant(X)};\texttt{string(Y)})$ *holds because* $\texttt{sf(Y)} \sqsubseteq \texttt{string(Y)}$.

Therefore, upon creating the answer table and generating the VC, the validity of the whole boolean condition is checked by resolving each conjunct separately. Note that each conjunct consists of comparisons of pairs of abstract substitutions, which simply return either true or false but do not compute any substitution. This validation may yield three different possible status: i) the VC is indeed checked and the answer table is considered a valid abstraction (marked as OK), ii) it is disproved, and thus the certificate is not valid and the code is definitely not safe to run (we should obviously correct the program before continuing the process); iii) it cannot be proved nor disproved. The latter case happens because some properties are undecidable and the analyzer performs approximations in order to always terminate. Therefore, it may not be able to infer precise enough information to verify the conditions. The user can then provide a more refined description of initial calling patterns or choose a different, finer-grained, domain. Although, it is not shown in the picture, in both the ii) and iii) cases, the certification process needs to be restarted until achieving a VC which meets i).

The following theorem states the soundness of the VC. Intuitively, it amounts to saying that if the VC holds, then the execution of the program will preserve all safety assertions. Following the notation of [33], we write $\rhd VC$ when VC is valid.

Theorem 6.3 (Soundness of the Verification Condition) *Let AT be an analysis answer table for a program P and a set of calling patterns S in an abstract domain D_α (as defined in Figure 4). Let AS be a set of assertions. Let $VC(AS, AT)$ be the verification condition for AS w.r.t. AT (generated as stated in Def. 6.1). If $\rhd VC(AS, AT)$, then P satisfies all assertions in AS for all computations described by S.*

This result directly derives from the fact that the static analysis algorithm computes a safe approximation of the states reached during computation (see Theorem 5.2).

7 Checking Safety in the Consumer

The checking process performed by the consumer is illustrated on the right hand side of Figure 1. Initially, the supplier sends the program P together with the certificate to the consumer. To retain the safety guarantees, the consumer can provide a new set of assertions, denoted $\mathcal{I}'_\alpha$, which specifies the Safety Policy required by this particular consumer. It should be noted that ACC is very flexible in that it allows different implementations on the way the safety policy is provided. Clearly, the same assertions AS used by the producer, denoted $\mathcal{I}_\alpha$, can be sent to the consumer. But, more interestingly, the consumer can decide to impose a weaker safety condition, i.e., $\mathcal{I}_\alpha \subseteq \mathcal{I}'_\alpha$, which can be proved with the submitted abstraction since $[\![P]\!]_\alpha \subseteq \mathcal{I}_\alpha$. Also, the imposed safety condition can be stronger, i.e., $\mathcal{I}'_\alpha \subseteq \mathcal{I}_\alpha$ and it may not be provable if it is not implied by the current abstraction $[\![P]\!]_\alpha$ (which means that the code would be rejected). From the provided assertions, the consumer must generate again a trustworthy VC and use the incoming certificate to efficiently check that the VC holds. Thus, in the *validation* process, a code consumer not only checks the validity of the answer table but it also (re-)generates a trustworthy VC. The validation of AT is carried out by the Analysis Checker. The re-generation of VC (and its corresponding validation) is identical to the process already discussed in the previous section.

7.1 Fixed-point Checking

Although global analysis is now routinely used as a practical tool, it is still unacceptable to run the whole *Analysis* to validate the certificate since it involves considerable cost. One of the main reasons is that the analysis algorithm is an iterative process which often computes answers (repeatedly) for the same call due to possible updates introduced by further iterations. At each iteration, the algorithm has to manipulate rather complex data structures—which involve performing updates, lookups, etc.—until the fixed point is reached. The whole validation process is centered around the following observation:

The checking algorithm can be defined as a very simplified "one-pass" analyzer.

The *Analysis* process can be understood as: $Analysis = fixed-point(analysis\_step)$. I.e., a process which repeatedly performs a traversal of the analysis graph (denoted by $analysis\_step$)

until the computed information does not change. The idea is that the simple, non-iterative, *analysis_step* process can play the role of abstract interpretation-based checker (or simply analysis checker). In other words, `check` $\equiv$ *analysis_step*. Intuitively, since the certification process should provide a correct fixed-point result (i.e., $[\![P]\!]_{\alpha+}$) as certificate, an additional analysis pass over this fixed point should not change the result, since if it thus then the current answer table is not a valid abstraction of the program. Thus, in our context, as long as the answer table is valid, one single execution of *analysis_step* is required to validate the certificate.

7.2 The Checking Algorithm

The next definition presents our *abstract interpretation-based checking* algorithm. It receives as an additional input a Certificate (which is the analysis fixed point). In a single traversal, it constructs a program analysis graph by using the information in Certificate. The algorithm is devised as a graph traversal procedure which places entries in a *local* answer table, AT, as new nodes in the program analysis graph are encountered. Thus, it handles two distinct answer tables: the local AT + the incoming Certificate. The final goal of the checking is to reconstruct the analysis graph and compare the results with the information stored in Certificate. As long as Certificate is valid, both results coincide and, thus, the certificate is guaranteed to be valid w.r.t. the program.

Definition 7.1 (Analysis Checker) *Let P be a normalized program and S be a set of calling patterns in the abstract domain D_α. Let* Certificate *be an answer table (or safety certificate) as defined in Figure 4. The validation of* Certificate *is performed by the procedure* `check` *depicted in Figure 5. The algorithm uses a local answer table, AT, to compute the results (initially it does not contain any entry).*

Following the presentation of the analysis algorithm in Section 5.3, we assume that the program P and the answer table are global parameters throughout the algorithm. The checking algorithm proceeds as follows: as in the analysis algorithm, the procedure process_arc is aimed at computing the resulting description CP_a after processing a given literal $B_{k,i}$. The computed result is used to process the next literal in the rule when $B_{k,i}$ is not the last one. Otherwise, the computed result constitutes indeed the computed answer for the rule. The difference w.r.t. the analyzer is that the answer is *combined* with the corresponding answer supplied by the certification process in Certificate. If Certificate is valid, the comparison should hold; otherwise the process prompts an error and the program is not safe to run. Therefore, no control structure is needed in order to guarantee that a fixed point is reached. This eliminates the need for the "event queue" of the analysis algorithm in Fig. 4. Moreover, since only one traversal of the analysis graph is to be performed, no detailed dependency information is required. This eliminates the need for the "dependency arc table" of the analysis algorithm. As a result, check is a suitable procedure for determining the validity of the certificate.

The following theorem ensures that algorithm check is able to validate safety certificates which are stored in a valid analysis answer table.

Theorem 7.2 (partial correctness) *Let P be a program, let S be a set of calling patterns in an abstract domain D_α. Let* Certificate *be an answer table for P and S as defined in Figure 4. Then,* `check`$(S,$ Certificate$)$ *terminates and, if it returns* Valid*, then* Certificate *is an abstraction of P and S.*

The theorem is implied by the definition of fixed point and the fact that check is a single pass of a correct *Analysis* algorithm [22].

Another issue is the efficiency of the checking algorithm. Our point to justify an efficient behavior of `check` for validating an answer table is that it performs a single graph traversal. Indeed, for a regular type domain, [10] demonstrates that directional type-checking for logic programs is fixed-parameter linear. The next section reports experimental evidence of efficiency issues.

```
check($S$, Certificate)
    foreach $A : CP \in S$
        process_node($A : CP$, Certificate)
    return Valid

process_node($A : CP$, Certificate)
    if ($\exists$ a renaming $\sigma$ s.t. $\sigma(A : CP \mapsto AP)$ in Certificate)
        then add $(A : CP \mapsto AP)$ to AT
        else return Error
    if (not external($A$))
        foreach rule $A_k \leftarrow B_{k,1}, \ldots, B_{k,n_k}$ in $P$
            $W := vars(A_k, B_{k,1}, \ldots, B_{k,n_k})$
            $CP_b$ :=Aextend($CP, vars(B_{k,1}, \ldots, B_{k,n_k})$)
            $CPR_b$ := Arestrict($CP_b, B_{k,1}$)
            foreach $B_{k,i}$ in the rule body $i = 1, ..., n_k$
                $CP_a$ := process_arc($B_{k,i} : CPR_b, CP_b, W$, Certificate)
                if ($i <> n_k$) then $CPR_a$ := Arestrict($CP_a, var(B_{k,i+1})$)
                $CP_b := CP_a$
                $CPR_b := CPR_a$
            $AP_1$ := Arestrict($CP_a, vars(A_k)$)
            $AP_2$ := Alub($AP_1, \sigma^{-1}(AP)$)
            if $AP <> AP_2$ then return Error
    else % external($A$))
        $AP_1$ := Atrust($A, CP$)
        if $\sigma^{-1}(AP) <> AP_1$ then return Error

process_arc($B_{k,i} : CPR_b, CP_b, W$, Certificate)
    if ($\nexists$ a renaming $\sigma$ s.t. $\sigma(B_{k,i} : CPR_b \mapsto AP')$ in AT)
    then
        process_node ($B_{k,i} : CPR_b$, Certificate)
    $AP_1$ := Aextend ($\rho^{-1}(AP), W$) where $\rho$ is a renaming s.t.
                                    $\rho(B_{k,i} : CPR_b \mapsto AP)$ in AT
    $CP_a$ := Aconj ($CP_b, AP_1$)
    return $CP_a$
```

Figure 5: Abstract Interpretation-based Checking in `CiaoPP`

7.3 An Example

We describe the more representative steps that algorithm check performs in order to validate the answer table of Example 5.3.

Example 7.3 *Consider the answer table, called* Certificate*, of Example 5.3. First, procedure* `process_node` *looks up an answer for the initial calling pattern in* Certificate *and adds the entry*

$\langle$`create_streams(X,Y)` $:list(X, num) \mapsto AP = \{list(X, num), list(Y, stream)\}\rangle$

to the answer table AT (note that, for short, we use AP to denote this particular answer pattern). Since there are two rules defining `create_streams` *the outermost loop performs two iterations:*

Iter 1. *We start by describing the processing of the first rule (although the order is irrelevant). Since the first literal* `X=[]` *in the rule body is a constraint, its description is computed within procedure* process_arc *by adding its abstract description, i.e.,* $\{nil(X)\}$*, to the initial description* $\{list(X, num)\}$*, resulting in* $\{nil(X), list(X, num)\}$*. Similarly, the analysis for the second constraint adds* $\{nil(Y)\}$ *to the former description producing* $\{nil(X),$

$nil(Y), list(X, num)\}$. Upon exiting the innermost loop, the disjunction of this description with the answer stored in Certificate *is calculated:*

$AP_2 := Alub(\{nil(X), nil(Y), list(X, num)\}, AP)$

since $nil(X) \sqsubseteq list(X, num)$ and $nil(Y) \sqsubseteq list(Y, stream)$, then $AP_2 = AP$. Thus, the certificate holds for this rule.

Iter 2. *In the second iteration, we find eight literals in the rule body. Thus, the innermost loop performs the following eight steps. The first two traversals deal with the constraints for* `X` *and* `Y`*, and are similar to* **Iter 1**. *They produce the calling pattern*

$$\{list(X, num), num(N), list(NL, num), Y = [F|FL]\}$$

The next literal, `number_codes`*, in the rule body is an external procedure, thus,* process_node *uses the parametric routine* Atrust *which gives the answer $\{num(N), list(ChInN, num\_code)\}$ for it. This answer is conjoined with the description of the program point immediately before the literal, i.e.:*

$$\{list(X, num), num(N), list(NL, num), Y = [F|FL], list(ChInN, num\_code)\}$$

The remaining intermediate literals are dealt in a similar way (see Example 5.3 for more specific details). Let us just consider the processing of the recursive call to `create_streams`*, for which we get as final description:*

$$\begin{array}{lll} CP & = & [\ list(X, num), num(N), list(NL, num),\ Y = [F|FL], list(ChInN, num\_code), \\ & & \quad sf(Fname), constant(File), Mode = write, stream(F),\ T = "/tmp/"] \end{array}$$

Now, process_node *finds out that AT already contains an answer pattern for this procedure. Then, both calling patterns are conjoined: $CP_a := Aconj(CP, AP)$ and restricted to variables* `X` *and* `Y`*, obtaining $CP_a = AP$ as final result. Upon return from* process_arc*, it performs the disjunction of the computed answer with the answer supplied by* Certificate*: $AP_2 := Alub(CP_a, AP)$. Since $CP_a = AP$ and also the result, $AP_2 = AP$, coincides with the one in the certificate, the proof is validated and the algorithm terminates in a single graph traversal for the initial query. Note that in the analysis example, there is an additional full iteration due to the existence of update events which make the analyzer re-process all arcs which depend on a calling pattern whose answer has been updated. It is well known that several passes over the program are often needed to reach a fixed point.*

8 Experimental Results

In this section we show some experimental results aimed at studying two crucial points for the practicality of our proposal: the checking time as compared to the analysis time, and the size of certificates. We have implemented the checker as a simplification of the generic abstract interpretation system of `CiaoPP`. It should be noted that this is an efficient, highly optimized, state-of-the-art analysis system and which is part of a working compiler. Both the analysis and checker are parametric w.r.t. the abstract domain. In these experiments they both use the same implementation of the domain-dependent functions of the *sharing+freeness* domain [31]. We have selected this domain because the information it infers is very useful for reasoning about instantiation errors, which is a crucial aspect for the safety of logic programs. The whole system is implemented in Ciao 1.11#200 [7] with compilation to bytecode. All of our experiments have been performed on a Pentium 4 at 2.4GHz and 512MB RAM running GNU Linux RH9.0. The Linux kernel used is 2.4.25, customized with the *hrtime* patch to provide improved precision and resolution in time measurements.

	Analysis			Checking			Speedup	
Bench	P_A	An	T_A	P_C	Ch	T_C	A/C	T_A/T_C
aiakl	2	87	89	2	71	72	1.2	1.2
ann	22	452	474	18	254	272	1.8	1.7
bid	4	56	60	4	35	38	1.6	1.6
boyer	9	143	151	7	85	92	1.7	1.6
browse	3	14	17	3	12	15	1.2	1.2
deriv	2	86	88	1	19	20	4.6	4.4
grammar	2	10	12	2	9	11	1.1	1.1
hanoiapp	2	25	26	2	16	18	1.5	1.5
mmatrix	1	13	14	1	10	11	1.3	1.3
occur	2	16	18	2	10	12	1.7	1.6
progeom	2	13	15	2	9	11	1.5	1.4
read	9	792	801	8	488	497	1.6	1.6
qplan	13	1411	1424	11	962	973	1.5	1.5
qsortapp	1	20	21	1	12	14	1.6	1.5
query	5	11	15	4	9	12	1.2	1.3
rdtok	8	141	149	6	43	49	3.3	3.1
serialize	2	40	42	2	17	19	2.3	2.2
warplan	8	173	181	7	108	115	1.6	1.6
witt	16	196	212	14	72	86	2.7	2.5
zebra	3	94	97	3	90	92	1.1	1.0
Overall							1.63	1.61

Table 1: Checking Time

8.1 Checking Time

Table 1 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. A relatively wide range of programs has been used as benchmarks. They are the same ones used in [22], where they are described in some detail. For each benchmark, the columns for `Analysis` are the following: $\mathtt{P}_A$ is the time required by the *preprocessing phase*, in which program rules are processed and stored in the format required by the analyzer. The *analysis* time proper is shown in column `An`. The actual time needed for analysis –the sum of these two times– is shown in column $\mathtt{T}_A$. Similarly, in the case of checking, three columns are shown. The preprocessing phase, $\mathtt{P}_C$, includes asserting the certificate in addition to asserting the program to be analyzed. As the figures show, the overhead required for asserting the certificate is negligible. Column `Ch` is the time for executing the checking algorithm. Finally, $\mathtt{T}_C$ is the total time for checking. The columns under `Speedup` compare analysis and checking times. As can be seen in columns `A/C` and $\mathtt{T}_A/\mathtt{T}_C$, the checking algorithm is faster than the analysis algorithm in all cases. The actual speedup ranges from almost none, as in the case of zebra, to over four times faster in the case of deriv. The last row summarizes the results for the different benchmarks using a weighted mean, which places more importance on those benchmarks with relatively larger analysis times. We use as weight for each program its actual analysis time. We believe that this weighted mean is more informative than the arithmetic mean, as, for example, doubling the speed in which a large and complex program is analyzed (checked) is more relevant than achieving this for small, simple programs. Overall, the speedup is 1.63 in just analysis time, or 1.61 if we also take into account the preprocessing time. We believe that the achieved speedup is significant taking into account that `CiaoPP`'s analyzer for this domain is highly optimized and converges very efficiently [40]. However, it is to be expected that, for other domains and implementations, the relative gains will be higher.

	Source	Byte Code		Certificate	
Bench	Source	ByteC	B/S	Cert	C/S
aiakl	1555	3805	2.4	3090	2.0
ann	12745	43884	3.4	24475	1.9
bid	4945	10376	2.1	5939	1.2
boyer	11010	32522	3.0	12300	1.1
browse	2589	8467	3.3	1661	0.6
deriv	957	4221	4.4	288	0.3
grammar	1598	3182	2.0	1259	0.8
hanoiapp	1172	2264	1.9	2325	2.0
mmatrix	557	1053	1.9	880	1.6
occur	1367	6903	5.0	1098	0.8
progeom	1619	3570	2.2	2148	1.3
read	11843	24619	2.1	25359	2.1
qplan	9983	33472	3.4	20509	2.1
qsortapp	664	1176	1.8	2355	3.5
query	2090	8833	4.2	531	0.3
rdtok	13704	15354	1.1	6533	0.5
serialize	987	3801	3.9	1779	1.8
warplan	5203	23971	4.6	15305	2.9
witt	17681	41760	2.4	19131	1.1
zebra	2284	5396	2.4	4058	1.8
Overall	1		2.66		1.44

Table 2: Certificate Size

8.2 Certificate Size

Table 2 shows our experimental results regarding certificate size, coded in compact (*fastread*) format, for the different benchmarks and compares it to the size of the source code for the same program and to the size of the corresponding bytecode. To make this comparison fair, 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 exception handlers for any executable). The results show the size of the certificate to be quite reasonable. It ranges from 0.3 times the size of the source code (for deriv) to 3.5 (in the case of qsortapp). Overall, it is 1.44 times the size of the source code. We consider this acceptable since in general (C)LP programs are quite compact (up to 10 times more compact than equivalent imperative programs). In fact, the size of source plus certificate is smaller (1+1.44) than that of the bytecode (2.66).

9 An Application of Abstraction Carrying Code

As already mentioned, abstract interpretation techniques allow inferring very rich information. This information will allow us to specify *safety policies* involving not only traditional safety issues (e.g., that the code will not write on specific areas of the disk) but also *resource*-related issues (e.g., that it will not compute for more than a given amount of time, or that it will not take up an amount of memory or other resources above a certain threshold) and, thus, achieving further expressiveness.

We illustrate through a simple example the fundamental intuition behind the application of ACC to resource-awareness. Consider the naive reverse Ciao program in Figure 6 written in functional syntax. The `entry` assertion states information on the *entry points* to the program module. In our case, initial calls to `nrev` must be performed with a totally instantiated list (i.e., a ground list of terms) in the first argument and a variable in the second one (the output), i.e., it will indeed be used as a function. Assume also that the cost unit is the number of procedure calls. With these assumptions, the exact cost function of procedure `append` is $\texttt{Cost}_{append}(x, y) = x + 1$,

```
:- module(reverse, [nrev/2], [assertions,functions]).
:- entry nrev/2 : {list, ground} * var.

nrev( [] )       := [].
nrev( [H|L] )    := ~append( nrev(L), [H] ).

append([],X)     := X.
append([H|X],Y) := [ H | append(X,Y) ].
```

Figure 6: The naive reverse program.

```
:- true pred nrev(A,B)      :  ( list(A), var(B) )
                            => ( list(A), list(B) )
                            + ( not_fails, covered, is_det, mut_exclusive ).

:- true pred nrev(A,B)      :  ( ground(A), var(B), mshare([[B]]) )
                            => ( ground(A), ground(B) ).
```

Figure 7: CiaoPP compiler output (types, modes, determinacy, non-failure).

where x and y are the sizes (lengths) of the first and second input lists respectively. Note that this cost function does not depend on the size of the second argument of append. Also, based on this cost function, the exact cost function of procedure nrev is $\texttt{Cost}_{nrev}(n) = 0.5\ n^2 + 1.5\ n + 1$, where n is the size (length) of the input list. In order to obtain a lower-bound approximation of the previous cost functions, CiaoPP first performs the following analyses (all using abstract interpretation techniques):

– A mode (and sharing) analysis. This determines which arguments (or parts of them) are inputs and which are outputs for each constructor operation, procedure and procedure call, as well as the dependencies between any variables (pointers) in the data structures passed via these arguments.
– A type analysis. This infers the types for all program variables. Note that type declarations are not compulsory in the language, so the relevant type definitions may also have to be inferred.
– A determinacy analysis. It requires the results of type and mode analysis, and it detects which procedures and procedure calls are deterministic.
– A non-failure analysis. This also requires the results of type and mode analysis, and can detect procedures and goals that can be guaranteed not to fail, i.e., to produce at least one solution or not terminate. The need for a non-failure analysis stems from an interesting problem with estimating lower bounds: in general it is necessary to account for the possibility of failure of a call to the procedure (because of, e.g., an inadmissible argument) leading to a trivial lower bound of 0.
– Inference of size metrics for relevant arguments. It is based on the type information.

The results of these analyses for nrev ($[\![P]\!]_\alpha$ for these domains), as produced by CiaoPP in the form of assertions are shown in Figure 7 (the information for append/3 has been left out for brevity). Once all this information is obtained, the work done by (recursive) rules is determined. To this end, it is first necessary to be able to estimate the size of input arguments in the procedure calls in the body of the procedure, relative to the sizes of the input arguments to the procedure, using the inferred metrics. The size of an output argument in a procedure call depends, in general, on the size of the input arguments in that call. For this reason, for each output argument, CiaoPP uses an expression which yields its size as a function of the input data sizes. For this, an abstraction of procedure definitions called a data dependency graph is used, built using all the abstract information inferred previously. The following steps are then performed:

```
:- true pred nrev(A,B)      :  ( list(A), var(B) )
                            => ( list(A), list(B),
                                 size_lb(A,length(A)), size_lb(B,length(A)),
                                 size_ub(A,length(A)), size_ub(B,length(A))
                            +  ( not_fails, covered, is_det, mut_exclusive,
                                 steps_lb(0.5*exp(length(A),2)+1.5*length(A)+1),
                                 steps_ub(0.5*exp(length(A),2)+1.5*length(A)+1)).

:- true pred nrev(A,B)      :  ( ground(A), var(B), mshare([[B]]) )
                            => ( ground(A), ground(B) ).
```

Figure 8: CiaoPP compiler output (including sizes and cost).

```
:- comp    nrev(_,_)  + ( not_fails, is_det, terminates ).   % A1
:- comp    nrev(_,_)  + seff(free).                          % A2
:- comp    nrev(A,_)  + steps_ub( o(exp(length(A),2)) ).     % A3
```

Figure 9: Some assertions for the nrev/2 program.

– The data dependency graphs are used to determine the relative sizes of variable bindings at different program points.
– The size information is used to set up difference equations representing the computational cost of procedures.
– Abstractions (lower and upper bounds) of the solutions of these difference equations are then obtained which provide the lower/upper-bound procedure cost and data size functions.

It is beyond the scope of this paper to fully explain all the techniques involved in inferring this information (see, e.g., [21, 18, 19] and their references). For illustration purposes, the concrete output from CiaoPP obtained after performing this process for the nrev program is presented in Figure 8. This output includes the *assertion* (simplified for brevity):

```
:- true pred nrev(A,B) : ( list(A), var(B) )
  => ( list(A), list(B), size_lb(B, length(A)))
   + ( not_fails, is_det, steps_lb(0.5*exp(length(A),2)+1.5*length(A)+1)).
```

Such a "pred" assertion specifies in a combined way properties of both: ":" the entry (i.e., upon calling) and "=>" the exit (i.e., upon success) points of all calls to the procedure, as well as some global properties of its execution. The assertion above, with a "true" prefix, expresses that the compiler has proved that procedure nrev will produce as output a list of numbers B, whose length is at least (size_lb) equal to the length of the input list, that the procedure will never fail (i.e., at least an output value will be computed for any possible input), that it is deterministic (only one solution will be produced as output for any input), and that a lower bound on its computational cost (steps_lb) is $0.5\ length(A)^2 + 1.5\ length(A) + 1$ execution steps (where the cost measure used in the example is again the number of procedure calls, but it can be any other arbitrary measure).

Then, "check" assertions can be added to a program in order to state its partial specification $\mathcal{I}_\alpha$. For our example, let us assume that the assertions shown in Figure 9 (the check prefix, meaning that such assertions are part of the specification and must be checked, is assumed when no prefix is given, as in the example) are given as specification for the nrev program. The properties used in these assertions, such as not_fails, terminates, costs and types, are external in the example from system libraries. These check assertions play the role of integrity constraints: if their properties do not hold at the corresponding program points (procedure call, procedure exit, etc.), the program is incorrect. Comp assertions specify *global* properties of the execution of

```
:- checked comp    nrev(_,_)  + ( not_fails, is_det, terminates ).   % A1
:- checked comp    nrev(_,_)  + seff(free).                          % A2
:- checked comp    nrev(A,_)  + steps_ub( o(exp(length(A),2)) ).     % A3
```

Figure 10: `CiaoPP` compiler output (assertion checking).

a procedure. These include complex properties such as determinacy or termination and are in general not amenable to run-time checking. They can also be restricted to a subset of the calls using "`:`".

Concretely, the first assertion (`A1`) in Figure 9 states that `nrev` should never fail, that it should be deterministic, and terminate. In addition, and directly related to our resource-awareness objective, assume that we know that the consumer will only accept tasks of polynomial (actually, at most quadratic) complexity, and only those which are purely computational, i.e., tasks that have no side effects. This safety policy can be expressed at the producer side for this particular program using the assertions `A2` and `A3`, respectively, of Figure 9. More concretely, `A2` states that it should be verified that the computation is pure in the sense that it does not produce any side effects (such as opening a file, etc.). `A3` states that it should be verified that if the procedure is called with a list in the first argument and a variable in the second one, then there is an upper bound for the cost of this procedure in $O(n^2)$, i.e., quadratic in n, where n is the length of the first list (represented as `length(A)`).

We are assuming that the code will be accepted at the receiving end, provided all assertions can be checked, i.e., that the intended semantics expressed in the assertions determines the safety condition. Stating the policy in this form will allow us to ensure during program development that we produce a program that adheres to our specifications and also to the known safety policy of the consumer. Indeed, during compilation of the `nrev` program, `CiaoPP` will check the assertions above (representing $\mathcal{I}_\alpha$) by comparing them with the assertions inferred by the types, modes, non-failure, determinism, and upper- and lower-bound cost analysis (representing $[\![P]\!]_\alpha$) and given in Figure 8. The result of compile-time checking the intended semantics (assertions in Fig. 9) against this output appears in Fig. 10 (refer also to the output of the VCGen). Note that all initial assertions have been marked as `checked`, i.e., they have been *validated*. Thus, the program has been *verified* which means that all calls to `nrev` performed within this program satisfy the resource-aware safety policy, i.e., the safety condition is met and the code is indeed safe to run, for now on the producer side.

Following the ACC scheme, (a subset of) the assertions in Fig. 8 (i.e., the analysis results) is used as the abstract *cost and safety certificate* to be used to check for a safe and efficient use of procedure `nrev` on the receiving side. The consumer will use this abstract certificate in order to accept/reject code depending on whether it adheres or not to some specification.

First of all, the code receiver proceeds to validate the certificate. This implies running the checker over the program assuming the information in Fig. 8 in the relevant points and checking that it is indeed a fixed point (and later a solution to the recurrence equations, for the case of cost analysis). This process clearly involves less effort that creating the certificate, since only a single pass over the program is required (and checking that an expression is a solution is typically cheaper than obtaining such solution). If the certificate is not valid, the code is discarded. If the certificate is valid, it is compared against the (local) specifications. The code will be accepted only if all assertions involved can be turned to "`checked`".

In our example, if we assume that the specification at the receiving end contains, e.g., (possibly a subset of) the assertions from Fig. 9, then the code would be accepted. Clearly, in order to guarantee that the cost assertion holds, the certificate has to contain upper bounds on computational cost. In contrast, let us assume that a consumer with very limited computing resources is assigned to perform a computation using this code. Then, the following "check" assertion (instead of `A3`) could perhaps represent one of the resource-related requirements at this particular node:

```
:- check comp nrev(A,_) : list * var + steps_ub( o(length(A)) ).  % A3R
```

i.e., this consumer node will not accept an implementation of `nrev` with larger complexity than linear.

In this case, given that the certificate contains the (valid) information that `nrev` will take at least 0.5 $(\mathit{length}(A))^2 + 1.5\ \mathit{length}(A) + 1$ resolution steps, this will be found incompatible with the assertion `A3R`, which requires the cost to be in $O(\mathit{length}(A))$ resolution steps. In our implementation the checker produces the following "complexity error:"

```
ERROR: false comp assertion:
  :- comp nrev(A,B) : true => steps_ub(o(length(A)))
  because in the computation the following holds:
  steps_lb(0.5*exp(length(A),2)+1.5*length(A)+1)
```

thus flagging that the program does not satisfy the efficiency requirements imposed. This means of course that the consumer will reject the code.

Note that if we had replaced `A3` with `A3R` during the compilation process at the producer end, this same error would have appeared during compilation, i.e., the compilation process would have flagged the "complexity error" at compile time (and reported this assertion as false in the output).

10 Conclusions and Related Work

We have presented *abstraction-carrying code* (ACC) as a novel enabling technology for PCC, which follows the standard strategy of associating safety certificates to programs but it is based throughout on the use of abstract interpretation techniques. We argue that ACC is highly flexible, one aspect being the parametricity on the abstract domain inherited from analysis engines as exemplified by those used in (C)LP. We argue that our proposal brings the expressiveness, flexibility and automation which is inherent in the abstract interpretation techniques developed in logic programming to this area. We have illustrated through an application of ACC for resource-aware security that our approach supports a very rich set of domains. We believe that ACC provides novel means for certifying security by enhancing mobile code with certificates which guarantee that the execution of the (in principle untrusted) code received from another node in the network is *safe* but also, as mentioned above, *efficient*, according to a predefined safety policy which includes properties related to *resource consumption*. We have illustrated the approach using the `CiaoPP` system. This system already uses a combination of abstract interpretation, abstract specialization, and a flexible assertion language, to perform program debugging, verification, and optimization with a wide variety of domains. Other approaches to abstract verification and debugging have also been proposed (see [13, 21] for further references). The system has been enhanced to produce certificates as dictated by the ACC scheme, as an integral part of the static debugging and verification performed during the program development process. A simplified version of the analysis framework of `CiaoPP` has also been developed that serves as an efficient checker of the certificates. The approach is currently being tested in a number of pervasive applications using an embedded version of the Ciao system which runs on PDAs and Gumstix processors. Ongoing work also includes the study of techniques for further reducing the size of certificates, such as only include information on recursive procedures and reducing the checking time.

Our approach differs from existing approaches to PCC in several aspects. In our case, the certificate is computed automatically on the producer side by an *abstract interpretation-based analyzer* and the certificate takes the form of a particular *subset* of the analysis results. The burden on the consumer side is reduced by using a simple one-*traversal* checker, which is a very simplified and efficient abstract interpreter which does not need to compute a fixed point.

A type-level dataflow analysis of Java virtual machine bytecode is also the basis of several existing verifiers [27, 26], and some are loosely based on abstract interpretation. These analyses allow proving that the program is correct w.r.t. type-related correctness conditions. In [41] a proposal is presented to split the type-based bytecode verification of the KVM (an embedded variant of the JVM) 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, these approaches are designed limited to types.

Let us note that the checker is part of the trusted computing base and, hence, the code consumer has to trust also the domain operations. Other approaches to PCC use logic-based verification methods as enabling technology, an example is [46] which formalizes a simple assembly language with procedures and presents a safety policy for arithmetic overflow in Isabelle/HOL. The coexistence of several abstract domains in our framework is somewhat related to the notion of *models* to capture the security-relevant properties of code, as addressed in the work on Model-Carrying Code (MCC) [43]. MCC enables code consumers to try out different security policies of interest and select one that can be statically proved to be consistent with the model associated to the untrusted code. However, models are intended to describe low-level properties and their combination has not been studied, which differs from our idea of combining (high-level) abstract domains.

Another difference between our work and other related work is that the particular instance that we have described is actually defined at the source-level, whereas in existing PCC frameworks the code supplier typically packages the certificate with the *object* code rather than with the *source* code (both are untrusted). Actually, both approaches are of interest from our point of view (and, in fact, ACC can also be applied to bytecode). Clearly, in many cases the source code is simply not available to the consumer and even when there is a choice between object and source code, using object code means reducing the trusted computing base in the consumer since there is no need for a compiler. However, open-source code is becoming much more relevant these days (in fact, `Ciao` and `CiaoPP` are themselves GNU-licensed and available in source code for reviewing and modification). As a result, it is now realistic to expect that a relatively large amount of untrusted source code is available to the consumer. The advantages of open-source with respect to safety are important since it allows inspecting the code and applying powerful techniques for program analysis and validation which allow inferring information which may be difficult to observe in low-level, compiled code. This allows handling richer properties which in turn potentially allow more expressive safety policies.

References

[1] E. Albert, G. Puebla, and M. Hermenegildo. An Abstract Interpretation-based Approach to Mobile Code Safety. In *Proc. of Compiler Optimization meets Compiler Verification (COCV'04)*, Electronic Notes in Theoretical Computer Science 132(1), pages 113–129. Elsevier - North Holland.

[2] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *11th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'04)*, number 3452 in LNAI, pages 380–397. Springer-Verlag, March 2005.

[3] A. Appel and A. Felty. Lightweight Lemmas in lambda-Prolog. In *Proc. of ICLP'99*, pages 411–425. MIT Press, 1999.

[4] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile resource guarantees for smart devices. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, LNCS. Springer, 2004. To appear.

[5] A. Bernard and P. Lee. Temporal logic for proof-carrying code. In *Proc. of CADE'02*, pages 31–46. Springer LNCS, 2002.

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

[7] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual (v1.8). The Ciao System Documentation Series–TR CLIP4/2002.1, School of Computer Science, Technical University of

Madrid (UPM), May 2002. System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.

[8] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.

[9] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

[10] W. Charatonik. Directional Type Checking for Logic Programs: Beyond Discriminative Types. In *Proc. of ESOP 2000*, pages 72–87. LNCS 1782, 2000.

[11] B. Le Charlier, O. Degimbe, L. Michael, and P. Van Hentenryck. Optimization Techniques for General Purpose Fixpoint Algorithms: Practical Efficiency for the Abstract Interpretation of Prolog. In *Workshop on Static Analysis*, pages 15–26. Springer-Verlag, September 1993.

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

[13] M. Comini, R. Gori, G. Levi, and P. Volpe. Abstract Interpretation based Verification of Logic Programs. *Electr. Notes Theor. Comput. Sci.*, 30(1), 2000.

[14] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.

[15] P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

[16] S. Debray, editor. *Journal of Logic Programming, Special Issue: Abstract Interpretation*, volume 13(1–2). North-Holland, July 1992.

[17] S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989.

[18] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[19] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[20] T. Früwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. LICS'91*, pages 300–309, 1991.

[21] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *Proc. of SAS'03*, pages 127–152. Springer LNCS 2694, 2003.

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

[23] M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

[24] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[25] A. Kelly, K. Marriott, H. Søndergaard, and P.J. Stuckey. A practical object-oriented analysis engine for CLP. *Software: Practice and Experience*, 28(2):188–224, 1998.

[26] Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.

[27] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[28] K. Marriott, H. Søndergaard, and N.D. Jones. Denotational Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.

[29] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

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

[31] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

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

[33] G. Necula. Proof-Carrying Code. In *Proc. of POPL'97*, pages 106–119. ACM Press, 1997.

[34] G. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proc. of PLDI'98.* ACM Press, 1998.

[35] G.C. Necula and S.P. Rahul. Oracle-based checking of untrusted software. In *Proceedings of POPL'01*, pages 142–154. ACM Press, 2001.

[36] G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.

[37] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, pages 23–61. Springer LNCS 1870, 2000.

[38] G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, 2000.

[39] G. Puebla, J. Correas, M. Hermenegildo, F. Bueno, M. García de la Banda, K. Marriott, and P. J. Stuckey. A Generic Framework for Context-Sensitive Analysis of Modular Programs. In M. Bruynooghe and K. Lau, editors, *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, number 3049 in LNCS, pages 234–261. Springer-Verlag, Heidelberg, Germany, August 2004.

[40] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *Proc. of SAS'96*, pages 270–284. Springer LNCS 1145, 1996.

[41] K. Rose, E. Rose. Lightweight bytecode verification. In *OOPSALA Workshop on Formal Underpinnings of Java*, 1998.

[42] V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.

[43] R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *Proc. of SOSP'03*, pages 15–28. ACM, 2003.

[44] P. Van Roy and A.M. Despain. High-Performance Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

[45] C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *International Static Analysis Symposium*, number 2477 in LNCS, pages 102–116. Springer-Verlag, September 2002.

[46] Martin Wildmoser and Tobias Nipkow. Certifying Machine Code Safety: Shallow Versus Deep Embedding. In *17th Int. Conference on Theorem Proving in Higher Order Logics*, number 3223 in LNCS. Springer, 2004.