INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Result certification for relational program analysis

Frédéric Besson — Thomas Jensen — David Pichardie — Tiphaine Turpin

N° 6333

October 2007

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--6333--FR+ENG

Result certification for relational program analysis

Frédéric Besson*, Thomas Jensen† , David Pichardie*, Tiphaine Turpin‡

Thème SYM — Systèmes symboliques
Projet Lande

Rapport de recherche n° 6333 — October 2007 — 29 pages

Abstract: We define a generic relational program analysis for an imperative, stack-oriented byte code language with procedures, arrays and global variables and instantiate it with an abstract domain of polyhedra. The analysis has automatic inference of loop invariants and method pre-/post-conditions, and efficient checking of analysis results by a simple checker. Invariants, which can be large, can be specialized for proving a safety policy using an automatic pruning technique which reduces their size. The result of the analysis can be checked efficiently by annotating the program with parts of the invariant together with certificates of polyhedral inclusions, which allow to avoid certain complex polyhedral computation such as the convex hull of two polyhedra. Small, easily checkable inclusion certificates are obtained using Farkas lemma for proving the absence of solutions to systems of linear inequalities. The resulting checker is sufficiently simple to be entirely certified within the Coq proof assistant.

Key-words: Static analysis, abstract interpretation, bytecode Java, Coq

* INRIA Rennes - Bretagne Atlantique/IRISA
† CNRS/IRISA
‡ Université Rennes I/IRISA

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 —Télécopie : +33 2 99 84 71 71

Certification de résultat pour l'analyse de programme relationnelle

Résumé : Nous proposons une analyse générique de programme relationnelle pour un langage de bytecode impératif avec pile d'opérande, procédures, tableaux et variables globales. Cette analyse est instanciée avec un domaine abstrait de polyèdres. Elle propose une inférence automatique d'invariants de boucle et de préconditions/postconditions de procédures, ainsi qu'une vérification efficace du résultat de l'analyse par un vérificateur simple. Les invariants, qui peuvent être grands, peuvent être spécialisés pour prouver une propriété de sûreté en utilisant une technique automatique de compression de taille de certificat. Le résultat de l'analyse peut être vérifié efficacement en annotant le programme avec une partie des invariants et quelques certificats d'inclusion de polyèdre, qui permettent d'éviter certaines calculs polyédriques complexes comme le calcul de l'enveloppe convexe de deux polyèdres. Nous obtenons des certificats d'inclusion petits et facilement vérifiables grâce au lemme de Farkas pour prouver l'absence de solution dans un système d'inégalités linéaires. Le vérificateur ainsi obtenu est suffisamment simple pour être entièrement certifié avec l'assistant à la preuve Coq.

Mots-clés : Analyse statique, interprétation abstraite, bytecode Java, Coq

1 Introduction

Logic-based, static program verification, be it in form of abstract interpretation, symbolic model checking or interactive proving of programs, is used in a number of ways to improve the confidence in safety-critical systems and for protecting host machines from malicious code, as *e.g.*, done by the Java byte code verifier. As applications and the program logics grow in complexity, an automated technique for verifying program invariants based on a program logics should ideally meet all of the following three requirements:

- *Automatic Inference*: the complexity of both programs and the underlying logic can quickly make it burdensome to conduct program proofs manually. Automatic inference of program properties is necessary to obtain a technique that scales.
- *Result certification*: when inference is available, it often relies on advanced deductive methods for inferring an invariant whose size and complexity make it difficult to ascertain its validity manually. Efficient checking of the result of the inference or of any proposed invariant in general becomes important.
- *Small Trusted Computing Base (TCB)*: the result checker becomes the cornerstone of the reliability of the verification framework. In order to reduce the part of the code base that needs to be trusted without proof, the checker should be kept sufficiently simple and small in order to be able to verify the checking algorithmics mechanically.

Program verification based on general Hoare-style program logics may follow the Verification Condition Generator (VCGen) approach of *e.g.*, Extended Static Checking by Flanagan, Leino *et al.* [14] or use expressive type systems such as the dependent type systems of Xi and Pfenning [27] for proving properties of programs. The approaches based on VCGens are generally complete for partial correctness and will produce a set of verification conditions which, when satisfied, will allow to conclude that a given program property holds in the logic. Verification conditions often fall into fragments of logic that require them to be proved by dedicated decision procedures or theorem provers. VCGens and the type-based approaches are primarily concerned with invariant checking and discard part of the inference problem by relying on loop invariants and pre-post-condition of methods to be provided by the programmer. In terms of small TCB, the VCGens remain complex software which are hard to prove correct *in extenso*. The machine-checked formalizations e.g., by Nipkow, Wildmoser *et al.* [25, 26] show that this is indeed possible to certify an entire VCGen inside a proof assistant but also that this remains a major software certification challenge.

Another strand of program verification is based on abstract interpretation. Abstract interpretation is an automatic technique for inferring program properties in the form of fixpoints of monotone data flow functions. As a theory of proving programs it has strong semantic foundations. At the same time it should be noted that the algorithmics of the domains underlying the more advanced analyses such as polyhedral analysis (initially described by Cousot and Halbwachs [13]) is highly non-trivial. Checking an invariant is in theory simple as it only requires one more iteration to check that a property is indeed a

fixpoint but, as said, this computation does in certain cases rely on non-trivial algorithmics that forms part of what must be trusted. In previous work [9, 21], some of the authors formalised the theory of abstract interpretation inside the proof assistant Coq and extracted Caml implementations of a variety of program analyses. This Certified Abstract Interpretation approach represents a systematic way of reducing the TCB of static analyzers and fulfills the three requirements listed above. However, a fully mechanised correctness proofs of more advanced program analysers such as an optimised, polyhedral-based analysis would require an enormous effort in terms of program certification.

The purpose of this paper is to demonstrate that by focusing on certifying the *result* of the analysis rather than the analysis itself, it is possible to develop a verification framework for advanced program properties that satisfies all of the three desired properties and, at the same time, requires a significantly smaller effort in order to be proved correct. This idea was previously used by Wildmoser *et al* [24] who use the result of an untrusted interval analysis in a VCGen for byte code and by Leroy [17] in his certification of a compiler back-end where he, rather than certifying the complex graph-coloring algorithms for register allocation, proves the correctness of a checker that verifies a given coloring returned by an untrusted graph-coloring algorithms. Here, we generalise this idea by developing a relational analysis framework together with a certified checker. The basic observation is that an abstract interpretation can be decomposed into an abstract domain of properties, a generic program logic for reasoning about these properties and a fixpoint engine for solving recursive equations over the abstract domains. The inference does not need to use certified abstract domain operations and fixpoint engines, and the checking of invariants does not need to use a fixpoint engine at all. We take advantage of this to design a checker that re-uses the program logic but replaces the more complex domain operations with simpler ones, at the expense of providing some extra information in the certificate accompanying a program.

2 Overview

In the first part of this paper, we will develop a fully relational, interprocedural analyser which 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. Relational analyses are useful for finding loop invariants needed for proving program safety, e.g. when verifying the resource usage of programs or verifying safety properties related to safe memory access such as checking that all array accesses are within bounds. We will take Safe Array Access as an example safety policy and illustrate our approach with the Binary Search example given in Fig. 1, showing how the 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

```
  //     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) / 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|
```

Figure 1: Binary search

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 (and also for the global variables, of which there are none in the example) in order to refer to its value when entering the procedure. The effect of this is that the invariant on exit of the program defines a relation between the input and the output of the procedure, thus yielding a *summary relation* for the procedure.

2.1 Compressing invariants

Abstract interpretations may give you more information than you need for proving a particular property. In the case of the Binary Search example, if we are only interested in proving the validity of array accesses, there are a number of relations between variables in the invariants that can be forgotten. Reducing the number of constraints and the number of variables under consideration can lead to a significant gain in execution time when it comes to checking a proposed invariant. For example, pruning the invariants in Fig. 1 with respect to this property yields the simpler invariant shown in Fig. 2:

Notice that the inferred loop invariant I_3' is close to what a specifying programmer of Binary Search might have come up with, but here produced automatically. We explain pruning of procedures in Section 7.

2.2 Analysing a stack-based language

Polyhedral analysis of While languages is well understood but we want our framework to be able to analyse byte code programs and not only source code. We could in theory avoid the problem by transforming the program into three-address code and treat each stack location as a local variable but this transformation is expensive from an algorithmic point of view, as it increases the number of times that the relation has to be updated. Instead, we achieve the effect of this transformation by defining an analysis for stack-oriented byte code

```
  //     PRE: True
static int bsearch(int key, int[] vec) {
  // (I'1) |vec0| = |vec| ∧ 0 ≤ |vec0|
   int low = 0, high = vec.length - 1;
  // (I'2) |vec0| = |vec| ∧ 0 ≤ low ≤ high + 1 ≤ |vec0|
   while (0 < high-low) {
  // (I'3) |vec0| = |vec| ∧ 0 ≤ low < high < |vec0|
      int mid = (low + high) / 2;
  // (I'4) |vec| − |vec0| = 0 ∧ low ≥ 0 ∧ mid − low ≥ 0∧
  //       2 · high − 2 · mid − 1 ≥ 0 ∧ |vec0| − high − 1 ≥ 0
      if (key == vec[mid]) return mid;
      else if (key < vec[mid]) high = mid - 1;
      else low = mid + 1;
  // (I'5) |vec0| = |vec| ∧ −1 + low ≤ high ∧ 0 ≤ low ∧ 5 + 2 · high ≤ 2 · |vec|
   }
  // (I'6) 0 ≤ |vec0|
   return -1;
} //    POST: −1 ≤ res < |vec0|
```

Figure 2: Binary search after invariant pruning

that combines relational abstract interpretation with symbolic execution, following an idea previously used for analysing Java byte code by Xi and Xia [28] and Wildmoser *et al* [24]. This technique abstracts the environment of local variables by a relation (e.g., a polyhedron) and replace the operand stack with a stack of symbolic expressions used to "decompile" the operations on the operand stack. For example, the comparison of variables `low` and `high` will be compiled to the byte codes below, which are analysed in a state consisting of the relation I_2 as defined in Fig. 1 and an abstract stack that evolves as values are pushed onto the stack.

		$[]$	I_2
7 :	ipush 0	0	I_2
8 :	iload high	$\mathtt{high} :: 0$	I_2
9 :	iload low	$\mathtt{low} :: \mathtt{high} :: 0$	I_2
10 :	isub	$(\mathtt{high}-\mathtt{low}) :: 0$	I_2
11 :	if_icmpge 56	$[]$	I_3

Before the comparison in instruction 11, the stack top contains the expression $\mathtt{high}-\mathtt{low}$, reflecting that in the real execution the stack top at this point will contain the value of this expression. When we learn from the test that the expression $\mathtt{high}>\mathtt{low}$ evaluates to true in the state immediately following the comparison (and only then), we update the relation accordingly to obtain invariant I_3. Similarly, we have to update the relation when assigning a new value to a variable. For example, the instruction that assigns $(\mathtt{high}+\mathtt{low})/2$ to `mid` is compiled and analysed as shown below. Again, the relation I_3 is only updated when the assignment to `mid` is done, to yield relation I_4.

		$[]$	I_3
14 :	iload low	$\texttt{low}$	I_3
15 :	iload high	$\texttt{high} :: \texttt{low}$	I_3
16 :	iadd	$(\texttt{high}+\texttt{low})$	I_3
17 :	ipush 2	$2 :: (\texttt{high}+\texttt{low})$	I_3
18 :	idiv	$((\texttt{high}+\texttt{low})/2)$	I_3
19 :	istore mid	$[]$	I_4

More generally, with the abstract stack of expressions, only the comparisons and assignment to variables require updating the relation. In a polyhedron-based analysis this is a substantial saving.

2.3 Result checking with certificates

Checking an invariant obtained by computing a post-fixpoint of an abstract interpretation is in theory simple as it only requires one more iteration to check that it is indeed a post-fixpoint. In addition, only invariants at certain program points such as loop headers are required for re-building an entire invariant in one iteration. Lightweight Bytecode Verification by Rose [22] and the more general Abstraction-Carrying Code by Albert, Puebla and Hermenegildo [1] exploit this to construct efficient checkers for invariant-based program certificates. For the code in Fig. 2, only I'_2 is required.

The inference of invariants using our relational analysis uses an iterative fixpoint solver over an abstract domain of polyhedra and is in principle amenable to the same technique. However, despite efficient implementations of basic polyhedral operations, the algorithmic complexity of operations such a computing the least upper bound (*i.e.* the convex hull) of two polyhedra remains high, and certifying them in a proof assistant would be a major undertaking.

Instead, we propose an enriched certificate format which has the virtue of being simpler to check, at the cost of sending more information than in basic fixpoint reconstruction. We exploit that, for the checker, the only important property of the convex hull operators is that it produces an upper bound of two polyhedra and therefore can be replaced by inclusion checks with respect to an upper bound that is proposed by the certificates. Upper bounds are computed at join points so in Fig. 2 we would also supply I'_5.

Safety checks also reduces to inclusions of polyhedra as verifying the array access $\texttt{vec}[\texttt{mid}]$ amounts to ensuring that I'_4 implies $0 \leq \texttt{mid} < |\texttt{vec}|$. By simple propositional reasoning, this reduces to proving that the linear systems of constraints $-\texttt{mid}-1 \geq 0 \wedge I'_4$ and $\texttt{mid}-|\texttt{vec}| \geq 0 \wedge I'_4$ have no solution. Due to a result by Farkas, such problems can be checked efficiently using certificates by a simple matrix computation. The key insight is that unsolvability follows from the existence of a *positive* combination of the constraints which yield a strict negative constant. This would lead to a contradiction because the sum and product of positive quantities cannot be strictly negative. The certificate is therefore a vector which records the coefficients of the positive combination. For example, the certificate $[\mathbf{2};\mathbf{2};\mathbf{0};\mathbf{0};\mathbf{1};\mathbf{2}]$ proves that the constraints $\texttt{mid}-|\texttt{vec}| \geq 0 \wedge I'_4$ are unsatisfiable, as the expression

$$\mathbf{2}\cdot(\texttt{mid}-|\texttt{vec}|)+\mathbf{2}\cdot(|\texttt{vec}|-|\texttt{vec}_0|)+\mathbf{0}\cdot\dots+\mathbf{0}\cdot\dots+$$
$$\mathbf{1}\cdot(2\cdot\texttt{high}-2\cdot\texttt{mid}-1)+\mathbf{2}\cdot(|\texttt{vec}_0|-\texttt{high}-1)$$

evaluates to -2. We explain these certificates in detail in Section 8.

2.4 Certified certificate checkers

The result checking technique explained above already drastically reduces the TCB of the analysis result which only rely on the result checker. To further reduce the TCB, we have machine-checked the result checker of our analysis in the Coq proof assistant. The main components of the formalisation are

1. a predicate `Safe:program→Prop` which models the safe programs with respects to the the semantics described in Section 4,
2. a function `checker:program→certificate→bool`, which checks the safety of a program using a certificate containing a (partial) result of an analysis and some inclusion certificates,
3. a machine checked proof establishing the correctness of the checker:

```
Theorem checker_correct :
  ∀ p cert, checker p cert = true → Safe p.
```

The Trusted Computed Base is hence reduced to the Coq type checker and the formal definition of program safety.

Once the certified result checker is verified (by the Coq type checker) and installed by the code consumer, two scenarios can be envisaged to verify the safety of programs sent by producers. In the first one, the consumer may use an efficient Ocaml version of the checker, extracted from the Coq version thanks to the Coq extraction mechanism. The other alternative is related to *proof by reflection*. For each program `p` and certificate `cert` the consumer may build a foundational Coq proof of `Safe p`. To do so he only has to check in Coq the term `checker_correct p cert refl_eqtrue` where `refl_eqtrue` denotes a proof of `true=true`. It is the role of the Coq reduction engine to verify during type checking if `true=true` is equivalent to `checker p cert = true` by running the checker inside Coq. In this way we combine two desirable features which are often difficult to reconcile in state-of-the art Proof Carrying Code: foundational proofs and small certificates.

3 Notations

Let A and B be sets. If A and B are disjoint then $A+B$ is the disjoint sum of A and B. We write $A_\perp$ the set $A+\{\perp\}$. For $f \in A \to B_\perp$, $dom(f) = \{a \in A \mid f(x) \neq \perp\}$. Let $f \in A \to B$, $f[x \mapsto v]$ is the function identical to f everywhere except for x for which it returns v. The notation $[x_1 \mapsto v_1; \ldots; x_n \to v_n]$ stands for a function f of domain $\{x_1, \ldots, x_n\}$ such that $f(x_i) = v_i$. A^* is the set of lists of elements of A. We write $[]$ for the empty list and $a_0 :: \ldots :: a_{n-1}$ is a list l of length n ($|l| = n$) whose head (resp. tail) is a_0 (resp. a_{n-1}). $l[i]$ is the i-th element of l. We write a^i the list that is the repetition of a, i times. Let V, W be totally ordered sets. For $x \in V$, $\iota_V(x)$ is the index of x in set V and ι_V^{-1} is the inverse function. We abuse notations and identify $A^{|V|}$ with $V \to A$ *i.e.,* given a finite ordered set, $V = \{x_1, \ldots, x_n\}$ such that $x_1 < \ldots < x_n$, we identify the finite mapping $[x_1 \mapsto v_1, \ldots x_n \mapsto v_n]$ with the n-tuple $(v_1, \ldots, v_n)$.

We will write A^V to denote both $A^{|V|}$ and $V \to A$. Let $\rho \in A^V$ and $V' \subseteq V$, $\rho_{|V'} \in A^{V'}$ is the restriction of e over the variables of V' such that for all $x \in V'$, $\rho_{|V'}(x) = e(x)$. Given V and W disjoint set of variables, $\rho_1 \in A^V$ and $e_2 \in A^W$, we write $\rho_1 \oplus \rho_2 \in A^{V+W}$ for the finite mapping such that $(\rho_1 \oplus \rho_2)_{|V} = \rho_1$ and $(\rho_1 \oplus \rho_2)_{|W} = \rho_2$. Let W and W' ordered sets of same cardinality. If $\rho \in A^{V+W}$, then $\rho_{W \to W'} \in A^{V+W'}$ is obtained by renaming the variables of W to the variables in W'. Formally, we have $\rho_{W \to W'}(x) = \rho(x)$ if $x \in V$ and $\rho_{W \to W'}(x) = \rho(\iota_W^{-1}(\iota_{W'}(x)))$ if $x \in W'$. To make the distinction clear between syntactic expressions and values, syntactic expressions are bracketed ($\llcorner \cdot \lrcorner$). For example, we write $\llcorner 1 + e \lrcorner$ a syntactic expression built by applying the + operator to the constant 1 and the syntactic expression e.

4 A byte code language and its semantics

We use a simple stack-based byte code language to illustrate our ideas. Features include integers, dynamically created (unidimensional) array of integers, static methods (procedures) and static fields (global variables).

Programs are lists of methods and a method consists of a name, a number of arguments and a list of instructions. In the following, f ranges over the set S of static field names, r ranges over the set $R = \{r_0, \ldots, r_{|R|}\}$ of local variables and *id* ranges over the set *MethId* of method names. Moreover, i and n range over $\mathbb{N}$ or $\mathbb{Z}$ depending on the context and p is used for control points.

$$
\begin{array}{rcll}
P & \in & \mathit{Prog} & = \mathit{Meth}^* \\
m & \in & \mathit{Meth} & = \mathit{Sig} \times \mathit{Code} \\
 & & \mathit{Sig} & = \mathit{MethId} \times \mathbb{N} \\
c & \in & \mathit{Code} & = \mathit{Instr}^* \\
\mathit{instr} & \in & \mathit{Instr} & \\
\mathit{instr} & ::= & \multicolumn{2}{l}{\mathit{Nop} \mid \mathit{Ipush}\ n \mid \mathit{Iinc}\ r\ n \quad \text{where } n \in \mathbb{Z}} \\
 & & \multicolumn{2}{l}{\mathit{Pop} \mid \mathit{Dup} \mid \mathit{Ineg} \mid \mathit{Iadd} \mid \mathit{Isub} \mid \mathit{Imult} \mid \mathit{Idiv}} \\
 & & \multicolumn{2}{l}{\mathit{Load}\ r \mid \mathit{Store}\ r} \\
 & & \multicolumn{2}{l}{\mathit{Getstatic}\ f \mid \mathit{Putstatic}\ f} \\
 & & \multicolumn{2}{l}{\mathit{Newarray} \mid \mathit{Arraylength} \mid \mathit{Iaload} \mid \mathit{Iastore}} \\
 & & \multicolumn{2}{l}{\mathit{Goto}\ p \mid \mathit{If_icmp}\ \mathit{cond}\ p} \\
 & & \multicolumn{2}{l}{\quad \text{where } \mathit{cond} \in \{=, \neq, <, \leq\}} \\
 & & \multicolumn{2}{l}{\mathit{Invoke}\ \mathit{sig} \quad \text{where } \mathit{sig} \in \mathit{Sig}} \\
 & & \multicolumn{2}{l}{\mathit{Iinput} \mid \mathit{Return}}
\end{array}
$$

The instruction set has operators for integer arithmetic and for manipulating local variable, static fields and an operand stack. Instructions on arrays permit to create, obtain the size of, access and update arrays. The flow of control can be modified unconditionally (with *Goto*), and conditionally with the family of conditional instructions *If_icmp cond* which compare the top elements of the run-time stack and branch according to the outcome. Input of data is modelled with the instruction *Iinput*. The inter-procedural layer of the language contains an instruction *Invoke* for invoking a method and an instruction *Return* which transfers control to the calling method, and, at the same time returns the top of the operand stack as result by pushing it onto the operand stack of the caller (see the operational semantics below).

A program state is composed of a frame stack, the value of static fields and a heap of arrays and has the form $<(m,p,s,l)^*, g, h>$. Each frame is a triple composed of a method m, a control point p to be executed next, an operand stack s local to a frame and l a partial mapping from local variables to values. The global heap h is used for storing allocated arrays and is modelled as a partial function from memory locations to arrays. A special error state *Error* models the run-time error arising from indexing an array outside its bounds.

$$
\begin{array}{rcrcl}
ref & \in & Location & & \\
v & \in & Val & = & \mathbb{Z} + Location \\
s & \in & Stack & = & Val^* \\
l & \in & LocVar & = & R \to Val \\
a & \in & Array & = & \mathbb{Z}^* \\
h & \in & Heap & = & Location \to Array_\bot \\
g & \in & Static & = & S \to Val \\
 & & Frame & = & Meth \times \mathbb{N} \times Stack \times LocVar \\
 & & State & = & Frame^* \times Static \times Heap \\
 & & & & + \{Error\}
\end{array}
$$

The byte code language is given an operational semantics via a transition relation $\to$ between states. Some of the rules of the definition of $\to$ are shown in Fig. 3. In the semantics, for a method $m = ((id, n), c)$, we write $m[p]$ for $c[p]$. Note that the language is untyped: registers and fields may (and will) point successively to values of different types during execution. Instructions that require arguments with a certain type get stuck in case of error. Also, the number of registers $|R|$ is the same for all methods. Unused registers and uninitialised fields have the value 0. Finally, we only consider states $<st, g, h>$ such that every location appearing in st, g is in $dom(h)$, which is clearly preserved by the semantics in Fig. 3.

5 Relational analysis of byte code

In this section, we describe a generic, relational analysis for byte code, parameterised with respect to a numeric relational domain used to abstract the values of the local and global variables of the program.

5.1 Symbolic analysis of the stack

Rather than treating each stack location as a new local variable and include this variable in the numeric abstraction describing the state, we integrate a *symbolic de-compilation* into the analysis that abstracts a stack location by a symbolic expression describing how the value at that stack location is computed from the values of the local variables. The operand stack is hence abstracted by a stack of symbolic expressions which represents relation between operands, static fields and local variables.

The following definition of expressions and guards has two purposes: they form the basis of the abstract domain for stacks (*Expr* only), which is specific to stack-based byte code, and they serve as the interface with the numeric relational domain, which is parametric. Note that those two aspects of the analysis are

$$\frac{}{s,l,g,h \xrightarrow{Ipush\ n} n :: s,l,g,h} \qquad \frac{}{n_2 :: n_1 :: s,l,g,h \xrightarrow{Iadd} n_1+n_2 :: s,l,g,h}$$

$$\frac{l(r)=n}{s,l,g,h \xrightarrow{Iinc\ r\ i} s,l[r\mapsto n+i],g,h} \qquad \frac{}{s,l,g,h \xrightarrow{Load\ r} l(r) :: s,l,g,h}$$

$$\frac{}{v :: s,l,g,h \xrightarrow{Store\ r} s,l[r\mapsto v],g,h} \qquad \frac{}{s,l,g,h \xrightarrow{Getstatic\ f} g(f) :: s,l,g,h}$$

$$\frac{h(ref)=\bot \quad n\geq 0}{n :: s,l,g,h \xrightarrow{Newarray} ref :: s,l,g,h[ref\mapsto 0^n]}$$

$$\frac{h(ref)=a \quad 0\leq i<|a|}{i :: ref :: s,l,g,h \xrightarrow{Iaload} a[i] :: s,l,g,h} \qquad \frac{h(ref)=a \quad \neg\, 0\leq i<|a|}{i :: ref :: s,l,g,h \xrightarrow{Iaload} Error}$$

$$\frac{m[p]=instr \quad s,l,g,h \xrightarrow{instr} s',l',g',h'}{<(m,p,s,l) :: st,g,h> \rightarrow_P <(m,p+1,s',l') :: st,g',h'>}$$

$$\frac{m[p]=If_icmp\ \ cond\ p' \quad n_1\ cond\ n_2}{<(m,p,n_2 :: n_1 :: s,l) :: st,g,h> \rightarrow_P <(m,p',s,l) :: st,g,h>}$$

$$\frac{m[p]=If_icmp\ \ cond\ p' \quad \neg\, n_1\ cond\ n_2}{<(m,p,n_2 :: n_1 :: s,l) :: st,g,h> \rightarrow_P <(m,p+1,s,l) :: st,g,h>}$$

$$\frac{m[p]=Invoke\ (mn,n) \quad m'=((mn,n),c)\in P}{\begin{array}{l}<(m,p,(v_{n-1} :: \ldots :: v_0 :: s),l) :: st,g,h> \rightarrow_P \\ \quad <m',0,[],[r_0\mapsto v_0;\ldots;r_{n-1}\mapsto v_{n-1};r_n\mapsto 0;\ldots;r_{|R|}\mapsto 0] :: (m,p,s,l) :: st,g,h>\end{array}}$$

$$\frac{m[p]=Return}{<(m,p,v :: s,l) :: (m',p',s',l') :: st,g,h> \rightarrow_P <(m',p'+1,v :: s',l') :: st,g,h>}$$

Figure 3: Operational semantics of the byte code language

completely independent apart from that.

$$\begin{array}{lll} Expr_V \ni e ::= & n \mid x \mid ? \mid e \diamond e & x\in V, \diamond\in\{+,-,\times,/\} \\ Guard_V \ni t ::= & e \bowtie e & \bowtie\in\{=,\neq,<,\leq,>,\geq\} \end{array}$$

The expression ? represents an unknown value and is responsible for the non-deterministic evaluation of expressions. Analyses will use this expression to model interactive inputs and abstract away numeric quantities not in the scope of the analysis. For instance, our analysis will not keep track of values stored in arrays.

The semantics $[\![e]\!]_\rho$ and $[\![t]\!]_\rho$ of expressions and guards with respect to an environment $\rho\in\mathbb{V}\rightarrow\mathbb{Z}$ are given below.

$$[\![n]\!]_\rho=\{n\} \qquad [\![x]\!]_\rho=\{\rho(x)\} \qquad [\![?]\!]_\rho=\mathbb{Z}$$

$$\begin{array}{lcl} [\![e_1\diamond e_2]\!]_\rho & = & \{n_1\diamond n_2 \mid n_1\in[\![e_1]\!],\ n_2\in[\![e_2]\!]\} \\ [\![e_1\bowtie e_2]\!]_\rho & \Longleftrightarrow & \exists\ n_1\in[\![e_1]\!]_\rho,\ n_2\in[\![e_2]\!]_\rho \quad n_1\bowtie n_2 \end{array}$$

Note that this is not the whole concretisation function for symbolic expressions, which is described later (see Fig. 4).

Symbolic stacks Concrete operand stacks are abstracted by lists of symbolic expressions. To deal correctly with values which are returned after a method

call we use auxiliary variables in a given set A, so the symbolic abstract domain for stacks is $Expr^*_{R+S+A}$.

5.2 Numeric relational domain specification

Apart from symbolic expressions stacks, the byte code analysis is specified with respect to an abstract numeric relational interface (defined below) that can be instantiated with standard relational abstract domains. We thus assume a domain $\mathbb{D}$ parameterised over a (finite) totally ordered set of variables V.

Language independent operators An abstract element is mapped to a set of environments in $\mathbb{Z}^V$ by the concretisation function $\gamma : \mathbb{D}_V \to \mathcal{P}(\mathbb{Z}^V)$. To manage sets of variables, $\mathbb{D}$ is equipped with a projection operator $\exists_{V'} : \mathbb{D}_{V+V'} \to \mathbb{D}_V$, an extension operator $\mathbb{E}_{V'} : \mathbb{D}_V \to \mathbb{D}_{V+V'}$ and a renaming operator $\cdot_{W \to W'} : \mathbb{D}_{V+W} \to \mathbb{D}_{V+W'}$.The abstract domain is also equipped with a partial order $\sqsubseteq \ \subseteq \ \mathbb{D}_V \times \mathbb{D}_V$ and meet and upper bound operators $\sqcap, \sqcup : \mathbb{D}_V \times \mathbb{D}_V \to \mathbb{D}_V$. These components are language-independent.

Language dependent operators The abstract assignment of an expression $e \in Expr_V$ to a variable $x \in V$ is modelled by the operator $[\![x := e]\!]^\sharp : \mathbb{D}_V \to \mathbb{D}_V$. A guard $t \in Guard_V$ may be abstracted by two operators $assume^\sharp(t)$, $ensure^\sharp(t) : \mathbb{D}_v$: the $assume^\sharp$ operator computes an over-approximation of the guard, while $ensure^\sharp$ computes an under-approximation.

Definition 5.1 states formally the requirements over the operators of abstract domain $\mathbb{D}_V$.

Definition 5.1. *An abstract domain $\mathbb{D}$ is a family of sets $\mathbb{D}_V$ with:*

- *a concretisation function $\gamma : \mathbb{D}_V \to \mathcal{P}(\mathbb{Z}^V)$,*
- *a decidable ordering relation $\sqsubseteq \ \subseteq \ \mathbb{D}_V \times \mathbb{D}_V$ such that*

$$d \sqsubseteq d' \Rightarrow \gamma(d) \subseteq \gamma(d'),$$

- *a projection $\exists_{V'} : \mathbb{D}_{V+V'} \to \mathbb{D}_V$, an extension $\mathbb{E}_{V'} : \mathbb{D}_V \to \mathbb{D}_{V+V'}$ and a renaming $\cdot_{W \to W'} : \mathbb{D}_{V+W} \to \mathbb{D}_{V+W'}$ operators such that:*

$$\begin{aligned}\gamma(\exists_{V'}(d)) &= \{\rho_{|_V} \mid \rho \in \gamma(d)\}\\ \gamma(\mathbb{E}_{V'}(d)) &= \{\rho \mid \rho_{|_V} \in \gamma(d)\}\\ \gamma(d_{W \to W'}) &= \{\rho_{W \to W'} \mid \rho \in \gamma(d)\},\end{aligned}$$

- *a meet operator $\sqcap : \mathbb{D}_V \times \mathbb{D}_V \to \mathbb{D}_V$ such that*

$$\gamma(d \sqcap d') = \gamma(d) \cap \gamma(d'),$$

- *an upper bound operator $\sqcup : \mathbb{D}_V \times \mathbb{D}_V \to \mathbb{D}_V$ such that*

$$\gamma(d \sqcup d') \supseteq \gamma(d) \cup \gamma(d'),$$

- *an abstract assignment operator* $[\![x := e]\!]^\sharp : \mathbb{D}_V \to \mathbb{D}_V$ s.t.

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

- $assume^\sharp, ensure^\sharp : Guard_V \to \mathbb{D}_V$ *such that*

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

With the operator $assume^\#$ of the numerical domain we define the abstract test $[\![t]\!]^\sharp : \mathbb{D}_V \to \mathbb{D}_V$ of a guard $t \in Guard_V$ by:

$$\begin{array}{ll} [\![e \bowtie e']\!]^\sharp(l^\sharp) = assume^\sharp(e \bowtie e') \sqcap l^\sharp & \text{if } \bowtie \in \{=,<,\leq,>,\geq\} \\ [\![e \neq e']\!]^\sharp(l^\sharp) = (assume^\sharp(e'<e) \sqcap l^\sharp) \sqcup (assume^\sharp(e<e') \sqcap l^\sharp) & \end{array}$$

The specific rule for $\neq$ is necessary to ensure a good precision with convex polyhedra.

5.3 Analysis specification

The byte code analysis is defined by specifying for each byte code an abstract transfer function which maps abstract states to abstract states (for non-jumping intraprocedural instruction 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 $r_0, \ldots, r_{|L|-1}$ of methods,

R_0: set of old local variables $r_0^{old}, \ldots, r_{|P|-1}^{old}$ of methods, representing their initial values t 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 $Pre \to Post$ whose 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 actually 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

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

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

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

$$\frac{m[p] = \textit{If_icmp cond p'} \quad Loc(m,p) = (e_2 :: e_1 :: s^\sharp, l^\sharp)}{\big(s^\sharp, [\![e_1 \; cond \; e_2]\!]^\sharp(l^\sharp)\big) \sqsubseteq Loc(m, p')}$$

$$\frac{m[p] = \textit{If_icmp cond p'} \quad Loc(m,p) = (e_2 :: e_1 :: s^\sharp, l^\sharp)}{\Big(s^\sharp, [\![e_1 \; \overline{cond} \; e_2]\!]^\sharp(l^\sharp)\Big) \sqsubseteq Loc(m, p+1)}$$

$$\frac{m[p] = \textit{Invoke (mn,n)} \quad \textit{((m',n),c')} \in P \quad Loc(m,p) = (e_{n-1} :: \cdots :: e_0 :: s^\sharp, l^\sharp)}{\left(\exists_{R+S_0+A}\left(\bigsqcap_{i=0}^{n-1} assume^\sharp(e_i = r_i^{old}) \sqcap \exists_{R_0}(l^\sharp)\right)\right)_{S \to S_0} \sqsubseteq Pre((mn,n),c')}$$

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

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

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

$$\frac{(m,n,c) \in P}{\bigsqcap_{i=0}^{|S|-1} assume^\sharp(f_i = f_i^{old}) \bigsqcap_{i=0}^{n-1} assume^\sharp(r_i^{old} = r_i) \sqcap Pre(m) \sqsubseteq Loc(m, 0)}$$

$$\frac{((\mathrm{main}, 0), c) \in P}{\top \sqsubseteq Pre((\mathrm{main}, 0), c)}$$

Figure 4: Relational byte code analysis with stack de-compilation

Definition 5.2 (Abstract domain)**.** *The abstract value for a program P is described by an element $(Pre, Post, Loc)$ of the lattice*

$$\begin{array}{rl} State^\# = & Meth \to \mathbb{D}_{R_0+S_0} \\ & \times \; Meth \to \mathbb{D}_{R_0+S_0+S+\{res\}} \\ & \times \; Meth \times \mathbb{N} \to \left(Expr_{R+S+A}{}^\star \times \mathbb{D}_{R_0+S_0+R+S+A}\right)_\perp \end{array}$$

The analysis is specified as a solution of a constraint (inequation) system associated to each program. The constraint system is formally defined

in Fig 4. Note that extensions are left implicit. For non-jumping intraprocedural instructions, the constraint is defined via a transfer function in $Expr^\star \times \mathbb{D}_{R_0+S_0+R+S+A} \rightarrow (Expr^\star \times \mathbb{D}_{R_0+S_0+R+S+A})_\perp$. We (ab)use notation and write $(e :: s^\sharp, l^\sharp) \rightarrow (e :: e :: s^\sharp, l^\sharp)$ for the function that maps a state of the form $(e :: s^\sharp, l^\sharp)$ to the resulting state $(e :: e :: s^\sharp, l^\sharp)$ and other states to $\perp$. The analysis maintains a symbolic version of the operand stack and most of the transfer functions are defined as symbolic executions. The transfer functions for the stack operations *Nop*, *Pop* and *Dup* mimic the semantics of those operations so *e.g.*, *Dup* will duplicate the expression on top of the (abstract) operand stack and hence is abstracted by the function $(e :: s^\sharp, l^\sharp) \rightarrow (e :: e :: s^\sharp, l^\sharp)$. The abstraction of the instruction *Load* r for fetching the value of local variable r just pushes the expression $\llcorner r \lrcorner$ onto the abstract stack (rather than projecting an abstract value of r from the relation describing the local variables). Similarly, the abstraction of the addition operation *Iadd* pops the two topmost expressions e_1 and e_2 from the abstract stack and replaces them with the symbolic expression $\llcorner e_2 + e_1 \lrcorner$.

The transfer function for the *Store* r operation updates the abstract environment of local variables with the constraint that r is now equal to the value given by the expression e on top of the abstract stack top. Formally, this is done using the operation $[\![x := e]\!]^\sharp$ provided by the interface of the relational domain. By the same token, all occurrences of the sub-expression $\llcorner x \lrcorner$ in the abstract stack become invalid, as r now (potentially) has changed value, and are replaced by the "don't know" expression $\llcorner ? \lrcorner$. The analysis abstracts arrays references by the length of the referenced array, so the transfer functions for *Newarray*(which takes the length as argument and returns a reference to the created array) becomes the identity function. Similarly for *Arraylength*.

For all non-jumping instructions, we generate a constraint saying that the state following the instruction should include the result of applying the transfer function of the instruction to the state preceding the instruction. For the conditional *If_icmp cond p'*, we use the abstract tests provided by the relational domain to take the outcome of the test into account, so *e.g.*, at program point p' we know that the condition *cond* holds between the two top elements of the stack. If these are given by expressions e_1 and e_2 then we know that the symbolic expression $\llcorner e_1\ cond\ e_2 \lrcorner$ evaluates to true in the current environment. The expression $[\![e_1\ cond\ e_2]\!]^\sharp(l^\sharp)$ in the rule for conditionals updates the environment of local variables ($l^\sharp$ to take this information into account. A similar constraint is generated for the program point $p + 1$ using this time the negation $\overline{cond}$ of the condition *cond*.

The analysis of method calls is the most complicated part. The complications partly arise because we have several kinds of variables (static fields, local and auxiliary variables) whose different scope 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 and one that registers the impact of the call on the state immediately following the call site.

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 m'. 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 inject-

ing 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 entire start state for m' is thus described by the expression

$$\left(\exists_{R+S_0+A}\left(\bigsqcap_i assume^\sharp(e_i = r_i^{old}) \sqcap \exists_{R_0}(l^\sharp)\right)\right)_{S\to S_0}$$

The second rule 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 $s^\#$) to name the result of a 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

$$l^\sharp_{S\to S'} \sqcap \exists_{R_0}\left(\bigsqcap_i assume^\sharp(e_i = r_i^{old})_{S\to S'} \sqcap Post(m')_{S_0\to S'}\right)$$

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 x_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, ensures that these values are identified.

Two rules are used to initiate the analysis of a method (constraint on $Loc(m, 0)$) and of the entire program (constraint on $Pre((\text{main}, n), c)$). To initialise the analysis of a method m, the precondition $Pre(m)$ is conjoined with the constraints linking the variable f_i^{old} to the current value of the static field f_i and linking the parameters r_i^{old} with the local variables r_i, in accordance with how parameters are handled in *e.g.* Java byte code. The analysis of the main method starts in the completely unconstrained state $\top$.

5.4 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 operator $F^\sharp \in State^\sharp \to State^\sharp$ operating on the global abstract domain $State^\sharp$ of the analysis. Assuming that $State^\sharp$ is a complete lattice[1] we know that the least solution $lfp F^\sharp$ of this problem exists and can be over-approximated by any post-fixpoint of $F^\sharp$. Computing such a post-fixpoint is the role of chaotic iterations [12] which operate on the equation

[1] For the polyhedra abstract domain this assumption is too strong but we can relax it by considering a complete lattice containing $State^\sharp$ and all its upper bounds [13].

system associated with the constraint system and choose a suitable iteration strategy [8]. Iteration is sped up by using widening on well-chosen control points. Neither the iteration strategy nor the widening operators belong to the TCB since the validity of the result can be checked with a post-fixpoint test.

5.5 Safety checks

Once the analysis has inferred correct invariants, this information is used to check if they enforce the 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.

Definition 5.3 (Abstract safety checks)**.** *We say a set of local invariant* $Loc \in (\mathbb{N} \to (Expr^\star \times \mathbb{D}_{P+S_0+L+S+A})_\perp)$ *verifies all safety checks of a program if and only if*

$$\begin{array}{l}\forall m \in P,\ \ pc \in \mathbb{N},\\ \quad m[p] = Iaload \ \Rightarrow\\ \quad\quad Loc(m,pc) = (e_2 :: e_1 :: s^\sharp, l^\sharp) \Rightarrow\\ \quad\quad l^\sharp \sqsubseteq ensure^\sharp(\llcorner 0 \leq e_2 \lrcorner) \ \wedge\ l^\sharp \sqsubseteq ensure^\sharp(\llcorner e_2 < e_1 \lrcorner)\\ \quad \wedge\\ \quad m[p] = Iastore \ \Rightarrow\\ \quad\quad Loc(m,pc) = (e_3 :: e_2 :: e_1 :: s^\sharp, l^\sharp) \Rightarrow\\ \quad\quad l^\sharp \sqsubseteq ensure^\sharp(\llcorner 0 \leq e_2 \lrcorner) \ \wedge\ l^\sharp \sqsubseteq ensure^\sharp(\llcorner e_2 < e_1 \lrcorner)\end{array}$$

5.6 Soundness of the analysis

Fig. 5 gives the concretisation functions for the abstract domains. The auxiliary abstraction function β maps everything to an integer, abstracting arrays by their length. γ^{expr} defines concretisation of a symbolic expression with respect to an environment. γ^{Pre} maps pre-conditions to sets of calling contexts, γ^{Post} maps post-conditions to relations between calling contexts and return contexts, and γ^{Loc} maps local invariants to relations between calling contexts and local program states. Note that concretisations contain only states such that all locations that are being referenced are defined in the heap.

Definition 5.4 (Reachable states)**.** *For a method m in a program P, a heap h, a static heap g, a set of local variables l, a frame stack st, the set $[\![P]\!]^m_{h,g,l,st}$ of reachable state from an execution of m starting in an initial configuration (h, g, l, st) is defined by*

$$[\![P]\!]^m_{h,g,l,st} = \left\{ s \mid <(m, 0, [], l') :: st, g, h> \xrightarrow{\geq st}{}^*_P\ s \right\}$$

where $\xrightarrow{\geq st}{}^*_P$ *is the reflexive transitive closure of* $\to_P$ *restricted to states who have a form* $< \ldots :: (m, \ldots) :: st, \ldots >$.

The purpose of $\xrightarrow{\geq st}{}^*_P$ is to collect only the states in between the start and the end of the execution of a particular stack frame.

$$
\begin{array}{llcl}
\beta_{\mathbb{V}}: & Heap \times (\mathbb{V} \to Val) & \to & (\mathbb{V} \to \mathbb{Z}_\perp) \\
 & (h, z) & \mapsto & \lambda x. \left| \begin{array}{ll} z(x) & \text{if } z(x) \in \mathbb{Z} \\ |h(z(x))| & \text{if } z(x) \in dom(h) \end{array} \right. \\
\gamma^{expr}_{h,g,l,a}: & Expr_{R+S+A} & \to & \mathcal{P}(Val), \quad h \in Heap, g \in Static,\ l \in LocVar \\
 & & & \qquad\qquad \text{and } a \in Val \to \mathbb{Z} \\
 & e & \mapsto & [\![e]\!]_{\beta_{R+S+A}(h, l \oplus g \oplus a)} \\
 & & & \cup \left\{ ref \in dom(h) \;\middle|\; |h(ref)| \in [\![e]\!]_{\beta_{R+S+A}(h, l \oplus g \oplus a)} \right\}
\end{array}
$$

$$
\begin{array}{llcl}
\gamma_{Pre}: & \mathbb{D}_{R_0+S_0} & \to & \mathcal{P}(Static \times Heap \times LocVar) \\
 & pre & \mapsto & \{\ (g_0, h_0, l_0) \mid \beta_{R_0+S_0}(h_0, l_0 \oplus g_0) \in \gamma(pre)\ \} \\
\gamma_{Post}: & \mathbb{D}_{R_0+S_0+S+\{res\}} & \to & \mathcal{P}((Static \times Heap \times LocVar) \times (Static \times Heap \times Val)) \\
 & post & \mapsto & \left\{ \begin{array}{l} ((g_0, h_0, l_0), (g, h, v)) \mid \\ \quad \beta_{S_0+R_0+S+\{res\}}(h_0, g_0 \oplus l_0 \oplus g \oplus [res \mapsto v]) \in \gamma(post) \end{array} \right\} \\
\gamma_{Loc}: & \begin{array}{l} Expr^\star \\ \times\ \mathbb{D}_{R_0+S_0+R+S+A} \end{array} & \to & \mathcal{P}\left(\begin{array}{l} (Static \times Heap \times LocVar) \\ \times\ (Static \times Heap \times Stack \times LocVar) \end{array} \right) \\
 & (e_1 :: \cdots :: e_n, loc) & \mapsto & \left\{ \begin{array}{l} ((g_0, h_0, l_0), (g, h, v_1 :: \cdots :: v_n, l)) \mid \\ \quad \exists a \in A \to \mathbb{Z},\ \forall i \in [\![1, n]\!],\ v_i \in \gamma^{expr}_{h,g,l,a}(e_i) \wedge \\ \qquad \beta_{S_0+R_0+S+R+A}(h_0, g_0 \oplus l_0 \oplus g \oplus l \oplus a) \in \gamma(loc) \end{array} \right\}
\end{array}
$$

Figure 5: Concretisation functions

Definition 5.5 (Safe method)**.** *A method m in a program P is said to be* safe *wrt. a precondition $Pre \subseteq Heap \times Static \times LocVar$ if for all stack frames st and all $(h, g, l) \in Pre$, $Error \notin [\![P]\!]^m_{h,g,l,st}$.*

Theorem 5.6 (Correctness)**.**
Let P be a program and $(Pre, Post, Loc)$ a solution of the constraint system associated with P. If Loc satisfies all safety checks then every method m in P is safe wrt. to $Pre(m)$. In particular,

$$< (((main, n), c), 0, [], \lambda r.0) :: [], \lambda f.0, \lambda ref.\bot > \not\to_P Error$$

Proof. The proof is divided into two parts. We first prove that each reachable intermediate state at a point (m, p) satisfies the property $\gamma_{Loc}(Loc(m, p))$, that each method m is called in a context satisfying $\gamma_{Pre}(Pre(m))$ and that its return value (if it exists) satisfies $\gamma_{Post}(Loc(m))$. In the second part we prove that if a state at some point (m, p) satisfies $\gamma_{Loc}(Loc(m, p))$ as well as the abstract safety check associated with this point, then no error happens in the next semantic step. To deal with the steps corresponding to procedure calls, the proof makes use of an intermediate big-step operational semantics. Details are omitted for lack of space. □

6 Polyhedral analysis

We now instantiate the relational analysis framework using linear relations in the form of convex polyhedra. Polyhedral program analysis has a well-established theory [13] with several implementations [4, 15]. Here, we recall the basics of this theory.

Definition 6.1. *Convex polyhedra of dimension n ($\mathbb{P}_n \subseteq \mathbb{Q}^n$) are (convex) subsets of $\mathbb{Q}^n$ that can be expressed as a finite intersection of half-planes of $\mathbb{Q}^n$.*

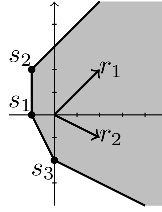

Figure 6: Dual representation of polyhedra

Polyhedra can be represented as sets of linear constraints. It is desirable to keep these sets in normal form *i.e.,* without redundant constraints. For this purpose, polyhedra libraries maintain a dual representation 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. Fig. 6 shows a a polyhedron with four constraints whose dual representation is made of three *vertices* (s_1,s_2,s_3) and two *rays* (r_1,r_2).

The efficiency of the algorithm that maintains the normal form of the double description is of crucial importance. For this task, state-of-the-art polyhedral libraries [4, 15] use Chernikova's algorithm [11]. In the worst case, the number of generators is exponential in the number of constraints (and *vice-versa*) but, in practise, the double description offers a good performance. To alleviate further the cost of normalising polyhedra, these libraries switch lazily from one representation to the other.

Polyhedral cannot directly handle expressions that fall outside the linear fragment. It would be sound but unsatisfactory to abstract those expressions towards an arbitrary value *i.e.*, the ? expression. More information can be retained by *linearising* expressions [19]. For instance, the precise analysis of Binary Search (Fig. 1) requires a precise model of euclidean divisions. Given an integer constant n, the guard $y = x/n$ is abstracted by the linear guards $0 \leq x - n \cdot y < n$. Multiplications can also be linearised by using the range of variables.

We now briefly explain how polyhedral algorithms implement the abstract numeric relational domain specified in Definition 5.1. To be implemented efficiently, the double description of polyhedra is needed, using Chernikova's algorithm to reconstruct the coherence of the double representation.

The convex polyhedron can directly be cast into an abstract numeric domain by mapping variables of the domain to dimensions of the polyhedron. Hence, we get $\mathbb{D}_V = \mathbb{P}_{|V|}$ and the concretisation:

$$\gamma(P) = \{\rho \in \mathbb{Z}^V \mid \rho \in P \cap \mathbb{Z}^V\}$$

Renaming of variables consists in applying a permutation to the dimensions of polyhedron. The **extension** operation which add new variables consists in inserting new unconstrained dimensions at the relevant indexes.

Projections can be efficiently performed on the *generator* description of polyhedra in linear time. Each generator is projected by erasing the now irrelevant dimensions.

Intersections are computed by taking the union of the constraints of each polyhedron.
The **convex hull**, *i.e.*, least upper bound, is computed by taking the union of the generators of both polyhedra.
An **assignment** $[\![x := e]\!]^\sharp$ is modelled by the linear transformation (if e is linear) that keeps all the variables unchanged except x which is mapped to e. The transformation is applied to the generators.
Inclusion tests are using both representation at once. Checking the containment of two polyhedra ($P \sqsubseteq Q$) amounts to verifying that the generators of P satisfy the constraints of Q.
Widening operators are used by the fixpoint iterator to ensure convergence. For convex polyhedra, there exist various widening operators [13, 3].
Assume and ensure operators are responsible for interpreting guards of the target language. If the guard t is linear, a polyhedron is built from it and no abstraction takes place. Otherwise, t has to be linearised. In the worst case, universal (resp. empty) polyhedra can be used as sound (though very imprecise) fallbacks.

7 Fixpoint pruning

The result of the polyhedral byte code analysis will be a fixpoint of the transfer functions, representing an invariant of the program under analysis. This invariant will often contain more information than necessary for proving a particular safety policy such as absence of indexing outside array bounds. In the following we show how to *prune* an invariant with respect to a given safety policy, resulting in an invariant that is smaller and cheaper to verify.

7.1 Witnesses and pruning

We have applied the technique described in [7] for pruning constraint-based invariants, with some adaptations allowing to handle our interprocedural polyhedral analysis on byte code better. First we recall the definition of witnesses for this particular analysis.

Definition 7.1. *A witness for a program P is a solution (Pre, Post, Loc) to the constraint system associated with P that satisfies the safety checks of P (see Definition 5.3).*

We use this as the basis for building certificates, relying on the fact that if there exists a witness for P then P is safe (see Theorem 5.6). Part of the witness is sent to the checker in the constraint representation only (see Section 6), so we aim at extracting a weaker witness with fewer linear constraints than the one produced by the inference algorithm of Section 5.4 (if the analysis is accurate enough for the program). Pruning leaves the symbolic expression stacks of the witness unchanged because the checker recomputes them (and hence nothing is transmitted about this part).

It is easy to see that there is generally no unique weakest witness nor a unique witness with the minimum number of constraints (because the analysis is not *distributive*). Also, the idea of starting from the safety requirements to compute backward a witness that satisfies them cannot achieve the same precision as a

prune(w) :=
  let $\overline{w'} = \emptyset$
  while w' is not a witness do
    choose a constraint C and $k \in w'_{|dep(C)}$ *s.t.* $\overline{w'}, \{k\} \not\vdash C$
      (or a check C such that $\overline{w'} \not\vdash C$)
    choose $\overline{x} \subseteq (\overline{w} \setminus \overline{w'})_{|dep(C)}$ such that $\overline{w'} \cup \overline{x}, \{k\} \vdash C$
      (respectively, $\overline{w'} \cup \overline{x} \vdash C$)
    $\overline{w'} := \overline{w'} \cup \overline{x}$
  done
  return w'

Figure 7: Witness pruning algorithm

forward analysis, because intuitively it would have to guess the invariants that a forward analysis naturally discovers. For these reasons we use a technique of pruning that removes as many linear constraints as possible from a given witness.

7.2 Abstract algorithm

We use a variation of the greedy heuristic presented in [7]. In the following we identify polyhedra with sets of constraints. We use

$$\begin{aligned} Var = \quad & \{\mathrm{pre}_m \mid m \in P\} \cup \{\mathrm{post}_m \mid m \in P\} \\ & \cup \{\mathrm{loc}_{m,p} \mid m \in P,\ m = ((mn, n), c),\ p < |c|\} \end{aligned}$$

to denote the set of unknowns of the constraint system associated with P. For an abstract element $x = (Pre, Post, Loc)$ we define the set of linear constraints of x:

$$\overline{x} = \bigcup_{m \in P} \begin{aligned} & \{(\mathrm{loc}_{m,p}, k) \mid Loc(m,p) = (s^\#, l^\#),\ k \in l^\#\} \\ \cup\ & \{(\mathrm{pre}_m, k) \mid k \in Pre(m)\} \\ \cup\ & \{(\mathrm{post}_m, k) \mid k \in Post(m)\} \end{aligned}$$

For $V \subseteq Var$ we define $\overline{x}_{|V} = \{(var, k) \in \overline{x} \mid var \in V\}$ and $x_{|V}$ is defined accordingly.

Recall that the constraint system for P is a set of constraints of the form $F(x) \sqsubseteq x_{|\{v\}}$ where $v \in Var$. For a constraint c we note $\overline{x}, \overline{y} \vdash C$ if $F(x) \sqsubseteq y_{|\{v\}}$ (we can do so since the expression stacks are fixed) and $\overline{x} \vdash C$ for $\overline{x}, \overline{x} \vdash C$. We will overload the notation and write also $\overline{x} \vdash C$ if x satisfies the safety check C. Then, for every such constraint C, we define a set $dep(C) \subseteq Var$ that represents the dependencies of this constraint, in the sense that if $\overline{x}, \overline{y} \vdash C$ then $\overline{x}_{|dep(C)}, \overline{y} \vdash C$. The definition of dep is straightforward. For example, if C is the constraint $F_{instr}(Loc(m,p)) \sqsubseteq Loc(m, p+1)$ corresponding to an non-jumping intraprocedural instruction (see the first part of Fig. 4), then $dep(C) = \{\mathrm{loc}_{m,p}\}$. For the constraint $\ldots \sqsubseteq Loc(m, p+1)$ of an *Invoke sig* instruction, $dep(C) = \{\mathrm{loc}_{m,p}, \mathrm{post}_{(sig,c)}\}$ where $(sig, c) \in P$.

The pruning algorithm is shown in Fig. 7. The main issue in this non-deterministic algorithm is the choice of the subset $\overline{x}$: we obviously want a minimal one in the sense of set inclusion (achievable in reasonable time by monotonicity), but it is not unique.

7.3 Efficient pruning for polyhedral byte code analysis

Our strategy is to take a minimal such $\overline{x}$ that almost minimizes a cost function taking into account the number of linear constraints, the number of non-null coefficients in them, and, for *Invoke*, the number of post constraints (as opposed to loc). This allows us to obtain a witness with simpler invariants and signatures. The heuristic blindly applies the definition of $\vdash$ while labelling (part of) the search space. The dependency function *dep* helps by reducing the number of linear constraints to be considered at each step.

Finally, we face a problem specific to the polyhedra domain when pruning an invariant: in order to keep things small, the polyhedra are usually represented in a minimal form in which the relation between a set of dimensions does not necessarily appear as a dedicated linear constraint, but often as a consequence of several other relations. For example, the constraint $x \leq z$ is implicit in $x \leq y \leq z$. For the purpose of finding a small invariant, we may benefit from being able to include such constraints. Our solution is to add some implicit constraints to the invariant before pruning it. More precisely, for a polyhedron in $\mathbb{D}_V$, we add all the projections $\exists_{V \setminus V'}$ (see Section 6) where V' is a subset of V of cardinality at most n. For the maximal number n of dimensions in the implicit constraints to be generated, ∞ seems too costly for non-trivial programs, and unnecessary. It turns out that 3 is enough for all of our examples, which is not surprising because very few correctness proofs actually rely on linear invariants involving more than three variables.

8 Result checking of polyhedral analysis

A result checker for abstract interpretation based static analysis can be reduced to an (optimised) fixpoint checker [1], with the downside that the abstract domains are still part of the TCB. Formally certifying optimised polyhedral libraries [4, 15] is feasible but would require an enormous certification effort. Instead, we propose a lightweight verifier of polyhedral analyses using a result checking methodology which has two advantages: i) the TCB is small, and ii) the checking time is optimised.

8.1 The polyhedral domain revisited

Chernikova's algorithm is at the origin of the computational complexity of convex polyhedra operations, so a first approach would be to design a result checker for Chernikova's algorithm *i*.e., a normal form checker. This has the inconvenience that most of the polyhedral operations would be annotated with their result together with a certificate attesting that it is in normal form. Instead, we develop a checker which only uses the constraint representation of polyhedra and which never need to normalise. 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 while $e \in E$ are (existential) variables that represent dimensions which have been projected out.

Definition 8.1. *Let V and E be disjoint sets of variables.*

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

where

$$Lin_{V+E} = \{\llcorner c_1 \times x_1 + \cdots + c_n \times x_n \lrcorner \mid c_i \in \mathbb{Z} \wedge x_i \in V + E\}.$$

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

$$\gamma_V(es) = \{\rho_{|V} \mid \forall k \in es, [\![k \geq 0]\!]_\rho\}$$

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. In addition, for this encoding, **extension** is a no-op because unused variables have no impact on the internal representation.

$$es \in \mathbb{P}_V \Rightarrow \forall W \supseteq V, es \in \mathbb{P}_W$$

Using Fourier-Motzkin elimination (see *e.g.*), [23], **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 variables.

To compute **intersections**, care must be taken not to mix up the existential variables. To avoid capture, existentially variables are renamed to variables that are fresh for both polyhedra. Thereafter, the intersection is implemented by taking the union of the expressions.

To implement the **assume** and **ensure** operators, the involved expressions are first linearised and the obtained linear inequality is put into the form $e \geq 0$ which now belongs to the set Lin defined above.

For convex polyhedra, **assignment** is efficiently implemented as an atomic operation. However, it can be expressed in terms of the previous operators: given x' a fresh variable, an assignment can be defined as follows.

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

It is this latter definition that we use.

Widening operators are only used during the fixpoint iteration, and are not needed at checking time.

Convex Hull is the typical operation that is straightforward to implement using the generator representation of polyhedra. Using a relaxation technique, it is possible to express the convex hull as the projection of a polyhedron of higher dimension [2] but since this requires to compute projections this does not scale. Even with our delaying of projections, the size of the polyhedron doubles. 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 double inclusion test.

$$isUpperBound(P, Q, UB) \equiv P \sqsubseteq UB \wedge Q \sqsubseteq UB$$

To implement **inclusion tests**, we push the result certification methodology further and use inclusion certificates. The form of certificates and their generation are described below.

8.2 Result certification for polyhedral inclusion

Farkas lemma (Lemma 8.2) is a theorem of linear programming (see for instance [23]) which gives a notion of *emptiness* certificate for polyhedra. In this part, we show how this result can be i) lifted to obtain an inclusion checker; ii) extended further to deal with existential variables. 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 allows to conclude that P is indeed included in Q ($P \sqsubseteq Q$).

Lemma 8.2 (Farkas Lemma)**.** *Let $A \in \mathbb{Q}^{m\times n}$ and $b \in \mathbb{Q}^n$. 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 ($\Leftarrow$) of certificates is the easy part and is all that is needed in the machine-checked proof. It follows that the existence of a certificate ensures the infeasibility of the linear constraints and therefore that the polyhedron made of these constraints is empty.

Thus, an *inclusion certificate ic* is a vector of $\mathbb{Q}^m$ and checking a certificate 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 ($b^t \cdot ic$); and 4) verifying that the result is strictly positive. All in all, the certificate checker runs in quadratic-time in terms of arithmetic operations.

Certificates generation can be recast as a linear programming problem that can be efficiently solved by either the Simplex or interior point methods. The set of certificates is characterised by the convex polyhedron

$$Cert = \left\{ ic \,\middle|\, ic \geq \bar{0} \wedge b^t \cdot ic > 0 \wedge A^t \cdot ic = \bar{0} \right\}$$

As a result, finding an *extremal* certificate amounts to solving a linear optimisation problem. For instance, the solution of the linear program $min\{c^t \cdot \bar{1} \mid c \in Cert\}$ minimises the sum of the coefficients of the certificate. In theory, such a minimisation might not yield a compact certificate because the optimisation is done over the rationals – there are very small rationals that require many bits. However, in practise, the technique is sufficiently efficient.

From emptiness to inclusion Lemma 8.3 states that in the absence of existential variables an inclusion check amounts to emptiness checks.

Lemma 8.3. *Given $P, P' \in \mathbb{P}_{V+E}$, we have*

$$\forall e' \in P', \gamma_{V+E}(-e'-1 :: P) = \emptyset$$

if and only if

$$\gamma_{V+E}(P) \subseteq \gamma_{V+E}(P').$$

Proof. By construction, polyhedra in $\mathbb{P}_{V+E}$ do not have existential variables. Hence, we have $\gamma_{V+E}(P') = \bigcap_{e' \in P'} \gamma_{V+E}(e'::[])$. Moreover, the complement of a linear constraint $e' \geq 0$ is $-e' - 1 \geq 0$. These facts allow to reduce inclusion to a set of emptiness tests. □

Lemma 8.4 states that to do an inclusion test, it is sound to drop existential variables.

Lemma 8.4. *Let P and P' be polyhedra in constraint form.*

$$\gamma_{V+E}(P) \subseteq \gamma_{V+E}(P') \Rightarrow \gamma_V(P) \subseteq \gamma_V(P')$$

Proof. The Lemma follows from the definition of γ and the fact that the restriction operator on environments is monotone. □

Together, Lemma 8.3 and Lemma 8.4 allow the design of a sound result checker for inclusion tests of form $P \subseteq P'$. In general, the checker is incomplete but this only shows up in cases where P' has existential variables. However, inclusions only need to be certified when P' is a polyhedron computed by the analyser and such a P' *does not* contain existential variables, so the inclusion checker is always used in a context where it is complete.

9 Implementation and Experiments

The relational byte code analysis has been implemented in Caml and instantiated with the efficient NewPolka polyhedral library [15] as its relational abstract domain. As the language presented in Section 4 is compatible with Java byte code, we analyse programs that are compiled from genuine Java programs. The result checker for polyhedral analysis described in Section 8 has been implemented in the Coq proof assistant.

The analyser computes a solution to the constraint system generated from a program and passes it on to the fixpoint pruning algorithm. From the compressed 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.

In Fig 8, we give a table of some of the benchmarks used by Xi to demonstrate the dependent type system for Xanadu [27]. HeapSort and QuickSort are particularly good illustrations of the capacities of our analysis for analysing automatically programs mixing recursive procedures and loops. The programs and the analysis results can be found at `http://www.irisa.fr/lande/polycert.html`.

	.class	certificates		checking time	
Program		before	after	before	after
BSearch	515	22	12	0.005	0.007
BubbleSort	528	15	14	0.0005	0.0003
HeapSort	858	72	32	0.053	0.025
QuickSort	833	87	44	0.54	0.25

Figure 8: Class files in bytes, certificates in number of constraints, time in seconds

The two checking times in the last column give the checking time with and without fixpoint pruning Three things are worth noticing. First, invariants that had to be produced manually for the Xanadu examples are now inferred

automatically. Second, the checking time is small, especially given that the checker is extracted from its Coq specification and thus is a purely functional implementation of a polyhedral domain. Third, pruning can halve the number of constraints to verify. This reduction can sometimes but not always produce a similar reduction in checking time. 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.

10 Related work

A number of relational abstract domains (octagons [18], convex polyhedra [13], polynomial equalities [20]) 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 byte code. Dependent type systems for Java-style byte code for removing array bounds checks have been proposed by Xi and Xia [28]. The analysis of the stack uses singleton types to track the values of stack elements, achieving the same as our symbolic stack expressions. The analysis is intra-procedural and does not consider methods (they are added in a later work [27] 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 has been an active field recently. Wildmoser, Nipkow *et al.* [25] were the first to develop a fully certified VCGen within Isabelle/HOL for verifying arithmetic overflow in Java byte code. The certification of abstract interpreters has been developed by Cachera, Pichardie *et al.* [9, 21]. for a variety of analyses including class analysis of Java byte code and interval analysis. Lee *et al.* [16] have certified the type analysis of a language close to Standard ML in LF and Leroy [17] has certified some of the data flow analyses of a compiler back-end. Leroy also observes that for certain, more involved analyses such as the register allocation, it is simpler and sufficient to certify a checker of the result than the analysis itself. The same idea is used by Wildmoser *et al.* [24] who certifies 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 byte code 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.

The idea of removing useless parts from an invariant was developed independently by Besson *et al.* [7] and by Yang *et al.* [29] who call it abstract value slicing. Both works deal with intra-procedural invariants and both are based on a dependency computation that selects, for every constraint $F(X) \sqsubseteq Y$ of the constraint system of P and every subset of an abstract state Y, a sufficient subset of X that satisfies the constraint. The two methods differ in the way that this choice is done but both have been shown viable for intra-procedural pruning of

relational invariants. The present work is an extension of the principles underlying the non-deterministic algorithm in [7] to handle the pre-/post-conditions arising from the interprocedural analysis. Finally, it should be noted that the fixpoint compression is orthogonal to and compatible with the optimisation of iteration strategies for fixpoint checking underlying Lightweight Bytecode Verification [22] and the more general abstraction-carrying code [1, 6]. Our checker combines both techniques.

11 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 byte code language. The machine-generated invariants can be pruned wrt. a particular safety policy to yield compact program certificates. To simplify the checking of these certificates, 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 completion of polyhedra or non-linear (polynomial) invariants. The certificate format and the result checker can accommodate the disjunctive completions, the inclusion certificates from Section 8.2 can be generalised to deal with non-linear inequalities as well [5]. 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 byte code. 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 analysis to keep track of aliases between objects.

A promising domain of application for our relational analysis technique is to verify the dynamic allocation and consumption of resources and in particular to ensure statically that a program always acquires a necessary amount of resources before consuming them. The approach of Chander *et al.* [10] relies on the programmer to provide loop invariants and pre- and post-conditions for methods in order to link program variables to the amount of resources available and perform powerful transformations such as hoisting resource allocations out of loops. Our inter-procedural byte code analyser could infer the necessary invariants and pre-/post-conditions and in the same vein provide the checker for integrating this into a mobile code resource certification scheme.

References

[1] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-carrying code. In *Proc. of 11th Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR'04)*, Springer LNAI vol. 3452, pages 380–397, 2004.

[2] B. De Backer and H. Beringer. A clp language handling disjunctions of linear constraints. In *Proc. of the 10th Int. Conf. on Logic Programming (ICLP'93)*, pages 550–563. MIT Press, 1993.

[3] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In *Proc. of 10th Int. Static Analysis Symposium*, pages 337–354. Springer LNCS vol. 2694, 2003.

[4] R. Bagnara, P.M Hill, and E. Zaffanella. The Parma polyhedral library user's manual, 2006.

[5] F. Besson. Fast reflexive arithmetic tactics: the linear case and beyond. In *Types for Proofs and Programs (TYPES'06)*, pages 48–62. Springer LNCS vol. 4502, 2006.

[6] F. Besson, T. Jensen, and D. Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theoretical Computer Science*, 364(3):273–291, 2006.

[7] F. Besson, T. Jensen, and T. Turpin. Small witnesses for abstract interpretation based proofs. In *Proc. of 16th European Symp. on Programming (ESOP 2007)*, pages 268 – 283. Springer LNCS vol. 4421, 2007.

[8] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the Int. Conf. on Formal Methods in Programming and their Applications*, pages 128–141. Springer LNCS vol. 735, 1993.

[9] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. In *Proc. of 13th European Symp. on Programming (ESOP'04)*, pages 385–400. Springer LNCS vol. 2986, 2004.

[10] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *Proc. of the 14th European Symposium on Programming (ESOP 2005)*, pages 311–325. Springer LNCS vol. 3444, 2005.

[11] N.V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities. *U.S.S.R Computational Mathematics and Mathematical Physics*, 5(2):228–233, 1965.

[12] 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.

[13] 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.

[14] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. of the ACM Conf. on Programming Language Design and Implementation (PLDI'2002)*, pages 234–245, 2002.

[15] B. Jeannet and the Apron team. The Apron library, 2007.

[16] 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.

[17] 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 (POPL'06)*, pages 42–54, New York, NY, USA, 2006. ACM Press.

[18] A. Miné. The octagon abstract domain. In *Proc. of Working Conf. on Reverse Engineering 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[19] A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *VMCAI'06*, volume 3855 of *LNCS*, pages 348–363. Springer, 2002.

[20] 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.

[21] David Pichardie. *Interprétation abstraite en logique intuitioniste: extraction d'analyseurs Java certifiés.* PhD thesis, Université de Rennes 1, Sept. 2005.

[22] E. Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, 2003.

[23] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1998.

[24] M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In *Proc. of 1st Workshop on Bytecode Semantics, Verification and Transformation, Electronic Notes in Computer Science*, 2005.

[25] M. Wildmoser and T. Nipkow. Asserting bytecode safety. In *Proc. of the 15th European Symp. on Programming (ESOP'05)*, 2005.

[26] M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping proof carrying code. In *Exploring New Frontiers of Theoretical Informatics, TC1 3rd Int. Conf. on Theoretical Computer Science (TCS2004)*, pages 333–347. Kluwer, 2004.

[27] Hongwei Xi. Imperative Programming with Dependent Types. In *Proc. of 15th IEEE Symposium on Logic in Computer Science (LICS'00)*, pages 375–387. IEEE, 2000.

[28] Hongwei Xi and Songtao Xia. Towards Array Bound Check Elimination in Java Virtual Machine Language. In *Proc. of CASCOON '99*, pages 110–125, 1999.

[29] H. Yang, S. Seo, K. Yi, and T. Han. Goal-directed weakening of abstract interpretation results. Submitted for publication.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399