Decomposing Bytecode Verification by Abstract Interpretation

C. BERNARDESCHI, N. DE FRANCESCO, G. LETTIERI,
L. MARTINI, and P. MASCI
Università di Pisa

Bytecode verification is a key point in the security chain of the Java platform. This feature is only optional in many embedded devices since the memory requirements of the verification process are too high. In this article we propose an approach that significantly reduces the use of memory by a serial/parallel decomposition of the verification into multiple specialized passes. The algorithm reduces the type encoding space by operating on different abstractions of the domain of types. The results of our evaluation show that this bytecode verification can be performed directly on small memory systems. The method is formalized in the framework of abstract interpretation.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Smartcards*

General Terms: Verification, Theory

Additional Key Words and Phrases: Abstract interpretation, bytecode verification

ACM Reference Format:

Bernardeschi, C., De Francesco, N., Lettieri, G., Martini, L., and Masci, P. 2008. Decomposing bytecode verification by abstract interpretation. ACM Trans. Prog. Lang. Syst. 31, 1, Article 3 (December 2008), 63 pages. DOI = 10.1145/1452044.1452047 http://doi.acm.org/10.1145/1452044.1452047

1. INTRODUCTION

The Java platform was developed in the early 1990s in order to give developers flexible tools for programming smart electronic devices. The core of the Java platform is a Java Virtual Machine (JVM). The JVM is a software CPU with

This work was partially supported by the ReSIST IST-026764 project.
A preliminary version of this article appeared in *Proceedings of the 1st Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'05)* with the title, A Space-Aware Bytecode Verifier for Java Cards
Authors' address: C. Bernardeschi, N. De Francesco, G. Lettieri, L. Martini, and P. Masci, Università di Pisa, Dipartimento di Ingegneria dell'Informazione, sez. Informatica, Via Diotisalvi 2, 56122 Pisa, Italy; email: {cinzia,nico,g.lettieri,luca.martini,paolo.masci}@iet.unipi.it

a stack-based architecture that creates an execution environment between the operating system of the device and the program. The JVM guarantees that the program is independent from the operating system, and programs are always compiled in a standardized binary code, called Java bytecode.

In the last decade, the Java platform moved to mobile and embedded systems as well, such as mobile phones and electronic smart cards. Ad hoc versions of the JVM were produced in order to suit the hardware constraints of the devices: the Kilo Virtual Machine (KVM), and the Java Card Virtual Machine (JCVM). These kinds of mobile and embedded technologies represent an interesting research challenge: such systems have limited memory, limited computing speed, and sometimes even limited energy budgets; on the other hand, they have to provide secure execution environments since programs may handle sensitive information.

The Java bytecode verifier is one of the key components of Java's security system. The verifier checks that the bytecode of programs is correct before execution. The verifier not only checks that bytecode is well formed, but it also ensures that no execution can violate any of the language typing rules. This is accomplished through a dataflow analysis which checks the type-correctness of the code. In the dataflow analysis performed by the bytecode verifier, each method is executed abstractly, using types instead of actual values. The verifier checks that the operands of each instruction match the required type. The verification algorithm must store the abstract execution state of the virtual machine (VM) at each branch target of the method [Leroy 2003]. An abstract execution state of the VM consists of the type information for the operand stack and the local variables.

With mobile and embedded devices, bytecode verification enables *post issuance* download of programs, even when such programs are not downloaded directly from the vendor but also from other sources. However, bytecode verification in its standard form cannot be performed directly on-board for some resource-constrained devices, since the analysis requires too much memory. Several approaches have been proposed in the literature to perform bytecode verification directly on-board. They modify the standard verification process so that the memory available is sufficient for the verification to be carried out. The current trend is to perform verification based on Proof Carrying Code (PCC) [Necula 1997] techniques, such as Lightweight Bytecode Verification (LBV) [Rose 2003]. In LBV verification is executed off-line and produces a certificate that must be distributed with the bytecode and checked on-board on the device. This on-board check requires much less memory than standard verification.

In this article, we propose and evaluate an alternative approach which checks the correctness of the bytecode by means of a progressive analysis (*multipass verification*) requiring much less memory than the standard analysis. The idea is to reduce the space needed for encoding the abstract execution states by decomposing the analysis into smaller subanalyses (requiring very small amount of memory) in such a way that the verification can be obtained by executing the subanalyses separately. We propose a *serial/parallel decomposition* of bytecode verification. In the parallel decomposition, a set of independent subanalyses is applied, each of which checks only a subset of types. In the serial decomposition,

a subanalysis can use some information produced by a previously applied analysis.

The approach is developed within the framework of abstract interpretation [Cousot and Cousot 1992, 1977]. An abstract interpreter executes the program in an abstract (approximated) way to statically check dynamic properties. The actual domain of computation (called the *concrete* domain) is replaced by an *abstract* domain, and the operators of the concrete computation are replaced by correct abstract operators. Correctness of the abstract interpretation is proved a standard way.

We give a formal semantics of the verification process, which is taken as the concrete semantics of verification. In the *parallel decomposition*, each subanalysis is modeled as a different abstract interpretation of bytecode verification and corresponds to a different abstraction of the domain of types. In each of these analyses, only a given subset of the type constraints are checked. Therefore, each type can be encoded with fewer bits (only one in the best case) and the memory requirements are reduced.

The best results can be obtained when we need to look only at one type at a time (the so-called decomposition into basic domains). Unfortunately, due to the nature of some bytecode instructions, this is not always possible. To solve this problem, we propose *serial decomposition* which simplifies some subanalyses by exploiting the results of some other subanalyses done previously. Serial decomposition is defined combining abstract interpretation with program transformation: at each step, the invariant properties discovered in the previous subanalyses are used to simplify the code to which the next analysis has to be applied. In this way decomposition into basic domains is always possible—there is only a small penalty of memory due to the saving of temporary results.

The paper proves that the proposed method is both sound and complete. This means that:

—if a program is not correct, that is, it does not pass standard bytecode verification, then at least one of the subanalyses reports an error;

—if a program is correct, that is, it passes standard bytecode verification, then all the subanalyses report no errors.

We remark the advantages of using abstract interpretation as a framework in which multipass verification is defined. First, abstract interpretation is a well-assessed theory allowing to prove correctness of static analyses in a standard way. Secondly, the complete shell theory [Giacobazzi et al. 2000], which is part of the abstract interpretation framework, suggests the effective definition and construction of the abstract domain for each subanalysis in which verification is decomposed. Moreover the abstract interpretation which we define does not affect the behavior of the bytecode verifier since only the domain of types is abstracted and the verifier is executed with the same algorithm on the abstract domain. Hence it is possible to combine our method with other verification algorithms, possibly achieving advantages in memory requirements. For example, the proposed method can be combined with LBV. This way a multipass algorithm can be obtained, where each pass is performed with LBV technique.

This adds flexibility in the management of the certificate, and in some cases produces a smaller certificate.

Finally, we think that the parallel/serial decomposition technique, defined in the paper for bytecode verification, can be generalized as a standard technique for static analysis decomposition, and formalized in the general case within the abstract interpretation theory.

The article is organized as follows. In Section 2 we recall basic notions about Java bytecode language, bytecode verification and abstract interpretation; moreover, we give an overview of our approach (Section 2.2). Then, we formalize the bytecode verification algorithm in such a way that abstract interpretation can be applied easily (Section 3). Parallel decomposition is presented in Section 4. Serial decomposition is described in Section 5. Section 6 evaluates the memory requirements of the multipass verification and compares this algorithm with the standard bytecode verification and with LBV. Related work is reviewed in Section 7, and Section 8 concludes the work.

2. OVERVIEW

In this section we summarize the basic notions about the Java Virtual Machine (JVM), its language and bytecode verification, and we outline our approach.

2.1 JVML and Bytecode Verification

For our purposes, a Java program consists of a set $\mathcal{C}$ of user defined *classes*. Each class $\tau \in \mathcal{C}$ defines a set of class fields and methods. Each field has a type, while each method accepts a fixed number of typed parameters and returns a typed value. The set $\mathcal{T}$ of types includes the set $\mathcal{B} = \{\texttt{i}, \texttt{f}, \texttt{b}, \dots\}$ of primitive (or base) types and the set $\mathcal{C}' = \{\texttt{Object}\} \cup \mathcal{C}$ of user defined classes, together with the predefined `Object` class, and the set of array types defined in Section 4.2.

Classes are related by a user-defined *extends* binary relation, $\mathit{extends} \subseteq \mathcal{C}' \times \mathcal{C}'$. Moreover, set $\mathcal{C}'$, together with the *extends* relation, is required to form a tree rooted at `Object`.

Given a class τ, we use the syntax $\tau.f : \tau'$ to denote field f of class τ, of type τ'. Each method is denoted by an expression of the form $\tau_0.m(\tau_1, \dots, \tau_n)\colon \bar{\tau}$, where $\tau_0 \in \mathcal{C}'$ is the class which method m belongs to, $\tau_1, \dots, \tau_n \in \mathcal{T}$ are the argument types, and $\bar{\tau} \in \mathcal{T}$ is the type of the return value.

The result of the compilation of a Java program is a set of *class files*. A class file is generated by the Java compiler for each class defined in the program, and it is made up of the declaration of the class and by the JVM Language (JVML) bytecode for each class method. Each method $\mu = \tau_0.m(\tau_1, \dots, \tau_n)\colon \bar{\tau}$ is compiled into a (finite) sequence of bytecode instructions. Figure 1 shows the bytecode instructions available. In the figure, $\mathcal{B}' = \mathcal{B} \cup \{\texttt{Object}\}$.

Let $\mathbb{I}$ be the set of bytecode instructions defined in Figure 1, $\mathbb{L} = \{1, 2, \dots\}$ be the set of instruction addresses, and $\mathbb{L}_\mu = [1, l_\mu]$ the set of instruction addresses of method μ, where l_μ is the size of the method. We use $B_\mu : \mathbb{L}_\mu \to \mathbb{I}$ to map each method address to the corresponding bytecode instruction. We assume that $B_\mu(1)$ is the entry point of method μ. Moreover, we assume that, if $B_\mu(h) = \tau\texttt{return}$ for some $h \in \mathbb{L}_\mu$, then $\tau = \bar{\tau}$ (the type returned by method μ). To

$\tau op\colon \tau'$	$(\tau, \tau' \in \mathcal{B})$ Takes two operands of type τ from the stack, performs operation *op* and pushes the result (of type τ') onto the stack.
τ`const` d	$(\tau \in \mathcal{B})$ Pushes a constant d of type τ onto the stack.
τ`load` x	$(\tau \in \mathcal{B}')$ Pushes the value of type τ from register x to the stack.
τ`store` x	$(\tau \in \mathcal{B}')$ Takes a value of type τ from the stack and stores it into register x.
`if`*cond* k	$(k \in \mathbb{L})$ Takes a value of type `i` from the stack, and jumps to k if the value satisfies *cond*
`goto` k	$(k \in \mathbb{L})$ Jumps to k.
`getfield` $\tau.f\colon \tau'$	$(\tau \in \mathcal{C}', \tau' \in \mathcal{T})$ Takes an object reference of class τ from the stack; fetches field f (of type τ') of the object and loads the field on top of the stack.
`putfield` $\tau.f\colon \tau'$	$(\tau \in \mathcal{C}', \tau' \in \mathcal{T})$ Takes a value of type τ' and an object reference of class τ from the stack; saves the value in the field f of the object.
`invoke` $\tau_0.m(\tau_1, \ldots, \tau_n)\colon \tau$	$(\tau_0 \in \mathcal{C}', \tau_1, \ldots \tau_n, \tau \in \mathcal{T})$ Takes the values $a_1, \ldots, a_n$ (of types $\tau_1, \ldots, \tau_n$) and an object reference of class τ_0 from the stack. Invokes method $\tau_0.m$ of the object with actual parameters $a_1, \ldots, a_n$; places the method return value (of type τ) on top of the stack.
τ`return`	$(\tau \in \mathcal{T})$ Takes the value of type τ from the stack and terminates the method.
	(1)
`newarray` τ	$(\tau \in \mathcal{T})$ Takes an integer i from the stack and creates an instance of an array of class τ and adds a reference to the instance on top of the stack. The created array has a number of elements equal to i.
τ`aload`	$(\tau \in \mathcal{B}')$ Takes a reference to an array and an integer index from the stack. The array reference is of type $[\tau$. Loads on the stack the value, of type τ, stored at the index position in the referenced array.
τ`astore`	$(\tau \in \mathcal{B}')$ Takes an array reference, an integer index and a value from the stack. The array reference is of type $[\tau$, the value of type τ. The value is saved in the referenced array at the index position.
	(2)
`new` τ	$(\tau \in \mathcal{C}')$ Creates an instance of class τ and adds a reference to the created instance on top of the stack.
`init` τ	$(\tau \in \mathcal{C}')$ takes a reference to an uninitialized object of class τ from the stack. Initializes the object by invoking the appropriate constructor.
	(3)
`jsr` k	$(k \in \mathbb{L})$ Places the address of the successor on the stack and jumps to k.
`ret` x	Jumps to the address specified in register x.
	(4)

Fig. 1. Instruction set $\mathbb{I}$.

simplify the exposition, we study JVML in steps, where each subsequent step considers a richer subset of the instructions in $\mathbb{I}$. We will denote by $\mathbb{I}_i$ the subset of $\mathbb{I}$ containing all instructions in Figure 1, from the top of the figure to the line labeled with (i).

The JVM interprets the bytecode instructions of a method using a *context*, made up of a fixed set of r_μ registers and an operand stack, whose maximum height, t_μ, is also fixed. Bytecode instructions are typed: for example, `iload` x (where `i` is an abbreviation for `int`) assumes that register x contains an integer

value and pushes this value into the stack, while `astore` x (where `a` stands for `Object`) assumes that the top of the stack contains a reference (which may point to any type in $\mathcal{C}'$), pops it off the stack, and stores it into register x.

Whenever a class file is loaded by the JVM, it is first examined by the bytecode verifier, whose main purpose is to check the type correctness of the class. The bytecode verifier performs a dataflow analysis of the code by abstractly executing the instructions over types instead of over actual values. A Java bytecode verification algorithm is presented in Lindholm and Yellin [1999]: almost all existing bytecode verifiers implement this algorithm. An overview can be found in Leroy [2002]. The verification process is performed method per method: when verifying a method, the other methods are assumed to be correct. The algorithm uses a lattice of types, containing a minimum type $\bot$ and a maximum type $\top$ representing an erroneous type. The order relation in the lattice models the "assignable to" relation in JVML. For example, basic types are unrelated, while a subclass is less than its superclass. The data structures used by the verifier can be modeled using a *context vector*, which maps each instruction onto a context. Given a context vector v, for each instruction address h, the context $v(h)$ models the abstract state (containing types instead of actual values) of the JVM whenever an instruction at address h is about to be executed (before state). Using the lattice of types, contexts and context vectors are pointwise ordered. When an instruction at address h is executed by the verifier (in state $v(h)$), three actions are performed: a) a check is made on whether the context contains the expected type for the instruction: for example, if the instruction is τ`load` x, then the contents of register x must be τ (or a subtype of τ); if this does not occur, the verifier stops and signals an error; b) if the check succedes, the instruction at h is executed and thus produces the *after state* of h: for example, in the case of τ`load` x, the after state is the context in which τ is pushed onto the stack; c) the after-state of h is merged with the state $v(k)$ of each successor instruction at address k, producing a new state for $v(k)$. Merging is necessary since, due to the conditional and unconditional jumps, exceptions and calls (returns) to (from) subroutines, there are instructions corresponding to a join of different paths of the control flow graph. Merging two states consists in merging (by a least upper bound operation on the type domain) the types of each memory register and stack element. Merging unrelated types gives $\top$ as the result. Only stacks with the same length can be merged. Whenever two stacks of different length have to be merged, the bytecode verifier rejects the code. This behavior simplifies the implementation of the verifier and rejects programs that may cause a stack overflow/underflow during execution (e.g., due to loops where the stack grows indefinitely).

Given a method $\tau_0.m(\tau_1, \ldots, \tau_n)\colon \bar{\tau}$, the initial context vector assigns $\bot$ to the stack and the registers in each context, except for context $v(1)$ (the before context of the entry point), where register 0 is assigned τ_0, register i, $i \in [1, n]$, are assigned the type τ_i of the ith actual parameter, the remaining registers are assigned $\top$ and the stack is empty. Starting from the execution of the first instruction, the verifier continues to iteratively execute the program instructions until an error is encountered or a fixpoint is reached. In the second case, the bytecode is certified and accepted for execution.

2.2 Multipass Verification

The goal of this work is to propose and evaluate an alternative approach to bytecode verification, which checks the correctness of the bytecode by means of a progressive analysis (*multipass verification*) which requires much less memory than the standard verification. The main idea consists of decomposing bytecode verification into multiple phases (or passes) that can be executed separately. Each phase analyzes the propagation and usage of a subset of types (and in particular of a single type), independently from the other types, and deals with a subset of errors, thereby allowing a more space-efficient handling of the verification.

Modeling bytecode verification. First we give a formal semantics of the bytecode verifier, based on a least fixpoint calculation on suitable data domains. In order to introduce a uniform formalism to define and compare standard verification and multipass verification, we adopt a verification algorithm for standard verifier which differs slightly from the one usually defined. We assign a type to each typing error, in such a way that errors with type τ are those that are found when τ is required and a type different from τ is encountered. Moreover, the verification algorithm does not stop on the first error encountered, but continues execution until the fixpoint is reached, collecting all errors found and maintaining their type. This is achieved by splitting the semantic function into one part that models the execution (next function) and another part that checks typing errors (error function).

Example 2.1. Figure 2(a) shows an example of bytecode verification according to our formalization. It shows $B(h)$, $1 \leq h \leq 2$, for a simple method $\mu =$ `C.m(i): f` of a class `C`. Note that $B(0)$ is a pseudo-instruction that models the initialization of context $v(1)$. Context vectors are represented as tables, where each line h is the before context of the instruction at address h. Each context shows the contents of the registers and of the stack. Registers and stack are visually separated by a double line. Stacks are represented with fixed size (t_μ), whose top is on the left and whose unused elements are equal to $\top$. The iteration starts with a context vector $v^{(0)}$ initialized with all values equal to $\bot$. At the first step all instructions are executed on $v^{(0)}$ to produce $v^{(1)}$. The only instruction modifying the context vector is `start C.m(i): f`, since the effect of any other instruction with an empty before context (composed of all $\bot$) is null. Instruction `start C.m(i): f` affects only context $v^{(1)}(1)$: it stores `C` in register 0 and `i` in register 1, and pushes $\top$ onto the stack (modeling the empty stack). At the second iteration the instructions are evaluated on context vector $v^{(1)}$. At this point an error with type `f` is found: instruction `fload` requires an `f` type in register 1, but the register contains `i`. The type `f` of the error is recorded and execution continues, producing $v^{(2)}$ from $v^{(1)}$. In this iteration `fload 1` produces context $v^{(2)}(2)$ from its before context $v^{(1)}(1)$ by propagating the values in the registers and copying register onto the stack. The other instructions do not modify the context vector. Finally, the three instructions are once more executed on context $v^{(2)}$. Now, when `ireturn` is executed, it is verified that `i` is on top of the stack. Note that the effect of the last instruction is not included in the context vector, since it is not before context of any instruction. The context

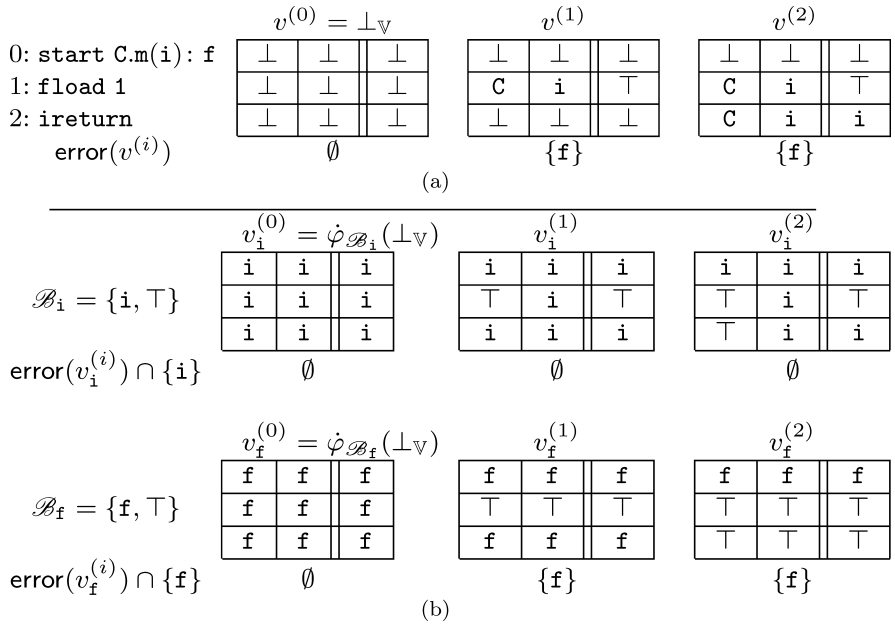

Fig. 2. Examples of verification: (a) standard verification, (b) multipass verification (analyses for integer and float errors).

vector is not modified by the instructions and hence $v^{(2)}$ is the least fixpoint of the iteration process. The code is not correct, since an `f` error has been found.

The formal semantics of the bytecode verifier is given in Section 3. In particular Section 3.5 shows our modeling of the verifier.

Parallel decomposition. In parallel decomposition, the verification is split into subanalyses (passes) each of which checks only errors in a subset of all possible types.

We use abstract interpretation to model the decomposition: the semantics of the verifier is taken as concrete semantics, and each subanalysis is modeled as an abstract interpretation of it. We show that each subanalysis is univocally determined by choosing a subset of the domain of types. The intended meaning is that the abstract interpretation should analyze the propagation and use of those types only. With a proper choice of a family of subsets of types, the results of the corresponding subanalyses can be put together to recover the result of the original analysis.

To prove the correctness and completeness of the Parallel decomposition we use the complete shell theory [Giacobazzi et al. 2000]. We show that the minimum set of types that have to be considered in each subanalysis coincides with the *complete shell* of the errors checked by that subanalysis. Thus, the complete shell theory is taken as the basis to constructively build the abstract type domain to be considered at each pass and ensures that considering only these types is sound and complete. In order to make this theory practically applicable, we characterize the semantic function next, which is a function from

context vectors to context vectors, by means of a set of simple one-argument functions from types to types, each one associated with a pair of context items (Section 4.1.2).

Example 2.2. In Figure 2(b), the analysis is split into two subanalyses. The first analysis considers only integer errors, that is, errors found when an integer is required and a noninteger type is encountered. All other errors are ignored. To perform this, complete shell calculations show that we can use the abstract interpretation determined by the set of types $\mathscr{B}_{\mathtt{i}} = \{\mathtt{i}, \top\}$. In the abstraction, type `i` represents any type assignable to `i`, while type $\top$ represents all types. Function $\varphi_{\mathscr{B}_{\mathtt{i}}}$ maps a type to its most precise representative in $\mathscr{B}_{\mathtt{i}}$. We place a "dot" above the function to denote its pointwise extension to context vectors. The first line in Figure 2(b) shows the iterations performed by the multipass verifier for this subanalysis. The subanalysis uses the same algorithm as the original analysis of Example 2.1, using the abstract domain $\mathscr{B}_{\mathtt{i}}$ instead of the domain of all types. Therefore, context vector $v^{(0)}$ is abstracted in context vector $v_{\mathtt{i}}^{(0)}$ where every $\bot$ is replaced by `i`. The effect of each instruction in B is interpreted over the $\mathscr{B}_{\mathtt{i}}$ domain. This corresponds to computing $\dot{\varphi}_{\mathscr{B}_{\mathtt{i}}} \circ \mathsf{next}$. No error is found, since $B_\mu(2)$ is type-correct and the execution of $B_\mu(1)$ ignores the error (`i` instead of `f` in register 1). We then apply the multipass verification to analyze errors of type `f` only. Again, we can calculate a new abstract domain for this purpose, and obtain $\mathscr{B}_{\mathtt{f}} = \{\mathtt{f}, \top\}$. The second line in Figure 2(b) show the resulting analysis. In this case, when executing $B_\mu(1)$, an `f` type is expected in register 1, which holds $\top$ instead. Hence, an `f` error is collected. Note that when executing $B_\mu(2)$, the top of the stack is $\top$, instead of the type `i` expected by the instruction. This is not a real error, since it is due to abstraction of type `i` onto $\top$. This error is ignored during this subanalysis. The execution of the two subanalyses, each of which corresponds to a single type, obtains exactly the same result as performing the standard verification.

Since we are interested in saving space, we want the set of types checked by the subanalyses to be as small as possible. The best results can be obtained when we only need to look at one type at a time, the so-called decomposition in basic domains. When this is possible (as in the above example), each context item can be implemented using a single bit, while standard implementations of bytecode verification usually require 24 bits for each context item [Leroy 2001]. A discussion of space requirements will be given in Section 6.

Unfortunately, due to the nature of some bytecode instructions, a decomposition in basic domains is not always possible. There are cases in which taking into account only one type achieves correctness, but not completeness. For example, let us consider instructions that manipulate arrays. We will denote the type "array of elements of type τ" as $[\tau$. Figure 3 shows a simple example of verification of a method $\mu =$ `C.m1([C): C` whose argument is an array of references and contains an `aaload` instruction. The first line in the figure shows the fixpoint of the standard verification (using the whole lattice of types), certifying that the method is correct. Let us now consider the parallel decomposition of the verification that analyses one error at a time. Consider the pass checking errors for type `C`. The complete shell for an error of type `C` is $\{\mathtt{C}, [\mathtt{C}, \mathtt{null}, \top\}$.

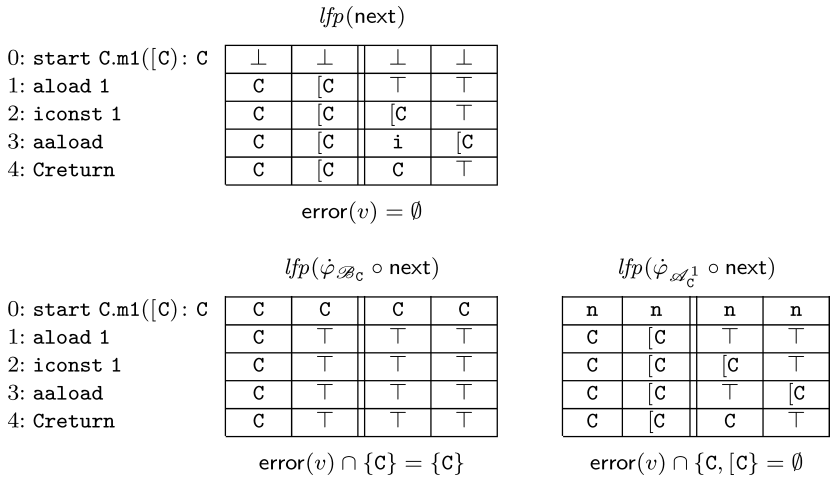

Fig. 3. Three analyses of the same method: using $\mathbb{T}$, only the basic domain $\mathscr{B}_{\mathtt{C}}$, and the domain $\mathscr{A}^1_{\mathtt{C}} = \{\top, \mathtt{C}, [\mathtt{C}, \mathtt{n}\}$ where n is shorthand for null.

Note that, in this case, the shell is not a basic domain, since other types besides C belong to the domain, and in particular [C and null (a type that models null references). This happens whenever some instruction in the method may create a type τ from an unrelated type σ, and τ is not specified in the opcode of the instruction. For array types, this is the case for the aaload instruction, that creates C from [C, but C does not appear in the aaload opcode.

Example 2.3. To see that a basic domain would cause the verification to be imprecise, let us consider domain {C, ⊤}, with both i and [C mapped onto ⊤ (see the left-hand side of the second line in Figure 3). Instructions $B_\mu(1)$ and $B_\mu(2)$ load ⊤ onto the stack (instead of [C and i, respectively). Instruction $B_\mu(3)$ leaves ⊤ on the stack. When $B_\mu(4)$ is executed, type C is expected on top of the stack, since the type of the returned value is C. Hence an error of type C is signaled, which is not ignored since we are in the pass that checks C. However, we know that the method is correct, so the analysis is imprecise. The problem is that, having abstracted [C onto ⊤, the relation between C and [C is lost and C can no longer be recovered by instructions such as aaload. The right hand side in the second line in Figure 3 shows the verification that uses {C, [C, null, ⊤}, which correctly certifies the method.

In this example, the implementation of each context item requires at least two bits to encode the four different types in {C, [C, null, ⊤}. In general, the number of required bits may be higher (see Section 4.6), but we found that it is statistically much less than the number of bits required by standard verification (Section 6.1).

The basic concepts of abstract interpretation and complete shell theory are recalled in Section 2.3. Parallel decomposition is defined in Section 4. In particular, the problem with array types is addressed in Section 4.2.1. In Section 4.3 we show that the same problem arises for object initialization.

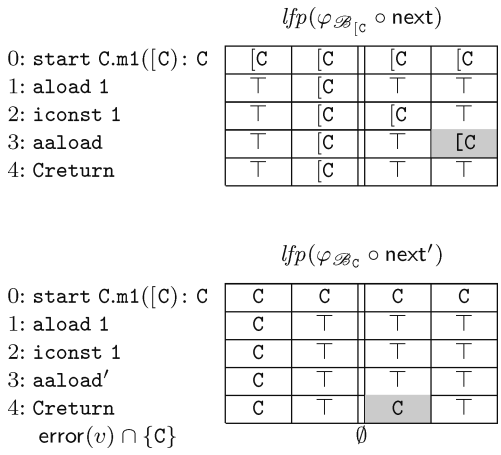

Fig. 4. Serial decomposition for method C.m1.

Serial decomposition. Taking more than one type in a pass of the verification increases the memory space needed by the verifier at each pass. To overcome this problem, we propose another kind of decomposition, denoted as *serial decomposition*, which allows us to maintain the "one type at a time" analysis. In serial decomposition some passes of the verification give some information to the next passes. This information is exploited to perform a complete analysis using a basic domain even when the complete shell is larger. There is only a small penalty in memory space due to the saving of temporary results. Consider a bytecode with an instruction having as operand a type σ and producing a type τ unrelated to σ (thus, the complete shell for τ errors is not a basic domain). We perform first an enriched pass for $\{\sigma, \top\}$ which, besides checking errors for σ, also records the elements of the fixpoint context vector where τ would occur. Using this information, the next pass for τ is performed on a transformed bytecode. Every instruction h that takes σ and produces τ is transformed into an instruction that produces the constant result τ irrespectively of the type of operands.

Example 2.4. Let us reconsider the method presented in Figure 3. In the serial decomposition approach, first the multipass analysis obtained with the basic domain $\{[\mathtt{C}, \top\}$ is performed (see the first line of Figure 4). Besides checking type [C, this pass releases also the information that the aaload instruction is applied to a stack whose second element is [C. In the pass of the analysis which uses the basic domain $\{\mathtt{C}, \top\}$, the aaload instruction is replaced by a fictitious instruction aaload′ which takes two elements out of the stack and pushes C onto the stack. In this way this occurrence of C type is also visible in the pass for C, the analysis does not raise any error and correctly accepts the method.

In the previous example, only one additional bit is required. This bit records whether the aaload instruction is applied to a [C type. The space requirements for serial decomposition are discussed in Section 6.

Symbol	Description	Section
$\mathcal{B}$, $\mathcal{B}'$	The set of base types of JVML ($\mathcal{B}'$ includes `Object`)	2.1
$\mathcal{C}$, $\mathcal{C}'$	User defined classes ($\mathcal{C}'$ includes `Object`)	2.1
$\mathcal{A}$	Array types in JVML	4.2
$\mathcal{T}$	The set of types of JVML, i.e., $\mathcal{B} \cup \mathcal{C}' \cup \mathcal{A}$	2.1
μ	The method being verified	2.1
l_μ, r_μ, t_μ	Size, number of registers and maximum stack size of method μ	2.1
B_μ	Map from labels to bytecode instructions of method μ	2.1
$\mathbb{L}_\mu$ / $\mathbb{L}_{\mu,0}$ / $\mathbb{L}'_{\mu,0}$	Set of labels of method μ/with additional label 0 for `start` instruction/with additional labels 0 and -1 (exit from method)	2.1
$\mathbb{M}_\mu, \mathbb{S}_\mu, \mathbb{C}_\mu, \mathbb{V}_\mu$	Domains for memories, stacks, contexts and context vectors of method μ	3.2
$\mathbb{I}$ / $\mathbb{I}_i$	All bytecode instructions/bytecode instructions up to line i (defined in Figure 1)	2.1
$\mathbb{T}_i$	Types used during verification of methods that use instructions in $\mathbb{I}_i$ only	3.1
$\mathbb{E}_i$	Type errors that may be generated by methods that use instructions in $\mathbb{I}_i$ only	3.5
$\mathbb{U}^\tau_\mu$ / $\mathbb{U}^{\sqsubseteq\tau}_\mu$ / $\mathbb{U}_\mu$	Set of uninitialized versions of type τ/types assignable to τ/any type that may be used in method μ	4.3
φ_M	The closure operator corresponding to Moore family M	2.3
$\mathscr{M}(A)$	Moore operator returning the Moore closure of set A	2.3
$\mathscr{S}^B_f(A)$	Relative complete shell of abstraction A wrt function f and abstraction B (f may also be a set of functions)	2.3.1
$\mathscr{S}_f(A)$	Absolute complete shell of abstraction A wrt function f (f may also be a set of functions)	2.3.1
$h \rightsquigarrow k$	Instruction at label k follows instruction at label h in the Control Flow Graph	3.3
$i \twoheadrightarrow j$	Context vector element at index i may affect element at index j	4.1.2
$\mathscr{F}_\mu$	Set of functions between context vector elements used in method μ	4.1.2
$\mathscr{B}_C$	Basic domain, containing only c and $\top$	4
$\mathscr{A}^k_\tau$	The lattice of arrays of τ up to k dimensions, depicted in Figure 15	4.2.1
$\kappa_{\mu,\tau}$	The maximum dimension of all arrays of type assignable to τ in method μ	4.2.1
$\mathscr{T}_\mu$	The set of types that are actually used in the standard verification of method μ	4.2.1
$\mathscr{E}_\mu$	The set of type errors that must be checked for method μ	4.6
$\mathscr{U}_{\mu,\tau}$ / $\mathscr{U}_{\mu,\sqsubseteq\tau}$	The least lattice containing all uninitialized versions of τ/all uninitialized versions of all types $\sigma \sqsubseteq \tau$ that may be created in method μ	4.3.1

Fig. 5. Table of symbols.

In Section 5 we define Serial Decomposition. In Section 5.1 we apply Serial Decomposition to array types. Section 5.2 shows how to obtain decomposition in basic domains in the presence of object initialization.

The reader may refer to Figure 5, where we have collected the main symbols used throughout the paper, together with a short description and the section number where they are first introduced.

2.3 Background on Abstract Interpretation

Abstract interpretation [Cousot and Cousot 1977] is a framework that allows an approximate evaluation of the semantics of a discrete dynamic system. The semantics is modeled as a couple (C, f) of a complete lattice $\langle C; \leq \rangle$, called the concrete domain, and a semantic function f, where f maps each element of the system (e.g., each program of a programming language) to an element in C. Abstraction is introduced by defining a new complete lattice $\langle A; \sqsubseteq \rangle$, called the abstract domain. The relationship between the concrete and abstract domain must be defined precisely by means of two functions $\alpha : C \to A$, the abstraction function, and $\gamma : A \to C$, the concretization function, such that (C, α, γ, A) is a Galois connection (i.e., $\forall c \in C, a \in A, \alpha(c) \sqsubseteq a \Leftrightarrow c \leq \gamma(a)$). Then, the theory shows how to design an abstract semantic function $f^\sharp$ that correctly approximates function f in the abstract domain.

In this article we use an alternative, but equivalent, characterization of abstractions, given by upper closure operators [Cousot and Cousot 1979]. An (upper) closure operator ρ on a poset $\langle C; \leq \rangle$ is a monotone, extensive (i.e., $\forall c \in C, c \leq \rho(c)$) and idempotent (i.e., $\rho \circ \rho = \rho$) operator on C. If (C, α, γ, A) is a Galois connection, then $\gamma \circ \alpha$ is an upper closure operator on C. Vice versa, if ρ is an upper closure operator on C and ι is any isomorphism between $\rho(C)$ and A, then $(C, \iota \circ \rho, \iota^{-1}, A)$ is a Galois connection. Thus, an upper closure operator ρ can be used, instead of functions α and γ, to define an abstraction on C (leaving ι implicit or letting $\iota = \lambda c.c$). The abstract domain A, in these settings, is simply the image of C through ρ, denoted by $A = \rho(C)$. Thus, A is a subset of C. The idea is to abstract each element of C by replacing it with a representative, taken from C itself.

Note that $\rho(C)$ coincides with the set of fixpoints of ρ. Moreover, any closure operator is uniquely determined by its set of fixpoints [Davey and Priestley 2002]. If $\langle C; \leq \rangle$ is a complete lattice, then $\rho(C)$ is a *Moore family*, that is, a poset closed under arbitrary meets ($\bigwedge$). Likewise, every Moore family $M \subseteq C$ uniquely determines a closure operator φ_M such that $\varphi_M(C) = M$. An expression for φ_M is given by

$$\varphi_M(x) = \bigwedge \{ y \in M \mid x \leq y \}. \tag{1}$$

All this means that any abstraction of C can be intuitively represented, and defined, by an appropriately chosen subset (i.e., a subset that is a Moore family) of C. For this reason, we denote with $abs(C)$ (abstractions of C) the set of all Moore families contained in C.

Any Moore family is a complete lattice (and vice versa). In particular, if $\langle C; \leq \rangle$ is a complete lattice with meet $\wedge$, join $\vee$, top element $\top$ and bottom element $\bot$, then $\rho(C)$ is a complete lattice with meet $\wedge$, join $\lambda x, y.\rho(x \vee y)$, top element $\top$ and bottom element $\rho(\bot)$ [Ward 1942]. In particular, note that the top element of C belongs to all Moore families of C.

Moore families of a complete lattice $\langle C; \leq \rangle$ can be characterized as the fixpoints of the *Moore operator* $\circ\!\!\mathscr{M}$ on $\wp(C)$, defined as

$$\circ\!\!\mathscr{M}(A) \triangleq \left\{ \bigwedge X \mid X \subseteq A \right\}. \tag{2}$$

Since the Moore operator is itself an upper closure operator on the complete lattice $\langle \wp(C); \subseteq \rangle$, the set of its fixpoints $\circ\mathscr{M}(\wp(C)) = abs(C)$ (i.e., the set of all Moore families of C) is a complete lattice, with meet given by set intersection, join given by $\lambda A, B.\circ\mathscr{M}(A \cup B)$, top element C and bottom element $\circ\mathscr{M}(\emptyset) = \{\top\}$ (taking $\bigwedge \emptyset = \top$).

The set of all upper closure operators on a complete lattice $\langle C; \leq \rangle$, denoted by $uco(C)$, is also a complete lattice, when closure operators are ordered pointwise. In particular, the meet operation in $uco(C)$ is simply the pointwise extension of the meet operation in C. Since each closure operator on C determines a unique Moore family of C and vice versa, there is a bijection between $uco(C)$ and $abs(C)$. This bijection is also an order isomorphism between $\langle uco(C); \leq \rangle$ and $\langle abs(C); \supseteq \rangle$, so that $\langle uco(C); \leq \rangle$ and the order dual of $\langle abs(C); \subseteq \rangle$ are isomorphic lattices.

2.3.1 *Soundness and Completeness.* Let us consider two complete lattices $\langle C; \leq \rangle$ and $\langle D; \sqsubseteq \rangle$, together with a semantic function $f : C \to D$. We want to abstractly interpret function f, when working with abstractions of C and D. Let us suppose that the abstractions of C and D are given in the form of a pair $\langle \rho, \eta \rangle$ of closure operators, where $\rho : C \to C$ and $\eta : D \to D$. Then, a function $f^\sharp : \rho(C) \to \eta(D)$ is a *correct* (or *sound*) abstraction of f with respect to $\langle \rho, \eta \rangle$, iff $\eta \circ f \sqsubseteq f^\sharp \circ \rho$. Among all correct abstractions of f wrt $\langle \rho, \eta \rangle$ there is a function f^b which is the *best abstraction*, in the sense that all correct abstractions $f^\sharp$ are such that $f^b \sqsubseteq f^\sharp$ (in the pointwise ordering). It is well known from abstract interpretation theory that $f^b = \eta \circ f|_{\rho(C)}$ [Cousot and Cousot 1979].

Completeness [Mycroft 1993] is the natural strengthening of correctness: $f^\sharp$ is a *complete abstraction* of f (is complete for f) wrt $\langle \rho, \eta \rangle$, iff

$$\eta \circ f = f^\sharp \circ \rho. \tag{3}$$

When $C = D$, we can define abstractions using a single closure operator. Moreover, we can calculate $lfp(f)$ and $lfp(f^\sharp)$. We say that $f^\sharp$ is *fixpoint complete* for f wrt (closure operator) ρ iff

$$\rho(lfp(f)) = lfp(f^\sharp). \tag{4}$$

It is well known that completeness implies fixpoint completeness: if $f^\sharp$ is complete for f wrt $\langle \rho, \rho \rangle$, then $f^\sharp$ is fixpoint complete for f wrt ρ (the converse, in general, is not true).

Giacobazzi et al. [2000] have observed that completeness and fixpoint completeness for f are properties of the closure operators (Moore families) that define the abstractions on C and D (ρ and η), rather than properties of the functions ($f^\sharp$) that interpret f. In fact, a function $f^\sharp$, complete for f wrt $\langle \rho, \eta \rangle$, can be found if and only if the *best* abstraction of f in $\langle \rho, \eta \rangle$ is itself complete for f wrt $\langle \rho, \eta \rangle$. Then, the observation follows from the fact that the best abstraction f^b of f in $\langle \rho, \eta \rangle$ is solely defined in terms of f, ρ and η. Thus, it is meaningful to talk about the completeness of $\langle \rho, \eta \rangle$ wrt f, and of fixpoint completeness of ρ wrt f. These can be checked directly, replacing $f^\sharp$ with the definition of f^b in (3) and (4).

In the same article, Giacobazzi et al. give a constructive characterization of what they call the *relative complete shell* of ρ wrt η and f. The meaning of this

construction is as follows: we imagine to keep η fixed, and we search for the "greatest" (i.e., the most abstract) abstraction ρ', which is "more precise" (i.e., less abstract) than ρ, such that $\langle \rho', \eta \rangle$ is complete wrt f. Here, "more precise" means that $\rho(C) \subseteq \rho'(C)$, and ρ' is the "greatest" abstraction when its codomain is the least in the inclusion order on $abs(C)$. If such an abstraction ρ' can be found, then ρ' is the relative complete shell of ρ wrt η and f. They show that, if f is a continuous function from C to D, then the codomain $\rho'(C)$ (and, thus, operator ρ') of the relative complete shell ρ' of ρ wrt η and f can be obtained as $\rho'(C) = \mathscr{S}_f^{\eta(C)}(\rho(C))$, where $\mathscr{S}_f^B : abs(C) \to abs(C)$, with $B \in abs(D)$, is defined as

$$\mathscr{S}_f^B(A) = \mathscr{M}(A \cup \mathscr{R}_f(B)), \tag{5}$$

where $\mathscr{R}_f : abs(D) \to \wp(C)$ is

$$\mathscr{R}_f(B) = \bigcup_{b \in B} \max\{c \in C \mid f(c) \sqsubseteq b\}. \tag{6}$$

They also show that, if δ is any closure operator on C such that $\mathscr{S}_f^{\eta(C)}(\rho(C)) \subseteq \delta(C)$, then $\langle \delta, \eta \rangle$ is also complete wrt f.

Example 2.5. Consider sets $C = \wp(\mathbb{Z})$, $D = \wp(\mathbb{R})$, lattices $\langle C; \subseteq \rangle$, $\langle D; \subseteq \rangle$, and the semantic function $f : C \to D$ such that $f(X) = \{2^n \mid n \in X\}$ (i.e., the exponential function extended to set of integers). If we have

$$\rho(X) = \begin{cases} \emptyset & \text{if } X = \emptyset, \\ \{n > 0\} & \text{if } X \neq \emptyset \text{ and } X \subseteq \{n > 0\}, \\ \{n < 0\} & \text{if } X \neq \emptyset \text{ and } X \subseteq \{n < 0\}, \\ \mathbb{Z} & \text{otherwise,} \end{cases} \quad \eta(Y) = \begin{cases} \{1\} & \text{if } Y \subseteq \{1\}, \\ \mathbb{R} & \text{otherwise,} \end{cases}$$

as closure operators on C and D, respectively, then the relative complete shell of ρ wrt η and f can be calculated as follows. From Equation (6) we have

$$\begin{aligned} \mathscr{R}_f(\eta(D)) = &\max\{X \in C \mid f(X) \subseteq \mathbb{R}\} \cup \\ &\max\{X \in C \mid f(X) \subseteq \{1\}\} = \{\mathbb{Z}, \{0\}\}, \end{aligned}$$

then,

$$\mathscr{S}_f^{\eta(D)}(\rho(C)) = \{\emptyset, \{n < 0\}, \{0\}, \{n > 0\}, \mathbb{Z}\}.$$

The lattices induced by ρ, η, and $\mathscr{S}_f^{\eta(D)}(\rho(C))$ are depicted in Figure 6.

In the cases where $C = D$, Giacobazzi et al. also define the *absolute complete shell* of ρ wrt f, as the greatest abstraction ρ', more precise than ρ, such that $\langle \rho', \rho' \rangle$ is complete wrt f. If f is continuous, then we can define $\mathscr{S}_f : abs(C) \to abs(C)$ as

$$\mathscr{S}_f(A) = lfp(\lambda Q \in abs(C).\mathscr{M}(A \cup \mathscr{R}_f(Q))). \tag{7}$$

The absolute complete shell of ρ wrt f is the closure operator induced by the Moore family $\mathscr{S}_f(\rho(C))$ (as given by (1)).

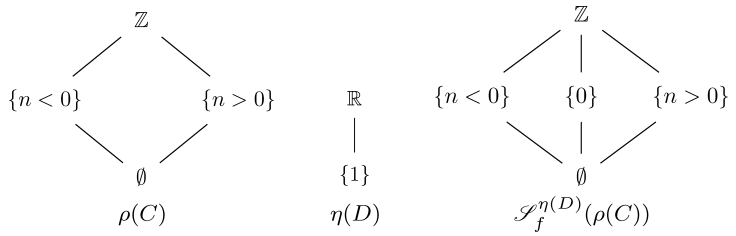

Fig. 6. An example of a relative complete shell ($C = \wp(\mathbb{Z})$ and $D = \wp(\mathbb{R})$).

All these notions are extended, in a natural way, by replacing function f with a set F of functions, where each function in F may have an arity greater than 1. In this case, $\mathscr{R}_F(B)$ must be calculated taking the union over all functions in F and, for each function, considering each argument in turn and taking the maximal by letting the argument vary, while keeping the remaining arguments fixed (and repeating this for each possible combination of values of the remaining arguments).

Other results of interest for us are that $\langle \rho, \rho \rangle$ is always complete for $\lambda x_1, \ldots, x_n.x_1 \vee \cdots \vee x_n$, and that if $\langle \rho, \rho \rangle$ is complete wrt f and g, it is also complete wrt $f \circ g$.

3. MODELING THE STANDARD BYTECODE VERIFIER

In this section we give a formalization of the informal Verification algorithm presented in Lindholm and Yellin [1999], to obtain a formal semantics suitable for abstract interpretation. Sections 3.1 to 3.3 contain some preliminary definitions. Section 3.4 contains an initial formalization of the Verification algorithm, for which we only give informal justifications, since it is sufficiently close to the specification in Lindholm and Yellin [1999]. Section 3.5 gives a second formalization, which is more suitable to our study, and shows its equivalence with the former. The main difference between the formalization given in Section 3.4 and the one given in Section 3.5 is that the former stops on the first error encountered during verification, while the latter goes on and collects all errors that may be present.

3.1 The Complete Lattice of Types

The types used during the verification of method $\mu = \tau_0.m(\tau_1, \ldots, \tau_n)\colon \tau$ are organized in a partially ordered set (poset) whose definition depends on the instruction set used. For instruction set $\mathbb{I}_1$, we can define $\langle \mathbb{T}_1; \sqsubseteq_1 \rangle$, where

$$\mathbb{T}_1 = \{\top, \bot, \texttt{null}\} \cup \mathcal{B} \cup \mathcal{C}'. \tag{8}$$

The partial order $\sqsubseteq_1$ is the smallest reflexive, antisymmetric and transitive relation containing all of the following:

$$\begin{aligned} (\forall \tau \in \mathbb{T}_1) \quad & \tau \sqsubseteq_1 \top \text{ and } \bot \sqsubseteq_1 \tau, \\ (\forall \tau \in \mathcal{C}') \quad & \texttt{null} \sqsubseteq_1 \tau, \\ (\forall \tau', \tau'' \in \mathcal{C}' \text{ such that } \tau' \textit{ extends } \tau'') \quad & \tau' \sqsubseteq_1 \tau''. \end{aligned}$$

The partial order is deduced from the Java language specification, in such a way that, if the bytecode verifier reports no error for a state containing type τ_1, it will report no error for a state where type τ_1 is replaced by type τ_2, with $\tau_2 \sqsubseteq \tau_1$. Informally, we call $\sqsubseteq$ the *assignable to* relation. The fictitious types $\top$ and $\bot$ have been added to turn $\langle \mathbb{T}_1; \sqsubseteq_1 \rangle$ into a *complete lattice*, that is, a poset where each subset of elements has a least upper bound (and, thus, also a greatest lower bound). Type $\top$ should be interpreted as an unknown type. If soundness must be preserved, the unknown type cannot safely satisfy any type constraint; thus, it can also be regarded as an erroneous type. It can be used to model the initial, undefined value of uninitialized registers. Type $\bot$, instead, should be regarded as an impossible type, that is, no actual value used by any reachable instruction may have this type. The fact that we require that all instructions should safely accept an impossible type (since $\bot$ is assignable to every other type) does not cause problems. Indeed, it will be clear that type $\bot$ may only be used by unreachable instructions (dead code). Since these instructions can never be executed, their type correctness is irrelevant.

PROPOSITION 3.1. *$\langle \mathbb{T}_1; \sqsubseteq_1 \rangle$ is a lattice, that is, each pair of elements $\tau_1, \tau_2 \in \mathbb{T}_1$ has a least upper bound (join) $\tau_1 \sqcup \tau_2$ and a greatest lower bound (meet) $\tau_1 \sqcap \tau_2$.*

PROOF SKETCH. By cases. □

In particular, if we take τ_1 and τ_2 in $\mathbb{T}_1$ with $\tau_1 \neq \tau_2$, we have $\tau_1 \sqcup_1 \tau_2 = \top$ and $\tau_1 \sqcap_1 \tau_2 = \bot$ if either τ_1, or τ_2, or both are in $\mathcal{B}$. If both τ_1 and τ_2 are in $\mathcal{C}'$ and neither $\tau_1 \sqsubset_1 \tau_2$, nor $\tau_2 \sqsubset_1 \tau_1$, then $\tau_1 \sqcap_1 \tau_2 = \texttt{null}$ and $\tau_1 \sqcup_1 \tau_2$ is the first common ancestor of τ_1 and τ_2 in the *extends* tree.

PROPOSITION 3.2. *$\langle \mathbb{T}_1; \sqsubseteq_1 \rangle$ is a complete lattice, that is, each subset D of $\mathbb{T}_1$ has a least upper bound $\bigsqcup_1 D$ (and a greatest lower bound $\bigsqcap_1 D$).*

PROOF. Since $\langle \mathbb{T}_1; \sqsubseteq_1 \rangle$ is a lattice by Prop. 3.1 and $\mathbb{T}_1$ is finite. □

In particular, we have $\bigsqcup_1 \emptyset = \bot$ and $\bigsqcap_1 \emptyset = \top$. Hereafter, we will omit the "1" subscript from $\mathbb{T}_1$, $\sqsubseteq_1$ etc., when not required.

3.2 Contexts and Context Vector

The standard verifier models the state of the JVM using registers and stacks that contain types taken from $\mathbb{T}$. For a method μ with a maximum number of registers r_μ and a maximum stack size t_μ, a memory is an element of $\mathbb{M}_\mu = [0, r_\mu) \to \mathbb{T}$ (registers contain types) and a stack is an element of $\mathbb{S}_\mu = \mathbb{T}^{[0,t_\mu]}$, the set of sequences of elements of $\mathbb{T}$, with size between 0 and t_μ. For any element $s \in \mathbb{S}_\mu$, we denote by $|s|$ the size (number of elements) of s. If $s \in \mathbb{S}_\mu$ and $|s| = n > 0$, then $s = s_0 \cdots s_{n-1}$, where $s_0, \ldots, s_{n-1}$ are the elements of s and s_0 is the top of s. If $|s| \geq i$, then $s\rfloor_i$ denotes a sequence of size $|s| - i$, which is equal to s with the i topmost elements removed (the i subscript will be omitted when $i = 1$). We use ϵ to denote the empty stack (where $|\epsilon| = 0$).

The partial order $\sqsubseteq$ on $\mathbb{T}$ is extended pointwise to $\mathbb{M}_\mu$ (for any $M_1, M_2 \in \mathbb{M}_\mu$, $M_1 \sqsubseteq M_2$ iff $M_1(x) \sqsubseteq M_2(x)$, $\forall x \in [0, r_\mu)$). Since $\langle \mathbb{T}; \sqsubseteq \rangle$ is a complete lattice (Prop. 3.2), so is $\langle \mathbb{M}_\mu; \sqsubseteq \rangle$. Ordering between any two elements s_1 and s_2 of $\mathbb{S}_\mu$ is

defined (by pointwise extension of $\sqsubseteq$) only when $|s_1| = |s_2|$, so that $\langle \mathbb{S}_\mu; \sqsubseteq \rangle$ is not a lattice (since there is no join, nor meet, of s_1 and s_2 when they have different size).

A *context* is an element of $\mathbb{C}_\mu = (\mathbb{M}_\mu \times \mathbb{S}_\mu) \cup \{\Omega\}$, where Ω denotes a context containing an error. By pointwise ordering $\mathbb{C}_\mu$, using the partial orders defined on $\mathbb{M}_\mu$ and $\mathbb{S}_\mu$, and letting $c \sqsubseteq \Omega$, $\forall c \in \mathbb{C}_\mu$, we have that join is defined for any non-empty subset of $\mathbb{C}_\mu$. In particular, $(M_1, s_1) \sqcup_\mu (M_2, s_2) = \Omega$ whenever $|s_1| \neq |s_2|$. Finally, if we add a fictitious bottom element $\bot_{\mathbb{S}}$ to $\mathbb{S}_\mu$, $\langle \mathbb{C}_\mu; \sqsubseteq \rangle$ becomes a complete lattice, whose bottom is given by $(\lambda x \in [0, r_\mu).\bot, \bot_{\mathbb{S}})$.

A *context vector* maps each instruction address to a context. The idea is that the context associated with address h models the state of the JVM whenever instruction at address h is about to be executed (before state). For the sake of uniformity, we also need an additional context, that models the state before the beginning of the execution of the method. We map this context to address 0. Thus, a context vector is an element of (omitting the μ subscript) $\mathbb{V} = \mathbb{L}_0 \to \mathbb{C}$, where $\mathbb{L}_0 = \mathbb{L} \cup \{0\}$. The partial order on $\mathbb{C}$ is extended pointwise to $\mathbb{V}$, turning $\langle \mathbb{V}; \sqsubseteq \rangle$ into a complete lattice.

3.3 Control Flow Graph

Given a method $\mu = \tau_0.m(\tau_1, \ldots, \tau_n)\colon \bar{\tau}$, with bytecode instructions B, we define the *control flow graph* of μ as follows. First, we extend B to address 0, by letting $B(0) = \texttt{start}\ \mu$, and add a special address, -1, to model method termination: $\mathbb{L}'_0 = \mathbb{L}_0 \cup \{-1\}$. Then, the control flow graph of μ is graph $(\mathbb{L}'_0, \rightsquigarrow)$, with $\rightsquigarrow \subseteq \mathbb{L}'_0 \times \mathbb{L}'_0$. We write $h \rightsquigarrow k$ to denote $(h, k) \in \rightsquigarrow$. The $\rightsquigarrow$ relation is defined as follows: for all $h \in [0, l)$, $h \rightsquigarrow h + 1$ if $B(h) \neq \texttt{goto}\ k$ and $B(h) \neq \tau\texttt{return}$; $h \rightsquigarrow k$ if $B(h) = \texttt{if}cond\ k$ or $B(h) = \texttt{goto}\ k$; $h \rightsquigarrow -1$ if $B(h) = \tau\texttt{return}$.

3.4 Standard Verifier

Figure 7 shows the standard interpretations for all instructions in $\mathbb{I}_1$, together with the additional instruction $\texttt{start}\ \mu$. The semantic function $[\![\bullet]\!]\colon \mathbb{I} \to (\mathbb{C} \to \mathbb{C})$ maps each instruction $I \in \mathbb{I}$ to a function that simulates the effect of I on its context. Figure 7 uses the following notation: given a function $f\colon A \to B$, function $f[z/x]\colon A \to B$, for $x \in A$ and $z \in B$, is defined as follows:

$$f[z/x](y) = \begin{cases} z & \text{if } y = x, \\ f(y) & \text{otherwise.} \end{cases}$$

A computation step is modeled by defining function $\mathsf{std}\colon \mathbb{V} \to \mathbb{V}$ as

$$\mathsf{std}(v) = \lambda k \in \mathbb{L}_0. \bigsqcup \{[\![B(h)]\!]v(h) \mid h \rightsquigarrow k\}. \tag{9}$$

Standard verification requires calculation of the least fixpoint of function std, denoted by $lfp(\mathsf{std})$. A method μ is rejected iff $\exists h \in \mathbb{L}_0$ such that $lfp(\mathsf{std})(h) = \Omega$.

PROPOSITION 3.3. *Function* std *is continuous (and, thus, monotone).*

PROOF SKETCH. Since the least upper bound in Equation (9) is obviously monotone, monotonicity of std follows from monotonicity of $[\![I]\!]$ for all $I \in \mathbb{I}_1$. Since

$$
\begin{aligned}
&[\![\texttt{start } \tau_0.m(\tau_1,\ldots,\tau_n)\colon \tau]\!]\langle M,s\rangle && = \langle M[\tau_i/i \ (\forall i \in [0,n]); \top/i \text{ otherwise}], \epsilon\rangle \\
&[\![\tau op\colon \tau']\!]\langle M,s\rangle && = \begin{cases} \langle M, \tau' s\rfloor_2\rangle & \text{if } |s| \geq 2,\ s_0 \sqsubseteq \tau \text{ and } s_1 \sqsubseteq \tau \\ \Omega & \text{otherwise} \end{cases} \\
&[\![\tau\texttt{const } d]\!]\langle M,s\rangle && = \begin{cases} \langle M, \tau s\rangle & \text{if } |s| < t \\ \Omega & \text{otherwise} \end{cases} \\
&[\![\tau\texttt{load } x]\!]\langle M,s\rangle && = \begin{cases} \langle M, M(x)s\rangle & \text{if } |s| < t \text{ and } M(x) \sqsubseteq \tau \\ \Omega & \text{otherwise} \end{cases} \\
&[\![\tau\texttt{store } x]\!]\langle M,s\rangle && = \begin{cases} \langle M[s_0/x], s\rfloor\rangle & \text{if } |s| > 0 \text{ and } s_0 \sqsubseteq \tau \\ \Omega & \text{otherwise} \end{cases} \\
&[\![\texttt{if}cond\ k]\!]\langle M,s\rangle && = \begin{cases} \langle M, s\rfloor\rangle & \text{if } |s| > 0 \text{ and } s_0 \sqsubseteq \texttt{i} \\ \Omega & \text{otherwise} \end{cases} \\
&[\![\texttt{goto } k]\!]\langle M,s\rangle && = \langle M,s\rangle \\
&[\![\texttt{getfield } \tau'.f\colon \tau']\!]\langle M,s\rangle && = \begin{cases} \langle M, \tau s\rfloor\rangle & \text{if } |s| > 0 \text{ and } s_0 \sqsubseteq \tau \\ \Omega & \text{otherwise} \end{cases} \\
&[\![\texttt{putfield } \tau.f\colon \tau']\!]\langle M,s\rangle && = \begin{cases} \langle M, s\rfloor_2\rangle & \text{if } |s| \geq 2,\ s_0 \sqsubseteq \tau \text{ and } s_1 \sqsubseteq \tau' \\ \Omega & \text{otherwise} \end{cases} \\
&[\![\texttt{invoke } \tau_n.m(\tau_0,\ldots,\tau_{n-1})\colon \tau]\!]\langle M,s\rangle && = \begin{cases} \langle M, \tau s\rfloor_{n+1}\rangle & \text{if } |s| \geq n+1 \text{ and } s_i \sqsubseteq \tau_i \ (\forall i \in [0,n]) \\ \Omega & \text{otherwise} \end{cases} \\
&[\![\tau\texttt{return}]\!]\langle M,s\rangle && = \begin{cases} \langle M, \epsilon\rangle & \text{if } |s| = 1 \text{ and } s_0 \sqsubseteq \tau \\ \Omega & \text{otherwise} \end{cases} \\
&[\![I]\!]\Omega && = \Omega \quad (\forall I \in \mathbb{I})
\end{aligned}
$$

Fig. 7. Standard interpretation of bytecode instructions in $\mathbb{I}_1$.

every monotone function on a finite poset is continuous, and $\langle \mathbb{V}; \sqsubseteq\rangle$ is finite, then std is continuous. □

Since function std is continuous, by Prop. 3.3, its least fixpoint can be calculated as the limit of an ascending Kleene sequence:

$$lfp(\mathsf{std}) = \bigsqcup_{n\geq 0} \mathsf{std}^n(\bot_{\mathbb{V}}),$$

where, for any function f, f^0 is the identity function, $f^{i+1} = f \circ f^i$ and $\bot_{\mathbb{V}} = \lambda h \in \mathbb{L}_0.\bot_{\mathbb{C}}$.

3.5 A Modified Verifier

In this section we define a different verification algorithm that does not stop when an error is found, but continues execution collecting all errors found with the associated type.

To simplify management of stack elements, we assume that all condition on stack size are checked in an initial separate verification step. For example, a condition is that, whenever an instruction is reachable from two (or more) different instructions, the incoming stack sizes are always the same. The step checking stack constraints can be performed by a simple data flow analysis,

$$\mathsf{next}[\![\texttt{start}\ \tau_0.m(\tau_1,\ldots,\tau_n)\colon\tau]\!]\langle M,s\rangle = \langle M[\tau_i/i\ (\forall i\in[0,n]);\top/i\ \text{otherwise}],\top^t\rangle$$
$$\mathsf{next}[\![\tau op\colon\tau']\!]\langle M,\sigma_1\sigma_2 s\rangle = \langle M,\tau' s\top\rangle$$
$$\mathsf{next}[\![\tau\texttt{const}\ d]\!]\langle M,s\rangle = \langle M,\tau s\rangle$$
$$\mathsf{next}[\![\tau\texttt{load}\ x]\!]\langle M,s\rangle = \langle M,M(x)s\rangle$$
$$\mathsf{next}[\![\tau\texttt{store}\ x]\!]\langle M,\sigma s\rangle = \langle M[\sigma/x],s\top\rangle$$
$$\mathsf{next}[\![\texttt{if}\,cond\ k]\!]\langle M,\sigma s\rangle = \langle M,s\top\rangle$$
$$\mathsf{next}[\![\texttt{goto}\ k]\!]\langle M,s\rangle = \langle M,s\rangle$$
$$\mathsf{next}[\![\texttt{getfield}\ \tau.f\colon\tau']\!]\langle M,\sigma s\rangle = \langle M,\tau' s\rangle$$
$$\mathsf{next}[\![\texttt{putfield}\ \tau.f\colon\tau']\!]\langle M,\sigma_1\sigma_2 s\rangle = \langle M,s\top\top\rangle$$
$$\mathsf{next}[\![\texttt{invoke}\ \tau_n.m(\tau_0,\ldots,\tau_{n-1})\colon\tau]\!]\langle M,\sigma_0\cdots\sigma_n s\rangle = \langle M,\tau s\top^n\rangle$$
$$\mathsf{next}[\![\tau\texttt{return}]\!]\langle M,\sigma s\rangle = \langle M,s\top\rangle$$

Fig. 8. The $\mathsf{next}[\![I]\!]$ function for all instruction $I\in\mathbb{I}_1$.

mapping a stack size to each method instruction. Once this step has been performed and method μ has not been rejected, we say that μ is *stack safe* and we can assume that stacks can be simply modeled using $\mathbb{T}^t$ (t-uples of elements in $\mathbb{T}$), ordered pointwise, where t is the maximum stack size declared for method μ. We still denote stacks using the sequence of their t elements, with the stack top being the leftmost element. To manage typed errors, we first split Ω into a set for errors. The definition of the set of errors depends on the instruction set. For instruction set $\mathbb{I}_1$ we define

$$\mathbb{E}_1 = \mathcal{B}\cup\mathcal{C}', \tag{10}$$

where $\tau\in\mathbb{E}_1$ represents an error on type τ. Such an error can be generated by an instruction that requires a type assignable to τ. Now, the special context Ω is no longer needed, and we can define $\mathbb{C}'$ simply as $\mathbb{M}\times\mathbb{T}^t$. Accordingly, we let $\mathbb{V}'=\mathbb{L}_0\to\mathbb{C}'$. Finally, we remove type constraints from instruction interpretations by introducing a new semantic function $\mathsf{next}[\![\bullet]\!]\colon\mathbb{I}\to(\mathbb{C}'\to\mathbb{C}')$, defined in Figure 8 for instructions in $\mathbb{I}_1$. Type checking is delegated to a separate function $\mathsf{error}[\![\bullet]\!]\colon\mathbb{I}\to(\mathbb{C}'\to\wp(\mathbb{E}))$, defined in Figure 9. Since all type constraints can be expressed as inequalities, we have defined the helper function $\mathsf{check}\colon\mathbb{T}\times\mathbb{E}\to\wp(\mathbb{E})$ as follows:

$$\mathsf{check}(\sigma,\tau) = \begin{cases}\emptyset & \sigma\sqsubseteq\tau,\\ \{\tau\} & \text{otherwise.}\end{cases} \tag{11}$$

The analogous of function (9) can be defined as

$$\mathsf{next}(v) = \lambda k\in\mathbb{L}_0.\bigsqcup\{\mathsf{next}[\![B(h)]\!]v(h)\mid h\rightsquigarrow k\}, \tag{12}$$

and we obtain the analogous properties:

PROPOSITION 3.4. *Function* $\mathsf{next}\colon\mathbb{V}'\to\mathbb{V}'$ *is monotone and continuous.*

Proposition 3.4 allows us to compute $lfp(\mathsf{next})$ as the limit of the ascending Kleene sequence $v^{(n)}=\mathsf{next}^n(\bot_{\mathbb{V}'})$, where $\bot_{\mathbb{V}'}=\lambda h\in\mathbb{L}_0.\bot_{\mathbb{C}'}$ and $\bot_{\mathbb{C}'}=(\lambda x\in[0,r_\mu).\bot,\bot^{t_\mu})$.

$$\mathsf{error}[\![\texttt{start}\ \tau_0.m(\tau_1,\ldots,\tau_n)\colon\tau]\!]\langle M,s\rangle = \emptyset$$
$$\mathsf{error}[\![\tau\texttt{op}\colon\tau']\!]\langle M,\sigma_1\sigma_2 s\rangle = \mathsf{check}(\sigma_1,\tau)\cup\mathsf{check}(\sigma_2,\tau)$$
$$\mathsf{error}[\![\tau\texttt{const}\ d]\!]\langle M,s\rangle = \emptyset$$
$$\mathsf{error}[\![\tau\texttt{load}\ x]\!]\langle M,s\rangle = \mathsf{check}(M(x),\tau)$$
$$\mathsf{error}[\![\tau\texttt{store}\ x]\!]\langle M,\sigma s\rangle = \mathsf{check}(\sigma,\tau)$$
$$\mathsf{error}[\![\texttt{if}\mathit{cond}\ k]\!]\langle M,\sigma s\rangle = \mathsf{check}(\sigma,\texttt{i})$$
$$\mathsf{error}[\![\texttt{goto}\ k]\!]\langle M,s\rangle = \emptyset$$
$$\mathsf{error}[\![\texttt{getfield}\ \tau.f\colon\tau']\!]\langle M,\sigma s\rangle = \mathsf{check}(\sigma,o)$$
$$\mathsf{error}[\![\texttt{putfield}\ \tau.f\colon\tau']\!]\langle M,\sigma_1\sigma_2 s\rangle = \mathsf{check}(\sigma_1,\tau')\cup\mathsf{check}(\sigma_2,\tau)$$
$$\mathsf{error}[\![\texttt{invoke}\ \tau_n.m(\tau_0,\ldots,\tau_{n-1}):\tau]\!]\langle M,\sigma_0\cdots\sigma_n s\rangle = \bigcup_{i=0}^{n}\mathsf{check}(\sigma_i,\tau_i)$$
$$\mathsf{error}[\![\tau\texttt{return}]\!]\langle M,\sigma\rangle = \mathsf{check}(\sigma,\tau)$$

Fig. 9. The error$[\![I]\!]$ function for all instructions $I\in\mathbb{I}_1$.

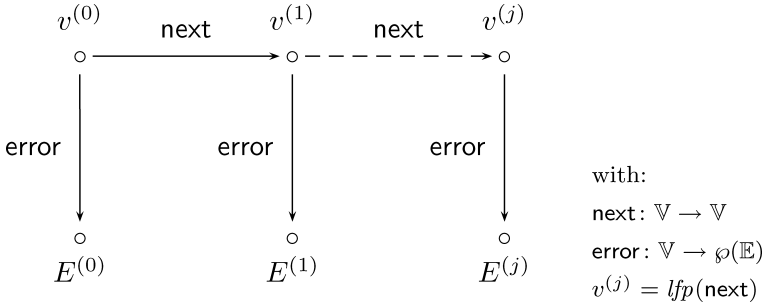

Fig. 10. The chain $E^{(i)}$ is derived from the chain $v^{(i)}$ by applying the function error.

Moreover, we define the function $\mathsf{error}\colon\mathbb{V}\to\wp(\mathbb{E})$, which, when applied to a context vector v, returns the set of types $E=\mathsf{error}(v)$ such that, if $\tau\in E$, then at least one constraint of type τ failed in checking the context vector v:

$$\mathsf{error}(v) = \bigcup_{h\in\mathbb{L}_0}\mathsf{error}[\![B(h)]\!]v(h). \tag{13}$$

Therefore from the context vector chain we can derive an "error" chain $E^{(i)}$, $i\geq 0$, such that $E^{(i)}=\mathsf{error}(v^{(i)})$ for all $i\geq 0$ (see Figure 10).

PROPOSITION 3.5. *Function* $\mathsf{error}\colon\mathbb{V}\to\wp(\mathbb{E})$ *is monotone.*

By Propositions 3.4 and 3.5, the sequence $E^{(i)}$ is an ascending chain; thus, the complete set of errors can be obtained by applying error to the fixpoint of next. A method is rejected iff $\mathsf{error}(lfp(\mathsf{next}))\neq\emptyset$. We show that the verification defined in this section, is equivalent to standard verification, as defined in Section 3.4.

THEOREM 3.6. *Let* μ *be a method. There exists* $k\in\mathbb{L}_0$ *such that* $lfp(\mathsf{std}_\mu)(k)=\Omega$*, iff either* μ *is not stack safe, or* $\mathsf{error}(lfp(\mathsf{next}))\neq\emptyset$.

PROOF SKETCH. Note that, if μ is not stack safe, then an error will be found in standard verification. On the other hand, if a stack size error is found in standard verification, then μ is not stack safe. Thus, it remains to show that, if μ is stack safe, then the two verifications are equivalent. The proof proceeds by induction on the iterates $q^{(i)}$ of std and $v^{(i)}$ of next, showing that, as long as there are no errors, the two iterates are equivalent. We say that $v \in \mathbb{V}$ and $v' \in \mathbb{V}'$ are equivalent iff, $\forall h \in \mathbb{L}_0$, context $v(h)$ is equivalent to context $v'(h)$. Two (correct) contexts, $c = (M, s) \in \mathbb{C}$ and $c' = (M', s') \in \mathbb{C}'$ are equivalent iff $M = M'$ and $s' = s\top^{t-|s|}$. Then, it must be shown, by cases, that, whenever $c \in \mathbb{C}$ and $c' \in \mathbb{C}'$ are equivalent, then $[\![I]\!]c = \Omega$ iff $\mathsf{error}(c') \neq \emptyset$, for all $I \in \mathbb{I}_1$. Thus, as soon as an error is found in $q^{(i)}$, an error is found in $v^{(i)}$, and vice versa. Monotonicity of std, next and error will then propagate the error to the fixpoint. We only examine a single case here, since they are all similar. Assume $I = \tau\, o\, p\colon \tau'$ and let $c = (M, s)$ and $c' = (M', s')$, with $M = M'$ and $s' = s\top^{t-|s|}$. Since we are assuming that μ is stack safe, we know that $|s| \geq 2$, so $s = s_0 s_1 s\rfloor_2$ and also $s' = s_0 s_1 s\rfloor_2 \top^{t-|s|}$. If $s_0 \sqsubseteq \tau$ and $s_1 \sqsubseteq \tau$, context c is correct and $[\![I]\!]c$ must be equivalent to $\mathsf{next}[\![I]\!]c'$. In fact, $[\![I]\!]c = (M, \tau' s\rfloor_2) = (M, \bar{s})$, with $|\bar{s}| = 1 + |s\rfloor_2| = |s| - 1$, and $\mathsf{next}[\![I]\!]c' = (M', \tau' s\rfloor_2 \top^{t-|s|}\top) = (M, \bar{s}\top^{t-|\bar{s}|})$. If $s_0 \not\sqsubseteq \tau$ or $s_1 \not\sqsubseteq \tau$, then $[\![I]\!]c = \Omega$ and also $\mathsf{error}(c') = \mathsf{check}(s_0, \tau) \cup \mathsf{check}(s_1, \tau) = \{\tau\} \neq \emptyset$, as required. □

Hereafter we will always use $\mathbb{C}'$ instead of $\mathbb{C}$, and we will omit the "prime," for the sake of simplicity.

4. PARALLEL DECOMPOSITION

In this section we will exploit abstract interpretation to decompose a fixpoint analysis into a set of simpler subanalyses. Each of the subanalysis will be an abstract interpretation of the original analysis. Under some conditions, the results of the subanalyses can be put together to recover the result of the original analysis. The lemma at the end of this section will describe such conditions.

We first define the term *decomposition* [Cortesi et al. 1997].

Definition 4.1. A *decomposition* of an abstraction $\rho\colon C \to C$ is a family $\{\rho_i\}_{i\in I}$ of abstractions such that $\bigsqcap_{i\in I} \rho_i = \rho$.

Given the isomorphism between $\langle uco(C); \sqsubseteq\rangle$ and $\langle abs(C); \supseteq\rangle$, we can also say that a decomposition of domain C is a set of abstract domains $\{C_i\}_{i\in I}$, such that $\circledcirc\mathscr{M}(\bigcup_{i\in I} C_i) = C$ (recall that join, in $abs(C)$, is given by the Moore closure of set union).

Note that the most space efficient decomposition of a domain is the decomposition into *basic domains* [Ward 1942]. Given a domain C and an element $c \in C$, a *basic domain* $\mathscr{B}_c$ is given by the two element domain $\{c, \top\}$. Instantiating formula (1) to this simple case, we obtain the associated closure operator, $\varphi_{\mathscr{B}_c}$, as

$$\varphi_{\mathscr{B}_c}(d) = \begin{cases} c & \text{if } d \sqsubseteq c, \\ \top & \text{otherwise.} \end{cases} \tag{14}$$

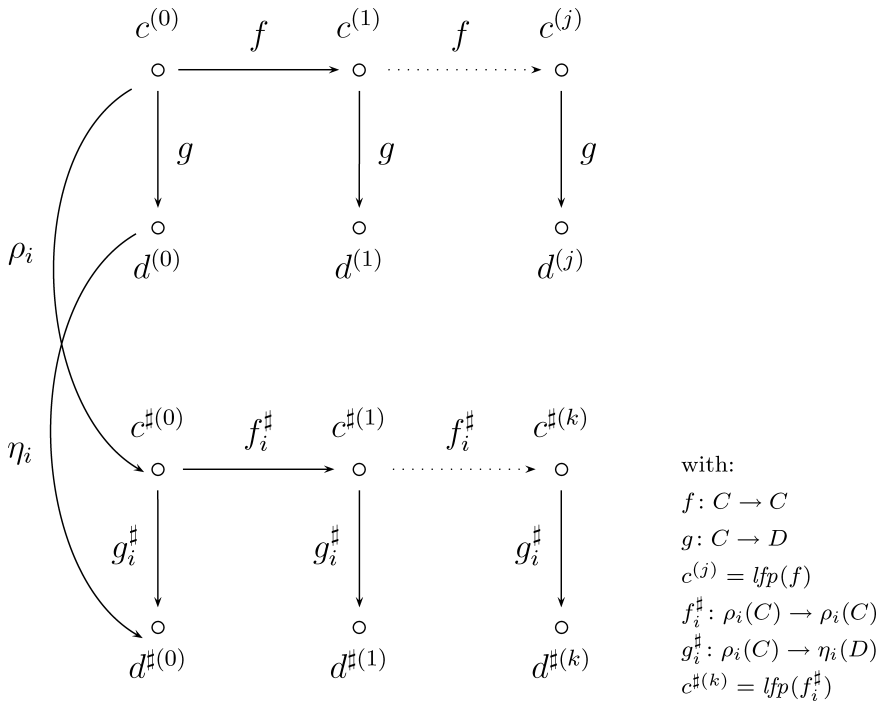

Fig. 11. Decomposition of the concrete analysis.

A decomposition in basic domains is always possible. However, it is not guaranteed that the decomposition preserves any property that the original domain may have, such as completeness wrt a given semantic function.

Consider two semantic operations: one is used to compute a fixpoint, and another is used to extract the requested information from the fixpoint. We want to "decompose" the fixpoint calculation into several fixpoint calculations. We start by decomposing the requested information into several parts and, from this, we show how to obtain a corresponding decomposition of the fixpoint calculation, such that from each calculation we can retrieve the corresponding information part. Then, we want complete information to be recovered, by putting all parts together. Figure 11 illustrates the idea. Function $f : C \to C$, is the function for which a fixpoint is calculated, and function $g : C \to D$ is the function that extracts the requested information. Function g is used to obtain a parallel chain of values in D, from the iterates of f. Then, we introduce a set of abstraction pairs $\langle \rho_i, \eta_i \rangle$, where each ρ_i abstracts set C and each η_i abstracts set D. Then we reproduce the fixpoint iteration and the parallel chain with respect to each abstraction pair.

We show that this can be done without losing precision, that is, that we can retrieve the result of the initial analysis by composing the results of the decomposed analyses.

The following straightforward lemma shows sufficient requirements for this decomposition. It acts as a guideline for the design of the decomposition.

LEMMA 4.2 PARALLEL DECOMPOSITION. *Let* $\langle C; \sqsubseteq \rangle$ *and* $\langle D; \leq \rangle$ *be complete lattices,* $f : C \to C$ *and* $g : C \to D$*. Moreover, let:*

(a) $\{\rho_i\}_{i \in I}$ *be a set of closure operators on* C*, and* $\{f_i^\sharp\}_{i \in I}$ *be a set of functions, where* $f_i^\sharp : \rho_i(C) \to \rho_i(C)$ *is fixpoint complete for* f *wrt* ρ_i*,* $\forall i \in I$*;*

(b) $\{\eta_i\}_{i \in I}$ *be a set of closure operators of* D*, and* $\{g_i^\sharp\}_{i \in I}$ *be a set of functions, where* $g_i^\sharp : \rho_i(C) \to \eta_i(D)$ *is complete for* g *wrt* $\langle \rho_i, \eta_i \rangle$*,* $\forall i \in I$*.*

If $\{\eta_i(D)\}_{i \in I}$ *is a decomposition of* D*, then*

$$\bigwedge_{i \in I} g_i^\sharp(lfp(f_i^\sharp)) = g(lfp(f)).$$

PROOF.

$$\begin{aligned}
\bigwedge_{i \in I} g_i^\sharp(lfp(f_i^\sharp)) &= \quad \text{since each } f_i^\sharp \text{ is fixpoint complete for } f \text{ wrt } \rho_i \\
\bigwedge_{i \in I} g_i^\sharp(\rho_i(lfp(f))) &= \quad \text{since each } g_i^\sharp \text{ is complete for } g \text{ wrt } \langle \rho_i, \eta_i \rangle \\
\bigwedge_{i \in I} \eta_i(g(lfp(f))) &= \quad \text{since } \{\eta_i\}_{i \in I} \text{ is a decomposition of } D \\
g(lfp(f)). & \quad \square
\end{aligned}$$

4.1 Parallel Decomposition of Bytecode Verification

We now apply the parallel decomposition method to bytecode verification, in order to execute simpler and more space efficient analyses. In particular, we use Lemma 4.2 with $C = \langle \mathbb{V}; \sqsubseteq \rangle$, $D = \langle \wp(\mathbb{E}); \subseteq \rangle$, $f = \mathsf{next}$ and $g = \mathsf{error}$ (compare Figure 10 with Figure 11). The idea is that each analysis in the decomposition should only look for a subset of all possible errors, in such a way that, when taken together, the set of analyses will check all possible errors. To focus on a subset $E \subseteq \mathbb{E}$ of errors, we consider the abstraction $\alpha_E : \wp(\mathbb{E}) \to \wp(E)$, where $\alpha_E(Q) = Q \cap E$. The corresponding concretization function $\gamma_E : \wp(E) \to \wp(\mathbb{E})$ is $\gamma_E(Q) = Q \cup \overline{E}$ (where $\overline{E} = \mathbb{E} \setminus E$ is the set-theoretic complement of E). It is easy to prove that $(\wp(\mathbb{E}), \alpha_E, \gamma_E, \wp(E))$ is indeed a Galois Connection. Note that the concretization function makes it explicit that, by focusing on a subset E of errors only, we are making no precise statement on all other possible errors. Thus, each abstraction α_E can only be used to prove the *absence* of errors on the types in E. The closure operator corresponding to the above Galois Connection is $\eta_E(Q) = \gamma_E(\alpha_E(Q)) = (Q \cap E) \cup \overline{E} = (Q \cup \overline{E}) \cap (E \cup \overline{E}) = Q \cup \overline{E}$.

Take now any family $\{E_i\}_{i \in I}$ of subsets of errors that covers $\mathbb{E}$ (i.e., $\bigcup_{i \in I} E_i = \mathbb{E}$) and let $\eta_i = \lambda Q \in \wp(\mathbb{E}).Q \cup \overline{E_i}$, for all $i \in I$. The family of abstractions $\{\eta_i\}_{i \in I}$ is a decomposition of $\wp(\mathbb{E})$. In fact, for any $Q \in \wp(\mathbb{E})$, we have:

$$\bigcap_{i \in I} \eta_i(Q) = \bigcap_{i \in I} Q \cup \overline{E_i} = Q \cup \bigcap_{i \in I} \overline{E_i} = Q \cup \overline{\bigcup_{i \in I} E_i} = Q \cup \overline{\mathbb{E}} = Q.$$

The intuitive meaning of the decomposition is that we can prove that a method is correct if we can separately prove the absence of each possible error. The family

of functions $\{g_i^\sharp\} = \{\mathsf{error}_i^\sharp\}$ can be taken as the family of best abstractions of error wrt each η_i, that is, $\mathsf{error}_i^b = \eta_i \circ \mathsf{error} = \lambda v \in \mathbb{V}.\ \mathsf{error}(v) \cup \overline{E_i}$. To apply Lemma 4.2, we need to find, for each abstraction η_i, a corresponding abstraction ρ_i of $\mathbb{V}$, such that each ρ_i is fixpoint complete for next, and each pair $\langle \rho_i, \eta_i \rangle$ is complete for error. Then Lemma 4.2 ensures that we can recover $\mathsf{error}(\mathit{lfp}(\mathsf{next}))$ by taking the intersection of all $\mathsf{error}_i^b(\mathit{lfp}(\mathsf{next}_i^b))$, where next_i^b is the best abstraction of next wrt ρ_i. The required family $\{\rho_i\}_{i \in I}$ can be found in two separate steps:

(1) for each η_i, find a ρ_i' such that $\langle \rho_i', \eta_i \rangle$ is complete for error;
(2) for each ρ_i', find a ρ_i more precise than ρ_i' (i.e., $\rho_i'(\mathbb{V}) \subseteq \rho_i(\mathbb{V})$), such that $\langle \rho_i, \rho_i \rangle$ is complete wrt next.

Step 2 ensures that $\langle \rho_i, \rho_i \rangle$ is complete wrt next and, thus, also that ρ_i is fixpoint complete for next. Moreover, since $\rho_i'(\mathbb{V}) \subseteq \rho_i(\mathbb{V})$, we also have that $\langle \rho_i, \eta_i \rangle$ is complete wrt error (see Section 2.3.1). We perform steps 1 and 2 in the following sections.

4.1.1 *Completeness for* error. In this section, given $E \in \wp(\mathbb{E})$ and $\eta_E = \lambda Q \in \wp(\mathbb{E}).Q \cup \overline{E}$, we find an abstraction ρ' of $\mathbb{V}$, such that $\langle \rho', \eta_E \rangle$ is complete for error. Abstraction ρ' can be obtained in the following way: start with the "most abstract" abstraction of $\mathbb{V}$ and find its relative complete shell wrt η_E and error. The most abstract abstraction of $\mathbb{V}$ is $\dot{\varphi}_{\{\top\}}$, the pointwise extension from $\mathbb{T}$ to $\mathbb{V}$ of $\varphi_{\{\top\}} = \lambda\sigma \in \mathbb{T}.\top$. Note that it is much simpler to begin calculating an abstraction ρ' of $\mathbb{T}$, rather than of $\mathbb{V}$, and then use ρ' to build the desired abstraction of $\mathbb{V}$ (by pointwise extension of ρ'). We will show that, for any $E \subseteq \mathbb{E}$, the corresponding abstraction of $\mathbb{T}$ can be chosen as $P = \circledcirc\mathscr{M}(E)$. Then, the desired abstraction of $\mathbb{V}$ is simply $\dot{\varphi}_P$. Intuitively, this means that, if we have a vector v and we want to find all errors contained in v, and related to a subset E of types, we can "forget everything" about those types in v that are not assignable to types in E. In fact, all types in $\mathbb{T}$ that are not assignable to a type in E are mapped to the unknown type $\top$ by φ_P.

First, let us consider function $f_\tau = \lambda\sigma \in \mathbb{T}.\,\mathsf{check}(\sigma, \tau)$ (i.e., function (11) with the second parameter fixed to τ) for any $\tau \in \mathbb{E}$ and let us compute the relative complete shell of $\varphi_{\{\top\}}(\mathbb{T}) = \{\top\}$ wrt η_E and f_τ. According to Equation (5), this is computed as

$$\mathscr{S}^B_{f_\tau}(\{\top\}) = \circledcirc\mathscr{M}(\{\top\} \cup \mathscr{R}_{f_\tau}(B)),$$

where $B = \eta_E(\wp(\mathbb{E}))$ is the Moore family of $\wp(\mathbb{E})$ that corresponds to closure η_E and (by Equation (6))

$$\mathscr{R}_{f_\tau}(B) = \bigcup_{Q \in B} \max\{\sigma \in \mathbb{T} \mid \mathsf{check}(\sigma, \tau) \subseteq Q\}.$$

Note that each $Q \in \eta_E(\wp(\mathbb{E}))$ has the form $Q = E' \cup \overline{E}$, with $E' \subseteq E$. Now, either $\tau \in E$ or $\tau \notin E$. In the former case, we have $\tau \notin \overline{E}$ and we can choose $E' \subseteq E$ such that $\tau \notin E'$, so that $\tau \notin Q = E' \cup \overline{E}$. Then, the maximal σ such that $\mathsf{check}(\sigma, \tau) \subseteq Q$ is $\sigma = \tau$, since $\mathsf{check}(\sigma, \tau) = \emptyset \subseteq Q$ when $\sigma \sqsubseteq \tau$, but $\mathsf{check}(\sigma, \tau) = \{\tau\} \not\subseteq Q$ when $\sigma \not\sqsubseteq \tau$. In all other cases, $\mathsf{check}(\sigma, \tau) \subseteq Q$, $\forall\sigma \in \mathbb{T}$, and the maximal is $\sigma = \top$. The net result is that $\mathscr{R}_{f_\tau}(\eta_E(\mathbb{E}))$ is $\{\tau, \top\}$

if $\tau \in \boldsymbol{E}$, and $\{\top\}$ otherwise. If we let τ vary over $\mathbb{E}$, we obtain the final result that $P = \mathscr{M}(\boldsymbol{E})$ is complete wrt η_E and each f_τ, that is,

$$(\forall \sigma \in \mathbb{T}, \tau \in \mathbb{E}) \quad \eta_E(\mathsf{check}(\sigma, \tau)) = \eta_E(\mathsf{check}(\varphi_P(\sigma), \tau)).$$

Since $\mathsf{error}(v)$ is simply the union of several $\mathsf{check}(v[i], \tau)$'s functions, each applied to an element $v[i] \in \mathbb{T}$ of vector v and a $\tau \in \mathbb{E}$, we obtain the final result that $\langle \dot{\varphi}_P, \eta_E \rangle$ is complete for error.

4.1.2 *Breaking up Bytecode Instructions.* In this and in the following section we turn to completeness wrt next. Given $P \in abs(\mathbb{T})$, we look for an abstraction ρ of $\mathbb{V}$ that is more precise than $\rho' = \dot{\varphi}_P$, and such that $\langle \rho, \rho \rangle$ is complete wrt next. This matches the definition of the absolute complete shell of $\dot{\varphi}_P$ wrt next, thus we could instantiate Equation (7) and try to compute $\rho = \mathscr{S}_{\mathsf{next}}(\dot{\varphi}_P(\mathbb{V}))$. However, this is a rather complex computation, since the exact form of method, including its Control Flow Graph, is taken into account. A more reasonable approach would be to use definition (12) for function next, disregarding relation $\rightsquigarrow$ (i.e., the Control Flow Graph). In this approach, we would find the complete shell $\mathscr{S}_F(\dot{\varphi}_P(\mathbb{C}))$ wrt the set F of all $\mathsf{next}[\![B(h)]\!]$ functions for $h \in \mathbb{L}_\mu$, that is, wrt the set of all bytecode instructions occurring in the method. This would greatly simplify the computation, since the complete shell wrt a set of functions is simply obtained from the union of the complete shells computed wrt each function taken separately. However, this is still a fairly complex computation, since all effects of the instructions are taken into account, even those not related to types, such as popping elements from the stack. Here, we go a step further, and define a new expression for next, where simple functions are composed according to a finer grained $\twoheadrightarrow$ relation between individual context elements (i.e., single registers or stack positions). Then, the complete shell will be computed wrt the set of simple functions, disregarding relation $\twoheadrightarrow$.

First, we represent contexts as arrays of types so that we can adopt a uniform index j for all elements in a context, where $c(j)$ is a register if $0 \leq j < r_\mu$, and it is a stack item if $r_\mu \leq j < r_\mu + t_\mu$. In particular, $c(r_\mu)$ denotes the top of the stack. Each transition function $\mathsf{next}[\![I]\!] : \mathbb{C} \to \mathbb{C}$ can be decomposed into subfunctions, each of which gives the "next" type of a different stack or memory item in the target context:

$$\mathsf{next}[\![I]\!]_j = \lambda c \in \mathbb{C}.(\mathsf{next}[\![I]\!]c)(j).$$

For the language subset we are considering, the subfunctions are particularly simple, since each of them depends only on the type stored in at most one specific stack or memory item of the source context c, and not on c as a whole. Let $n_\mu = r_\mu + t_\mu$ be the size of a context for method μ. Relation $\twoheadrightarrow_I \subseteq [0, n_\mu) \times [0, n_\mu)$ is the relation between each item in the target context and the item(s) in the source context that may affect it in the execution of instruction I. We define relation $\twoheadrightarrow_I$ indirectly as the complement of relation $i \not\twoheadrightarrow_I j$ (i does not affect j in instruction I) where

$$i \not\twoheadrightarrow_I j \quad \text{iff} \quad (\forall c', c'' \in \mathbb{C})\, c' = c''[c'(i)/i] \implies \mathsf{next}[\![I]\!]_j c' = \mathsf{next}[\![I]\!]_j c'',$$

that is, if element j in the after context of instruction I is always the same for all before contexts that only differ in element i, then i does not affect j in instruction I.

Example 4.3. Consider instruction $I = \tau\texttt{load}\ x$. We have:

—for all $j \in [0, r_\mu)$, $j \twoheadrightarrow_I j$, with $\mathsf{next}[\![I]\!]_j = \lambda c.c(j)$, modeling the fact that registers are not modified by instruction I;
—$x \twoheadrightarrow_I r_\mu$, with $\mathsf{next}[\![I]\!]_{r_\mu} = \lambda c.c(x)$, modeling the push of register x on the stack;
—for all $j \in (r_\mu, n_\mu)$, $j - 1 \twoheadrightarrow_I j$ with $\mathsf{next}[\![I]\!]_j = \lambda c.c(j-1)$, modeling stack shift of one position.

And $i \not\twoheadrightarrow_I j$ in all other cases.

For all instructions in the language subset we are considering, any j may be affected by at most one i, that is, $i' \twoheadrightarrow_I j$ and $i'' \twoheadrightarrow_I j$ implies $i' = i''$ for all $I \in \mathbb{I}_1$ (the only exception will be found in $\mathbb{I}_3$ in Section 4.3). For some instructions, some element j of the after context may be not affected by any element of the before context, that is, function $\mathsf{next}[\![I]\!]_j$ may be constant. For example, instruction $I = \tau op\ \tau'$ always pushes τ on the stack, irrespectively of the before context. Thus, $\mathsf{next}[\![I]\!]_{r_\mu} = \lambda c.\tau$ and $i \not\twoheadrightarrow_I r_\mu$, for all $i \in [0, n_\mu)$. To simplify the notation and to have each element affected by exactly one other element, we will assume a fictitious dependency $j \twoheadrightarrow_I j$ whenever $\mathsf{next}[\![I]\!]_j$ is a constant function.

Now, we turn to context vectors and linearize them as unidimensional vectors of types. Each context vector for a method μ will contain $m_\mu = (l_\mu + 1) \cdot n_\mu$ types. Let $\div$ denote natural division and mod division remainder. Then, if $i \in [0, m_\mu)$ is the index of an item in context vector v, $(i \div n_\mu) \in [0, l_\mu]$ is the index of the context c that contains item i, and $(i \bmod n_\mu) \in [0, n_\mu)$ is the index of item i within context c. We define relation $\twoheadrightarrow_\mu \subseteq [0, m_\mu) \times [0, m_\mu)$ in the following way:

$$i \twoheadrightarrow_{B_\mu(h)} j \text{ and } h \rightsquigarrow k \implies n_\mu h + i \twoheadrightarrow_\mu n_\mu k + j.$$

Relation $\twoheadrightarrow_\mu$ encodes all the dependencies between individual context vector elements for the whole method μ. We now make it explicit that (for the language subset we are considering) all functions $\mathsf{next}[\![I]\!]_j$ depend on (at most) one element of the appropriate before context by defining functions $\mathsf{next}_{ij} : \mathbb{T} \to \mathbb{T}$ as

$$\mathsf{next}_{ij}(\tau) = \mathsf{next}[\![B_\mu(i \div n_\mu)]\!]_{j \bmod n_\mu} c[\tau / i \bmod n_\mu].$$

For any $c \in \mathbb{C}$ and $i \twoheadrightarrow_\mu j$. The definition is independent of c, since $j \bmod n_\mu$ is affected only by $i \bmod n_\mu$ in instruction $B_\mu(i \div n_\mu)$.

Example 4.4. Let $h \in [0, l_\mu]$ and assume $B_\mu(h) = \tau'\texttt{add}\colon \tau$, $r_\mu = 1$, and $t_\mu = 3$ (see Figure 12 for a graphical representation of this example). In this case, $h \rightsquigarrow k$ means that $k = h + 1$. Consider $j \in [4k, 4k + 4)$ (that is the context

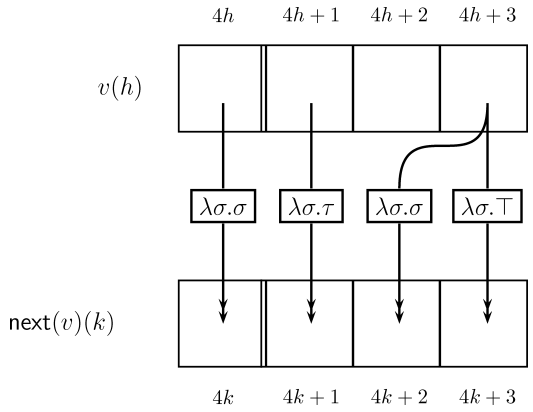

Fig. 12. The detail of the execution of a $\tau'\texttt{add}\colon\tau$ instruction when there is only one register and the stack contains up to three items: the register is not affected, the two topmost items of the stack are consumed, and τ is pushed into the stack.

corresponding to the instruction k). Then, $i \twoheadrightarrow_\mu j$ only when:

—$i = 4h$ and $j = 4k$. Then, $\mathsf{next}_{ij} = \lambda\sigma.\sigma$ (identity function on $\mathbb{T}$). This models the fact that the register is not modified by $\tau'\texttt{add}\colon\tau$

—$i = 4h + 3$ and $j = 4k + 2$. Then, $\mathsf{next}_{ij} = \lambda\sigma.\sigma$. This models the stack shift of one position, due to a) the pop of the two operands and b) the subsequent push of the result.

—$i = 4h + 1$ and $j = 4k + 1$. Then, $\mathsf{next}_{ij} = \lambda\sigma.\tau$. This models the push of the result on the stack

—$i = 4h + 3$ and $j = 4k + 3$. Then, $\mathsf{next}_{ij} = \lambda\sigma.\top$. This is the last element in the stack, after stack pop.

Note that, in the last two cases, we have used the convention that $j \twoheadrightarrow_I j$ whenever the effect of instruction I on j is constant.

The effect that the $\tau'\texttt{add}\colon\tau$ instruction of Example 4.4 has on the types contained in the context vectors can be summarized by the set of simple functions $\lambda\sigma.\sigma$, $\lambda\sigma.\tau$ and $\lambda\sigma.\top$ (i.e., the functions in the boxes in the middle of Figure 12). The other effects of the instruction (e.g., shifting elements of the stack) are captured by the $\twoheadrightarrow_\mu$ relation. The point is that relation $\twoheadrightarrow_\mu$, no matter how complex, will be automatically preserved by any abstraction we will introduce. This means that abstractions can be defined on types and then readily extended to context vectors. Moreover, in the complete shell computations, we can ignore relation $\twoheadrightarrow_\mu$ and use only the set of subfunctions.

Using this decomposition, we can finally write another expression for next, as

$$\mathsf{next}(v) = \lambda j \in [0, m).\bigsqcup\{\mathsf{next}_{ij}(v[i]) \mid i \twoheadrightarrow j\}. \tag{15}$$

Where $v[i] = v(i \div n_\mu)(i \bmod n_\mu)$ and we have omitted subscript μ for simplicity. We can simplify this expression by assuming $\mathsf{next}_{ij} = \lambda\sigma.\bot$ whenever $i \not\twoheadrightarrow j$

$$\mathscr{F}[\![\texttt{start}\ \tau_0.\mu(\tau_1,\ldots,\tau_n)\colon\tau]\!] = \{\lambda\sigma.\tau_0,\ \ldots,\ \lambda\sigma.\tau_n,\ \lambda\sigma.\top\}$$
$$\mathscr{F}[\![\tau\texttt{op}\colon\tau']\!] = \{\lambda\sigma.\sigma,\ \lambda\sigma.\tau',\ \lambda\sigma.\top\}$$
$$\mathscr{F}[\![\tau\texttt{const}\ d]\!] = \{\lambda\sigma.\sigma,\ \lambda\sigma.\tau\}$$
$$\mathscr{F}[\![\tau\texttt{load}\ x]\!] = \{\lambda\sigma.\sigma\}$$
$$\mathscr{F}[\![\tau\texttt{store}\ x]\!] = \{\lambda\sigma.\sigma,\ \lambda\sigma.\top\}$$
$$\mathscr{F}[\![\texttt{if}\,cond\ k]\!] = \{\lambda\sigma.\sigma,\ \lambda\sigma.\top\}$$
$$\mathscr{F}[\![\texttt{goto}\ k]\!] = \{\lambda\sigma.\sigma\}$$
$$\mathscr{F}[\![\texttt{getfield}\ \tau.f\colon\tau']\!] = \{\lambda\sigma.\sigma,\ \lambda\sigma.\tau'\}$$
$$\mathscr{F}[\![\texttt{putfield}\ \tau.f\colon\tau']\!] = \{\lambda\sigma.\sigma,\ \lambda\sigma.\top\}$$
$$\mathscr{F}[\![\texttt{invoke}\ \tau_0.m(\tau_1,\ \ldots,\ \tau_n)\colon\tau]\!] = \{\lambda\sigma.\sigma,\ \lambda\sigma.\tau,\ \lambda\sigma.\top\}$$
$$\mathscr{F}[\![\tau\texttt{return}]\!] = \{\lambda\sigma.\sigma,\ \lambda\sigma.\top\}$$

Fig. 13. The $\mathscr{F}[\![I]\!]$ function for all $I \in \mathbb{I}_1$.

(since $\bot$ has no effect on join) and rewrite Equation (15) as

$$\mathsf{next}(v) = \lambda j \in [0,m).\ \bigsqcup_{i=0}^{m-1} \mathsf{next}_{ij}(v[i]). \tag{16}$$

A method is characterized by the set of different next_{ij} functions that are implied by its bytecode. Thus, for a method μ, we introduce the set $\mathscr{F}_\mu$ that contains all distinct next_{ij} functions appearing in formula (15) for method μ. First, we define $\mathscr{F}[\![\bullet]\!]\colon \mathbb{I} \to \wp(\mathbb{T} \to \mathbb{T})$ as follows. Given $f\colon \mathbb{T} \to \mathbb{T}$ and $I \in \mathbb{I}$, then $f \in \mathscr{F}[\![I]\!]$ iff there exists a method μ and $i, j \in [0, m_\mu)$ such that $B_\mu(i \div n_\mu) = I$, $i \twoheadrightarrow_\mu j$ and $f = \mathsf{next}_{ij}$. That is, $\mathscr{F}[\![I]\!]$ contains all subfunctions of $\mathsf{next}[\![I]\!]$. Figure 13 shows function $\mathscr{F}$ for all functions in $\mathbb{I}_1$ (plus instruction `start`). Finally, we define the set of functions that may be used in the interpretation of method μ:

$$\mathscr{F}_\mu \triangleq \bigcup_{h \in \mathbb{I}_{\mu,0}} \mathscr{F}[\![B_\mu(h)]\!]. \tag{17}$$

4.1.3 *Completeness for* next. In this section, we use the characterization of next given in the previous section and find an abstraction ρ of $\mathbb{T}$, such that $\rho \sqsubseteq \varphi_P$ and $\langle\rho, \rho\rangle$ is complete for $\mathscr{F}_\mu$. Then, we show that $\dot{\rho}$ is an abstraction of $\mathbb{V}$ that meets our requirements.

The following calculation shows that we can choose $\rho = \varphi_P$, that is, that, for any $P \in abs(\mathbb{T})$ and for any method μ (in instruction set $\mathbb{I}_1$), $\langle\dot{\varphi}_P, \dot{\varphi}_P\rangle$ is complete for next_μ. Intuitively, this can be expressed as follows: assume that $P = \mathscr{M}(E)$ (as obtained in Section 4.1.1) and let us call E-assignable types all types assignable to types in E. Let us assume that we know, for a vector v, where all E-assignable types can be found, while we know nothing about all other types. This means that we are able to precisely know where all E-assignable types in $\mathsf{next}(v)$ will be (assuming we are using instruction set $\mathbb{I}_1$ only). This is a direct consequence of the fact that all subfunctions in Figure 13 are either constant functions (whose value we know from the program text) or

the identity function. Thus, E-assignable types in $\mathsf{next}(v)$ can only come from facts we already know. Together with Section 4.1.1, this shows that keeping track of E-assignable types is not only necessary, but also sufficient to find all errors in E.

Closure ρ can be easily calculated as $\rho = \mathscr{S}_{\mathscr{F}_\mu}(\varphi_P(P)) = \mathscr{S}_{\mathscr{F}_\mu}(P)$. According to (7), we have

$$\mathscr{S}_{\mathscr{F}_\mu}(P) = lfp(\lambda A \in abs(\mathbb{T}).\circ\!\mathscr{M}(P \cup \mathscr{R}_{\mathscr{F}_\mu}(A))),$$

where

$$\mathscr{R}_{\mathscr{F}_\mu}(A) = \bigcup_{f \in \mathscr{F}_\mu} \bigcup_{\sigma \in A} \max\{\tau \in \mathbb{T} \mid f(\tau) \sqsubseteq \sigma\}.$$

The least fixpoint can be calculated using an ascending Kleene sequence starting at $\bot_{abs(\mathbb{T})} = \{\top\}$. We note that, for instruction set $\mathbb{I}_1$, $\mathscr{S}_{\mathscr{F}_\mu}(P) = P$, for any $P \in abs(\mathbb{T})$. In fact:

$$\begin{aligned}
&\mathscr{R}_{\mathscr{F}_\mu}(\{\top\}) = \bigcup_{f \in \mathscr{F}_\mu} \max\{\tau \in \mathbb{T} \mid f(\tau) \sqsubseteq \top\} = \{\top\}\\
&S^{(0)} = \circ\!\mathscr{M}(P \cup \{\top\}) = \circ\!\mathscr{M}(P) = P,\\
&\mathscr{R}_{\mathscr{F}_\mu}(P) = \bigcup_{f \in \mathscr{F}_\mu} \bigcup_{\sigma \in P} \max\{\tau \in \mathbb{T} \mid f(\tau) \sqsubseteq \sigma\} = P' \subseteq P\\
&S^{(1)} = \circ\!\mathscr{M}(P \cup P') = \circ\!\mathscr{M}(P) = P.
\end{aligned}$$

The reason for $\mathscr{R}_{\mathscr{F}_\mu}(P) \subseteq P$ lies in the fact that all $f \in \mathscr{F}_\mu$ are either constant functions or the identity function (see Figure 13). For the identity function, the maximum value τ such that $\tau \sqsubseteq \sigma$ is σ itself, which is in P. Consider now a constant function $f = \lambda t.\bar{\tau}$. If $\bar{\tau} \sqsubseteq \sigma$, then $\max\{\tau \mid f(\tau) \sqsubseteq \sigma\} = \top \in P$ (since P is a Moore family). If $\bar{\tau} \not\sqsubseteq \sigma$, f gives no contribution to $\mathscr{R}_{\mathscr{F}_\mu}(P)$.

Thus, we have found that, for any $P \in abs(\mathbb{T})$, if we let $\rho = \varphi_P$, then $\langle \rho, \rho \rangle$ is complete wrt all functions f in $\mathscr{F}_\mu$, that is,

$$(\forall f \in \mathscr{F}_\mu) \quad \rho \circ f = \rho \circ f \circ \rho.$$

Note that this result could have been obtained directly, since it holds trivially whenever f is a constant or identity function. However, we have shown the absolute complete shell computation, since it will be useful in the rest of the paper. The above equation also holds for $f = \lambda\sigma.\bot$, which we have used in Equation (15) but which is not in $\mathscr{F}_\mu$.

Now, we show that $\langle \dot{\rho}, \dot{\rho} \rangle$, the pointwise extension of $\langle \rho, \rho \rangle$, is complete for next, that is,

$$\dot{\rho} \circ \mathsf{next} = \dot{\rho} \circ \mathsf{next} \circ \dot{\rho}.$$

In fact, for any $v \in \mathbb{V}$, we have:

$$\begin{aligned}
\dot{\rho}(\mathsf{next}(\dot{\rho}(v))) &= \text{by (16) and definition of } \dot{\rho} \\
\lambda j.\rho\left(\bigsqcup_{i=0}^{m-1} \mathsf{next}_{ij}(\rho(v[i]))\right) &= \text{since } \langle \rho, \rho \rangle \text{ is complete wrt all } \mathsf{next}_{ij} \text{ and join} \\
\lambda j.\rho\left(\bigsqcup_{i=0}^{m-1} \mathsf{next}_{ij}(v[i])\right) &= \text{by definition of } \dot{\rho} \text{ and (16)} \\
\dot{\rho}(\mathsf{next}(v)).
\end{aligned}$$

We are interested in reducing the space required by each pass of the verification and this space is directly related to the number of different types that must be distinguished in each pass. Therefore we would like to choose the abstractions $\{\rho_i\}_{i \in I}$ so that the maximum cardinality of $\rho_i(\mathbb{T})$, $i \in I$, is as small as possible. Since, for the language we are considering, $\mathscr{S}_{\mathscr{F}_\mu}(P) = P$ for any $P \in abs(\mathbb{T})$, we are free to choose a decomposition in basic domains (Section 4.1.2).

Thus, we partition $\mathbb{E}$ into singletons $E_i = \{\tau_i\}$ and choose $\{P_i\}_{i \in I} \subseteq \wp(\mathbb{E})$, such that $P_i = \mathscr{M}(E_i) = \mathscr{B}_{\tau_i} = \{\top, \tau_i\}$, with the associated closure operator, as given in (14). Given our interpretation of $\sqsubseteq$, φ_{P_i} has an intuitive meaning: all types that are assignable to τ_i are identified with τ_i itself, while all other types are identified with the unknown/error type.

4.2 Array Types

In this section, we consider instruction set $\mathbb{I}_2 \supset \mathbb{I}_1$. Set $\mathbb{I}_2$ includes instructions that manipulate arrays. Following the Java language specification [Gosling et al. 2000]:

> In the Java programming language *arrays* are objects, are dynamically created, and may be assigned to variables of type `Object`. All methods of class `Object` may be invoked on an array.

and later:

> All the components of an array type have the same type, called the *component type* of the array.

If the component type of an array is τ, we will denote the type "array of τ" as $[\tau$. The component type may be itself an array type, therefore we define the set $\mathcal{A}$ of (multidimensional) arrays of types $\tau \in \mathcal{B} \cup \mathcal{C}'$, as follows:

$[^0\tau = \tau$;
$[^n\tau = [[^{n-1}\tau$ is an array of elements of type $[^{n-1}\tau$;
$\mathcal{A} = \{[^n\tau \mid \tau \in \mathcal{B} \cup \mathcal{C}', n \geq 1\}$.

We define a new poset $\langle \mathbb{T}_2; \sqsubseteq_2 \rangle$ and a new set of errors, $\mathbb{E}_2$, in this way:

$$\mathbb{T}_2 = \mathbb{T}_1 \cup \mathcal{A}, \tag{18}$$
$$\mathbb{E}_2 = \mathbb{E}_1 \cup \mathcal{A}. \tag{19}$$

and the partial order $\sqsubseteq_2$ (omitting the μ subscript, as before) is the smallest reflexive, antisymmetric and transitive relation such that $(\forall \tau_1, \tau_2 \in \mathbb{T}_1)$, $\tau_1 \sqsubseteq_1 \tau_2 \implies \tau_1 \sqsubseteq_2 \tau_2$ and:

$$(\forall \alpha \in \mathcal{A}) \quad \alpha \sqsubseteq_2 \texttt{Object}, \ \texttt{null} \sqsubseteq_2 \alpha,$$
$$(\forall [\tau', [\tau'' \in \mathcal{A}) \quad \tau' \sqsubseteq_2 \tau'' \implies [\tau' \sqsubseteq_2 [\tau''.$$

Poset $\langle \mathbb{T}_2; \sqsubseteq_2 \rangle$ is still a lattice. In particular, note that $[^{n'}\texttt{Object} \sqsubseteq [^{n''}\texttt{Object}$ whenever $1 \leq n'' \leq n'$. However, since $\mathcal{A}$ is an infinite set,[1] we have to prove that $\langle \mathbb{T}_2; \sqsubseteq_2 \rangle$ is still a complete lattice.

PROPOSITION 4.5. *$\langle \mathbb{T}_2; \sqsubseteq_2 \rangle$ satisfies the Ascending Chain Condition (ACC): given any sequence $\tau_1 \sqsubseteq_2 \tau_2 \sqsubseteq_2 \cdots \sqsubseteq_2 \tau_n \sqsubseteq_2 \cdots$ of elements of $\mathbb{T}_2$, there exists $k \in \mathbb{N}$ such that $\tau_k = \tau_{k+1} = \ldots$.*

PROOF SKETCH. Note that (ACC) may fail in $\mathbb{T}_2$ only for chains containing arrays, since all other chains in $\mathbb{T}_2$ are of finite height. So, we can restrict ourselves to chains where $\tau_i \in \mathcal{A}$ for some $i \in \mathbb{N}$. Moreover, if there exists a $\bar{\imath} \in \mathbb{N}$ such that $\tau_j \notin \mathcal{A}, \forall j \geq \bar{\imath}$, then (ACC) is still satisfied. So, (ACC) may fail only if there exists a chain containing an infinite number of different array types, but this is impossible, since each pair of successive different array elements in the chain must either be of type $[^n\tau' \sqsubset_2 [^n\tau''$, with both τ' and τ'' in $\mathcal{B} \cup \mathcal{C}'$ (which is finite), or be of type $[^{n'}\texttt{Object} \sqsubset_2 [^{n''}\texttt{Object}$, with $1 \leq n'' < n'$. □

PROPOSITION 4.6. *$\langle \mathbb{T}_2; \sqsubseteq_2 \rangle$ is a complete lattice.*

PROOF. By a well known result of lattice theory, since it has a bottom element and satisfies (ACC) (by Proposition 4.5). □

Instruction set $\mathbb{I}_2$ contains $\mathbb{I}_1$, plus the instructions that manipulate arrays: `newarray`, τ`aload`, τ`astore`. Figure 14 shows the $\mathsf{next}[\![\bullet]\!]$ function for the new instructions. It is worth noting that, while in instruction `newarray` τ type τ ranges over $\mathcal{T}$, in τ`aload` and τ`astore` type τ ranges over $\mathcal{B}'$, that is the basic types (`i`, `f`, `b`, ...) plus `Object`. In the case of τ`astore`, not all type errors can be caught at verification time, but some are delayed to execution time, raising the exception `ArrayStoreException` [Lindholm and Yellin 1999]. This is because only an approximation (from above) of the runtime type of the array reference is available during verification, when class types are involved. Assume, for example, that `B` *extends* `A` and `C` *extends* `A`, but classes `B` and `C` are unrelated. Assume also that, during verification, the stack in the before context of an `aastore` (where the first `a` stands for `Object`) is [`AiBs`. Then, checking that $\texttt{B} \sqsubseteq \texttt{A}$ is unsafe, since the actual run-time type of the array reference may well be [`C`.

Note that we have considered separately the case of τ`aload`, when $\tau = \texttt{Object}$. This case corresponds to the actual JVM opcode `aaload`.

The `aaload` instruction is used to access arrays of objects and pops two items from the stack: the first type, let us say τ, should be an array reference, and the

[1]According to the virtual machine specification [Lindholm and Yellin 1999] the maximum number of dimensions of an array is 256. However, the generalization to any number of dimensions does not cause problems.

$$\mathsf{next}[\![\texttt{newarray}\ \tau]\!]\langle M, \sigma s\rangle = \langle M, [\tau s\rangle$$
$$\mathsf{next}[\![\tau\texttt{aload}]\!]\langle M, \sigma_1\sigma_2 s\rangle = \langle M, \tau s\top\rangle$$
$$\mathsf{next}[\![\texttt{aaload}]\!]\langle M, \sigma_1\sigma_2 s\rangle = \langle M, [^{-1}\sigma_1 s\top\rangle$$
$$\mathsf{next}[\![\tau\texttt{astore}]\!]\langle M, \sigma_1\sigma_2\sigma_3 s\rangle = \langle M, s\top\top\top\rangle$$

$$\mathsf{error}[\![\texttt{newarray}\ \tau]\!]\langle M, \sigma s\rangle = \mathsf{check}(\sigma, \texttt{i})$$
$$\mathsf{error}[\![\tau\texttt{aload}]\!]\langle M, \sigma_1\sigma_2 s\rangle = \mathsf{check}(\sigma_1, [\tau) \cup \mathsf{check}(\sigma_2, \texttt{i})$$
$$\mathsf{error}[\![\texttt{aaload}]\!]\langle M, \sigma_1\sigma_2 s\rangle = \mathsf{check}(\sigma_1, [\texttt{Object}) \cup \mathsf{check}(\sigma_2, \texttt{i})$$
$$\mathsf{error}[\![\tau\texttt{astore}]\!]\langle M, \sigma_1\sigma_2\sigma_3 s\rangle = \mathsf{check}(\sigma_1, [\tau) \cup \mathsf{check}(\sigma_2, \texttt{i}) \cup \mathsf{check}(\sigma_3, \tau)$$

$$\mathscr{F}[\![\texttt{newarray}\ \tau]\!] = \{\lambda\sigma.\sigma,\ \lambda\sigma.[\tau\}$$
$$\mathscr{F}[\![\tau\texttt{aload}]\!] = \{\lambda\sigma.\sigma,\ \lambda\sigma.\top,\ \lambda\sigma.\tau\}$$
$$\mathscr{F}[\![\texttt{aaload}]\!] = \{\lambda\sigma.\sigma,\ \lambda\sigma.\top,\ [^{-1}\}$$
$$\mathscr{F}[\![\tau\texttt{astore}]\!] = \{\lambda\sigma.\sigma,\ \lambda\sigma.\top\}$$

Fig. 14. The functions $\mathsf{next}[\![I]\!]$, $\mathsf{error}[\![I]\!]$ and $\mathscr{F}[\![I]\!]$ for $I \in \mathbb{I}_2 \setminus \mathbb{I}_1$.

second type should be an integer index. In the after-state, the type of the top of the stack is inferred from type τ using a function $[^{-1}\colon \mathbb{T} \to \mathbb{T}$ such that:

$$[^{-1}\tau = \begin{cases} \tau' & \text{if } \tau = [\tau' \text{ and } \texttt{null} \sqsubset \tau' \sqsubseteq \texttt{Object}, \\ \tau & \text{if } \tau \sqsubseteq \texttt{null}, \\ \top & \text{otherwise.} \end{cases} \tag{20}$$

Function $[^{-1}$ removes a dimension from a reference to an array of objects. It is the identity function when applied to $\bot$ or `null`, and it is $\top$ otherwise. Note that in semantics of the standard verifier (not shown here) for the `aaload` instruction, the third case in the definition of $[^{-1}$ would cause a type error (i.e., a transition to state Ω). Function $[^{-1}$ is a monotonic extension to $\mathbb{T}$ of the function used in the standard semantics. Figure 14 shows the value of $\mathscr{F}[\![I]\!]$ for I equal to `newarray` τ, τ`aload`, `aaload` or τ`astore` and the $\mathsf{error}[\![I]\!]$ function for the same instructions.

4.2.1 *Completeness in the Presence of Arrays.* In Section 4.1.3 we have shown that, in instruction set $\mathbb{I}_1$, for any $E \in \wp(\mathbb{E})$, $P = \mathscr{M}(E) = \mathscr{S}_{\mathscr{F}_\mu}(P)$. This means that abstraction P can be used to find all errors in E, and only them. This is no longer true if we have array types, as anticipated in the overview. This is due to the presence of subfunction $[^{-1}$, which can generate E-assignable types from non-E-assignable types. Worse, $\mathscr{S}_{\mathscr{F}_\mu}(P)$ may also be infinite. In fact, let us suppose $\exists h \in \mathbb{L}_\mu$ such that $B_\mu(h) =$ `aaload`, and consider $P \in \mathit{abs}(\mathbb{T})$ such that P contains a type $\bar\tau$, with $\texttt{null} \sqsubset \bar\tau \sqsubseteq \texttt{Object}$. Now, let us compute $\mathscr{S}_{\mathscr{F}_\mu}(P)$. We obtain $S^{(0)} = \{\top\}$ and $S^{(1)} = P$. To compute $S^{(2)}$, we must calculate $\mathscr{R}_{\mathscr{F}_\mu}(P)$. Since μ contains the `aaload` instruction, $\mathscr{F}_\mu$ contains function $[^{-1}$. The maximal τ in $\mathbb{T}$, such that $[^{-1}\tau \sqsubseteq \bar\tau$ is $[\bar\tau$, thus $[\bar\tau \in S^{(2)}$. Then, $S^{(3)}$ will contain $[[\bar\tau$ (since this is the maximal $\tau \in \mathbb{T}$ such that $[^{-1}\tau \sqsubseteq [\bar\tau$) and so on. Finally, the fixpoint will contain all (infinite) multidimensional arrays of $\bar\tau$. Note that this

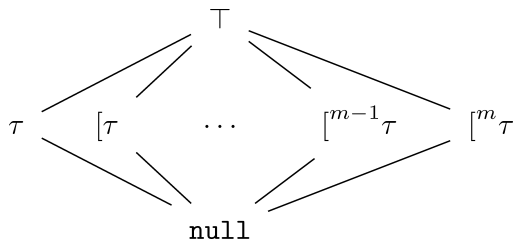

Fig. 15. Lattice $\mathscr{A}^k_\tau$ for $\texttt{null} \sqsubset \tau \sqsubset \texttt{Object}$.

would not be the case if we had calculated $\mathscr{S}_{\mathsf{next}}$ instead of $\mathscr{S}_{\mathscr{T}_\mu}$; thus, this is a limitation of breaking up bytecode instructions. However, the problem can be overcome by noting that most of these array types will never appear in any of the iterates when calculating $lfp(\mathsf{next})$, therefore we can remove them from the set of possible types. More precisely, let $\mathscr{T}_\mu$ be the set of types that may actually appear during the fixpoint iteration of function next_μ. It can be shown that, if we use $\mathscr{T}_\mu$ instead of $\mathbb{T}$ as concrete domain for types, then $\mathscr{S}_{\mathscr{T}_\mu}(\mathscr{B}_\tau)$ will only contain the finite set $\{[^i\tau \mid 0 \le i \le \kappa_{\mu,\tau}\}$, where $\kappa_{\mu,\tau} = \max\{i \mid \exists\sigma \sqsubseteq \tau, [^i\sigma \in \mathscr{T}_\mu\}$. More precisely, let us define $\mathscr{A}^k_\tau = \{[^i\tau \mid 0 \le i \le k\} \cup \{\texttt{null}, \top\}$. An example $\mathscr{A}^k_\tau$ is depicted in Figure 15. Using κ as shorthand for $\kappa_{\mu,\tau}$, we have $\mathscr{A}^\kappa_\tau = \mathscr{S}_{\mathscr{T}_\mu}(\mathscr{B}_\tau)$. Note that the only parameter that is required to build $\mathscr{A}^\kappa_\tau$ is κ, and this can be easily obtained by inspecting the program text. In fact, if $[^i\sigma \in \mathscr{T}_\mu$, then there exists a function $f \in \mathscr{T}_\mu$, f constant, such that $[^j\tau \in rng(f)$ with $\sigma \sqsubseteq \tau$ and $i \le j$ ($rng(f)$ is the range of f). Thus, $\kappa_{\mu,\tau}$ must appear explicitly in the opcode of at least one instruction in B_μ.

Reconsider Figure 3. Parallel decomposition using the basic domain $\mathscr{B}_\texttt{C}$ is incomplete since $[^{-1} \in \mathscr{T}_\mu$ and $\mathscr{S}_{\mathscr{T}_\mu}(\mathscr{B}_\texttt{C}) \neq \mathscr{B}_\texttt{C}$. The complete shell is $\mathscr{A}^1_\texttt{C} = \{\top, \texttt{C}, [\texttt{C}, \texttt{null}\}$. In fact, $\kappa_{\mu,\texttt{C}} = 1$ since the only array of `C` in the method is `[C` created by instruction `start` and, during verification, the `aaload` instruction may only reduce its dimension.

4.3 Object Creation and Initialization

The creation of a class instance in Java is split into two phases. In the first phase, the object is created, and its fields are filled with the default values. In the second phase, one of the constructors is called on the object, to perform other initializations. In JVML, the first phase is performed by executing an instruction `new` τ that has the class $\tau \in \mathcal{C}'$ as argument. Its effect is to leave on the top of the operand stack a reference to an uninitialized instance of class τ. The second phase is simply an `invoke` τ.`<init>`, where `<init>` is the name that the JVM uses for all constructors. For simplicity's sake we will consider only constructors without arguments and use the abbreviation `init` τ for `invoke` τ.`<init>` (see Figure 1). Both phases are mandatory and must be performed in order. The bytecode verifier checks that:

—objects are not initialized more than once;

—as long as an object stays uninitialized, it is not used for field manipulation (either `putfield` or `getfield`) or method invocation.

However, a reference to an object that has not yet been initialized can be stored in a local variable or in a stack location. The Sun verifier tags each uninitialized reference of class τ with the label of the `new` τ instruction that created it. When processing an `init` τ, if the top of the stack is a reference to an uninitialized object of type τ created at instruction h, (we denote it by $\tau^{(h)}$), then all occurrences of $\tau^{(h)}$ in memory and stack are replaced by τ, and the stack is modified by popping the reference. Otherwise, if the top of the stack is not a reference to an uninitialized object of type τ, the verification ends with a failure. The Java language specification also mandates that no uninitialized object may exist when a "backward branch" is taken [Leroy 2003]. The purpose of this requirement is to avoid that distinct objects created by the same `new` τ instruction inside a loop may be mistakenly assumed to be the same object. However, Coglio [2003] has shown that under some conditions, that are met by our proposed analysis, this requirement is unnecessary. Accordingly, we can avoid to cope with this additional complexity.

We introduce the new instruction set $\mathbb{I}_3$, which includes $\mathbb{I}_2$ and the instructions for object creation and initialization, namely `new` and `init`. These are described in Figure 1. We require that, if $B(k) = h$: `new` τ, then $k = h$. Note that we have included the label h of the `new` instruction in the opcode itself, since this is the only instruction that produces a type that depends on h (i.e., $\tau^{(h)}$). Otherwise, we would have had to let $\mathsf{next}[\![\bullet]\!]$ depend on B, thus making the notation heavier.

Note that types $\tau^{(h)}$ and $\sigma^{(k)}$, with $k \neq h$, are not assignable to each other, even when $\tau = \sigma$. Moreover, $\tau^{(h)}$, for any τ and h, is not assignable to `Object`, but can be used by the same `astore` and `aload` instructions used for `Object`. To correctly express the typing condition of these two instructions, we introduce a new type `addr`, which is the least upper bound of `Object` and all $\tau^{(h)}$'s, and we assume that the `a` in `astore` and `aload` stands for `addr`.

We introduce the new lattice $\langle \mathbb{T}_3; \sqsubseteq_3 \rangle$ and the new set $\mathbb{E}_3$ of errors, where:

$$\mathbb{T}_3 = \mathbb{T}_2 \cup \{\texttt{addr}\} \cup \mathbb{U}, \tag{21}$$

$$\mathbb{E}_3 = \mathbb{E}_2 \cup \{\texttt{addr}\} \cup \{\tau^{(*)} \mid \exists \tau^{(h)} \in \mathbb{U}\}, \tag{22}$$

where $\mathbb{U} = \{\tau^{(h)} \mid \tau \in \mathcal{C}' \text{ and } h \in \mathbb{L}\}$ is the set of uninitialized types. We also define $\mathbb{U}_\mu$ as the set of all uninitialized types that may be created in method μ. It is useful to define this set stepwise by defining $\mathbb{U}_\mu^\tau = \{\tau^{(h)} \mid B_\mu(h) = h$: `new` $\tau\}$ and $\mathbb{U}_\mu^{\sqsubseteq\tau} = \bigcup_{\sigma \sqsubseteq \tau} \mathbb{U}_\mu^\sigma$. Then, $\mathbb{U}_\mu = \mathbb{U}_\mu^{\sqsubseteq \texttt{Object}}$.

Note that $\mathbb{E}_3$ is the first set of errors that differs substantially from the corresponding $\mathbb{T}_3$, since it contains the new type $\tau^{(*)}$ instead of all $\tau^{(h)}$'s in $\mathbb{U}$. This is due to the fact that an error of the form $\tau^{(h)}$ does not make sense, since an `init` τ instruction may accept an uninitialized version of τ created by any h' : `new` τ instruction in the method.

The partial order $\sqsubseteq_3$ is the smallest reflexive, transitive and antisymmetric relation such that $(\forall \tau_1, \tau_2 \in \mathbb{T}_3)$, $\tau_1 \sqsubseteq_2 \tau_2 \implies \tau_1 \sqsubseteq_3 \tau_2$ and

$$(\forall \tau^{(h)} \in \mathbb{U}) \quad \bot \sqsubset_3 \tau^{(h)} \sqsubset_3 \texttt{addr},$$
$$\texttt{Object} \sqsubset_3 \texttt{addr} \sqsubset_3 \top.$$

$$\mathsf{next}[\![h\colon \texttt{new}\ \tau]\!]\langle M, \sigma s\rangle = \langle M, \tau^{(h)} s\rangle$$
$$\mathsf{next}[\![\texttt{init}\ \tau]\!]\langle M, \sigma s\rangle = \langle \dot{\mathcal{I}}^{\tau}(\sigma, M), \dot{\mathcal{I}}^{\tau}(\sigma, s)\top\rangle$$

$$\mathsf{error}[\![h\colon \texttt{new}\ \tau]\!]\langle M, s\rangle = \emptyset$$
$$\mathsf{error}[\![\texttt{init}\ \tau]\!]\langle M, \sigma s\rangle = \mathsf{check}^{*}(\sigma, \tau)$$

$$\mathscr{F}[\![h\colon \texttt{new}\ \tau]\!] = \{\lambda\sigma.\sigma,\ \lambda\sigma.\tau^{(h)}\}$$
$$\mathscr{F}[\![\texttt{init}\ \tau]\!] = \{\lambda\sigma.\top,\ \mathcal{I}^{\tau}\}$$

Fig. 16. Functions $\mathsf{next}[\![I]\!]$, $\mathsf{error}[\![I]\!]$ and $\mathscr{F}[\![I]\!]$ for $I \in \mathbb{I}_3 \setminus \mathbb{I}_2$.

It is easy to prove that Theorem 4.5 and Proposition 4.6 are also valid for $\langle \mathbb{T}_3; \sqsubseteq_3\rangle$. Figure 16 shows function $\mathsf{next}[\![\bullet]\!]$, $\mathsf{error}[\![\bullet]\!]$ and $\mathscr{F}[\![\bullet]\!]$ for the new instructions.

Functions $\dot{\mathcal{I}}^{\tau}$ in the definition of $\mathsf{next}[\![\texttt{init}\ \tau]\!]$ in Figure 16 are the pointwise extensions to memories and stacks of function $\mathcal{I}^{\tau}\colon \mathbb{T}\times\mathbb{T}\to\mathbb{T}$, defined as

$$\mathcal{I}^{\tau}(\sigma,\delta) = \begin{cases} \tau & \text{if } \sigma = \delta = \tau^{(h)} \in \mathbb{U}^{\tau}_{\mu}, \\ \delta & \text{otherwise.} \end{cases} \tag{23}$$

The purpose of these functions is to model the effect of the $\texttt{init}\ \tau$ instruction on a single item of the context: the item must be compared with the stack top and, if they are equal to the same uninitialized version of τ, the item must be converted into τ (thus modeling initialization). Functions $\mathcal{I}^{\tau}$ are monotone in both their arguments. Thus they do not alter the monotonicity of next and, more generally, all the previous theorems and propositions. Unlike all other functions introduced so far, they are functions of two arguments. However, everything still works, with the appropriate modifications.

Function $\mathsf{check}^{*}\colon \mathbb{T}\times\mathbb{T}\to\wp(\mathbb{E})$ is defined as

$$\mathsf{check}^{*}(\sigma,\tau) = \begin{cases} \emptyset & \text{if } \exists h > 0, \sigma \sqsubseteq \tau^{(h)}, \\ \{\tau^{(*)}\} & \text{otherwise.} \end{cases} \tag{24}$$

This function reflects the fact that an $\texttt{init}\ \tau$ instruction may accept any $\tau^{(h)}$ type.

4.3.1 *Completeness in the Presence of Object Creation.* We now repeat steps 1 and 2, introduced in Section 4.1, for language $\mathbb{I}_3$. We choose a family $\{E_i\}_{i\in I}$ of sets of errors that covers $\mathbb{E}_3$. If method μ contains a $h\colon \texttt{new}\ \bar{\tau}$ instruction, then at least one E_i set will contain $\tau^{(*)}$. Without loss of generality, let us assume that there exists $j \in I$ such that $E_j = \{\bar{\tau}^{(*)}\}$ and let us focus on this set only. The closure operator on $\wp(\mathbb{E})$ that checks for absence of errors of type $\bar{\tau}^{(*)}$ is $\eta_{\bar{\tau}^{(*)}} = \lambda Q.Q \cup \overline{\{\bar{\tau}^{(*)}\}}$. Using arguments similar to those used in Section 4.1.1, it is easy to see that, for any $\bar{\tau} \in \mathcal{C}'$, any complete abstraction of $\mathbb{V}$ wrt error and $\eta_{\bar{\tau}^{(*)}}$ must contain all $\bar{\tau}^{(h)}$'s in $\mathbb{U}^{\bar{\tau}}_{\mu}$. This is also a sufficient condition for completeness wrt error and $\eta_{\bar{\tau}^{(*)}}$. More precisely, if we let $\mathscr{V}_{\mu,\tau} = \mathscr{M}(\mathbb{U}^{\tau}_{\mu}) = \mathbb{U}^{\tau}_{\mu} \cup \{\top, \bot\}$, $f_{\tau} = \lambda\sigma.\,\mathsf{check}^{*}(\sigma,\tau)$ and $B = \eta_{\bar{\tau}^{(*)}}(\wp(\mathbb{E}))$, then $\mathscr{V}_{\mu,\bar{\tau}} = \mathscr{S}^{B}_{f_{\bar{\tau}}}(\{\top\})$. Intuitively, the knowledge of where all uninitialized versions of $\bar{\tau}$ can be found

in a vector v, is necessary and sufficient to check all $\mathsf{error}[\![\texttt{init}\ \bar{\tau}]\!]$ conditions in the method. This completes step 1.

For step 2, we have to find the absolute complete shell of any abstraction P of $\mathbb{T}_3$ wrt $\mathscr{F}_\mu$. The interesting case is when P contains a class type $\bar{\tau}$ and the method contains an `init` $\bar{\sigma}$ instruction, with $\bar{\sigma} \sqsubseteq \bar{\tau}$. In this case, $\mathscr{S}_{\mathscr{F}_\mu}(P)$ must contain all uninitialized versions of $\bar{\sigma}$ (taken from $\mathbb{U}_\mu^{\bar{\sigma}}$). We call such domain $\mathscr{U}_{\mu,\sqsubseteq\bar{\tau}}$ where

$$\mathscr{U}_{\mu,\sqsubseteq\tau} = \mathscr{M}\left(\mathbb{U}_\mu^{\sqsubseteq\tau}\right).$$

The intuition behind this is the same as for array types: we must keep track of all $\bar{\sigma}^{(h)}$'s, since function $\mathcal{I}^{\bar{\sigma}}$ may produce new $\{\bar{\tau}\}$-assignable types from them.

We show that all types in $\mathscr{U}_{\sqsubseteq\bar{\tau}}$ (μ subscript omitted for simplicity) are needed by means of a simple example, which can be easily generalized. Let $P = \{\bar{\tau}, \top\}$, for some $\bar{\tau} \in \mathcal{C}'$, and assume that $\mathscr{F}_\mu$ contains $\mathcal{I}^{\bar{\sigma}}$ only, with $\bar{\sigma} \sqsubseteq \bar{\tau}$. According to Section 4.1.3, we have to compute $\mathscr{S}_{\mathscr{F}_\mu}(P)$ using equation (7). The arity of $\mathcal{I}^{\bar{\sigma}}$ is 2 and, according to the last paragraph of Section 2.3, $\mathscr{R}_{\mathscr{F}_\mu}$ becomes

$$\mathscr{R}_{\mathscr{F}_\mu}(A) = \bigcup_{\sigma\in A}\left(\bigcup_{\delta'\in\mathbb{T}} \max\{\tau \in \mathbb{T} \mid \mathcal{I}^{\bar{\sigma}}(\tau,\delta') \sqsubseteq \sigma\} \cup \bigcup_{\delta''\in\mathbb{T}} \max\{\tau \in \mathbb{T} \mid \mathcal{I}^{\bar{\sigma}}(\delta'',\tau) \sqsubseteq \sigma\}\right).$$

The fixpoint iteration starts at $A = \{\top\}$ and we have that $\mathscr{R}_{\mathscr{F}_\mu}(\{\top\}) = \{\top\}$. Thus, the first iterate of the fixpoint calculation is $S^{(0)} = P$. Next, we have to calculate $\mathscr{R}_{\mathscr{F}_\mu}(P)$. When we take $\sigma = \top$, the above formula gives $\top$. When we take $\sigma = \bar{\tau}$, we have the following cases, for the first maximal in the expression.

—$(\forall h)\ \delta' \neq \bar{\sigma}^{(h)}$. Then $\mathcal{I}^{\bar{\sigma}}(\tau,\delta') = \delta'$ and either the first maximal does not exist (if $\delta' \not\sqsubseteq \bar{\tau}$), or it is $\top$.

—$(\exists h)\ \delta' = \bar{\sigma}^{(h)}$. Then, $\mathcal{I}^{\bar{\sigma}}(\tau,\bar{\sigma}^{(h)})$ gives $\bar{\sigma} \sqsubseteq \bar{\tau}$ if $\tau = \bar{\sigma}^{(h)}$, and $\bar{\sigma}^{(h)} \not\sqsubseteq \bar{\tau}$ otherwise. Thus, the first maximal is $\bar{\sigma}^{(h)}$.

For the second maximal in the expression, we have the following cases.

—$(\forall h)\ \delta'' \neq \bar{\sigma}^{(h)}$. Then $\mathcal{I}^{\bar{\sigma}}(\delta'',\tau) = \tau$ and the second maximal is $\bar{\tau}$.

—$(\exists h)\ \delta'' = \bar{\sigma}^{(h)}$. Then, $\mathcal{I}^{\bar{\sigma}}(\bar{\sigma}^{(h)},\tau)$ gives $\bar{\sigma}$ if $\tau = \bar{\sigma}^{(h)}$, and τ otherwise. Thus, the second maximal is either $\bar{\sigma}^{(h)}$ or $\bar{\tau}$.

Summarizing, we have that $\mathscr{R}_{\mathscr{F}_\mu}(P)$ contains $\top$, $\bar{\tau}$ and all uninitialized versions of all types $\bar{\sigma}$ assignable to $\bar{\tau}$. Then, $S^{(1)}$ will add $\bot$ (because of the Moore closure) and a further iteration will add no more types, signaling that the fixpoint has been reached.

4.3.2 *Instance Initialization Methods.* When the method μ being analyzed is an instance initialization method $\bar{\tau}$.`<init>` (constructor from now on), some additional constraints must be checked [Lindholm and Yellin 1999, Section 4.9.4]. Let us assume that $\bar{\tau}$ *extends* $\hat{\tau}$ (i.e., $\hat{\tau}$ is the direct base class of class $\bar{\tau}$). At the beginning of the analysis, register 0 contains a special

uninitialized type $\bar{\tau}^{(0)}$. Before returning normally, the method must initialize $\bar{\tau}^{(0)}$ by invoking either another constructor of class $\bar{\tau}$, or a constructor of class $\hat{\tau}$. Until either constructor is called, only assignments to instance fields declared in class $\bar{\tau}$ (and not, for example, in a super class of $\bar{\tau}$) are allowed on $\bar{\tau}^{(0)}$. Once $\bar{\tau}^{(0)}$ has been initialized, all its occurrences in the current context are transformed into type $\bar{\tau}$.

Here, we discuss how we can deal with these constraints in our framework, without going into full details. The semantics of the verifier must be refined to encode the additional constraints. Type $\bar{\tau}^{(0)}$ must be added to the type hierarchy, with $\bot \sqsubset \bar{\tau}^{(0)} \sqsubset \texttt{addr}$. Contexts must be extended with a flag that remembers whether the proper constructor has been called on $\bar{\tau}^{(0)}$. More formally, we introduce $\mathbb{C}'' = \{\bot_{\bar{\tau}}, \top_{\bar{\tau}}\} \times \mathbb{C}'$, with $\bot_{\bar{\tau}} \leq \bot_{\bar{\tau}} < \top_{\bar{\tau}} \leq \top_{\bar{\tau}}$, ordered pointwise. The flag is the first component of the context, with $\top_{\bar{\tau}}$ representing a state where the proper constructor has not yet been called, and $\bot_{\bar{\tau}}$ representing the opposite situation. Note that, if a context coming from a path where the constructor has not been called (flag set to $\top_{\bar{\tau}}$) joins a path where the constructor has been called (flag set to $\bot_{\bar{\tau}}$), the result of the merge operation on the flag is $\top_{\bar{\tau}} \vee \bot_{\bar{\tau}} = \top_{\bar{\tau}}$, correctly encoding the fact the constructor has not been called in all paths. Semantic function $\mathsf{next}[\![\bullet]\!]$ must be modified for instructions `start` and `init`. Instruction `start` must set the flag to $\top_{\bar{\tau}}$, and register 0 to $\bar{\tau}^{(0)}$. Instruction `init` τ must behave differently whenever $\tau \in \{\bar{\tau}, \hat{\tau}\}$ and the top of the stack is $\bar{\tau}^{(0)}$. In this case, all occurrences of $\bar{\tau}^{(0)}$ must be transformed into $\bar{\tau}$ and the flag must be set to $\bot_{\bar{\tau}}$. All other instructions must leave the flag unaltered. The semantic function $\mathsf{error}[\![\bullet]\!]$ must be modified for instructions `init`, `putfield` and `return`. Besides their normal behavior, instruction `init` τ must also accept $\bar{\tau}^{(0)}$ when $\tau \in \{\bar{\tau}, \hat{\tau}\}$ and instruction `putfield` $\bar{\tau}.f : \tau'$ must also accept $\bar{\tau}^{(0)}$ (if f is declared in class $\bar{\tau}$). Finally, instruction `return` must check that the flag is set to $\bot_{\bar{\tau}}$, returning a $\bot_{\bar{\tau}}$ error otherwise. A $\bot_{\bar{\tau}}$ errors signals that the method may reach a `return` without calling the proper constructor on the object.

When this changes are in place, we can apply our theory and obtain the following results:

—to check for absence of the new error $\bot_{\bar{\tau}}$, it is sufficient to remember the value of the flag and use $\mathscr{B}_{\bar{\tau}^{(0)}} = \{\bar{\tau}^{(0)}, \top\}$ for registers and stack items;

—because of the modifications to instruction `init`, type $\bar{\tau}^{(0)}$ belongs to the complete shells of both $\mathscr{B}_{\bar{\tau}}$ and $\mathscr{B}_{\hat{\tau}}$.

In summary, the pass that checks for absence of errors on type $\bar{\tau}$ must also track type $\bar{\tau}^{(0)}$. If, during this pass, we also remember the value of the flag, we can use this pass to check for absence of error $\bot_{\bar{\tau}}$.

4.4 Subroutines

The bytecode instructions that deal with subroutines are `jsr` and `ret`. Subroutines are used by the Java compiler to translate `try...finally` statements of the Java language: since the `finally` block may be reached from several points in the method (e.g., normal or exceptional exit from the corresponding `try` block), it is translated only once and reached through the use of the `jsr`

$$\mathsf{next}[\![\texttt{jsr}\ k]\!]\langle M, s\sigma\rangle = \langle \dot{\mathcal{J}}(M), \texttt{raddr}\ \dot{\mathcal{J}}(s)\rangle$$
$$\mathsf{next}[\![\texttt{ret}\ x]\!]\langle M, s\rangle = \langle \dot{\mathcal{J}}(M), \dot{\mathcal{J}}(s)\rangle$$

$$\mathsf{error}[\![\texttt{jsr}\ k]\!]\langle M, s\rangle = \emptyset$$
$$\mathsf{error}[\![\texttt{ret}\ x]\!]\langle M, vs\rangle = \mathsf{check}(M(x), \texttt{raddr})$$

$$\mathscr{F}[\![\texttt{jsr}\ k]\!] = \{\mathcal{J},\ \lambda\sigma.\texttt{raddr},\ \lambda\sigma.\sigma\}$$
$$\mathscr{F}[\![\texttt{ret}\ x]\!] = \{\lambda\sigma.\sigma\}$$

Fig. 17. Functions $\mathsf{next}[\![I]\!]$, $\mathsf{error}[\![I]\!]$ and $\mathscr{F}[\![I]\!]$ for $I \in \mathbb{I}_4 \setminus \mathbb{I}_3$.

instruction. A `jsr` k instruction at label h saves the address $h+1$ of the following instruction on the stack, then jumps at instruction k. The subroutine can store label $h+1$ in a register x using an `astore` x instruction, and then resume execution from instruction at label $h+1$ using a `ret` x instruction.

To model these new instructions, the lattice of types must include a new type `raddr`, which is the type of the return address saved on the stack by the `jsr` instruction and checked by the `ret` instruction. Moreover, since a `raddr` may be used by an `astore` instruction, but not by an `aload` instruction, we must assume the existence of a type `laddr` (loadable address) to which both `Object` and uninitialized types can be assigned, but which is unrelated to `raddr`. Then, we assume that `aload` stands for `laddrload`, while `astore` stands for `addrstore` as before.

Thus we have:

$$\mathbb{T}_4 \ = \ \mathbb{T}_3 \cup \{\texttt{raddr}, \texttt{laddr}\}, \tag{25}$$
$$\mathbb{E}_4 \ = \ \mathbb{E}_3 \cup \{\texttt{raddr}, \texttt{laddr}\}. \tag{26}$$

The partial order $\sqsubseteq_4$ is the smallest reflexive, transitive, and antisymmetric relation such that $(\forall \tau_1, \tau_2 \in \mathbb{T}_4)\ \tau_1 \sqsubseteq_3 \tau_2 \implies \tau_1 \sqsubseteq_4 \tau_2$ and

$$\begin{aligned} & \bot \sqsubseteq \texttt{raddr} \sqsubset \texttt{addr}, \\ (\forall \tau^{(h)} \in \mathbb{U}_\mu) \quad & \bot \sqsubset_4 \tau^{(h)} \sqsubset_4 \texttt{laddr}, \\ & \texttt{Object} \sqsubset_4 \texttt{laddr} \sqsubset_3 \texttt{addr}. \end{aligned}$$

Subroutine verification has been extensively studied in the literature (see, for instance, the related section of Coglio [2004] for a complete state of the art). Here we refer mostly to the implementation suggested by Sun. Figure 17 shows functions $\mathsf{next}[\![\bullet]\!]$, $\mathsf{error}[\![\bullet]\!]$ and $\mathscr{F}[\![\bullet]\!]$ for `jsr` and `ret` instructions. These new instructions also require modifications to the definition of the control flow graph. The modification is obvious for `jsr` k. Instruction `ret` x, instead, is more difficult. Following Sun, we make the simple assumption that $h \rightsquigarrow k+1$ for all $k \in \texttt{callers}(\texttt{subr}(h))$, whenever $B(h) = \texttt{ret}\ x$. The purpose of function $\texttt{subr} : \mathbb{L} \to \mathbb{L}$ is to map each `ret` x instruction to a subroutine entry point k, as specified in some `jsr` k instruction, also found in the method. Function `subr` has to be calculated using a separate data flow analysis. Then, function $\texttt{callers}\colon \mathbb{L} \to \wp(\mathbb{L})$ is simply defined as $\texttt{callers}(k) = \{h \in \mathbb{L} \mid B(h) = \texttt{jsr}\ k\}$. The effect of this assumption is that all call sites of a given subroutine are considered successors of the same `ret` instruction. The new function $\dot{\mathcal{J}}$ is the

pointwise extension to $\mathbb{M}$ and $\mathbb{S}$ of function $\mathcal{J}: \mathbb{T} \to \mathbb{T}$ defined as

$$\mathcal{J}(\tau) = \begin{cases} \top & \text{if } \tau \in \mathbb{U}_\mu, \\ \tau & \text{otherwise.} \end{cases}$$

The purpose of this function is to (conservatively) solve the problem pointed out by Freund and Mitchell [1999].

Note that $\mathcal{J}$ is a new nonconstant, nonidentity function that adds to $\mathscr{T}_\mu$. However, unlike $[^{-1}$ and $\mathcal{I}^\tau$, this new function is "benign," since it does not cause types to be added to any complete shell. In fact, for any $D \subseteq \mathbb{T}$ and $\sigma \in D$, $\max\{\tau \mid \mathcal{J}(\tau) \sqsubseteq \sigma\}$ is either σ itself (if $\sigma \notin \mathbb{U}_\mu$), $\top$ (if $\sigma = \top$) or does not exist.

Another problem with subroutines is related to the join of the contexts at the instructions that follow a `jsr`. Due to the simple assumption made in the construction of the control flow graph, contexts coming from different call sites of the same subroutine get joined at all possible call sites of that subroutine. This approximation is too rough and causes code actually generated by the Java compiler to be rejected. The solution proposed by Sun is to treat as a special case the join of contexts for instructions that follow a `jsr`: the state of registers used by the subroutine is taken by the context generated by the `ret` instruction, while the state of registers not used by the subroutine is taken by the context of the `jsr` instruction. This strategy is implemented as follows. First, for each subroutine k, a set $\texttt{used}(k) \subseteq [0, r_\mu)$ is built. This set contains the names of all registers read or written by any instruction belonging to subroutine k. Assume subroutine k is exited by a `ret` instruction at label w and that $\bar{h} \in \texttt{callers}(k)$. Then:

$$\mathsf{next}(v)(\bar{h}+1) = \left(\bigsqcup_{\substack{h \rightsquigarrow \bar{h}+1 \\ h \neq w}} \mathsf{next}[\![B(h)]\!]v(h) \right) \sqcup (\mathsf{next}[\![B(\bar{h})]\!]v(\bar{h}) \sqcup_k \mathsf{next}[\![B(w)]\!]v(w)),$$

where:

$$(M_1, s_1) \sqcup_k (M_2, s_2) = (M_1 \sqcup_k M_2, s_2),$$
$$M_1 \sqcup_k M_2 = \lambda x \in [0, r). \begin{cases} M_1(x) & \text{if } x \notin \texttt{used}(k), \\ M_2(x) & \text{otherwise.} \end{cases}$$

It should be clear that, for our purposes, this algorithm only affects the $\twoheadrightarrow$ relation, and, thus, has no influence on our proposed decompositions.

Example 4.7. Consider a method with $r_\mu = 4$ and $t_\mu = 2$ and $n = r_\mu + t_\mu = 6$. Assume the method contains a subroutine, located at instruction k, and called at instruction $\bar{h}$. Moreover, suppose the subroutine ends at instruction w and modifies only register 0 and 2. Then, the relation $\twoheadrightarrow$ is such that (see Figure 18):

—for $i \in [r, n)$, $wn + i \twoheadrightarrow (\bar{h}+1)n + i$, (the stack comes from the subroutine),
—for $i \in \texttt{used}(k)$, $wn + i \twoheadrightarrow (\bar{h}+1)n + i$ (the registers used by the subroutine affect the instruction following the `jsr`, and
—for $i \notin \texttt{used}(k)$ and $i \in [0, r)$, $\bar{h}n + i \twoheadrightarrow (\bar{h}+1)n + i$ (the register not touched by the subroutine are taken from the `jsr` instruction).

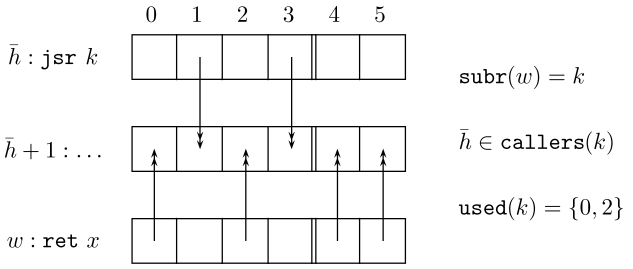

Fig. 18. How the relation $\twoheadrightarrow_\mu$ describes registers propagation across method μ and subroutine located at instruction k. The next_{ij} functions, all equal to $\lambda\sigma.\sigma$, are omitted.

4.5 Other Features of JVML

Other features of JVML have been left out of our discussion, mainly because they are largely orthogonal to our proposed decomposition.

The first feature is interface types. For our purposes, interfaces are an additional set of user defined types, similar to classes. Interfaces are arranged in an *extends* acyclic relation which is separated from the class tree (apart from common derivation of classes and interfaces from `Object`). The connection between classes and interfaces is introduced by a distinct *implements* relation. Each class may implement any number of interfaces. The $\sqsubseteq$ order relation includes both the class and interface *extends* relations, and the *implements* relation. This means that the inclusion of the set of types, ordered by $\sqsubseteq$, in a lattice is generally no longer trivial. For example, when two unrelated classes both implement two unrelated interfaces there is no least upper bound ($\sqcup$) for the two classes. This is a problem in standard verification, since $\sqcup$ is used whenever two contexts need to be merged. For this reason, the Sun implementation of the bytecode verifier ignores type constraints on interface types. These constraints, instead, are checked at runtime by the bytecode interpreter. In our proposed verification we do the same. However, we discuss some alternatives in the Related Work section.

The second feature is the JVM exception handling mechanism. Exceptions are represented by objects derived from the predefined `Throwable` class. Exceptions are thrown either explicitly (through the `athrow` bytecode instruction) or implicitly by the JVM itself. This causes a transfer of control to a matching exception handler entry (an index in the bytecode array). Each exception handler entry is mapped to a specific type of exception, and a specific range of bytecode instructions (the active range of the exception handler entry). This mapping is implemented in an exception table for each method. As far as bytecode verification is concerned, each exception handler entry must be considered as a possible successor of each instruction in its active range, thus causing a modification of the Control Flow Graph. Again, this does not affect our proposed decomposition.

Finally, not all bytecode instructions in JVML are listed in Figure 1. However, all missing instructions do not add any nonconstant nonidentity function to $\mathscr{F}_\mu$, so they have been omitted for simplicity. Note that type conversion instructions like `i2f` or `checkcast` contain the target type in their opcode, so their effect can be modeled using constant functions.

4.6 Implementation

In this section we present some implementation guidelines for multipass verification. The implementation of the multipass algorithm for the verification of a method μ proceeds as follows.

(1) Structural constraints and stack safeness are checked (Section 3.5).
(2) If subroutines are used, tables implementing `subr` and `used` are built (Section 4.4).
(3) The set $\mathscr{E}_\mu \subseteq \mathbb{E}$ of types that must be checked is computed. This amounts to evaluating $\mathscr{E}_\mu = \bigcup_{h \in \mathbb{I}_{0,\mu}} rng(\mathsf{error}[\![B_\mu(h)]\!])$, and can be performed in a single pass over the bytecode of the method.
(4) For each type τ in $\mathscr{E}_\mu$, the complete shell $S_\tau = \mathscr{S}_{\mathscr{T}_\mu}(\mathscr{B}_\tau)$ is computed and the verification algorithm is performed using abstraction S_τ.

The verification algorithm is performed using a standard chaotic fixpoint iteration [Qian 2000] and a dictionary of contexts. The dictionary only stores contexts for instructions that are targets of a branch instruction (i.e., `goto` or `if`*cond*) [Leroy 2001].

In step 3, S_τ may be different from $\{\top, \tau\}$ only when $\texttt{null} \sqsubset \tau \sqsubseteq \texttt{Object}$. In this case, the following steps are sufficient to determine S_τ:

(1) if the method contains an `aaload` instruction, find $\kappa_{\mu,\tau}$ (see Section 4.2.1);
(2) find all h: `new` σ instructions, where $\sigma \sqsubseteq \tau$ (see Section 4.3.1).

When S_τ contains both arrays of τ and uninitialized versions of types assignable to τ, we represent a type $\sigma \in S_\tau$ using a two-fields representation: the first field, with a fixed size of two bits, determines four kinds of types (e.g., "11" for $\top$, "01" for array, "10" for uninitialized type, "00" for `null`); the second field contains, depending on the value of the first field, the dimension of the array (0 for type τ itself), the index in the method bytecode of the corresponding `new` instruction for an uninitialized type, or an undefined value. The size of the second field depends on the method being verified and can be calculated during step 3. There is no need to have a representation for $\bot$, since this type never appears. The two-fields representation wastes some space, but allows for an easy implementation of all operations that must be performed on types.

5. SERIAL DECOMPOSITION

Parallel decomposition gives a satisfactory solution to the problem of simplifying the analysis of programs written in $\mathbb{I}_1$. However, this is due to the extreme simplicity of the characteristic functions involved in $\mathbb{I}_1$: if we are verifying a method μ, we only focus on a subset E of types in $\mathbb{T}$, and, for a given context vector v, we know precisely where all E-assignable types can be found, then we can obtain the same precise knowledge on $\mathsf{next}(v)$. For more complex instruction sets, however, we can obtain only partial information; that is, $\mathsf{next}(v)$ may contain some E-assignable types which we miss. As we have seen, this is possible if some characteristic function in $\mathscr{T}_\mu$ may produce an E-assignable type from a non-E-assignable type. We would miss this new type, since we

have no knowledge of where non-E-assignable types can be found in v. This partial knowledge on $\mathsf{next}(v)$ implies loss of precision in type checking. In fact, for some instruction of μ that requires an E-assignable type, we may not be able to prove that it can be correctly executed in $\mathsf{next}(v)$. Thus, we would have to conservatively reject the method, even though it might be correct.

In Sections 4.2.1 and 4.3.1 this problem has been solved by enlarging the set of types that we must keep track of at the same time. In fact, the purpose of the complete shell computation is to include the set E' of types that may generate (through functions in $\mathscr{T}_\mu$) E-assignable types, then the set E'' of types that may generate E'-assignable types, and so on. However, keeping track of several types at the same time requires more space. An alternative solution is to compute $\mathsf{next}(v)$ considering only types in E'', but remembering where new E'-assignable types are generated, then use this information to compute $\mathsf{next}(v)$ for types in E', and so on.

Let us consider, for example, a class `C` and the corresponding array type `[C`, which both appear in the method we are verifying. Assume that no $[^k$`C` with $k > 1$ is used in the method, and the method contains no `init C` instruction. We know from Section 4.2.1 that, if the method contains an `aaload` instruction, both `C` and `[C` must be analyzed together using abstraction $\mathscr{A}^1_{\mathtt{C}} = \{$`[C`, `C`, `null`, $\top\}$, otherwise completeness is lost. This is because the `aaload` instruction brings function $[^{-1}$ into the set of functions used in the method, and this function may produce a `C` from a `[C`. The idea developed in this section is to perform an initial analysis using type `[C` only (i.e., compute the fixpoint of next using abstraction $\{$`[C`, $\top\}$). This analysis is used to discover which `aaload` instructions have a `[C` type on the proper stack element of their fixpoint before context. Then, this information is used to replace $[^{-1}$ functions with constant functions, each of which either produces `C` or $\top$. After this replacement, abstraction $\{$`C`, $\top\}$ becomes (fixpoint) complete. The effect is that we have decomposed abstraction $\{$`[C`, `C`, `null`, $\top\}$ into abstractions $\{$`[C`, $\top\}$ and $\{$`C`, $\top\}$. However, this decomposition is serial rather than parallel, since information obtained in the former is used to make the latter possible (without losing completeness). The crucial property that makes the serial decomposition possible is that $\{$`[C`, $\top\}$ is complete wrt $\{$`C`, $\top\}$ and $[^{-1}$ (as shown in Section 5.1 below, Proposition 5.4), that is, knowing whether a type τ is assignable to `[C` is all that is needed for knowing whether $[^{-1}\tau$ is assignable to `C`.

Let us consider type `C` again, but assume now that instructions h: `new C` and `init C` are found in the method. Assume that the method contains no other h': `new` τ instruction, with τ assignable to `C`, and either no array of `C`, or no `aaload` instruction. Recall from Section 4.3.1 that a complete abstraction for this method is $\mathscr{M}(\{\mathtt{C}\} \cup \mathscr{V}_{\mu,\sqsubseteq\mathtt{C}}) = \{\mathtt{C}, \mathtt{C}^{(h)}, \top, \bot\}$, since instruction `init C` may produce `C` from $\mathtt{C}^{(h)}$ using function $\mathcal{I}^{\mathtt{C}}$. Applying the idea of serial decomposition, we could perform an initial analysis using abstraction $\mathscr{B}_{\mathtt{C}^{(h)}} = \{\mathtt{C}^{(h)}, \top\}$ and discover which instances of function $\mathcal{I}^{\mathtt{C}}$ (there is a distinct instance for each element of the before context of the `init C` instruction) produce a `C` type. Unfortunately, abstraction $\{\mathtt{C}^{(h)}, \top\}$ is *not* complete wrt $\{$`C`, $\top\}$ and $\mathcal{I}^{\mathtt{C}}$. In fact, since $\mathcal{I}^{\mathtt{C}}(\sigma, \tau)$ produces τ whenever $\sigma \neq \tau$ or $\sigma \neq \mathtt{C}^{(h)}$, then $\mathcal{I}^{\mathtt{C}}(\sigma, \mathtt{C}) = \mathtt{C}$. Since `C` is unknown in abstraction $\{\mathtt{C}^{(h)}, \top\}$, there is no way to know if $\tau = \mathtt{C}$, so this

abstraction may miss some $\mathtt{C}$'s produced by some $\mathcal{I}^{\mathtt{C}}(\sigma, \tau)$ function. In other words, abstraction $\{\mathtt{C}^{(h)}, \top\}$ is able to predict *some* of the necessary information on the output of $\mathcal{I}^{\mathtt{C}}$ functions, but not all. Thus, functions $\mathcal{I}^{\mathtt{C}}$ cannot be replaced by constant functions. To overcome this problem, we define the *residual* of a semantic function f with respect to a given abstraction ρ of its domain. The idea is that the residual function gives all that is missed of the output of f, after its abstraction through ρ.

Definition 5.1 *Residual.* Let $\langle C; \leq \rangle$ and $\langle D; \sqsubseteq \rangle$ be complete lattices, $\rho \in uco(C)$ and $f : C \to D$. We call a *residual* of f wrt ρ any function $g : C \to D$ such that $g \sqcap (f \circ \rho) = f$.

In our example, f is $\varphi_{\mathscr{B}_{\mathtt{C}}} \circ \mathcal{I}^{\mathtt{C}}$ and ρ is (the closure operator corresponding to) abstraction $\{\mathtt{C}^{(h)}, \top\} \times \{\mathtt{C}^{(h)}, \top\}$ (the Cartesian product is necessary, since $\mathcal{I}^{\mathtt{C}}$ is a function of a pair of types). Note that, if $f \sqsubseteq f \circ \rho$ (and, in particular, if f is monotone), there always exists at least one residual of f wrt ρ, namely f itself. If $f = f \circ \rho$, we can take $g = \lambda c \in C.\top_D$ as a residual of f. This reflects the intuition that, if ρ keeps all information that is necessary to compute f, then any residual g of f wrt ρ does not need to give any additional information.

Now, instead of replacing each function $\mathcal{I}^{\mathtt{C}}$ with a constant function, the idea is that we can replace them with the meet of a constant and their residual function. The constant is discovered using abstraction $\{\mathtt{C}^{(h)}, \top\}$, while the residual function gives the rest of the information produced by $\mathcal{I}^{\mathtt{C}}$ which cannot be discovered by the abstraction. Recall that we want to replace $\mathcal{I}^{\mathtt{C}}$ functions because they make abstraction $\{\mathtt{C}, \top\}$ incomplete. In the case of arrays, we are able to replace functions $[^{-1}$'s with constant functions. This is certainly better, since all abstractions are complete wrt constant functions. However in the case of $\mathcal{I}^{\mathtt{C}}$, the replacement introduces other non constant functions. Nevertheless, the replacement is still useful, if abstraction $\{\mathtt{C}, \top\}$ is complete for the residual function.

In the following theorem, we show how information obtained from an abstract interpretation φ_B can be used to replace functions in $\mathscr{F}_\mu$, to be used in a different abstract interpretation φ_A.

THEOREM 5.2 SERIAL DECOMPOSITION. *Let $A, B \in abs(\mathbb{T})$, $F = \{f_h\}_{h \in H} \subseteq \mathscr{F}_\mu$ and $G = \{g_h\}_{h \in H}$ with g_h a monotone residual of $\varphi_A \circ f_h$ wrt φ_B, $\forall h \in H$. Let $\mathcal{L} = lfp(\mathsf{next}_\mu)$ and define method μ' using next'_{ij} where*

$$\mathsf{next}'_{ij} = \begin{cases} \lambda\tau.g_h(\tau) \sqcap \varphi_A(f_h(\varphi_B(\mathcal{L}[i]))) & \text{if } \mathsf{next}_{ij} = f_h \in F, \\ \mathsf{next}_{ij} & \text{otherwise.} \end{cases}$$

If $\langle \varphi_A, \varphi_A \rangle$ is complete for $(\mathscr{F}_\mu \setminus F) \cup G$ and φ_A is join-distributive, then

$$lfp(\dot{\varphi}_A \circ \mathsf{next}_{\mu'}) = \dot{\varphi}_A(lfp(\mathsf{next}_\mu)).$$

PROOF SKETCH. [2] First, we show that $\dot{\varphi}_A(\mathcal{L})$ is a fixpoint of $\dot{\varphi}_A \circ \mathsf{next}_{\mu'}$ and thus $lfp(\dot{\varphi}_A \circ \mathsf{next}_{\mu'}) \sqsubseteq \dot{\varphi}_A(\mathcal{L})$. Then, we show that the reverse inequality also

[2] See the companion technical report [Bernardeschi et al. 2007] for a full proof.

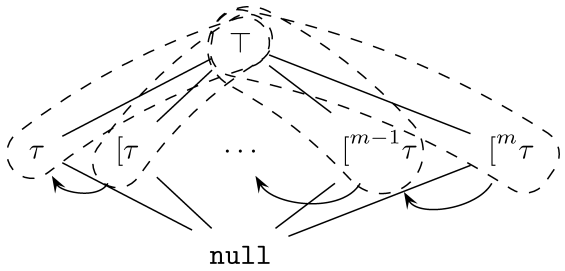

Fig. 19. Serial decomposition in the presence of array types.

holds, by showing that each iterate of the Kleene sequence for next_μ is $\sqsubseteq$ of the corresponding iterate for $\mathsf{next}_{\mu'}$. □

In the definition of method μ', we replace each function f_h in F with a function built from the meet of the residual g_h of $\varphi_A \circ f_h$, and a constant which is the value that $\varphi_A \circ f_h$ would assume when evaluated on the proper element of $\varphi_B(\mathcal{L})$. An important special case is when $\langle \varphi_B, \varphi_A \rangle$ is complete for each f_h, so that there is no need for a residual.

THEOREM 5.3. *Let $A, B \in abs(\mathbb{T})$ such that $\langle \varphi_B, \varphi_A \rangle$ is complete for $F \subseteq \mathscr{T}_\mu$, $\langle \varphi_A, \varphi_A \rangle$ is complete for $\mathscr{T}_\mu \setminus F$ and φ_A is join-distributive. Then, there exists a method μ' such that $\langle \varphi_A, \varphi_A \rangle$ is complete for $\mathscr{T}_{\mu'}$ and $\dot{\varphi}_A \circ \mathsf{next}_{\mu'}$ is fixpoint complete wrt $\dot{\varphi}_A$ and next_μ.*

PROOF. Let f be any function in F. Since $\langle \varphi_B, \varphi_A \rangle$ is complete for F, meaning that $\varphi_A \circ f \circ \varphi_B = \varphi_A \circ f$, a residual g of $\varphi_A \circ f$ wrt φ_B is simply $g = \lambda\tau.\top$. Since this is a constant function, it is monotone and $\langle \varphi_A, \varphi_A \rangle$ is trivially complete for it. Thus, we can apply Theorem 5.2, obtaining the requested method μ'. Note that the definition of next'_{ij} simplifies to

$$\mathsf{next}'_{ij} = \begin{cases} \lambda\tau.\varphi_A(\mathsf{next}_{ij}(\mathcal{L}[i])) & \text{if } \mathsf{next}_{ij} \in F; \\ \mathsf{next}_{ij} & \text{otherwise.} \end{cases}$$

The additional claim that $\langle \varphi_A, \varphi_A \rangle$ is complete for $\mathscr{T}_{\mu'}$ is trivially proved as follows: $\mathscr{T}_{\mu'}$ contains all functions in $\mathscr{T}_\mu \setminus F$, for which $\langle \varphi_A, \varphi_A \rangle$ was already complete by hypothesis, plus a set of constant functions, for which any abstraction is complete. □

5.1 Serial Decomposition in the Presence of Arrays

In this section we show how to combine Parallel and Serial Decomposition to further decompose lattice $\mathscr{A}^k_\tau$, defined at the end of Section 4.2.1, so that each separate analysis is performed using a two-element lattice, without losing completeness. We illustrate the idea in Figure 19, where $m = \kappa_{\mu,\tau}$. We decompose $\mathscr{A}^m_\tau$ into basic domains, disregarding type `null` (since it cannot generate errors). We then start from domain $\{[^m\tau, \top\}$, compute the corresponding fixpoint and remember where $[^{m-1}\tau$'s appear (if any). Finally, we proceed to domain $\{[^{m-1}\tau, \top\}$ and so on, until we reach domain $\{\tau, \top\}$.

First, we use Lemma 4.2 with $C = \mathscr{A}^m_\tau$ and $D = \wp(\mathscr{A}^m_\tau \cap \mathbb{E}) = \wp(\mathscr{A}^m_\tau \setminus \{\top, \texttt{null}\})$. Now let $L = [0, m]$, $\{D_l\}_{l \in L} = \{[^l\tau\}_{l \in L}$ and $\eta_l = \lambda Q.Q \cup \overline{D_l}$. We know from Section 4.1.1, that if we decompose $\mathscr{A}^m_\tau$ into the basic domains $P_l = \mathscr{B}_{[^l\tau} = \{[^l\tau, \top\}$, for each $l \in L$, then each pair $\langle \dot{\varphi}_{P_l}, \eta_l \rangle$ is complete wrt error. Abstraction $\dot{\varphi}_{P_l}$ is not complete for next, instead, for all $l < m$. The reason for this incompleteness is that an `aaload` instruction in the method may produce an $[^l\tau$ from an $[^{l+1}\tau$. In fact, when we abstract a context vector according to P_l, $[^{l+1}\tau$ is abstracted into $\top$, and the presence of a new $[^l\tau$, produced by the `aaload` instruction, is missed. Nonetheless, we note that Lemma 4.2 only requires fixpoint completeness wrt next, and fixpoint completeness is a weaker requirement than completeness [Giacobazzi et al. 2000].

Note that $\dot{\varphi}_{P_m}$ is complete wrt next, since $m = \max\{l \mid [^l\sigma \in \mathscr{T} \text{and } \sigma \sqsubseteq \tau\}$ and therefore type $[^{m+1}\tau$ may never appear in the verification. Therefore, we can start the analysis of $\mathscr{A}^m_\tau$ using abstraction $\dot{\varphi}_{P_m}$, and obtain fixpoint $\mathcal{L}_m = \dot{\varphi}_{P_m}(lfp(\mathsf{next}_\mu))$. Fixpoint $\mathcal{L}_m$ can be used to find all errors, if any, on type $[^m\tau$. Then, we want to use Theorem 5.3, with $A = P_{m-1}$, $B = P_m$, and $F = \{[^{-1}\}$, to compute $\mathcal{L}_{m-1} = \dot{\varphi}_{P_{m-1}}(lfp(\mathsf{next}_\mu))$. Since $\varphi_{P_{m-1}}$ is join distributive and $\langle \varphi_{P_{m-1}}, \varphi_{P_{m-1}} \rangle$ is certainly complete for $\mathscr{T}_\mu \setminus \{[^{-1}\}$ (Section 4.1.3). We then still have to prove that $\langle \varphi_{P_m}, \varphi_{P_{m-1}} \rangle$ is complete for $[^{-1}$. We prove a more general result in the following proposition.

PROPOSITION 5.4 $[^{-1}$ COMPLETENESS. *Let* $\tau \in \mathbb{T}$, *with* $\texttt{null} \sqsubset \tau \sqsubseteq \texttt{Object}$ *and let* $P_l = \mathscr{B}_{[^l\tau} = \{[^l\tau, \top\}$. *Then, for all* $l \geq 0$, $\langle \varphi_{P_{l+1}}, \varphi_{P_l} \rangle$ *is complete wrt* $[^{-1}$.

PROOF SKETCH. We will show that, for every $\sigma \in \mathbb{T}$, $\varphi_{P_l}([^{-1}(\varphi_{P_{l+1}}(\sigma))) = \varphi_{P_l}([^{-1}(\sigma))$. The proof proceeds by cases on σ.

- $\{\sigma \sqsubseteq \texttt{null}\}$ It holds since $\varphi_{P_l}([^{-1}(\varphi_{P_{l+1}}(\sigma))) = \varphi_{P_l}([^{-1}([^{l+1}\tau)) = \varphi_{P_l}([^l\tau) = [^l\tau$, and $\varphi_{P_l}([^{-1}(\sigma)) = \varphi_{P_l}(\sigma) = [^l\tau$.
- $\{\sigma \not\sqsubseteq \texttt{null},\ \sigma \sqsubseteq [^{l+1}\tau\}$ Thus, the left-hand can be re-written as $\varphi_{P_l}([^{-1}(\varphi_{P_{l+1}}(\sigma))) = \varphi_{P_l}([^{-1}([^{l+1}\tau)) = \varphi_{P_l}([^l\tau) = [^l\tau$. We have that $\sigma = [^{l+1}\sigma'$, with $\sigma' \sqsubseteq \tau$. Then the right-hand member can be simplified as $\varphi_{P_l}([^{-1}(\sigma)) = \varphi_{P_l}([^l\sigma') = [^l\tau$.
- $\{\sigma \not\sqsubseteq \texttt{null},\ \sigma \not\sqsubseteq [^{l+1}\tau\}$ Then, the left-hand member is equal to $\varphi_{P_l}([^{-1}(\varphi_{P_{l+1}}(\sigma))) = \varphi_{P_l}([^{-1}(\top)) = \varphi_{P_l}(\top) = \top$. The right-hand member is also equal to $\top$, since it can be shown that, whatever the value of $[^{-1}(\sigma)$ is, it cannot be $\sqsubseteq [^l\tau$. □

Thus, we can use Theorem 5.3 and find a new method, say μ_{m-1}, such that $lfp(\varphi_{P_{m-1}} \circ \mathsf{next}_{\mu_{m-1}}) = \varphi_{P_{m-1}}(lfp(\mathsf{next}_\mu)) = \mathcal{L}_{m-1}$. We can use $\mathcal{L}_{m-1}$ to find all errors, if any, for type $[^{m-1}\tau$. Moreover, we are now in the position to apply Theorem 5.3 again, with $A = P_{m-2}$, $B = P_{m-1}$ and $F = \{[^{-1}\}$, to find $\mathcal{L}_{m-2}$, and so on. At each step l, a new method μ_l must be found. When we have performed all steps, we will have built $\{\mathsf{next}_{\mu l}\}_{l \in L}$ (with $\mu_m = \mu$), which is a set of functions that are fixpoint complete for next_μ wrt φ_{P_l}. Thus, we have checked

all the hypotheses of Lemma 4.2, and we can conclude that the process we have described allows us to precisely find all errors in $\mathscr{A}_\tau^m \cap \mathbb{E}$.

Using the same definition introduced in the proof of Theorem 5.3, we can define, for all $l \in [0, m)$:

$$\mathsf{next}_{\mu_l}(v) = \lambda j. \bigsqcup_{i=0}^{m-1} \mathsf{next}_{ij}^{(l)}(v[i]),$$

where

$$\mathsf{next}_{ij}^{(l)} = \begin{cases} \varphi_{P_i}([^{-1}(\mathcal{L}_{l+1}[i])) & \text{if } \mathsf{next}_{ij} = [^{-1}, \\ \mathsf{next}_{ij} & \text{otherwise.} \end{cases}$$

Note that function next_{μ_l}, used in the computation of fixpoint $\mathcal{L}_l$, needs some information from fixpoint $\mathcal{L}_{l+1}$. The memory needed to store this information adds to the total memory requirements of the decomposition. However in this simple case the additional memory requirements amount to a single bit for each `aaload` instruction in method μ. In fact, to compute $\mathsf{next}_{ij}^{(l)}$, we only need to know whether the stack top in the before context of each `aaload` instruction, taken from $\mathcal{L}_{l+1}$, is $\sqsubseteq [^{l+1}\tau$ or not.

5.2 Serial Decomposition in the Presence of Object Initialization

As noted in Section 4.2.1, the presence of a nonconstant, nonidentity function in $\mathscr{F}_\mu$ may cause the complete shell of some abstractions of $\mathbb{T}$ to contain more than two elements. In this case, the complete shell (wrt next) of any $\tau \in \mathcal{C}'$, for which some `new` instruction is found in the method, must contain all the uninitialized versions of types assignable to τ.

Object initialization poses an additional problem, since the form of its error function is different from all the others. In particular, function check^* defined in Section 4.3.1 implies that all uninitialized versions of τ must be tracked to check for the presence of error $\tau^{(*)}$ (for any $\tau \in \mathcal{C}'$).

In this section, we apply similar ideas to those in Section 5.1 to obtain a decomposition into basic domains, without losing completeness. We do this in two steps: first, we show that the check of initialization errors, for any $\tau \in \mathcal{C}'$, can be decomposed using a basic domain. Then, we show how to use information, gathered during the first step, to obtain a fixpoint complete (wrt next) analysis that only uses the basic domain $\mathscr{B}_\tau = \{\tau, \top\}$.

5.2.1 *Decomposition of the Check for Initialization Errors.* For a fixed method μ and any $\tau \in \mathcal{C}'$, recall that $\mathscr{U}_\tau$ is the least domain that contains the set of uninitialized versions of τ (restricted to those that may actually be created by method μ). We can easily verify that $\dot{\varphi}_{\mathscr{U}_\tau}$ is complete wrt error and $\lambda Q.Q \cup \overline{\{\tau^{(*)}\}}$. Moreover, it holds that $\mathscr{S}_{\mathscr{F}_\mu}(\mathscr{U}_\tau) = \mathscr{U}_\tau$ for any μ, so that $\langle \dot{\varphi}_{\mathscr{U}_\tau}, \dot{\varphi}_{\mathscr{U}_\tau} \rangle$ is also complete (and, thus, fixpoint complete) for next_μ. This means that, as far as error $\tau^{(*)}$ is concerned, we can work in domain $\mathscr{U}_\tau$, without losing precision wrt standard verification.

We now want to decompose domain $\mathscr{U}_\tau$. Assume that domain $\mathscr{U}_\tau$ is not a basic domain (otherwise there would be no need for a decomposition). This is

the case when there is more than one new τ instruction in the method, since $\mathscr{U}_\tau$ will contain a distinct $\tau^{(h)}$ for each h : new τ instruction, together with $\top$ and $\bot$. Now, we apply Lemma 4.2, with $C = \mathbb{V} = D$, $f = \mathsf{next}$, $g = \dot{\varphi}_{\mathscr{U}_t}$, $I = \mathscr{U}_\tau$ and $\rho_i = \eta_i = \dot{\varphi}_{P_i}$, where P_i, given $i \in \mathscr{U}_\tau$, is the basic domain $\mathscr{B}_{\tau^{(h)}} = \{\tau^{(h)}, \top\}$ for some $i = \tau^{(h)} \in \mathscr{U}_\tau$. Finally, we choose $f_i^\sharp = \mathsf{next}_i^b = \dot{\varphi}_{P_i} \circ \mathsf{next}$, the abstraction of f wrt $\dot{\varphi}_{P_i}$, and $g_i^\sharp$ as the best abstraction g_i^b of $g = \dot{\varphi}_{P_i}$ wrt $\dot{\varphi}_{P_i}$, which is $g_i^b = \dot{\varphi}_{P_i}$ itself. Now, an easy calculation shows that $\mathscr{S}_{\mathscr{T}_\mu}(P_i) = P_i$, for any $i \in \mathscr{U}_\tau$. Thus, $\langle \dot{\varphi}_{P_i}, \dot{\varphi}_{P_i} \rangle$ is always complete and, thus, fixpoint complete for next and hypothesis (a) is satisfied. Hypothesis (b) requires completeness of $\langle \rho_i, \eta_i \rangle$ wrt g, for all $i \in \mathscr{U}_\tau$. This requires proving that $\eta_i \circ g = \eta_i \circ g \circ \rho_i$. Note that all functions involved are functions on $\mathbb{V}$. However, the equivalent condition can be checked on $\mathbb{T}$ and then pointwise extended to $\mathbb{V}$. When checked on $\mathbb{T}$, the previous condition translates to $\varphi_{P_i} \circ \varphi_{\mathscr{U}_\tau} = \varphi_{P_i} \circ \varphi_{\mathscr{U}_\tau} \circ \varphi_{P_i}$. It is well known that, if ρ and η are two closure operators on C, then $\rho(C) \subseteq \eta(C)$ implies that $\rho \circ \eta = \rho$. Thus, the previous condition holds, due to $P_i \subseteq \mathscr{U}_\tau$ and idempotency of φ_{P_i}. This satisfies hypothesis (b). Finally, hypothesis (c) is also satisfied, since $\mathscr{M}(\bigcup_{i \in \mathscr{U}_\tau} P_i) = \mathscr{U}_\tau$ (the only element that may be missing from the union, namely $\bot$, is added by the Moore closure, as the meet of any two, or more, distinct $\tau^{(h)}$ types in $\mathscr{U}_\tau$). Thus, we can apply Lemma 4.2 and obtain:

$$\mathcal{L} = \dot{\varphi}_{\mathscr{U}_\tau}(\mathit{lfp}(\mathsf{next})) = \bigsqcap_{i \in \mathscr{U}_\tau} \dot{\varphi}_{P_i}(\mathit{lfp}(\mathsf{next}_i^b)) = \bigsqcap_{i \in \mathscr{U}_\tau} \mathcal{L}_i.$$

This means that we can recover the least fixpoint $\mathcal{L}$ of the analysis that uses $\dot{\varphi}\,\mathscr{U}_\tau$, by calculating the meet of the least fixpoints $\mathcal{L}_i$, obtained in the separate analyses, each involving a different P_i. We recall that we need $\mathcal{L}$ only to apply error to it and look for the presence, or absence, of a $\tau^{(*)}$ error. Since this error can only be produced by the init τ instructions, we do not need all of vector $\mathcal{L}$, but only some relevant elements (i.e., the stack top of the before context of all init τ instructions). Thus, during computation of least fixpoint $\mathcal{L}_i$, we only need to remember the type observed in these elements, in order to perform the final meet. This meet can, obviously, be performed incrementally, so there is no need to store the partial results of each $\mathcal{L}_i$ in separate locations. Moreover, an analysis of the possible cases that may occur, shows that, for each element, we only need to know if it is $\top$ or not. In conclusion, this decomposition only requires an additional bit for each init τ instruction in the method.

5.2.2 *Decoupling Class Types from Their Uninitialized Versions.* The existence of function $\mathcal{I}^\sigma$ in $\mathscr{T}_\mu$, with $\sigma \sqsubseteq \tau$, forces the complete shell of $\mathscr{B}_\tau = \{\tau, \top\}$ wrt $\mathscr{T}_\mu$ to contain all types in $\mathscr{U}_{\sqsubseteq\tau}$. Thus, type errors on τ cannot be directly checked using basic domain $\mathscr{B}_\tau$. However, assume for the moment that method μ contains no aaload instructions or no arrays of τ. Then, we can use Theorem 5.2, with $A = \mathscr{B}_\tau$, $B = \mathscr{U}_{\sqsubseteq\tau}$ and $F = \{\mathcal{I}^\sigma \in \mathscr{T}_\mu \mid \sigma \sqsubseteq \tau\}$[3]. This allows us to gather sufficient information during the calculation of $\dot{\varphi}\,\mathscr{U}_{\sqsubseteq\tau}(\mathit{lfp}(\mathsf{next}_\mu))$ and use this information to compute $\dot{\varphi}\,\mathscr{B}_\tau(\mathit{lfp}(\mathsf{next}_\mu))$.

[3]Theorem 5.2 only considers functions with arity 1. However, it can be generalized in a natural way to functions of any arity.

We have that $\langle \varphi_{\mathscr{B}_\tau}, \varphi_{\mathscr{B}_\tau} \rangle$ is complete for $\mathscr{F}_\mu \setminus \boldsymbol{F}$ (since all functions in this set are either constant or the identity function). Moreover, $\varphi_{\mathscr{B}_\tau}$ is join distributive. Finally, we have to find for each function $\mathcal{I}^\sigma \in \boldsymbol{F}$, a residual of $\varphi_{\mathscr{B}_\tau} \circ \mathcal{I}^\sigma$ wrt $\varphi_{\mathscr{U}_{\sqsubseteq\tau}}$. We have:

$$\varphi_{\mathscr{B}_\tau}(\mathcal{I}^\sigma(\tau_1, \tau_2)) = \begin{cases} \tau & \text{if } \mathcal{I}^\sigma(\tau_1, \tau_2) \sqsubseteq \tau \Leftrightarrow \tau_1 = \tau_2 \in \mathbb{U}^\sigma_\mu \text{ or } \tau_2 \sqsubseteq \tau, \\ \top & \text{otherwise,} \end{cases}$$

while:

$$\varphi_{\mathscr{B}_\tau}(\mathcal{I}^\sigma(\varphi_{\mathscr{U}_{\sqsubseteq\tau}}(\tau_1), \varphi_{\mathscr{U}_{\sqsubseteq\tau}}(\tau_2))) = \begin{cases} \varphi_{\mathscr{B}_\tau}(\sigma) & \text{if } \varphi_{\mathscr{U}_{\sqsubseteq\tau}}(\tau_1) = \varphi_{\mathscr{U}_{\sqsubseteq\tau}}(\tau_2) \in \mathbb{U}^\sigma_\mu \\ \varphi_{\mathscr{B}_\tau}(\varphi_{\mathscr{U}_{\sqsubseteq\tau}}(\tau_2)) & \text{otherwise} \end{cases} =$$

$$= \begin{cases} \tau & \text{if } \tau_1 = \tau_2 \in \mathbb{U}^\sigma_\mu \\ \top & \text{otherwise.} \end{cases}$$

The main point in these calculations is that the result of the test $\tau_1 = \tau_2 \in \mathbb{U}^\sigma_\mu$ can be completely determined in $\mathscr{U}_{\sqsubseteq\tau}$. A residual function g such that $\forall(\tau_1, \tau_2) \in \mathbb{T} \times \mathbb{T}$, $g(\tau_1, \tau_2) \sqcap f(\rho(\tau_1), \rho(\tau_2)) = f(\tau_1, \tau_2)$ is $g = \lambda\tau_1, \tau_2.\varphi_{\mathscr{B}_\tau}(\tau_2)$. This function is monotone and $\langle \varphi_{\mathscr{B}_\tau}, \varphi_{\mathscr{B}_\tau} \rangle$ is trivially complete for it. Thus, we can apply Theorem 5.2, and implement the following strategy:

(1) during fixpoint calculation of $\dot{\varphi}_{\mathscr{U}_{\sqsubseteq\tau}} \circ \mathsf{next}_\mu$, we collect $\varphi_{\mathscr{B}_\tau} \circ \mathcal{I}^\sigma$ for all relevant elements in the context vector. This requires a bitmap of size $t_\mu + r_\mu$ for each `init` σ instruction in μ, $\sigma \sqsubseteq \tau$. Note that parallel decomposition can be applied to $\mathscr{U}_{\sqsubseteq\tau}$, so these bitmaps can be easily calculated, incrementally, as the meet of the corresponding bitmaps obtained in each substep (this is another application of Lemma 4.2).

(2) $\mathscr{B}_\tau$ is analyzed by replacing each $\mathcal{I}^\sigma(\tau_1, \tau_2)$ function with the meet of τ_2 and the appropriate element computed in step 1.

In summary, a basic domain can be used even in this case, with only a (generally) small penalty in memory requirements.

If $\mathscr{F}_\mu$ contains both $[^{-1}$ and $\mathcal{I}^\sigma$, $\sigma \sqsubseteq \tau$, then $\mathscr{S}_{\mathscr{F}_\mu}(\mathscr{B}_\tau) = \mathscr{M}(\mathscr{A}^{\kappa_{\mu,\tau}}_\tau \cup \mathscr{U}_{\mu,\sqsubseteq\tau})$. In this case, serial decomposition can still be used, but all $[^{-1}$ and $\mathcal{I}^\sigma$ must be replaced before abstraction $\mathscr{B}_\tau$ can be used. First, serial decomposition is used to decouple $\mathscr{B}_\tau$ from $S = \mathscr{M}(\mathscr{A}^{\kappa_{\mu,\tau}}_{[\tau} \cup \mathscr{U}_{\mu,\sqsubseteq\tau})$. Then, parallel decomposition is used to decompose S in $\mathscr{A}^{\kappa_{\mu,\tau}}_{[\tau}$ and the set of basic domains contained in $\mathscr{U}_{\mu,\sqsubseteq\tau}$. Finally, serial decomposition is applied to $\mathscr{A}^{\kappa_{\mu,\tau}}_{[\tau}$.

5.3 Implementation

To implement the strategies described in Sections 5.1 and 5.2, we can replace step 4 in the implementation outlined in Section 4.6 with the procedure that follows. For each type $\tau \in \mathcal{C}'$, found in step 3, $S_\tau = \mathscr{S}_{\mathscr{F}_\mu}(\mathscr{B}_\tau)$ is computed as before, but now the result is used to implement the strategy illustrated in Section 5.1 and/or the strategy of Section 5.2 as needed. Note that this simple implementation may cause some abstract interpretation to be repeated two or more times, but has minimal memory requirements.

6. DISCUSSION

In this section we will compare multipass verification to two other bytecode verification techniques: the standard verification and the Lightweight Bytecode Verification (LBV).

6.1 Multipass vs. Standard Verification

To perform the fixpoint iteration, both standard and multipass verification must save the types of each register and stack item at each branch target. This information is stored in a data structure which we call *dictionary*.

In the standard verification each type is represented with three bytes [Leroy 2002]. Hence, the size in bits of the dictionary used in the standard verification is

$$D_{\mathrm{Std}} = 24nb,$$

where b is the number of branch targets and n is the number of items in a context (number of registers plus stack size) of a given method.

If we use the multipass algorithm without applying serial decomposition the space occupancy of the dictionary is equal to

$$D_{\mathrm{Par}} = \max_{\sigma \in \mathscr{E}'_\mu}(f_{\mathrm{Par}}(\sigma)),$$

where $\mathscr{E}'_\mu$ is the set of types corresponding to the passes of the parallel verification of method μ, and $f_{\mathrm{Par}}(\sigma)$ is the space occupancy of the dictionary during the analysis that checks σ-correctness. Note that $\mathscr{E}'_\mu \subseteq \mathscr{E}_\mu$, since some passes of the parallel verification may keep track of more than one type at a time, and thus can also check more than one type error in a single pass. Function f_{Par} can be calculated as (see Section 4.6)

$$f_{\mathrm{Par}}(\sigma) = \begin{cases} nb & N_{\texttt{aaload}} = 0 \text{ or } \sigma \in \mathcal{B} \text{ or } \kappa_{\mu,\tau} = 0, \\ (2 + \lceil \log(\max(\kappa_{\mu,\sigma}, \left|\mathbb{U}_\mu^{\sqsubseteq\tau}\right|)) \rceil) nb & \text{otherwise,} \end{cases}$$

where $N_{\texttt{aaload}}$ is the number of `aaload` instructions in the method. On the other hand, when using serial decomposition for dealing with arrays and object initialization, the size of the dictionary is

$$D_{\mathrm{Ser}} = \max_{\sigma \in \mathscr{E}''_\mu}(f_{\mathrm{Ser}}(\sigma)),$$

where $\mathscr{E}''_\mu$ is the set of types corresponding to the passes of the serial verification of method μ, and $f_{\mathrm{Ser}}(\sigma)$ is equal to the memory requirements of the analysis with the basic domain $\{\sigma, \top\}$. In serial decomposition we have $\mathscr{E}''_\mu \supseteq \mathscr{E}_\mu$, since this decomposition may introduce some passes over types that are used but not checked in method μ. When we use the decomposition in basic domains, we can represent each type with a single bit. However, we have to use some additional memory for `aaload` and `init` instructions. We must use at least one bit for each `aaload` (Section 5.1) and then save one whole context for each `init` (Section 5.2.2). Therefore the value $f_{\mathrm{Ser}}(\sigma)$ can be calculated as

$$f_{\mathrm{Ser}}(\sigma) = nb + N_{\texttt{aaload}} + n\left|\mathbb{U}_\mu^{\sqsubseteq\sigma}\right|.$$

We measured the value of D_{Par}, D_{Ser}, and D_{Std} for two classes of packages: Java Card applets and J2ME applets. For the first class we have chosen the examples bundled with JCDK v.2.2.2 and the PACAP case study [Bieber et al. 2001]. This set of applets is a superset of the benchmarks used by other papers on this topic (see Section 7 for references). For the second class we chose the examples contained in the Sun Wireless Toolkit v. 2.2 because we would like to show some statistics about bigger applications that run on memory constrained devices.

When there are no `inits` and no `aaloads` the ratios $D_{\text{Par}}/D_{\text{Std}}$ and $D_{\text{Ser}}/D_{\text{Std}}$ are minimum and equal to $1/24 = 4.17\%$, therefore the space occupancy of the dictionary in multipass verification is substantially lower than in standard verification. However, for particular methods that have high number of uninitialized objects and a small number of branch targets, the standard verification may require less memory—for instance when a class has a long list of static object fields of the same type. The initialization method of such a class will have many `inits` without having many targets. Luckily, such methods are extremely rare (e.g., for each of the tested packages only less than 0.1% of the methods have a ratio $D_{\text{Ser}}/D_{\text{Std}} > 0.5$). Applet developers could find such pathological methods and refactor them in order to have less object initializations per method.

Figure 20 shows the values of D_{Par}, D_{Ser} (in bytes), of the ratios $D_{\text{Par}}/D_{\text{Std}}$ and $D_{\text{Ser}}/D_{\text{Std}}$ in the worst case, that is the ratio is between the most space consuming method with multipass and the most space consuming method with standard verification (i.e., the ratio can refer to two distinct methods). When the serial decomposition is not used, in packages that use arrays and objects, the ratio grows a bit, but without going beyond 13% (Java Card applets) and 21% (J2ME applets). Using serial decomposition, the ratio is around 5%. These results confirm that multipass verification greatly reduces the space occupancy of the dictionary.

Figure 21 shows the average number of passes needed (i.e., the number of abstraction interpretations) with and without serial decomposition. In most cases the number of passes ranges from 5 to 10.

However, even though we believe that multipass verification is slower than Standard Verification, some facts suggest that this slowdown might be tolerable. First, the total time of verification is not entirely due to the dataflow analysis, but also to the verification of structural constraints, which must be performed in any case. Moreover, verification is performed only when the applet is installed, and in Java Card most of the installation time is due to the transfer of code to the device (Deville and Grimaud [2002] report that the load time is from 5 to 8 times greater than the verification time). Finally, even if the algorithm must be repeated n times, the time penalty with respect to Standard Verification is generally not n. In fact, in bitmap representation, a join of two contexts can be easily implemented as bitwise logical OR, which is always a simpler and faster operation than that performed by Standard Verifier. Comparing the other operations is less clear, though. For example, take $\tau, \sigma \in \mathcal{C}'$. If we want to check for $\tau \sqsubseteq \sigma$, we need to traverse the class hierarchy, starting at τ, until σ is found, or the top of the hierarchy is reached. In Standard Verification, this traversal must be performed whenever a type checking involving class types is performed. On the other hand, in our bitmap representation this

		Parallel		Serial	
Package	Mth	Size	%Std	Size	%Std
mmademo	395	254	14.10%	81	4.48%
Demos	332	518	16.67%	140	4.50%
Games	166	88	5.90%	62	4.17%
UIDemo	88	85	13.49%	27	4.17%
Audiodemo	80	60	12.50%	21	4.30%
Demo3D	71	210	16.67%	64	5.03%
WMADemo	65	90	16.67%	31	5.58%
PDAPDemo	59	230	16.67%	64	4.63%
BluetoothDemo	56	309	20.83%	73	4.89%
NetworkDemo	54	38	12.50%	15	4.92%
ObexDemo	50	49	12.50%	20	5.00%
Photoalbum	41	93	12.50%	33	4.39%
JSR172Demo	34	150	20.83%	42	5.83%
FPDemo	6	18	4.17%	18	4.17%
PACAP	539	45	4.20%	45	4.17%
odSample	33	17	4.73%	15	4.17%
JavaPurseCrypto	32	68	7.21%	39	4.17%
JavaPurse	32	77	8.17%	39	4.17%
ChannelsDemo	25	14	4.17%	14	4.17%
utilitydemo	21	24	4.17%	24	4.17%
SecureRMIDemo	20	9	4.17%	9	4.17%
biometry	20	12	12.50%	5	5.56%
transit	18	41	11.16%	16	4.17%
photocard	11	37	12.50%	13	4.21%
wallet	9	8	4.17%	8	4.17%
RMIDemo	9	2	4.17%	2	4.17%
sigMsgRec	7	11	10.23%	5	4.55%
ServiceDemo	7	4	4.17%	4	4.17%
JavaLoyalty	7	11	9.38%	5	4.17%
NullApp	3	3	12.50%	2	8.33%
HelloWorld	3	4	4.17%	4	4.17%

Fig. 20. The space occupancy of the dictionary for some J2ME applets (at the top) and some Java Card applets (at the bottom). The Mth column reports the number of methods. The Size column indicates the maximum size of the dictionary using our algorithm (in bytes). The %Std column shows the percentage ratio of maximum sizes of the dictionary between our algorithm and Standard Verification.

traversal must be performed whenever a new class type is created (since we must know whether this new class type must be abstracted into 1 or 0), while type checking only requires testing for a 0. Thus, the comparison depends on the ratio of the number of instructions that create new class types versus the number of class type constraints, and on the average "distance" in the class hierarchy of the types to be compared in the two cases. The check $\tau \sqsubseteq \sigma$ can also be done more efficiently with the aid of some sort of caching [Click and Rose 2002; Cohen 1991] and therefore at the expense of some additional memory.

Package	Par	Ser
mmademo	4.92	5.96
Demos	5.78	7.27
Games	5.30	6.16
UIDemo	4.73	6.19
Audiodemo	5.88	6.94
Demo3D	5.07	5.86
WMADemo	7.00	8.66
PDAPDemo	8.00	10.29
BluetoothDemo	7.82	9.96
NetworkDemo	6.52	8.31
ObexDemo	7.52	8.92
Photoalbum	6.20	7.07
JSR172Demo	5.74	7.88
FPDemo	8.67	11.33

(a)

Package	Par	Ser
PACAP	5.85	6.25
odSample	4.03	4.73
JavaPurseCrypto	8.19	8.97
JavaPurse	8.16	8.94
ChannelsDemo	5.28	5.44
utilitydemo	8.38	9.14
SecureRMIDemo	5.55	6.05
biometry	5.55	5.75
transit	7.61	7.78
photocard	6.55	7.73
wallet	7.56	7.89
RMIDemo	4.33	5.00
sigMsgRec	8.29	8.57
ServiceDemo	3.29	4.29
JavaLoyalty	6.14	6.43
NullApp	4.00	4.67
HelloWorld	4.33	5.00

(b)

Fig. 21. The average number of passes needed to perform multipass verification of (a) some J2ME applets, and (b) some JavaCard applets (with and without serial decomposition).

		Parallel			Serial		
Package	Mth	Size	%Std	Passes	Size	%Std	Passes
JDK1.5	107555	7575	24.63%	4.87	1285	4.18%	5.80
Azureus	17507	7898	37.50%	4.78	1353	6.42%	6.07
Jedit	5980	3364	29.17%	5.46	582	5.04%	6.59
Bcel	2869	1044	22.44%	4.52	205	4.40%	5.64
Jclasslib	1494	150	19.61%	4.38	45	5.88%	5.14
JCDK	861	102	5.40%	4.93	79	4.17%	5.35

Fig. 22. The statistics concerning some large/medium applications/libraries. The Mth column reports the number of methods. The Size column indicates the maximum size of the dictionary using our algorithm (in bytes). The %Std column shows the percentage ratio of maximum sizes of the dictionary between our algorithm and Standard Verification. The Passes column shows the average number of analyses per method.

Finally, in Figure 22, we present statistics on a set of large/medium applications/libraries. We analyzed: a) *JDK* (the runtime environment of Sun Java Development Kit, v.1.5.0.12), b) *Azureus* (a Bittorrent client), c) *Jedit* (a programmer's text editor), d) *Bcel* (the Byte Code Engineering Library, a library for manipulating Java classfiles, developed by the Jakarta project of the Apache Foundation), e) *Jclasslib* (a Java class file library with a Java byte code viewer, developed by ej-technologies GmbH) e) *JCDK* (the Sun Java Card Development Kit APIs, v. 2.2.2). We believe that this class of statistics is less significant than the others because these applications run on devices for which the memory is not a concern. However, the results confirm the trend predicted by the other experiments.

6.2 Multipass vs. LBV

The Proof Carrying Code was proposed by Necula [1997] and tailored to bytecode verification by Rose and Rose [1998]. The verification process is split into two phases: lightweight bytecode certification (LBC) and lightweight bytecode verification (LBV). LBC is performed off-board and produces a certificate that must be distributed and downloaded into the device together with the bytecode. LBV is performed on-board and checks that the certificate is correct. LBV is currently used in the KVM of Sun's Java 2 Micro Edition. In this implementation, LBC performs a standard verification through fixpoint iteration, then it records the fixpoint into the certificate. Thus, the certificate contains, for each method, the contents of each before context (registers plus stack) of each branch target. The certificate is then stored in the classfile as a `StackMap` attribute. LBV is a linear verification process that exploits the certificate to avoid a full dataflow analysis. During LBV, only one context is constantly updated and needs to be stored in RAM. Certificates, instead, are never updated and therefore can be stored in slow-write memories (Flash, EEPROM) together with the application code. Thus, LBV can be performed in RAM constrained devices. Moreover, the linear verification performed by LBV is quite faster than standard verification, and for this reason LBV has become [JSR 2006] the predefined method of verification in the JSE as well, leaving the standard verification as a fall-back strategy when LBV fails (for instance due to an invalid or missing `StackMap` attribute).

One drawback of this technique is that each code must be distributed with its certificate, thus increasing the size of data that must be downloaded to the device. Leroy [2003] reports that the certificate is about 50% the size of the corresponding bytecode, and Deville and Grimaud report an increase of 10 to 30% in the size of the downloaded code due to the certificate [Deville and Grimaud 2002]. However, Rose [2003] reports that the certificate could be furtherly optimized at the cost of using more memory during the lightweight verification. In the following, we refer to the implementation present in the KVM. Reducing the size of the certificate is important in embedded devices, where: 1) the transmission of the additional bytes of the certificate could have a nonnegligible cost; 2) even slow-write memories may have severe size constraints. To make the certificate smaller, the encoding of types in LBC is not of uniform size: reference types need 3 bytes, while basic types need only one. Moreover, also the number of context elements stored for each branch target is not uniform: the certificate stores only the registers up to the first unused one, and only the stack elements up to the actual size of the stack for each branch.

We now compare the memory requirements of LBV wrt multipass verification, then suggest a combined multipass+LBV strategy.

Since LBV only needs one context, the requested amount of RAM is

$$D_{\mathrm{LBV}} = 24n = D_{\mathrm{Std}}/b,$$

therefore the ratio of D_{LBV} and D_{Std} only depends on the number of branch targets. In multipass verification, each context is much smaller than the context used in both LBV and standard verification. However, multipass verification still needs a context for each branch target.

	$D_{\mathrm{Par}}/D_{\mathrm{LBV}}$		$D_{\mathrm{Ser}}/D_{\mathrm{LBV}}$	
Class	Min	Max	Min	Max
Java Card applets	0.2	1.5	0.1	0.8
J2ME applets	0.5	6.2	0.4	1.6
Large app/lib	1.3	30.0	0.8	5.0

Fig. 23. Dictionary requirements of the multipass verification versus LBV.

Figure 23 summarizes the ratio of D_{Ser} and D_{LBV}, and of D_{Par} and D_{LBV}, respectively, on the same benchmarks used in Section 6.1. The ratio refers to the worst case (i.e., between the largest dictionary per application). Multipass verification is competitive only on very small applications, such as Java Card applets—serial decomposition uses at most 80% of the RAM used by LBV. However, the computation performed by serial decomposition is more time-consuming that that of LBV, and this time penalty should be compared with the time gained by avoiding the download of the certificate. The LBV, with only one context to store in RAM, outperforms multipass verification on the other benchmarks. This is because the larger the package is, the greater the chance of finding a method with many branch targets.

However, LBV is orthogonal wrt our strategy. In fact LBV and multipass verification do not exclude each other and can be combined in a "multipass+LBV" strategy: LBC can be used to build a subcertificate for each subanalysis of multipass verification. The multipass+LBV certificate, containing all subcertificates, can be downloaded in the memory-constrained device. The on board algorithm (multipass+LBV) checks the certificate by first computing the sequence of passes, as in multipass, then checks the subcertificate that corresponds to each pass, as in LBV. The combined multipass+LBV strategy has two potential benefits.

(1) Each subcertificate can be loaded separately on the device and deleted as soon as it has been checked. This would allow the use of RAM memory to store the certificate, instead of slower memories. Moreover, when some form of DMA is available the verification can proceed in pipeline with the download of the certificate.
(2) Even if the multipass+LBV certificate contains a subcertificate for each type error checked in each method, each subcertificate is very small. Since the vast majority of methods, in typical applications, use a few types only, the multipass+LBV certificate may be smaller than the LBV certificate in some cases.

To check claim 2, we have instrumented the bytecode verifier implemented in BCEL 5.2 [Bernardeschi et al. 2003] to obtain: 1) the total number N_μ of registers and stack elements stored in the fixpoint of each method μ, taking into account the nonuniform size of each context at each branch target of the same method in the LBV certificate; 2) the total size in bytes S_μ of the fixpoint of each method μ, taking into account the nonuniform size (either 1 or 3 bytes) of each context element identified in point 1). An estimate of the size of the LBV certificate of a given package is obtained by summing S_μ over all methods μ

in the package. For multipass+LBV, we have to distinguish between serial and parallel decomposition. Let $C_{\mathrm{Par},\mu}$ and $C_{\mathrm{Ser},\mu}$ be the size of the subcertificate for method μ using parallel and serial decomposition, respectively. Then, $C_{\mathrm{Par},\mu}$ and $C_{\mathrm{Ser},\mu}$ can be estimated as follows:

$$C_{\mathrm{Par},\mu} = N_\mu \sum_{\sigma \in \mathscr{E}'_\mu} (f_{\mathrm{Par}}(\sigma)) \quad \text{and} \quad C_{\mathrm{Ser},\mu} = N_\mu \left| \mathscr{E}''_\mu \right|.$$

Note that $C_{\mathrm{Ser},\mu}$ does not contain the interphase result propagation needed by `aaload` and object initialization, since these can be easily calculated during the on board lightweight verification. Finally, an estimate of the size of the certificate for multipass+LBV for a given package is obtained by summing the size of the subcertificates for each method in the package ($C_{\mathrm{Par},\mu}$ if multipass uses only parallel decomposition, and $C_{\mathrm{Ser},\mu}$ if multipass uses serial decomposition).

Figure 24 shows the size of the certificates for LBV and multipass+LBV when using parallel and serial decompositions. We have used the same benchmarks of Section 6.1, grouped by certificate size. For small applets, multipass+LBV leads to a certificate that is about half the size of the LBV certificate. For larger applets, the size reduction is smaller, and in some cases the multipass+LBV certificate is bigger than the LBV certificate. This happens, for example, in Demo3D, where our decomposition is penalized by the presence of methods that create a large number of objects. Figure 25 shows the size of the certificate for larger packages, confirming the trend observed in small and medium applets.

7. RELATED WORK

Bytecode verification is a well-studied technique—see Leroy [2003] for an extensive survey on the topic. As regards the Java Card platform, many investigations has been carried out to formalize the semantics of the virtual machine and its relation with bytecode verification.

Barthe et al. [2005] were able to formally derive from a defensive virtual machine (a virtual machine that performs type checks at runtime) both an offensive virtual machine (a virtual machine that does not perform type checks) and a bytecode verifier. Both the offensive machine and the bytecode verifier are seen as abstract interpretations of the defensive machine (in the former they discard types, in the latter they discard values). On the contrary, our formalization of the bytecode verification is not aimed at proving the correctness of the verification, but is the starting point for the decomposition. In fact, we abstract the verifier and not the virtual machine. Moreover, Barthe et al. define a specification and verification tool (Jakarta) that can be used to model a generic low-level language and its defensive VM, and they used it for proving the correctness of the Java Card Platform. The major benefits of this kind of analysis are its generality (e.g., assigning different meanings to types different properties can be checked) and the exploitation of the proof automation capabilities of the Coq proof assistant. A similar kind of refinement were also proposed by Requet, Lanet and others [Lanet and Requet 2000; Requet 2003; Casset et al. 2002] but without the full generality of Barthe et al. [2005]. They use the B method and generate executable code by translating B specifications into C programs.

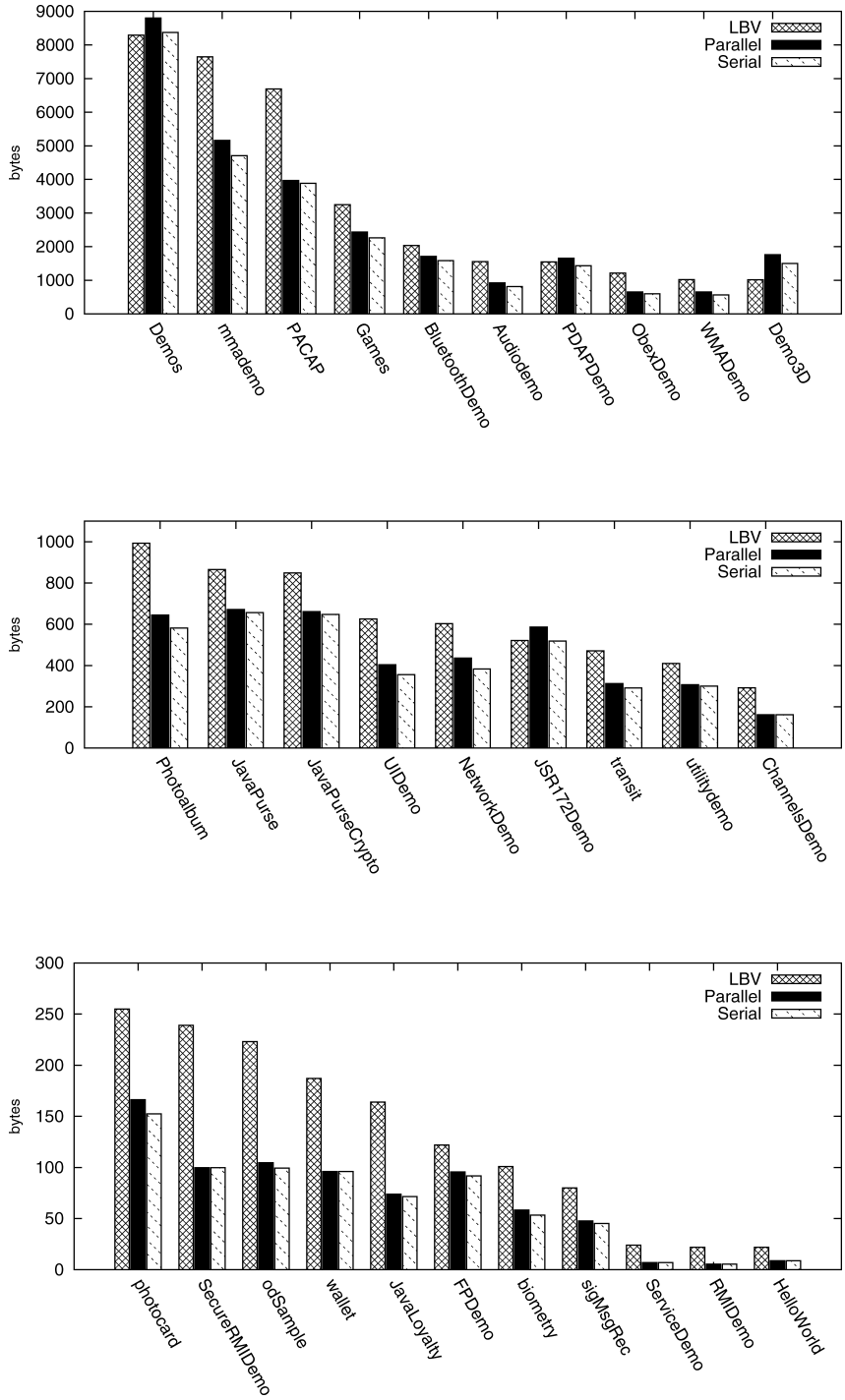

Fig. 24. The size of the proof carrying code certificate using Lightweight Bytecode Verification, and Parallel and Serial decompositions.

			Parallel		Serial	
Package	Methods	LBVSize	Size	RatioLBV	Size	RatioLBV
JDK1.5	107744	2073985	1989874	95.95%	1877554	90.53%
Azureus	17507	319967	424592	132.70%	450932	140.93%
Jedit	5980	161035	169450	105.23%	158270	98.28%
Bcel	2869	37940	45704	120.46%	44163	116.40%
JCDK	861	13537	9651	71.29%	9215	68.07%
Jclasslib	1494	11467	7776	67.81%	6904	60.21%

Fig. 25. The space occupancy of the certificate for some medium/large applications/libraries. The Methods column reports the number of methods.

For an in-depth specification of the Java platform, the reader can refer to the recent work by Klein and Nipkow [2006]. They provide a machine-checked (using Isabelle/HOL) formalization and verification of the entire Java language architecture (language, type system, compiler, virtual machine). This unified model also includes the bytecode verification and the proof of its correctness.

Many approaches have been presented to reduce the memory requirements of the verification with the aim of developing an on-card verifier.

Leroy [2002] proposes to reduce memory requirements with an off-card code transformation, also known as code normalization. The transformed code complies with the following constraints: every register contains the same type for all method instructions and the stack is empty at the merge points. The verification of a normalized code is not expensive: only one global state is required since the type of registers and stack items never change. Since a single context is used, the memory requirements of this verification are very similar to the memory requirements of LBV (probably slightly worse, since some new registers may be added by code transformation). Thus, it is probably both smaller and faster than multipass, and probably faster than multipass+LBV, even accounting for the increase in code size due to normalization. However, normalization requires a nonstandard toolchain, while multipass is compatible with standard verification.

Naccache et al. [2003] propose a different method to reduce the size of contexts. This size is normally equal for all contexts of a given method, and is computed using the maximum values for the stack height and for the number of local variables used throughout the method (these values are provided by the Java compiler). They propose to preprocess each method, before verification, to compute a different context size for each basic block. The size of the context depends on the stack height at the merge points. Moreover, they unify local variables according to definitions and uses in each basic blocks. Since the stack is generally empty at the merge points and each basic block uses only a subset of the local variables, this optimization results in a significant reduction in memory requirements. The reductions are significant but comparable with ours (the dictionary size is from 6% to 50% smaller than in standard verification). This technique could be combined with multipass verification to reduce the space required by each subanalysis.

Deville and Grimaud [2002] propose using the persistent memory for storing the data structures needed for the verification process. Their strategy holds all data structures in RAM as long as possible, and swaps them in persistent memory when RAM space is missing. A special type of encoding is proposed since persistent memory cells have a limited number of writing cycles. Both multipass and multipass+LBV have the advantage that most writes to persistent memory can be avoided.

Another work by Hyppönen et al. [2003] relies on a distrusted and not memory constrained terminal to store the dictionary. Dictionary entries are loaded/stored on demand from/to the terminal that is used as a remote repository. Each context is saved out in the terminal together with a message authentication code (MAC) computed on the card. Whenever the verifier on the card needs to read a context, it retrieves it from the terminal and checks for its integrity by verifying the MAC. In this way, only one context at time (plus some other minor data structures) is saved in the RAM of the card. We think that multipass+LBV should be a simpler and more effective way to exploit continued communication with the terminal during verification, since there would be no need for a MAC.

We have proposed in a previous paper [Bernardeschi et al. 2006a] a bytecode verifier that reduces the size of the dictionary. This verifier uses a dataflow algorithm that exploits the control flow dependencies to partition the method in control regions, and analyzes the regions one-by-one [Bernardeschi et al. 2006b]. Each region is analyzed by applying the standard verification algorithm and the size of the dictionary is reduced since not all entries need to be kept in the memory at the same time. This algorithm can reduce memory requirements from 27% to 58% compared to the standard verifier. Note that this algorithm reduces the maximum number of contexts that must be stored in the dictionary, and it can be combined with a strategy to reduce the size of each context. In particular, each pass of multipass could be performed using this algorithm. However, since both algorithms trade time for space, the combined algorithm would probably be too slow.

The problem of bytecode verification of interface types has been addressed by several authors [Goldberg 1998; Knoblock and Rehof 2001]. Recall that, in the presence of interface types, the set $\mathbb{T}$ of types ordered by the "assignable to" relation is generally no longer a lattice. The problem is solved by switching to $\wp(\mathbb{T})$, which is always a complete lattice. Each context element (register or stack element) may now contain a set of types, the least upper bound becomes set union and type constraints hold if they hold for all elements in the set. Dedekind-MacNeille completion [Davey and Priestley 2002] of $\mathbb{T}$ may be used to find a sufficient set of sets of types before verification starts. The advantage is that these sets of types may be given a name and used as simple types, so that the implementation of the verification does not need to deal with sets of types. In our decomposition there is the potential for a novel way to deal with interfaces. Namely, interface types may be used in a multipass verification just like all other types. The fact is that the closure operator φ_τ is defined only in terms of $\sqsubseteq$, so its domain does not need to be a lattice. However, its codomain is $\mathscr{B}_\tau$ which always is the two elements lattice $\{\tau, \top\}$, with $\tau \sqcup \tau = \tau$ and

$\tau \sqcup \top = \top \sqcup \tau = \top \sqcup \top = \top$. So, even if τ is an interface type, we can use lattice $\mathscr{B}_\tau$ in a verification pass that checks for absence of type errors on τ only. The intuition is that the least upper bound of two types in domain $\mathscr{B}_\tau$ only has to answer the simple question: "are both types assignable to interface τ?", instead of: "to which interfaces are both types assignable?" We think that the possible presence of arrays of interface types does not cause problems, and that lattice $\mathscr{A}^k_\tau$ can be used even when τ is an interface type, but we have not explored all the details yet.

An attractive goal of abstract interpretation theory is the systematic design of abstract domains [Cortesi et al. 1997; Cousot and Cousot 1979]. The idea is to define operators that combine or refine abstract domains, in order to obtain new domains with a desired property, such as greater precision or expressiveness. Relative and absolute complete shells [Bernardeschi et al. 2006a] are examples of refinement operators: they enhance an abstract domain, adding the minimal set of elements taken from the concrete domain, in order to achieve completeness wrt a given semantic function. Other examples include reduced product, disjunctive completion [Cousot and Cousot 1979], and Heyting completion [Giacobazzi and Scozzari 1998]. Parallel and Serial decomposition are related to this set of ideas, but they go in the opposite direction of simplifying existing abstractions. In this paper, simplification was motivated by the need to perform the analysis on a memory constrained device, but there are clearly other scenarios where a very complex analysis could benefit from being decomposed into a set of simpler analyses. Examples of domain operations that decompose or simplify abstract domains are complementation [Cortesi et al. 1997] and least disjunctive basis [Giacobazzi and Ranzato 1998]. Parallel decomposition is not a new domain operator, but it may be regarded as a guideline on how to combine the use of relative and absolute complete shells [Bernardeschi et al. 2006a] with reduced product [Cousot and Cousot 1979]. This has worked well in the settings of bytecode verification, and its practical applicability outside this domain should be explored. Serial decomposition, that is, using information obtained in an analysis to simplify a second analysis, seems more attractive. However, its present formulation is very specific to the kind of analysis used in bytecode verification, as formalized by our definition of next in Section 4.1.2. Thus, a generalization is required. Finally, there is a potential relation between the concept of residual function (definition 5.1) and the concept of complement of an abstract domain [Cousot and Cousot 1979], and this relation should be fully explored.

8. CONCLUSIONS

In this article we have presented a set of strategies for the decomposition of standard bytecode verification into simpler and more space efficient subanalyses. For each strategy, we have shown its equivalence to standard verification: the same set of errors is found in all cases. Each subanalysis is an abstract interpretation of standard verification, where only a subset of types is remembered and all other types are mapped to the "unknown" type. Thus, each analysis requires less space than standard verification, since each stack or register element in

the analysis requires fewer bits to encode its contents. In particular, analyses that remember a single type require a single bit for each element in order for them, to distinguish between the remembered type and the unknown type.

The first strategy we proposed is parallel decomposition. This strategy starts from the set of errors that may be generated by a given method. This set can be obtained by simply inspecting the method. Then, the set of errors is decomposed into a family of subsets. For each subset, abstract interpretation theory is used to constructively build the smallest domain of types that must be remembered to prove precisely the absence of all errors in the subset. The shape of the domains depends on the bytecode instructions used in the method. Finally, each subanalysis is run in turn, and the method is considered correct if it passes all analyses. We applied this strategy assuming a decomposition of the set of errors into singleton sets; that is, each analysis focuses on a single error. We have shown that, under this assumption, the domains that must be used for each subanalysis have simple shapes that only depend on some parameters, and these parameters can be determined by simply inspecting the method.

Our second strategy is serial decomposition. This can be used to further decompose the subanalyses of parallel decomposition when these analyses have to remember more than one type. The idea is to find those types T, inside the domain, that bytecode instructions may transform into other types Q in the same domain. The existence of this relation among types is what causes the domain to contain more than one type. We break this relation, by first performing an analysis that remembers types in T, and noting where new instances of types in Q appear. This information is then used for an analysis that only remembers types in Q and maps all types in T to the unknown type. We have shown that the right combination of parallel and serial decomposition produces a decomposition where each subanalysis remembers a single type only. This is obtained at the cost of storing some inter-analysis information.

Finally, we have shown some statistics taken from a large set of applications. These statistics highlight the viability of our approach to bytecode verification, as far as memory requirements are concerned. Serial decomposition generally only needs 4% of the memory required by standard verification, never exceeding 9% even for very large applications. Parallel decomposition generally uses 10–20% of the memory required by standard verification, never exceeding 40%. Both parallel and serial decomposition can be applied to lightweight bytecode verification (LBV) as well. In this case, they are often able to reduce the size of the certificate. In particular, the size of the certificate in serial decomposition ranges from 26% to 70% of the certificate in LBV.

ACKNOWLEDGMENTS

The authors would like to thank all the anonymous reviewers for their valuable comments and suggestions that greatly improved the content of this work.

REFERENCES

Barthe, G., Courtieu, P., Dufay, G., and de Sousa, S. M. 2005. Tool-assisted specification and verification of typed low-level languages. *J. Autom. Reason. 35,* 4, 295–354.

BERNARDESCHI, C., DE FRANCESCO, N., AND MARTINI, L. 2003. Efficient bytecode verification using immediate postdominators in control flow graphs. In *Proceedings of the OTM Workshops*. Lecture Notes in Computer Science, vol. 2889. Springer, 425–436.

BERNARDESCHI, C., FRANCESCO, N. D., LETTIERI, G., MARTINI, L., AND MASCI, P. 2007. Decomposing bytecode verification by abstract interpretation for space awareness in embedded systems. Tech. rep. IET-07-01, Dipartimento di Ingegneria dell'Informazione, Università di Pisa, `http://www.ing.unipi.it/~o1103499/papers/IET-07-01.pdf`.

BERNARDESCHI, C., LETTIERI, G., MARTINI, L., AND MASCI, P. 2006a. Using control dependencies for space-aware bytecode verification. *Comput. J. 49,* 2, 234–248.

BERNARDESCHI, C., LETTIERI, G., MARTINI, L., AND MASCI, P. 2006b. Using postdomination to reduce space requirements of data flow analysis. *Inform. Proc. Lett. 98,* 1, 11–18.

BIEBER, P., CAZIN, J., MAROUANI, A. E., GIRARD, P., LANET, J.-L., WIELS, V., AND ZANON, G. 2001. The PACAP prototype: A tool for detecting Java card illegal flow. Lecture Notes in Computer Science vol. 2041.

CASSET, L., BURDY, L., AND REQUET, A. 2002. Formal development of an embedded verifier for Java Card byte code. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*. IEEE Computer Society, 51–58.

CLICK, C. AND ROSE, J. 2002. Fast subtype checking in the HotSpot JVM. In *Proceedings of the Joint ACM-ISCOPE Conference on Java Grande (JGI'02)*. ACM Press, New York, 96–107.

COGLIO, A. 2003. Improving the official specification of Java bytecode verification. *Concur. Computat. Pract. Exper. 15,* 2, 155–179.

COGLIO, A. 2004. Simple verification technique for complex Java bytecode subroutines. *Concur.—Pract. Exper. 16,* 7, 647–670.

COHEN, N. H. 1991. Type-extension type test can be performed in constant time. *ACM Trans. Program. Lang. Syst. 13,* 4, 626–629.

CORTESI, A., FILÉ, G., RANZATO, F., GIACOBAZZI, R., AND PALAMIDESSI, C. 1997. Complementation in abstract interpretation. *ACM Trans. Program. Lang. Syst. 19,* 1, 7–47.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 238–252.

COUSOT, P. AND COUSOT, R. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 269–282.

COUSOT, P. AND COUSOT, R. 1992. Abstract interpretation frameworks. *J. Logic Comput. 2*, 511–547.

DAVEY, B. A. AND PRIESTLEY, H. A. 2002. *Introduction to Lattices and Order*. Cambridge University Press.

DEVILLE, D. AND GRIMAUD, G. 2002. Building an "impossible" verifier on a Java Card. In *Proceedings of the 2nd USENIX Workshop on Industrial Experiences with Systems Software*. USENIX, 15–24.

FREUND, S. N. AND MITCHELL, J. C. 1999. The type system for object initialization in the Java bytecode language. *ACM Trans. Program. Lang. Syst. 21,* 6, 1196–1250.

GIACOBAZZI, R. AND RANZATO, F. 1998. Optimal domains for disjunctive abstract interpretation. *Sci. Comput. Program. 32,* 1-3, 177–210.

GIACOBAZZI, R., RANZATO, F., AND SCOZZARI, F. 2000. Making abstract interpretations complete. *J. ACM 47,* 2, 361–416.

GIACOBAZZI, R. AND SCOZZARI, F. 1998. A logical model for relational abstract domains. *ACM Trans. Program. Lang. Syst. 20,* 5, 1067–1109.

GOLDBERG, A. 1998. A specification of Java loading and bytecode verification. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM Press, New York, 49–58.

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2000. *Java Language Specification*, 2nd Ed.: The Java Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

HYPPÖNEN, K., NACCACHE, D., TRICHINA, E., AND TCHOULKINE, A. 2003. Trading-off type-inference memory complexity against communication. In *Proceedings of the 5th International Conference Information and Communications Security (ICICS'03)*, S. Qing, D. Gollmann, and J. Zhou, Eds. Lecture Notes in Computer Science. Springer, 60–71.

JSR 2006. Jsr-000202 Java class file specification update. Tech. rep. JSR202, Java Community Process, http://jcp.org/en/jsr/detail?id=202.

KLEIN, G. AND NIPKOW, T. 2006. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst. 28,* 4, 619–695.

KNOBLOCK, T. B. AND REHOF, J. 2001. Type elaboration and subtype completion for Java bytecode. *ACM Trans. Program. Lang. Syst. 23,* 2, 243–272.

LANET, J.-L. AND REQUET, A. 2000. Formal proof of smart card applets correctness. In *Proceedings of International Conference on Smart Card Research and Applications (CARDIS'98)*, J.-J. Quisquater and B. Schneier, Eds. Lecture Notes in Computer Science, vol. 1820. Springer, 85–97.

LEROY, X. 2001. Java bytecode verification: an overview. In *Proceedings of the 13th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, vol. 2102. 265–285.

LEROY, X. 2002. Bytecode verification for Java smart card. *Softw. Pract. Exper. 32*, 319–340.

LEROY, X. 2003. Java bytecode verification: algorithms and formalizations. *J. Autom. Reason. 30*, 235–269.

LINDHOLM, T. AND YELLIN, F. 1999. *Java Virtual Machine Specification* 2nd Ed. Addison-Wesley Longman Publishing Co., Inc., Reading, MA.

MYCROFT, A. 1993. Completeness and predicate-based abstract interpretation. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*. ACM Press, New York, 179–185.

NACCACHE, D., TCHOULKINE, A., TYMEN, C., AND TRICHINA, E. 2003. Reducing the memory complexity of type-inference algorithms. In *Information and Communications Security*, R. Deng, S.Quing, F.Bao, and J. Zhou, Eds. Lecture Notes in Computer Science, vol. 2513, Springer-Verlag, 109–121.

NECULA, G. C. 1997. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*. ACM Press, New York, USA, 106–119.

QIAN, Z. 2000. Standard fixpoint iteration for Java bytecode verification. *ACM Trans. Program. Lang. Syst. 22,* 4, 638–672.

REQUET, A. 2003. A B model for ensuring soundness of a large subset of the Java Card virtual machine. *Sci. Comput. Program. 46,* 3, 283–306.

ROSE, E. 2003. Lightweight bytecode verification. *J. Autom. Reason. 31,* 3-4, 303–334.

ROSE, E. AND ROSE, K. 1998. Lightweight bytecode verification. In *Proceeding of the Workshop on the Formal Underpinning of Java*.

WARD, M. 1942. The closure operators of a lattice. *Annals Math. 43,* 2, 191–196.

Received January 2007; revised August 2007, December 2007; accepted March 2008