HAL
open science

Certified Result Checking for Polyhedral Analysis of Bytecode Programs

Frédéric Besson, Thomas Jensen, David Pichardie, Tiphaine Turpin

To cite this version:

Frédéric Besson, Thomas Jensen, David Pichardie, Tiphaine Turpin. Certified Result Checking for Polyhedral Analysis of Bytecode Programs. The 5th International Symposium on Trustworthy Global Computing (TGC), 2010, Munich, Germany. inria-00537816

HAL Id: inria-00537816

https://inria.hal.science/inria-00537816v1

Submitted on 19 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Certified Result Checking for Polyhedral Analysis of Bytecode Programs*

Frédéric Besson, Thomas Jensen, David Pichardie, and Tiphaine Turpin

INRIA Rennes - Bretagne Atlantique
Campus de Beaulieu, F-35042 Rennes, France

Abstract. Static analysers are becoming so complex that it is crucial to ascertain the soundness of their results in a provable way. In this paper we develop a certified checker in Coq that is able to certify the results of a polyhedral array-bound analysis for an imperative, stack-oriented bytecode language with procedures, arrays and global variables. The checker uses, in addition to the analysis result, certificates which at the same time improve efficiency and make correctness proofs much easier. In particular, our result certifier avoids complex polyhedral computations such as convex hulls and is using easily checkable inclusion certificates based on Farkas lemma. Benchmarks demonstrate that our approach is effective and produces certificates that can be efficiently checked not only by an extracted Caml checker but also directly in Coq.

1 Introduction

Bytecode verification is an important component for making Java a trustworthy platform for mobile computing. Several researchers have investigated how to develop machine-checked bytecode verifiers in order to increase the confidence in this component itself [13, 2]. The standard bytecode verifier ensures one kind of security policy that is proved by a simple data flow analysis. The static verification of other security and safety policies (*e.g.*, to check that all array accesses are within bounds) requires more sophisticated static program analysers, which themselves are sophisticated pieces of software. A significant example of this is the state-of-the-art Astrée static analyser for C [9] which proves the absence of run-time errors for the primary flight control software of the Airbus A340 fly-by-wire system.

In this paper we show that it is possible to use advanced analysers to enhance the security of a mobile code platform by developing a machine-verified extended bytecode verifier that can check the result of such analysers. One approach would be to certify the analyser entirely within a proof checker, as done for the key components of the Java bytecode verifier [13, 2]. In previous work, Pichardie *et. al* [18, 6] formalised the theory of abstract interpretation inside the Coq proof assistant and proved the correctness of a variety of program analysers.

* This work was partially funded by the FET Global Computing project Mobius, by the Brittany region project CertLogS and the FRAE project Ascert.

This approach is ambitious since it would require to program and certify in Coq the whole analyser with all its abstract operators (least upper bound, closure, widening...) and to prove termination of the fixpoint iteration process. Formally certifying a polyhedral analyser with this technique would require a tremendous certification effort. Moreover, efficiency is a major concern when considering the expensive symbolic manipulations of a polyhedral library [12] and the problem becomes even more perceptible in a pure lambda-calculus language such as Coq.

As noticed by Leroy in the context of certified compilation [15], static analyses and optimisation heuristics are algorithms for which it is generally easier to prove the correctness of a result verifier than the algorithm itself. In this paper we apply this *result certification* methodology [20] to a polyhedral analysis [10] for an imperative, stack-oriented bytecode language with procedures, arrays and global variables. We design in parallel a polyhedral analyser and a certified result checker using the abstract interpretation theory. The analyser and the checker share the same constraint-based specification whose soundness is formally proved in Coq. The analyser uses an optimised polyhedral C library [12] to compute a post-fixpoint solution while the checker uses a certified simplified abstract domain to check the post-fixpoint. One particularity of our approach is that, in addition to the program and the post-fixpoint, the checker receives hints that enable it to use a simplified abstract domain when verifying the fixpoint. In particular, the expensive operations of computing the convex hull of polyhedra is replaced by polyhedral inclusion checks which can be performed efficiently by an application of Farkas's lemma. More precisely, we propose the following three contributions:

- A certified constraint based specification of a polyhedral analysis for bytecode programs.
- A notion of certificate for result checking of polyhedral analysers.
- A certificate result checker, obtained by Coq extraction, able to perform static array bound checking on resource constrained devices.

2 Polyhedral Analysis of Bytecode

We consider a cut-down language of Java bytecode which includes integers, dynamically created (unidimensional) arrays of integers, static methods (procedures) and static fields (global variables). The formal syntax and small-step operational semantics are rather straightforward and can be found in the companion report [4].

The analysis is inter-procedural, relational and parametrised with respect to a numeric abstract domain used to abstract the values of the local and global variables of the program. The analyser automatically infers an invariant for each control point in the program, a pre-condition that must hold at the point of calling a procedure and a post-condition that is guaranteed to hold when the procedure returns.

2.1 Motivating example

The Binary Search example (in source format here for readability considerations) given in Fig. 1 shows how our analysis will prove that the instruction that accesses the array `vec` with index `mid` will not index out of bounds. We have annotated the code of Binary Search with the invariants that have been inferred automatically. Invariants refer to values of local and global variables and can also refer to the length of an array. For example, the invariant (I_3) asserts among other properties that when entering the while loop, the relation $0 \leq \texttt{low} < \texttt{high} < |\texttt{vec}|$ is satisfied. Similarly, the post-condition ensures that the result is a valid index into the array being searched, or -1, indicating that the element was not found. In addition, the analysis introduces a 0-indexed variable (such as *e.g.* $\texttt{key}_0$ in the example) for each parameter in order to refer to its value when entering the procedure. As a result, the invariant on exit of the method defines a *summary relation* between its input and its output.

```
  //     PRE: 0 ≤ |vec_0|
static int bsearch(int key, int[] vec) {
  // (I_1) key_0 = key ∧ |vec_0| = |vec| ∧ 0 ≤ |vec_0|
   int low = 0, high = vec.length - 1;
  // (I_2) key_0 = key ∧ |vec_0| = |vec| ∧ 0 ≤ low ≤ high + 1 ≤ |vec_0|
   while (0 < high-low) {
  // (I_3) key_0 = key ∧ |vec_0| = |vec| ∧ 0 ≤ low < high < |vec_0|
       int mid = low + (high - low) / 2;
  // (I_4) key_0 = key ∧ |vec_0| = |vec|∧
  //       0 ≤ low < high < |vec_0| ∧  low + high − 1 ≤ 2 · mid ≤ low + high
       if (key == vec[mid]) return mid;
       else if (key < vec[mid]) high = mid - 1;
       else low = mid + 1;
  // (I_5) key_0 = key ∧ |vec_0| = |vec| ∧ −2 + 3 · low ≤ 2 · high + mid∧
  //        − 1 + 2 · low ≤ high + 2 · mid ∧ −1 + low ≤ mid ≤ 1 + high∧
  //       high ≤ low+mid∧1+high ≤ 2·low+mid∧1+low+mid ≤ |vec_0|+high∧
  //       2 ≤ |vec_0| ∧ 2 + high + mid ≤ |vec_0| + low
   }
  // (I_6) key_0 = key∧|vec_0| = |vec|∧low−1 ≤ high ≤ low∧0 ≤ low∧high < |vec_0|
   return -1;
} //    POST: −1 ≤ res < |vec_0|
```

Fig. 1. Binary search

2.2 Numeric relational domain specification

The bytecode analysis is specified with respect to an abstract numeric relational interface (defined below) that can be instantiated with standard relational abstract domains [10, 16, 17]. The numeric abstract domain $\mathbb{D}$ is a family of sets $\mathbb{D}_V$

indexed with a finite set V of variables. The abstract operators and associated properties listed below furnish the interface needed to specify and prove correct our generic numeric relational bytecode analysis.

To establish the connection between abstract elements and sets of numeric environments $\mathcal{P}(V \to \mathbb{Z})$, $\mathbb{D}$ is equipped with a concretisation function $\gamma : \mathbb{D}_V \to \mathcal{P}(V \to \mathbb{Z})$ compatible with a decidable partial order relation $\sqsubseteq$ *i.e.,* $d \sqsubseteq d' \Rightarrow \gamma(d) \subseteq \gamma(d')$. The domain $\mathbb{D}$ provides an upper-bound ($\sqcup$) and a lower bound ($\sqcap$) operators. To handle variable scopes, the domain is also equipped with a renaming and a projection operator. The renaming operator $[\cdot]_{W \to W'} : \mathbb{D}_{V+W} \to \mathbb{D}_{V+W'}$ is purely syntactic and maps a variable w_i in the ordered set W to the corresponding variable w'_i in W' (+ denotes disjoint union here). The projection operator $\exists_{V'} : \mathbb{D}_{V+V'} \to \mathbb{D}_V$ allows to project an abstract element onto a subset of the variables. For instance, $\exists_{\{y\}}.x \leq y \leq z$ would (by transitivity) compute $x \leq z$.

All the previous operators are language independent. The interface of the numeric domain with the programming language is made through expressions (*Expr*) and guards (*Guard*).

$$\begin{array}{rll} \mathit{Expr}_V \ni e ::= n \mid x \mid ? \mid e \diamond e & & x \in V,\ \diamond \in \{+, -, \times, /\} \\ \mathit{Guard}_V \ni t ::= e \bowtie e & & \bowtie \in \{=, \neq, <, \leq, >, \geq\} \end{array}$$

In the rest of the paper $\overline{\bowtie}$ will denotes the negation of a binary test $\bowtie$. Expressions denote sets of numerical values (due to the question mark symbol ? that is used to model an arbitrary value) while guards denote predicates on environments. The meaning $[\![\cdot]\!]_\rho$ of such expressions is defined relative to an environment $\rho \in V \to \mathbb{Z}$.

$$\begin{array}{rcl} [\![n]\!]_\rho = \{n\} & [\![x]\!]_\rho = \{\rho(x)\} & [\![?]\!]_\rho = \mathbb{Z} \\ [\![e_1 \diamond e_2]\!]_\rho & = & \{n_1 \diamond n_2 \mid n_1 \in [\![e_1]\!], n_2 \in [\![e_2]\!]\} \\ [\![e_1 \bowtie e_2]\!]_\rho & \iff & \exists\ n_1 \in [\![e_1]\!]_\rho, n_2 \in [\![e_2]\!]_\rho.\ n_1 \bowtie n_2 \end{array}$$

The abstract assignment of an expression $e \in \mathit{Expr}_V$ to a variable $x \in V$ is modelled by the operator $[\![x := e]\!]^\sharp : \mathbb{D}_V \to \mathbb{D}_V$.

$$\{\rho[x \mapsto v] \mid \rho \in \gamma(d) \wedge v \in [\![e]\!]_\rho\} \subseteq \gamma([\![x := e]\!]^\sharp(d))$$

The set of environments for which a guard $t \in \mathit{Guard}_V$ is true may be over-approximated by $\mathit{assume}^\sharp(t)$. Formally, the following holds:

$$\{\rho \mid [\![t]\!]_\rho\} \subseteq \gamma(\mathit{assume}^\sharp(t)).$$

The analyser used in the benchmarks is obtained by instantiating the operators described above with the domain of convex polyhedra [10]. In addition, the analyser uses a widening operator whose purpose is to ensure the termination of fixpoint iterations—this operator is therefore not needed at checking time.

2.3 Constraint-based specification

The bytecode analysis is defined by specifying for each bytecode an abstract transfer function which maps abstract states to abstract states (for non-jumping intraprocedural instructions at least). The abstract states are pairs of the form $(s^\sharp, l^\sharp)$ where $l^\sharp$ is a relation between local, global and auxiliary variables and $s^\sharp$ is an abstract stack whose elements are symbolic expressions built from these variables. More precisely, the analysis manipulates the following sets of variables:

- R: set of local variables $r_0, \ldots, r_{|R|-1}$ of methods,
- R_0: set of old local variables $r_0^{old}, \ldots, r_{|R|-1}^{old}$ of methods, representing their initial values at the beginning of method execution,
- S: set of static fields $f_0, \ldots, f_{|S|-1}$ of the program,
- S_0: set of old static fields $f_0^{old}, \ldots, f_{|S|-1}^{old}$ of the program used to model values of static fields at the beginning of method execution,
- A: set of auxiliary variable $aux_0, \ldots, aux_{|A|-1}$ used to keep track of results of methods in the symbolic operand stack.

Moreover, we use a "primed" version X' of the variable set X for renaming purposes. For each method the analysis computes a signature $Pre \rightarrow Post$ whose informal meaning is

> if the method is called with in a context where its arguments and the static fields satisfy the property *Pre* then if the method returns, then its result, its arguments, and the initial and final values of static fields satisfy the property *Post*.

Preconditions are chosen by over-approximating the context in which each method may actually be invoked. Additionally the analysis computes at each control point of each method a local invariant between the current (R) and initial (R_0) values of local variables, the current (S) and initial (S_0) values of static fields, and some auxiliary variables (A) which are used temporarily to remember results of method calls which are still on the stack

The stack of symbolic expressions is used to "decompile" the operations on the operand stack. For example, for the instruction *Load r* that fetches the value of local variable r, the analysis just pushes the symbolic expression r onto the abstract stack $s^\sharp$. More generally, the effect of most instructions can be represented symbolically and only the comparisons and assignment to variables require updating the relation $l^\sharp$ between variables. In a polyhedron-based analysis this kind of symbolic manipulation [24, 21] is a substantial saving.

Definition 1 (Abstract domain). *The abstract value for a program P is described by an element $(Pre, Post, Loc)$ of the lattice*

$$\begin{aligned} State^\sharp = (Meth \rightarrow \mathbb{D}_{R_0+S_0}) \times \big(Meth \rightarrow \mathbb{D}_{R_0+S_0+S+\{res\}}\big) \\ \times \big(Meth \times \mathbb{N} \rightarrow \big(Expr^\star_{R+S+A} \times \mathbb{D}_{R_0+S_0+R+S+A}\big) + \{\bot\}\big) \end{aligned}$$

instr	F_{instr}
Nop	$(s^\sharp, l^\sharp) \rightarrow (s^\sharp, l^\sharp)$
Ipush n	$(s^\sharp, l^\sharp) \rightarrow \big(n :: s^\sharp, l^\sharp\big)$
Pop	$(e :: s^\sharp, l^\sharp) \rightarrow \big(s^\sharp, l^\sharp\big)$
Dup	$(e :: s^\sharp, l^\sharp) \rightarrow \big(e :: e :: s^\sharp, l^\sharp\big)$
Iadd	$(e_2 :: e_1 :: s^\sharp, l^\sharp) \rightarrow \big(e_2 + e_1 :: s^\sharp, l^\sharp\big)$
Isub	$(e_2 :: e_1 :: s^\sharp, l^\sharp) \rightarrow \big(e_2 - e_1 :: s^\sharp, l^\sharp\big)$
Imult	$(e_2 :: e_1 :: s^\sharp, l^\sharp) \rightarrow \big(e_2 \times e_1 :: s^\sharp, l^\sharp\big)$
Idiv	$(e_2 :: e_1 :: s^\sharp, l^\sharp) \rightarrow \big(e_2/e_1 :: s^\sharp, l^\sharp\big)$
Ineg	$(e :: s^\sharp, l^\sharp) \rightarrow \big(0 - e :: s^\sharp, l^\sharp\big)$
Iinput	$(s^\sharp, l^\sharp) \rightarrow (? :: s^\sharp, l^\sharp)$
Load r	$(s^\sharp, l^\sharp) \rightarrow \big(r :: s^\sharp, l^\sharp\big)$
Store r	$(e :: s^\sharp, l^\sharp) \rightarrow \big(s^\sharp[?/r], [\![r := e]\!]^\sharp(l^\sharp)\big)$
Getstatic f	$(s^\sharp, l^\sharp) \rightarrow \big(f :: s^\sharp, l^\sharp\big)$
Putstatic f	$(e :: s^\sharp, l^\sharp) \rightarrow \big(s^\sharp[?/f], [\![f := e]\!]^\sharp(l^\sharp)\big)$
Iinc r n	$(s^\sharp, l^\sharp) \rightarrow \big(s^\sharp[r - n/r], [\![r := r + n]\!]^\sharp(l^\sharp)\big)$
Newarray	$(e :: s^\sharp, l^\sharp) \rightarrow \big(e :: s^\sharp, l^\sharp\big)$
Arraylength	$(e :: s^\sharp, l^\sharp) \rightarrow \big(e :: s^\sharp, l^\sharp\big)$
Iaload	$(e_2 :: e_1 :: s^\sharp, l^\sharp) \rightarrow \big(? :: s^\sharp, l^\sharp\big)$
Iastore	$(e_3 :: e_2 :: e_1 :: s^\sharp, l^\sharp) \rightarrow \big(s^\sharp, l^\sharp\big)$

$$\frac{m[p] = instr \notin \{Goto\ p', If_icmp\ cond\ p', Invoke\ m', Return\}}{F_{instr}(Loc(m,p)) \sqsubseteq Loc(m,p+1)}\,[\mathsf{Intra}]$$

$$\frac{m[p] = Goto\ p'}{Loc(m,p) \sqsubseteq Loc(m,p')}\,[\mathsf{Goto}]$$

$$\frac{m[p] = If_icmp \bowtie p' \quad Loc(m,p) = (e_2 :: e_1 :: s^\sharp, l^\sharp)}{(s^\sharp, assume^\sharp(e_1 \bowtie e_2) \sqcap^\sharp l^\sharp) \sqsubseteq Loc(m,p')}\,[\mathsf{If1}]$$

$$\frac{m[p] = If_icmp \bowtie p' \quad Loc(m,p) = (e_2 :: e_1 :: s^\sharp, l^\sharp)}{(s^\sharp, assume^\sharp(e_1 \overline{\bowtie} e_2) \sqcap^\sharp l^\sharp)) \sqsubseteq Loc(m,p+1)}\,[\mathsf{If2}]$$

$$\frac{m[p] = Invoke\ m' \quad n = nbArgs(m') \quad Loc(m,p) = (e_{n-1} :: \cdots :: e_0 :: s^\sharp, l^\sharp)}{\left(\exists_{R+S_0+A}\left(\prod_{i=0}^{n-1} assume^\sharp(e_i = r_i^{old}) \sqcap \exists_{R_0}(l^\sharp)\right)\right)_{S \rightarrow S_0} \sqsubseteq Pre(m')}\,[\mathsf{Call1}]$$

$$\frac{\begin{array}{c} m[p] = Invoke\ m' \quad Loc(m,p) = (e_{n-1} :: \cdots :: e_0 :: s^\sharp, l^\sharp) \\ l^\sharp_{m'} = \exists_{R_0}\left(\prod_{i=0}^{n-1} assume^\sharp(e_i = r_i^{old})_{S \rightarrow S'} \sqcap Post(m')_{S_0 \rightarrow S'}\right) \end{array}}{\left(aux_j :: s^\sharp[?/aux_j], \exists_{S'+\{res\}}[\![aux_j := res]\!]\left(l^\sharp_{S \rightarrow S'} \sqcap\ l^\sharp_{m'}\right)\right) \sqsubseteq Loc(m,p+1)}\,[\mathsf{Call2}]$$

where p is the index of the $j-$th *Invoke* in m

$$\frac{m[p] = Return \quad Loc(m,p) = (e :: s^\sharp, l^\sharp)}{\exists_{R+A}([\![res := e]\!]^\sharp(l^\sharp)) \sqsubseteq Post(m)}\,[\mathsf{Return}]$$

$$\frac{m \in P \quad n = nbArgs(m)}{\prod_{i=0}^{|S|-1} assume^\sharp(f_i = f_i^{old}) \prod_{i=0}^{n-1} assume^\sharp(r_i^{old} = r_i) \sqcap Pre(m) \sqsubseteq Loc(m,0)}\,[\mathsf{Init}]$$

$$\frac{}{\top \sqsubseteq Pre(\mathrm{main})}\,[\mathsf{PreMain}]$$

Fig. 2. Analysis specification

The analysis result is specified as a solution of a constraint (inequation) system associated to each program. The constraint system is given in Fig. 2. Array references are abstracted by the length of the array they point to. As a consequence, the instruction *Newarray* which takes an integer n on top of the stack and replaces it with a reference to a newly allocated array of length n, is simply abstracted by the identity function. The constraints [Call1] and [Call2] associated with a method call are the most complicated parts of the analysis. The complications partly arise because we have several kinds of variables (static fields, local and auxiliary variables) whose different scopes must be catered for. The analysis gives rise to two constraints: one that relates the state before the call to the pre-condition of the method ([Call1]) and one that registers the impact of the call on the state immediately following the call site ([Call2]).

When invoking a method m' from method m, we compute an abstract state that holds before starting executing m' and which constrains the $Pre(m')$ component of the abstract element describing P. This state registers that the n topmost expressions $e_1, \ldots, e_n$ on the abstract stack corresponds to the actual arguments that will be bound to the local variables of the callee m', by injecting the constraints $e_i = r_i^{old}$ into the relational domain and adding them to the current state as given by $l^\sharp$. Care must be exercised not to confound the parameters R_0 of the caller with the parameters of the callee, hence the projecting out of R_0 before joining the constraints. Furthermore, the local variables R, the initial values of static fields S_0 and the auxiliary variables A of method m have a different meaning in the context of method m' and are removed from the abstract state at the start of m' too. Finally, the current value of static fields S in m at the point of the method call becomes the initial value of the static fields when analysing m', hence the renaming of S into S_0.

The second rule [Call2] for *Invoke* describes the impact of the method call on its successor state. We use an auxiliary variable aux_j (chosen to be free in $s^\sharp$) to name the result of the method call which is pushed onto the stack. This variable is constrained to be equal to the variable *res* which receives the value returned by m'. The rest of the left-hand side expression of the constraint $l^\sharp_{S \to S'} \sqcap \exists_{R_0} (\ldots)$ serves to link the post-condition $Post(m')$ of the method with the state $l^\sharp$ of the call site. These are linked via the local variables r_i constrained to be equal to the argument expressions e_i and via the global static fields S. Again, some renaming and hiding of variables is required: *e.g.*, the initial values of the static fields in m', referred to by S_0, correspond to the values of the static fields before the call in the state $l^\sharp$ and in the expressions e_i, referred to by S. The renamings $S_0 \to S'$ and $S \to S'$, respectively, ensure that these values are identified.

The purpose of the invariants specified by the analysis is to enforce a suitable safety policy. In a context of array bound checking we must check that each array access is within the bounds of the array. As a consequence, for each occurence of an instruction *Iaload* or *Iastore* at a program point (m, pc), we test if the local invariant $Loc(m, pc)$ computed by the analysis ensures a safe array access. If these tests succeed we say that *Loc satisfies all safety checks*.

2.4 Inference

The constraint system presented in the previous section can be turned into a post-fixpoint problem by standard techniques. Consequently, the solutions of the system can be characterised as the set of post-fixpoints $\{x \mid F^\sharp(x) \sqsubseteq x\}$ of a suitable monotone function $F^\sharp$ operating on the global abstract domain $State^\sharp$ of the analysis. Computing such a post-fixpoint is then the role of chaotic iterations [8]. Iteration is sped up by using widening on well-chosen control points. Neither the iteration strategy nor the widening operators belong to the Trusted Computing Base (TCB) since the validity of the result can be checked with a post-fixpoint test.

2.5 Soundness of the analysis

To prove the soundness of the analysis we prove that for each method of the program, the signature $Pre \rightarrow Post$ and the local invariants in Loc that are specified by the constraint system, are correct with respect to the semantics of the execution of the method. The full proof has been machine checked in Coq (see [23]) in order to prove the soundness of the result checker. Details (see [4]) are omitted here for lack of space but we comment the main theorems now.

First we define the safety policy using semantic ingredients. A program is safe if all reachable states w.r.t. to the small-step semantics are distinct from the error state. The semantics enters the error state when an array is accessed via the instructions *Iaload* and *Iastore* with a value outside the array bounds.

```
Definition safe (p:program) : Prop :=
  ∀ st, reachable p st → st <> error.
```

The constraint based specification of Fig. 2 is turned into a suitable Coq predicate `AnalysisSolution` (including safety checks) and we prove that the existence of a suitable $(Pre, Post, Loc)$ solution implies the safety of the program.

```
Theorem sound_analysis : ∀ p loc pre post,
    AnalysisSolution p loc pre post → safe p.
```

The purpose of Section 3 is to define an executable checker able to check if a candidate $(Pre, Post, Loc)$ is a solution to the constraint based specification. The candidate is included in a certificate `cert` with extra information that we will describe in the next section.

```
Theorem bin_checker_correct_wrt_analysis_spec : ∀ p cert,
  checker p cert = true →
  ∃ loc, ∃ pre, ∃ post, AnalysisSolution p loc pre post.
```

Combined together these two theorems prove the semantic soundness of the executable checker that can be run in Coq or extracted into a Caml version.

```
Theorem bin_checker_correct_wrt_semantic :
  ∀ p cert, checker p cert = true → safe p.
```

3 Result Checking of Polyhedral Operations

In this section, we show how to efficiently implement convex polyhedra operators using a result checking approach.

3.1 The polyhedral domain revisited

Polyhedra can be represented as sets of linear constraints. For efficiency, it is desirable to keep these sets in normal form *i.e.,* without redundant constraints. For this purpose, polyhedra libraries maintain a dual description of polyhedra based on *generators* in which a convex polyhedron is the convex hull of a (finite) set of *vertices*, *rays* and *lines*. Vertices, rays and lines are respectively extremal points, infinite directions and bi-directional infinite directions of the polyhedron.

At the origin of the efficiency (and complexity) of convex polyhedra algorithms is Chernikova's algorithm which is used to maintain the coherence of the double description of polyhedra [7]. The main insight of our approach is that we develop a checker which only uses the constraint description of polyhedra and which never needs to detect redundant constraints. Moreover, projections are not computed but delayed using a set of extra *existential* variables. More precisely, our polyhedra are represented by a list of linear expression over two disjoint sets of variables V and E. Variables in $v \in V$ are genuine variables. The set E is fixed. Variables $e \in E$ are (existential) variables that represent dimensions which have been projected out.

Definition 2. *Let V and E be disjoint sets of variables.*

$$\mathbb{P}_V = Lin^{\star}_{V+E}$$

where $Lin_X = \{c_0 + c_1 \times x_1 + \cdots + c_n \times x_n \mid c_i \in \mathbb{Z}, x_i \in X\}$.

Given $es \in \mathbb{P}_V$, the concretisation function is defined by

$$\gamma(es) = \{\rho_{|V} \mid \rho \in (V + E) \to \mathbb{Z} \wedge \forall lc \in es, [\![lc \geq 0]\!]_\rho\}$$

Efficient Coq implementation of $\mathbb{P}_V$ We have implemented (and proved correct) a result checker for convex polyhedra based on an efficient implementation of $\mathbb{P}_V$. To ensure the efficiency of the checker, we have carefully fine-tuned algorithms and data-structures. Variables are coded by binary integers *i.e.,* the Coq `positive` type.

```
Inductive positive : Set
 := xH | x0 (p:positive) | xI (p:positive).
```

Variables in $v \in V$ start with a `x0` constructor while existential variables $v \in E$ start with a `xI` constructor. A linear expression $e \in Lin$ is coded by a radix tree whose node labels record integer coefficients of the linear expression.

```
Inductive tree : Set :=
 | Leaf
 | Node (left:tree) (label:Z) (right:tree).
```

Therefore, looking-up a variable coefficient can be done by following a path in the tree. This operation executes in time linear in the length of the variable *i.e.,* logarithmic in the number of variables. For efficiency again, Coq polyhedra $p \in \mathbb{P}_V$ are not simply lists of linear expressions but are dependent records which store: i) a list `lin_cstr` of linear constraints coded as trees, ii) a variable `fresh_v` $\in V$ for which all successors are fresh, iii) a variable `fresh_e` $\in E$ for which all successors are fresh, iv) a set `used_v` that stores the variables $v \in V$ that are used in `lin_cstr`, v) and all the proofs *i.e.,* the data-structure invariants, that ensure that `fresh_v` and `fresh_e` are really fresh and that the set `used_v` indeed over-approximates the variables used in `lin_cstr`.

Checking convex polyhedra operations In the following, we show how to implement the polyhedral operations using (only) polyhedra in constraint form. **Renaming** simply consists in applying the renaming to the expressions within the polyhedron. Because the existential variables belong to a disjoint set, no capture can occur. Using Fourier-Motzkin elimination (see *e.g.,* [19]), **projections** can be computed directly over the constraint representation of polyhedra. However, in the worst case, the number of constraints grows exponentially in the number of variables to project. To solve this problem, we delay the projection and simply register them as existentially quantified. This is done by renaming these variables to fresh existential variables.
To compute **intersections**, care must be taken not to mix up the existential variables. To avoid captures, existentially variables are renamed to variables that are fresh for both polyhedra. Interestingly, with our tree encoding, renaming all the existential variables is a constant time operation. Thereafter, the intersection is obtained by concatenating the lists of linear expressions.
To implement the **assume** operator, the involved expressions are first linearised and the obtained linear inequalities are put into the form $e \geq 0$ where e now belongs to the set Lin defined above. A special care is taken to precisely handle euclidean division (which is the semantics we give to the division operator in this work). For instance, the expression $x = y/c$ where c is a strictly positive constant, gives rise to a polyhedron made of the linear constraints $c \cdot x \leq y$ and $y \leq c \cdot x + c - 1$. Dealing with the round-to-zero integer division can be done via a program transformation that does case analysis on the signs of the arguments. We do not detail this here.
Assignment can be expressed in terms of the previous operators. Given x' a fresh existential variable, we have:

$$[\![x := e]\!]^\sharp(P) = \left(\exists_{\{x\}} \left(P \sqcap \mathit{assume}^\sharp(x' = e)\right)\right)_{\{x'\} \to \{x\}}$$

The **least upper bound** operator *i.e.,* convex hull is the typical operation that is straightforward to implement using the generator representation of polyhedra.

Instead of computing a convex hull, we follow the result certification methodology and provide a certificate polyhedron that is the result of the convex hull computation. Furthermore, our result checker need not check that the result is exactly the convex hull but only that it is an upper bound by doing a two inclusion tests.
To implement **inclusion tests**, we push the methodology further and use inclusion certificates. The form of certificates and their generation are described below.

3.2 Result certification for polyhedral inclusion

Our inclusion checker $\sqsubseteq_{check}$ takes as input a pair of polyhedra (P, Q) and an inclusion certificate. It will only return true if the certificate contains enough information to conclude that P is indeed included in Q ($P \sqsubseteq Q$).

In practice, we only use our checker where Q does not contain existential variables (because Q is computed by the untrusted analyser). This allows us to reduce the problem of inclusion into n problems of polyhedron emptiness where n is the number of constraints in Q. Such a problem admits a nice result certification technique thanks to Farkas's lemma (see for instance [19]) that gives a notion of *emptiness* certificate for polyhedra.

Lemma 1 (Farkas Lemma). *Let $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^m$. The following statements are equivalent:*

- *For all $x \in \mathbb{Q}^n$, $\neg(A \cdot x \geq b)$*
- *There exists $ic \in \mathbb{Q}^{+m}$ satisfying $A^t \cdot ic = \bar{0}$ and $b^t \cdot ic > 0$.*

The soundness ($\Leftarrow$) proof is the easy part and is all that is needed in the machine-checked proof. The existence of a certificate ensures the infeasibility of the linear constraints and therefore that the corresponding polyhedron is empty.

Thus, an *inclusion certificate* $ic_1 :: \cdots :: ic_n$ for an entry (P, Q) is a collection of n vectors of $\mathbb{Q}^m$ (with $n = |Q|$) and checking each emptiness certificate ic_k consists of 1) computing a matrix-vector product ($A^t \cdot ic$); 2) verifying that the result is a null vector; 3) computing a scalar product ($b^t \cdot ic_k$); and 4) verifying that the result is strictly positive. All in all, the certificate checker runs in quadratic-time in terms of arithmetic operations for each emptiness certificate.

Moreover, certificate generation can be recast as a linear programming problem that can be efficiently solved by either the Simplex or interior point methods.

4 Implementation and Experiments

The relational bytecode analysis has been implemented in Caml and instantiated with the efficient NewPolka polyhedral library [12] as its relational abstract domain. The programs we analyse are genuine Java programs where unsupported instructions have been automatically replaced by conservative numerical instructions (*e.g.*, a *Getfield* is replaced by a sequence *Pop*; *Iinput*). *Iinput* is a

dummy instruction placing an arbitrary value on top of the operand stack. The analyser then computes a solution to the constraint system generated from a program. From these invariants, loop headers and join points are extracted and the inclusion certificates required by the checker are produced using the Simplex algorithm. A binary form of loop headers, join point invariants and their inclusion certificates constitute the final program certificate.

As invariants computed by static analysers often contain more information than necessary for proving a particular safety policy *i.e.,* the absence of array out-of-bounds accesses, it is interesting to *prune* the analysis result and eliminate invariants that are useless for proving a given safety property. The advantages are twofold: invariants to check are smaller and their verification cheaper. We have applied the technique described in [5] for pruning constraint-based invariants, with some adaptations to deal with the interprocedural aspects of our polyhedral analysis. The algorithm is not described here for space reasons but can be found in the companion report [4].

The result checker for polyhedral analysis described in Section 2 and Section 3 has been implemented in Coq. For our benchmarks we consider a refined version of the safety property where all but a designated subset of array accesses are required to be correct.

For each program we compare the checking time with (before) and without (after) fixpoint pruning, using either an extracted checker (Caml) or the checker running in Coq. In the first approach the Coq result checker is automatically transformed into a Caml program by the Coq extraction mechanism. In the second approach, the result checker is directly run inside the reduction engine of Coq to compute a foundational proof of safety of the program (using the technique of proof by *reflection* [1]). Fig. 3 presents our experimental results. The benchmarks are relatively modest in size and it is well known that full-blown polyhedral analyses have scalability problems. Our analyser will not avoid this but can be instantiated with simpler relational domains such as *e.g.*, octagons, without having to change the checker. The programs and the analysis results can be found online [23] and replayed in Coq or with an extracted Caml checker. We consider two families of programs. The first one consists of benchmarks used by Xi to demonstrate the dependent type system for Xanadu [24]. For this family we automatically prove the absence of out-of-bound accesses. The second is taken from the Java benchmark suite SciMark for scientific and numerical computing where our polyhedral analysis prove safety for array accesses except for the more intricate multi-dimensional arrays representing matrices.

Two things are worth noticing. First, the checking time is very small (less than one second), which is especially noteworthy given that the checker is run in Coq. We clearly benefit here from our efficient implementation and the optimised reduction engine of Coq [11]. Compared to the extracted version, the Coq verifier has at most a factor 10 of efficiency penalty. Second, pruning can halve the number of constraints to verify. This reduction can sometimes but not always produce a similar reduction in checking time. The reduction is especially visible when the analyser tends to generate huge invariants which cannot be exploited.

	size	score	certificate size		checking time (Caml)		checking time (Coq)	
Program			before	after	before	after	before	after
BSearch	80	100%	20	11	2.0	1.4	14.1	11.6
HeapSort	143	100%	65	25	6.1	3.7	45.0	35.5
QuickSort	276	100%	90	42	144.5	128.7	1036.7	974.0
Random	883	83%	50	31	7.3	8.0	46.9	44.3
Jacobi	135	50%	31	10	1.6	1.7	12.8	9.2
LU	559	45%	206	96	20.1	17.4	100.5	91.5
SparseCompRow	90	33%	34	6	1.5	1.1	10.3	6.1
FFT	591	78%	194	50	38.8	22.7	263.2	193.8

Fig. 3. Size in number of instructions, score in ratio succeeded checks / total checks, certificates in number of constraints, time in milliseconds

This is *e.g.*, the case for FFT where the analyser approximates an exponential with a complex polyhedron.

As part of the Mobius project and collaboration with Pierre Crégut from France Télécom, we have experimented with using the polyhedral result checker to check array bounds on a mobile phone. This is part of the Mobius demo that is available online[1]. The experiment shows that it is feasible to perform extended bytecode verification with the polyhedral certificates that we have developed.

5 Related Work

A number of relational abstract domains (octagons [16], convex polyhedra [10], polynomial equalities [17]) have been proposed with various trade-offs between precision and efficiency, and intra-procedural relational abstract interpretation for high-level imperative languages is by now a mature analysis technique. However, to the best of our knowledge the present work is the first extension of this to an inter-procedural analysis for bytecode. Dependent type systems for Java-style bytecode for removing array bounds checks have been proposed by Xi and Xia [25]. The analysis of the stack uses singleton types to track the values of stack elements, in the same spirit as our symbolic stack expressions. The analysis is intra-procedural and does not consider methods (they are added in a later work [24] which also adds a richer set of types). The type checking relies on loop invariants. We have run our analysis on the example Xanadu programs given by Xi and have been able to infer the invariants necessary for verifying safe array access automatically.

The area of certified program verifiers is an active field. Wildmoser, Nipkow *et al.* [22] were the first to develop a fully certified VCGen within Isabelle/HOL for verifying arithmetic overflow in Java bytecode. The certification of abstract interpreters has been developed by Pichardie *et al.* [18, 6]. Lee *et al.* [14] have certified the type analysis of a language close to Standard ML in LF and Leroy [15]

[1] `http://mobius.inria.fr/`

has certified some of the data flow analyses of a compiler back-end. Wildmoser *et al.* [21] certify a VCGen that uses untrusted interval analysis for producing invariants and that relies on Isabelle/HOL decision procedures to check the verification conditions generated with the help of these invariants. Their technique for analysing bytecode is close to ours in that they also use symbolic expressions to analyse the operand stack and the main contribution of the work reported here with respect to theirs is to develop this result checking approach for a fully relational analysis.

6 Conclusions and Future Work

This paper demonstrates the feasibility of an interprocedural relational analysis which automatically infers polyhedral loop invariants and pre-/post-condition for programs in an imperative bytecode language. To simplify the checking of these invariants, we have devised a result checker for polyhedra which uses inclusion certificates (issued from a result due to Farkas) instead of computing convex hulls of polyhedra at join points. This checker is much simpler to prove correct mechanically than the polyhedral analyser and provides a means of building a foundational proof carrying code that can make use of industrial strength relational program analysis.

Future work concerns extensions to incorporate richer domains of properties such as disjunctive completions of linear domains or non-linear (polynomial) invariants. Using propositional reasoning, checking disjunctive invariants can be reduced to emptiness tests. As a result, parts of the polyhedral checker could be reused. Emptiness certificates from Section 3.2 can be generalised to deal with non-linear inequalities [3]. However, the analyses for *inferring* such properties are in their infancy. On a language level, the challenge is to extend the analysis to cover the object oriented aspects of Java bytecode. The inclusion of static fields and arrays in our framework provides a first step in that direction but a full extension would notably require an additional alias analysis.

References

1. Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William E. Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 95–105. IEEE Computer Society, 1990.
2. G. Barthe and G. Dufay. A tool-assisted framework for certified bytecode verification. In *FASE*, volume 2984 of *LNCS*, pages 99–113. Springer, 2004.
3. F. Besson. Fast reflexive arithmetic tactics: the linear case and beyond. In *Types for Proofs and Programs*, pages 48–62. Springer LNCS vol. 4502, 2006.
4. F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Result certification for relational program analysis. Research Report 6333, Inria, 2007. `http://hal.inria.fr/inria-00166930/`.
5. F. Besson, T. Jensen, and T. Turpin. Small witnesses for abstract interpretation based proofs. In *Proc. of 16th Europ. Symp. on Programming (ESOP 2007)*, pages 268–283. Springer LNCS vol. 4421, 2007.

6. D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a Data Flow Analyser in Constructive Logic. *Theoretical Computer Science*, 342(1):56–78, September 2005.
7. N.V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities. *U.S.S.R Comp. Mathematics and Mathematical Physics*, 5(2):228–233, 1965.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proc. of 4th ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
9. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The Astrée analyser. In *Proc. of 14th European Symp. on Programming (ESOP'05)*, pages 21–30. Springer LNCS vol. 3444, 2005.
10. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of 5th ACM Symp. on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM Press, 1978.
11. B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proc. of the 7th ACM international conference on Functional programming (ICFP'02)*, pages 235–246. ACM Press, 2002.
12. B. Jeannet and the Apron team. The Apron library, 2007.
13. Gerwin Klein and Tobias Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298(3):583–626, 2002.
14. D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of Standard ML. In *Proc. of 34th ACM Symp. on Principles of Programming Languages (POPL'07)*, pages 173–184. ACM Press, 2007.
15. X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. of the 33rd ACM Symp. on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.
16. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.
17. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Proc. of 31st ACM Symp. on Principles of Programming Languages (POPL'04)*, pages 330–341. ACM Press, 2004.
18. D. Pichardie. *Interprétation abstraite en logique intuitioniste: extraction d'analyseurs Java certifiés*. PhD thesis, Université de Rennes 1, 2005.
19. A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1998.
20. H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 1997.
21. M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In *Proc. of 1st Workshop on Bytecode Semantics, Verification and Transformation, ENTCS*, 2005.
22. M. Wildmoser and T. Nipkow. Asserting bytecode safety. In *Proc. of the 15th European Symp. on Programming (ESOP'05)*, 2005.
23. The Coq development of the work. `http://www.irisa.fr/celtique/ext/polycert/`.
24. H. Xi. Imperative Programming with Dependent Types. In *Proc. of 15th IEEE Symposium on Logic in Computer Science (LICS'00)*, pages 375–387. IEEE, 2000.
25. Hongwei Xi and Songtao Xia. Towards Array Bound Check Elimination in Java Virtual Machine Language. In *Proc. of CASCOON '99*, pages 110–125, 1999.