Combining a Verification Condition Generator for a Bytecode Language with Static Analyses*

Benjamin Grégoire[1] and Jorge Luis Sacchini[1,2]

[1] INRIA Sophia Antipolis - Méditerranée, France
[2] FCEIA, Univesidad Nacional de Rosario, Argentina
{Benjamin.Gregoire,Jorge-Luis.Sacchini}@sophia.inria.fr

Abstract. In Proof-Carrying Code, the verification condition generator (VCgen) generates a set of formulas whose validity implies that the code satisfies the consumer policy. Applying a VCgen to a bytecode language with exceptions (such as Java bytecode) can result in a large number of proof obligations, due to the amount of branching instructions. We present a VCgen for Java bytecode that uses static analyses to reduce the number of proof obligations. As a result, the task of producing a proof is simpler, and the subsequent proof terms smaller. We formalize the VCgen as a deep embedding in Coq and prove soundness with respect to the Bicolano formalization of the Java bytecode semantics.

1 Introduction

Proof-Carrying Code (PCC) [8] has been developed as a framework to guarantee safety in mobile scenarios. The code that is to be executed by a consumer needs to be accompanied with a proof (certificate) that it satisfies a required safety policy. The consumer checks that the certificate corresponds with a proof of safety of the code. Once the certificate is checked, the code can be safely executed. The task of generating such certificate, which can be a complex task depending on the safety policy, is delegated to the producer. The task of the consumer reduces to checking the certificate, which is in general much simpler.

A verification condition generator (VCgen) is used to generate the proof obligations that will ensure that the code satisfies the given safety policy. The VCgen is usually applied to annotated bytecode. It ensures, no matter which path in the control flow graph of the code is taken at runtime, that the safety policy is satisfied. Programs written in bytecode languages such as Java, that includes objects creation, dynamic method calls, and exception mechanism, have a high degree of branching code, due to the instructions that can throw runtime exceptions.

* This work was funded in part by the Sixth Framework programme of the European Community under the MOBIUS project FP6-015905. This paper reflects only the authors views and the Community is not liable for any use that may be made of the information contained therein.

G. Barthe and C. Fournet (Eds.): TGC 2007, LNCS 4912, pp. 23–40, 2008.

Consider the following excerpt of Java bytecode:

$$\begin{array}{ll} pc_1 & \texttt{istore}\ x \\ pc_2 & \texttt{getfield}\ f \\ pc_3 & \ldots \end{array}$$

A VCgen (denoted by VC) generates two proof obligations for the program point pc_2:

$$\begin{aligned} lv(x) \neq \text{null} \Rightarrow\ & VC(pc_3) \\ \wedge\ lv(x) = \text{null} \Rightarrow\ & VC(pc_{\text{exc}}), \end{aligned}$$

where lv access the local variable array, and pc_{exc} is the program point corresponding to the exception handler. For every instruction that can throw a runtime exception, the VCgen returns two proof obligations: one corresponding to normal execution, and another corresponding to exceptional execution. Usually a program contains many of these instructions, which results in an explosion in the number of proof obligations.

The use of static analyses, such as null-pointer analysis, can ensure that the reference above is non-null and, therefore, it is not necessary to generate a proof obligation for the exceptional execution. In such case, the VCgen will generate the following condition:

$$lv(x) \neq \text{null} \Rightarrow VC(pc_3)\ .$$

Static analyses can provide the required information to reduce many proof obligations that are generated from instructions that may throw exceptions, as in the example above.

We show, in Sect. 3, a way to combine a VCgen with static analyses, to reduce the control flow graph of the program, and hence, the number of proof obligations. We will exemplify the approach using a simple null-pointer analysis, and sketch the proof of soundness of the VCgen.

We have formalized the VCgen as a deep embedding in Coq based on the Bicolano formalization of the Java bytecode semantics, which is described in Sect. 2.

The certificates for our VCgen need to include, besides the proofs of safety, the information collected from the static analyses. We discuss the generation and checking of these certificates in Sect. 4.

2 Preliminaries

We will base our development on the Bicolano formalization [10]. Bicolano is a formalization in Coq of the Java Virtual Machine (JVM), which includes object creation, virtual methods, exception handling, and arrays. We will describe only a small and reduced fragment of the formalization, needed for our purposes.

A program consists of a set of classes, each containing a set of fields and methods. A method is composed by a body (sequence of instructions) and a specification (this component will be described later, when describing the VCgen). The

instructions considered in this paper are: `getfield` *FieldId*, `putfield` *FieldId*, `iload` $\mathbb{Z}$, `istore` $\mathbb{Z}$, `invokevirtual` *Method*, `athrow` *ClassName*, `ireturn`.

For each method m, PC_m denotes the set of program points corresponding to the instructions of m. Most of our definitions refer to a single method, therefore, for simplicity, we will omit the reference to the method when is clear from the context. *State* denotes the type of program states; each $s : State$ is a triple, $s = (h, os, l)$, where h is the heap, os is the operand stack, and l is the local variables. The type of values is defined as $Value = Int + Loc$, where *Int* is the type of integers, and *Loc* the type of reference values. The operand stack is modeled by a list, $Stack = list\ Value$. The local variables are modeled by a function $LocalVar = \mathbb{Z} \to Value$. The heap is modeled by an abstract data type, *Heap*, with operations for creating objects (*newObj*) and accessing their fields (*get*, *update*). The type of *initial states* for a method is $State_i = Heap \times LocalVar$, and the type of *final states* is $State_r = Heap \times ReturnVal$, where $ReturnVal = Value + Loc$, representing normal termination of a method with a value, or abnormal termination with the location of an exception object. *Exc* is the type of possible exceptions (e.g. `NullPointer`, `ArrayBound`, ...).

Operational Semantics. The operational semantics is defined only for well-typed programs, so we will assume that all programs considered are well-typed.

The semantics is defined by two relations $\longrightarrow : Method \to PC \times State \to PC \times State \to Prop$ and $\downarrow : Method \to PC \times State \to State_r \to Prop$, where $m \vdash (pc, s) \longrightarrow (pc', s')$ represents execution of one instruction in a method, and $m \vdash (pc, s) \downarrow s'$ represents execution of one instruction that reaches a final state. We will write $\longrightarrow^*$ to mean the reflexive, transitive closure of $\longrightarrow$, and $\downarrow^*$ to mean the relation $\longrightarrow^* \circ \downarrow$ (i.e., many steps of $\longrightarrow$ followed by one step of $\downarrow$).

To make the presentation clearer, we define two auxiliary relations: $\to_{\mathsf{JVM}} : Method \to PC \times State \to Exc \to Prop$ and $\to_{\mathsf{EXC}} : Method \to PC \times State \to Heap \times Loc \to Prop$, where $m \vdash (pc, s) \to_{\mathsf{JVM}} e$ indicates that executing the instruction at pc in state s results in the JVM exception e being thrown (e.g., the exception `NullPointer` is thrown when accessing a null reference), and $m \vdash (pc, s) \to_{\mathsf{EXC}} (h, loc)$ indicates that the exception pointed by loc in heap h was thrown when executing the instruction at pc in state s, and we need to look for an exception handler. To search for the handler code corresponding to an exception, we have a function $excHandler : Method \to PC \times Heap \times Loc \to PC + \bot$, that returns $\bot$ when no handler is found in the current method.

Figure 1 shows a few rules of the big-step operational semantics. The function *instructionAt* returns the instruction corresponding to a given program point. The function initArgs : $Value \times list\ Value \to LocalVar$ builds the initial local variables for a method call, where the first argument is a reference to the object, and the second argument is the list of arguments of the method. The infix operators :: and ++ represent the cons function for lists and the concatenation of lists, respectively.

In the rules for the instruction `invokevirtual`, there is the implicit assumption that the length of *args* is the same as the number of arguments of the method m'. The first rule for `invokevirtual` corresponds to the case where the

$$\frac{\mathit{instructionAt}(pc) = \texttt{athrow} \qquad loc \neq \text{null}}{(pc, (h, loc :: os, l)) \rightarrow_{\mathsf{EXC}} (h, loc)}$$

$$\frac{\mathit{instructionAt}(pc) = \texttt{athrow} \qquad loc = \text{null}}{(pc, (h, loc :: os, l)) \rightarrow_{\mathsf{JVM}} \texttt{NullPointer}}$$

$$\frac{\mathit{instructionAt}(pc) = \texttt{getfield}\ f \qquad \mathit{get}(h, loc, f) = v \qquad loc \neq \text{null}}{(pc, (h, loc :: os, l)) \longrightarrow (pc + 1, (h, v :: os, l))}$$

$$\frac{\mathit{instructionAt}(pc) = \texttt{getfield}\ f \qquad loc = \text{null}}{(pc, (h, loc :: os, l)) \rightarrow_{\mathsf{JVM}} \texttt{NullPointer}}$$

$$\frac{\mathit{instructionAt}(pc) = \texttt{putfield}\ f \qquad \mathit{update}(h, (loc, f), v) = h' \qquad loc \neq \text{null}}{(pc, (h, v :: loc :: os, l)) \longrightarrow (pc + 1, (h', os, l))}$$

$$\frac{\mathit{instructionAt}(pc) = \texttt{iload}\ x \qquad l(x) = v}{(pc, (h, os, l)) \longrightarrow (pc + 1, (h, v :: os, l))} \qquad \frac{\mathit{instructionAt}(pc) = \texttt{ireturn}}{(pc, (h, v :: os, l)) \downarrow (h, v)}$$

$$\frac{\mathit{instructionAt}(pc) = \texttt{istore}\ x \qquad l\,[x \mapsto v] = l'}{(pc, (h, v :: os, l)) \longrightarrow (pc + 1, (h, os, l'))}$$

$$\frac{\begin{array}{c}\mathit{instructionAt}(pc) = \texttt{invokevirtual}\ m' \\ l' = \text{initArgs}(loc, args) \qquad loc \neq \text{null} \qquad m' \vdash (pc_0, (h, [], l') \downarrow^* (h', v)\end{array}}{(pc, (h, args +\!\!+ loc :: os, l)) \longrightarrow (pc + 1, (h', v :: os, l))}$$

$$\frac{\begin{array}{c}\mathit{instructionAt}(pc) = \texttt{invokevirtual}\ m' \\ l' = \text{initArgs}(loc, args) \qquad loc \neq \text{null} \qquad m' \vdash (pc_0, (h, [], l') \downarrow^* (h', loc')\end{array}}{(pc, (h, args +\!\!+ loc :: os, l)) \rightarrow_{\mathsf{EXC}} (h', loc')}$$

$$\frac{\mathit{instructionAt}(pc) = \texttt{invokevirtual}\ m' \qquad loc = \text{null}}{(pc, (h, args +\!\!+ loc :: os, l)) \rightarrow_{\mathsf{JVM}} \texttt{NullPointer}}$$

$$\frac{(pc, (h, os, l)) \rightarrow_{\mathsf{JVM}} e \qquad (h', loc) = \mathit{newObj}(h, e)}{(pc, (h, os, l)) \rightarrow_{\mathsf{EXC}} (h', loc)}$$

$$\frac{(pc, (h, os, l)) \rightarrow_{\mathsf{EXC}} (h', loc) \qquad \mathit{excHandler}(pc, h', loc) = pc'}{(pc, (h, os, l)) \longrightarrow (pc', (h', loc :: [], l))}$$

$$\frac{(pc, (h, os, l)) \rightarrow_{\mathsf{EXC}} (h', loc) \qquad \mathit{excHandler}(pc, h', loc) = \bot}{(pc, (h, os, l)) \downarrow (h', loc)}$$

Fig. 1. Operational semantics (excerpt)

called method returns successfully a value, the second one corresponds to the case where the called method throws an exception (so we need to find a handler in the current method), and the third one corresponds to the case where the object is null.

The control flow graph of method m, denoted $\mathcal{G}_m$ is the set of edges (pairs of program points) (pc, pc') such that the program can go from pc to pc' in one step. This means, for instance, that instructions like `getfield` and `putfield` have an edge to the null-pointer exception handler (if there is one), and instructions

`athrow` and `invokevirtual` have edges to all handlers in their range, since we cannot (statically) determine which exceptions will be thrown.

VCgen. We consider a deep embedding of the VCgen in Coq. As shown in [13], deep embeddings have several advantages over shallow embeddings, such as, smaller proof terms, and the possibility to manipulate the generated proof obligation (e.g. by structural analysis).

The language for expressing assertions, Assrt, used by the VCgen is the following (excerpt):

$$
\begin{aligned}
\mathsf{Assrt} ::= {} & \mathsf{Assrt} \mathbin{\underline{\wedge}} \mathsf{Assrt} \mid \mathsf{Assrt} \mathbin{\underline{\vee}} \mathsf{Assrt} \mid \underline{\neg}\mathsf{Assrt} \mid \mathsf{Assrt} \mathbin{\underline{\Rightarrow}} \mathsf{Assrt} \\
\mid {} & \mathsf{V}\ \mathit{CompOp}\ \mathsf{V} \ldots \qquad \textit{(* assertions *)} \\
\mathsf{V} ::= {} & \mathsf{Lv}\ \mathbb{Z} \mid \mathsf{Hget}\ \mathsf{H}\ \mathsf{V}\ \mathit{FieldId} \mid \mathsf{St}\ \mathbb{Z} \mid \mathsf{Vvar}\ \mathit{Value} \mid \mathsf{Old}\ \mathsf{V} \mid \mathsf{result} \\
\mid {} & \mathsf{null} \mid \mathsf{V}\ \mathit{BinOp}\ \mathsf{V} \ldots \qquad \textit{(* values *)} \\
\mathsf{H} ::= {} & \mathsf{Hupd}\ \mathsf{H}\ \mathsf{V}\ \mathit{FieldId}\ \mathsf{V} \mid \mathsf{Hvar}\ \mathit{Heap} \mid \mathsf{CurrHeap} \qquad \textit{(* heap *)} \\
\mathsf{S_i} ::= {} & \mathsf{H} \times (\mathbb{Z} \to \mathsf{V}) \qquad \textit{(* initial states *)} \\
\mathsf{S} ::= {} & \mathsf{H} \times (\mathbb{Z} \to \mathsf{V}) \times (\mathbb{Z} \to \mathsf{V}) \qquad \textit{(* local states *)} \\
\mathsf{S_r} ::= {} & \mathsf{H} \times \mathsf{V} \qquad \textit{(* final states *)} \\
\mathit{BinOp} ::= {} & \underline{+} \mid \underline{-} \ldots \qquad \mathit{CompOp} ::= \underline{=} \mid \underline{\neq} \mid \underline{\leq} \mid \underline{\leq} \ldots
\end{aligned}
$$

In Assrt we have the usual logical operators ($\underline{\wedge}$, $\underline{\vee}$, $\underline{\Rightarrow}$, ...), including equality and comparison. The operators are underlined to differentiate them from the operators of Coq. The type of values, V, allows to access the local variables (Lv), the stack (St), the heap (Hget(h, loc, f) access the field f of object loc in h), values in the initial state of a method (Old), the result of a method (result), and permits to express binary operations between values. The heap, represented by H, allows to update values (Hupd(h, loc, f, v) updates the field f of object loc with the value v), and access to the current heap (CurrHeap). Note that using Vvar and Hvar we can define a *lift* function that takes a *State* (resp. $State_i$, $State_r$) and returns a S (resp. $\mathsf{S_i}$, $\mathsf{S_r}$), so we will consider an element $s : State$ as having also type S (and similarly with $State_i$ and $State_r$).

The specification of a method is a tuple, $\mathcal{S}_m = (Pre, Post_{Nrml}, Post_{Exc}, A)$, where Pre : Assrt is the precondition; $Post_{Nrml}, Post_{Exc}$: Assrt are the postconditions corresponding to normal termination, and abnormal termination (due to an uncaught exception), respectively; and $A : PC \mapsto$ Assrt is a partial mapping called the *annotation table* containing assertions that are used by the VCgen to construct the proof obligations. We assume that all cycles in the control flow graph contain at least one annotated point.

The precondition states properties of the initial state, the postcondition relates the initial state with the final state, and the annotations relate the initial state with the local state. We also assume a well-formedness condition for specifications: accesses to the local variables or to the stack are in bound, only the postconditions can refer to result, expressions are well-typed, and preconditions do not use the Old construct.

The VCgen is based on weakest precondition calculus, defined by two mutually recursive functions: $\mathrm{wp}_{\mathsf{instr}} : \mathit{Method} \to PC \to$ Assertion, and $\mathrm{wp}_{\mathsf{annot}} : \mathit{Method} \to PC \to$ Assertion, where Assertion $= \mathsf{S_i} \to \mathsf{S} \to$ Assrt.

$\mathrm{wp}_{\mathsf{instr}}(pc)$ computes the weakest precondition (WP) corresponding to the instruction at pc, while $\mathrm{wp}_{\mathsf{annot}}(pc)$ returns the annotation of pc, or calls $\mathrm{wp}_{\mathsf{instr}}(pc)$ if pc is not annotated. To simplify the presentation, we define the functions $\mathrm{wp}_{\mathsf{JVM}} : \mathit{Method} \to \mathit{PC} \to \mathit{State}_i \to \mathit{Heap} \times \mathit{LocalVar} \to \mathit{Exc} \to \mathsf{Assrt}$, and $\mathrm{wp}_{\mathsf{EXC}} : \mathit{Method} \to \mathit{PC} \to \mathit{State}_i \to \mathit{Heap} \times \mathit{LocalVar} \to \mathit{Loc} \to \mathsf{Assrt}$, that roughly corresponds to relations $\to_{\mathsf{JVM}}$ and $\to_{\mathsf{EXC}}$, and returns the WP when an exception is thrown. They look for the exception handler and return the WP of the first point of the handler, or return the postcondition corresponding to abnormal termination if no handler is found in the method.

The general form of $\mathrm{wp}_{\mathsf{instr}}(pc)$ contains a conjunction for each branch of $\mathcal{G}$:

$$\mathrm{wp}_{\mathsf{instr}}(pc, s_0, s) = \bigwedge_{(pc,pc')\in\mathcal{G}} C_{(pc,pc')}(s) \mathrel{\underline{\Rightarrow}} P_{(pc,pc')}(\mathrm{wp}_{\mathsf{annot}}(pc'), s_0, s), \qquad (1)$$

where $C_{(pc,pc')}(s)$ is a necessary condition that needs to be satisfied in order for the program to go from pc to pc' in one step, and $P_{(pc,pc')}(\mathrm{wp}_{\mathsf{annot}}(pc'), s_0, s)$ is a predicate transformer that updates s in correspondence with the instruction at pc and applies it to $\mathrm{wp}_{\mathsf{annot}}(pc')$. To compute $\mathrm{wp}_{\mathsf{instr}}(pc, s_0, s)$ we proceed by case analysis on the instruction at pc, and state s. We show a few cases in Fig. 2. For readability, we change the first parameter, pc, for the corresponding instruction. For instance, the condition $C_{(pc,pc')}$ for the instructions `getfield` and `putfield` is that the top of the stack contains a null or non-null value depending on the branch. For `iload` and `ireturn`, the condition is simply $\underline{\mathsf{true}}$.

The function $\mathrm{wp}_{\mathsf{annot}}$ is defined as follows:

$$\mathrm{wp}_{\mathsf{annot}}(pc, s_0, s) = \begin{cases} \mathsf{subst}(s_0, s, A(pc)) & \text{if } pc \in \mathrm{dom}(A), \\ \mathrm{wp}_{\mathsf{instr}}(pc, s_0, s) & \text{otherwise .} \end{cases}$$

The function $\mathsf{subst} : \mathsf{S_i} \to \mathsf{S} \to \mathsf{Assrt} \to \mathsf{Assrt}$, performs a substitution on an expression; $\mathsf{subst}(s_0, (h, os, lv), a)$ traverses a replacing $\mathsf{CurrHeap}$ by h, $\mathsf{St}\ n$ by $os(n)$, and $\mathsf{Lv}\ x$ by $lv(x)$. The values protected by Old are substituted using the initial state. The function $\mathsf{subst}_{\mathsf{Post}} : \mathsf{S_i} \to \mathsf{S_r} \to \mathsf{Assrt} \to \mathsf{Assrt}$ does the same as subst, but also replacing result.

We need an interpretation function, $\mathsf{interp} : \mathsf{Assrt} \to \mathit{State}_i \to \mathit{State} \to \mathit{Prop}$ to transform an expression into a Coq proposition. $\mathsf{interp}(a, s_0, s)$ traverses a replacing the constructors for the corresponding functions in the Bicolano formalization, and replacing the references to the state with the values in s and s_0. The function $\mathsf{interp}_{\mathsf{Post}} : \mathsf{Assrt} \to \mathit{State}_i \to \mathit{State}_r \to \mathit{Prop}$ is the same as interp except that it also replaces result. This function are defined for well-formed specifications, returning an undefined value otherwise.

We say an assertion $a : \mathsf{Assrt}$ is valid in state s and initial state s_0, and write it $s_0, s \models a$, if $\mathsf{interp}(a, s_0, s)$ is valid in Coq. Similarly with $\mathsf{interp}_{\mathsf{Post}}$. We say an assertion $a : \mathsf{Assertion}$ is valid in state s and initial state s_0, and write it $s_0, s \models a$, if $(s_0, s \models a(\mathbf{s_i}, \mathbf{s}))$, where $\mathbf{s_i} = (\mathsf{CurrHeap}, \lambda x.\mathsf{Lv}\, x)$ and $\mathbf{s} = (\mathsf{CurrHeap}, \lambda x.\mathsf{St}\, x, \lambda x.\mathsf{Lv}\, x)$. We will write $\models a$ to mean $\forall s_0, s, (s_0, s \models a)$.

The following definition states the proof obligations needed to verify that a method complies with its specification.

$$\mathrm{wp}_{\mathsf{instr}}(\texttt{athrow}\ f, s_0, (h, loc :: os, l)) =$$
$$loc \underline{\neq} \mathrm{null} \underline{\Rightarrow} \mathrm{wp}_{\mathsf{EXC}}(pc, s_0, (h, l), loc)$$
$$\underline{\wedge}\ loc \underline{=} \mathrm{null} \underline{\Rightarrow} \mathrm{wp}_{\mathsf{JVM}}(pc, s_0, (h, l), \texttt{NullPointer})$$

$$\mathrm{wp}_{\mathsf{instr}}(\texttt{getfield}\ f, s_0, (h, loc :: os, l)) =$$
$$loc \underline{\neq} \mathrm{null} \underline{\Rightarrow} \mathrm{wp}_{\mathsf{annot}}(pc+1, s_0, (h, \mathsf{Hget}(h, loc, f) :: os, l))$$
$$\underline{\wedge}\ loc \underline{=} \mathrm{null} \underline{\Rightarrow} \mathrm{wp}_{\mathsf{JVM}}(pc, s_0, (h, l), \texttt{NullPointer})$$

$$\mathrm{wp}_{\mathsf{instr}}(\texttt{iload}\ x, s_0, (h, os, l)) = \mathrm{wp}_{\mathsf{annot}}(pc+1, s_0, (h, l(x) :: os, l))$$

$$\mathrm{wp}_{\mathsf{instr}}(\texttt{ireturn}, s_0, (h, v :: os, l)) = \mathsf{subst}_{\mathsf{Post}}(s_0, (h, v), Post_{Nrml})$$

$$\mathrm{wp}_{\mathsf{instr}}(\texttt{istore}\ x, s_0, (h, v :: os, l)) = \mathrm{wp}_{\mathsf{annot}}(pc+1, s_0, (h, os, l\,[x \mapsto v]))$$

$$\mathrm{wp}_{\mathsf{instr}}(\texttt{putfield}\ f, s_0, (h, v :: loc :: os, l)) =$$
$$loc \underline{\neq} \mathrm{null} \underline{\Rightarrow} \mathrm{wp}_{\mathsf{annot}}(pc+1, s_0, (\mathsf{Hupd}(h, loc, f, v), os, l))$$
$$\underline{\wedge}\ loc \underline{=} \mathrm{null} \underline{\Rightarrow} \mathrm{wp}_{\mathsf{JVM}}(pc, s_0, (h, l), \texttt{NullPointer})$$

$$\mathrm{wp}_{\mathsf{instr}}(\texttt{invokevirtual}\ m, s_0, (h, args +\!\!+ loc :: os, l) =$$
$$loc \underline{\neq} \mathsf{null} \underline{\Rightarrow} \left(\begin{array}{l} \mathsf{subst}((h, li), (h, [], li), Pre(m)) \\ \underline{\wedge}\ \mathrm{PostNormal} \\ \underline{\wedge}\ \mathrm{PostExc} \end{array} \right)$$
$$\underline{\wedge}\ loc \underline{=} \mathsf{null} \underline{\Rightarrow} \mathrm{wp}_{\mathsf{JVM}}(pc, s_0, (h, l), \texttt{NullPointer})$$

$$li = \mathrm{initArgs}(loc, args)$$
$$\mathrm{PostNormal} = \left\{ \begin{array}{l} \underline{\forall}\ r, \underline{\forall}\ h', \mathsf{subst}_{\mathsf{Post}}((h, li), (h', r), Post_{Nrml}(m)) \underline{\Rightarrow} \\ \mathrm{wp}_{\mathsf{annot}}(pc+1, s_0, (h', r :: os, l)) \end{array} \right.$$
$$\mathrm{PostExc} = \left\{ \begin{array}{l} \underline{\forall}\ loc'\ \underline{\forall}\ h', \mathsf{subst}_{\mathsf{Post}}((h, li), (h', loc'), Post_{Exc}(m)) \underline{\Rightarrow} \\ \mathrm{wp}_{\mathsf{EXC}}(pc, s_0, (h, l), loc') \end{array} \right.$$

Fig. 2. Weakest precondition for instructions (excerpt)

Definition 1. *Given a program p and a method m,* $\mathsf{certifiedMethod}(m)$ *stands for the following proposition:*

$$\forall s_0, (s_0, \overline{s_0} \models Pre(m) \underline{\Rightarrow} \mathrm{wp}_{\mathsf{annot}}(pc_0, \mathbf{s_i}, \mathbf{s}))$$
$$\wedge \bigwedge_{pc \in \mathrm{dom}(A)} \forall s_0\ s, (s_0, s \models A(pc) \underline{\Rightarrow} \mathrm{wp}_{\mathsf{instr}}(pc, \mathbf{s_i}, \mathbf{s}))\,,$$

where $\overline{s_0} = (h, [], l)$ *if* $s_0 = (h, l)$*, i.e.* $\overline{s_0}$ *is the state obtained by extending the initial state* s_0 *with an empty operand stack.*

To verify a method, we need to check that the precondition implies the WP of the first instruction, and for each annotated point pc, the annotation implies the WP of the instruction at pc.

The soundness is proved with respect to the operational semantics.

Theorem 1 (Soundness of the VCgen). *Let* p *be a program and* m *a method. Assume we have a proof of* $\mathsf{certifiedMethod}(m')$*, for all methods* m' *in the program, and a state* (pc, s) *such that* $s_0, s \models \mathrm{wp}_{\mathsf{annot}}(m, pc)$*. Then the following holds:*

- *if* $(pc, s) \longrightarrow (pc', s')$*, then* $s_0, s' \models \mathrm{wp}_{\mathsf{annot}}(m, pc')$*,*
- *if* $(pc, s)\downarrow(h, r)$*, with* $r \in Value$*, then* $s_0, (h, r) \models Post_{Nrml}(m)$*,*
- *if* $(pc, s)\downarrow(h, loc)$*, with* $loc \in Loc$*, then* $s_0, (h, loc) \models Post_{Exc}(m)$*.*

The proof is divided in the following lemmas.

Lemma 1. *If* $(pc, s) \longrightarrow (pc', s')$*, then* $s \models C_{(pc,pc')}(s)$*.*

Lemma 2. *If* $(pc, s) \longrightarrow (pc', s')$*, where* $instructionAt(pc) \neq$ `invokevirtual`*, and* $s_0, s \models \mathrm{wp}_{\mathsf{instr}}(m, pc)$*, then* $s_0, s' \models \mathrm{wp}_{\mathsf{annot}}(m, pc')$*.*

Proof. By case analysis on the current instruction, using Lemma 1.

Lemma 3. *If we have a proof of* $\mathsf{certifiedMethod}(m)$*, and* $s_0, s \models \mathrm{wp}_{\mathsf{annot}}(m, pc)$*, then* $s_0, s \models \mathrm{wp}_{\mathsf{instr}}(m, pc)$*.*

Proof. If pc is not annotated it is trivial, since $\mathrm{wp}_{\mathsf{annot}}(m, pc)$ is the same as $\mathrm{wp}_{\mathsf{instr}}(m, pc)$. Otherwise, we have $\mathrm{wp}_{\mathsf{annot}}(m, pc, s_0, s) = \mathsf{subst}(s_0, s, A(pc))$, and we conclude using the fact that we have a proof of $\mathsf{certifiedMethod}(m)$. □

Lemma 4. *Let* p *be a program and* m *a method. Assume we have a proof of* $\mathsf{certifiedMethod}(m')$*, for all methods* m' *in the program, and a state* (pc, s) *such that* $s_0, s \models \mathrm{wp}_{\mathsf{annot}}(m, pc)$*. Then the following holds:*

- *if* $(pc, s) \hookrightarrow (pc', s')$*, then* $s_0, s' \models \mathrm{wp}_{\mathsf{annot}}(m, pc')$*,*
- *if* $(pc, s) \hookrightarrow (h, r)$*, with* $r \in Value$*, then* $s_0, (h, r) \models Post_{Nrml}(m)$*,*
- *if* $(pc, s) \hookrightarrow (h, loc)$*, with* $loc \in Loc$*, then* $s_0, (h, loc) \models Post_{Exc}(m)$*,*

where the relation $\hookrightarrow$*:* $Method \to PC \times State \to PC \times State + State_r \to Prop$ *is defined in Fig. 3.*

Proof. The proof proceeds by induction in the relation $\hookrightarrow$. The relation $call : Method \to PC \times State \to State_r \to Method \to PC \times State \to PC \times State + State_r \to Prop$ determines the connection between the states of execution when calling a method. If $call(m, (pc, s), r, m', (pc_0(m'), s'), t)$ is valid, then it means that in method m, $instructionAt(pc) =$ `invokevirtual` m', $(pc_0(m'), s')$ is the initial state of execution in m' (i.e. it has an empty operand stack and the local

$$\frac{}{(pc,s) \hookrightarrow (pc,s)} \qquad \frac{(pc,s)\downarrow r}{(pc,s) \hookrightarrow r} \qquad \frac{\begin{array}{c}\mathit{instructionAt}(pc) \neq \texttt{invokevirtual} \\ (pc,s) \longrightarrow (pc',s') \qquad (pc',s') \hookrightarrow t\end{array}}{(pc,s) \hookrightarrow t}$$

$$\frac{m' \vdash (pc_0(m'),s') \hookrightarrow (h,loc) \qquad \mathit{call}(m,(pc,s),(h,loc),m',(pc_0(m'),s'),(h,loc))}{m \vdash (pc,s) \hookrightarrow (h,loc)}$$

$$\frac{\begin{array}{c}m' \vdash (pc_0(m'),s') \hookrightarrow r \\ m \vdash (pc'',s'') \hookrightarrow t \qquad \mathit{call}(m,(pc,s),r,m',(pc_0(m'),s'),(pc'',s''))\end{array}}{m \vdash (pc,s) \hookrightarrow t}$$

Fig. 3. Alternative big-step relation

variables are built from the arguments in the stack of s), and if r is the final state in m', then t is the next state of execution in m. If t is a final state, then it means that m' has thrown an exception that is uncaught in m. If t is a normal state, it means that, either m' has returned successfully (and the return value of r is in the top of the stack of t), or that has thrown an exception that was caught in m (and t contains the location of the exception handler). Note that *call* does not enforce any relation between the initial state s' and the final state r in m', which will be enforced by $\hookrightarrow$.

The relation $\hookrightarrow$ gives us the right induction principle for the `invokevirtual` instruction that was not addressed in Lemma 2. □

Finally, using Lemma 4 we can prove Theorem 1, by proving that the relation $\hookrightarrow$ is equivalent to the reflexive, transitive closure of the operational semantics.

3 Reducing Proof Obligations

In this section, we show a way to use static analysis to reduce the number of proof obligations generated by the VCgen described in the previous section. Roughly, the analysis is applied to the program, and the results are given to the VCgen. The VCgen can use this information to remove the proof obligations corresponding to paths in the code that cannot be taken at runtime. For example, if a null-pointer analysis can prove the absence of null-pointer exceptions, then the VCgen does not generate proof obligations corresponding to null-pointer exception handlers.

3.1 Preliminary Definitions

We consider a fixed program p and a method m with specification $\mathcal{S}$. pc_0 denotes the first instruction. We will make a small modification to the control flow graph and the semantics. To the set of program points we add two nodes: pc_{N} that represents normal termination, and pc_{E} that represents abnormal termination. The control flow graph $\mathcal{G}$ is augmented with edges of the form (pc, pc_{N}) for each pc that corresponds to a `ireturn` instruction, and (pc, pc_{E}) for each pc that corresponds to an instruction that can throw an exception that is not caught in m

(this includes `athrow` and `invokevirtual`). We make a small modification to the rules of the operational semantics. We change the relation $\downarrow$, so that, instead of $(pc, s)\downarrow(h, v)$ we have $(pc, s)\longrightarrow(pc_{\mathrm{N}}, (h, v :: [], lv))$, and instead of $(pc, s)\downarrow(h, loc)$ we have $(pc, s)\longrightarrow(pc_{\mathrm{E}}, (h, loc :: [], lv))$. The state considered at the nodes pc_{N} and pc_{E} consist of a heap, an operand stack with just one element (the return value, or location of the exception object, respectively), and undefined local variables (since a return state does not contain a local variable array).

Definition 2. *A* static analysis $\mathcal{A}$ *is a tuple* (D, t, I, f)*, where*

- $D = (D, \sqsubseteq, \bot, \top, \sqcap, \sqcup)$ *is a complete lattice that denotes the domain of the analysis,*
- $t : \mathcal{G} \to (D \to D)$ *is the transfer function, such that for each* $(pc, pc') \in \mathcal{G}$*,* $t_{(pc,pc')}$ *is a monotone function in* D*,*
- $I : PC \to D$ *is the initial value, and*
- $f \in \{\uparrow, \downarrow\}$ *denotes the direction of the analysis. If* $f = \uparrow$ *we say the analysis is* backward*, and if* $f = \downarrow$ *we say is* forward*.*

Definition 3. *A* solution *(or* table*) for a forward analysis* $\mathcal{A} = (D, t, I, \downarrow)$ *is a function* $S : PC \to D$*, such that* $I(pc_0) \sqsubseteq S(pc_0)$*, and*

$$\forall pc \in PC, \bigsqcup_{(pc',pc)\in\mathcal{G}} t_{(pc',pc)}(S(pc')) \sqsubseteq S(pc) \ .$$

A solution *(or* table*) for a backward analysis* $\mathcal{A} = (D, t, I, \uparrow)$ *is a function* $S : PC \to D$*, such that* $S(pc_{\mathrm{N}}) \sqsubseteq I(pc_{\mathrm{N}})$*,* $S(pc_{\mathrm{E}}) \sqsubseteq I(pc_{\mathrm{E}})$*, and*

$$\forall pc \in PC, S(pc) \sqsubseteq \bigsqcap_{(pc,pc')\in\mathcal{G}} t_{(pc,pc')}(S(pc')) \ .$$

To find a solution for a given analysis, one needs to find a post-fixpoint to a specific function defined using the transfer function. We will not delve in this, see, e.g., [9] for more details.

To illustrate the combination of analysis and the VCgen, we will define a simple null-pointer analysis. We use a technique described in [3,12] for defining domains for bytecode analysis, where the values stored in the stack are related to their meaning.

Example 1. The null-pointer analysis $\mathcal{A}_{\mathsf{NP}} = (D_{\mathsf{NP}}, t_{\mathsf{NP}}, I_{\mathsf{NP}}, \downarrow)$ is defined as follows. The domain D_{NP} represents the operand stack and the local variables, and is defined by:

$$\begin{aligned} D_{\mathsf{NP}} &= (\mathit{list}\ E)^{\top}_{\bot} \times (\mathbb{Z} \to NP), \\ NP &= \{\mathit{null}, \mathit{nonnull}\}^{\top}_{\bot}, \\ E &::= \mathit{const}\ NP \mid \mathit{localvar}\ \mathbb{Z} \ . \end{aligned}$$

The transfer functions, $t_{\mathsf{NP}(pc,pc')}(d)$ is defined by case analysis in the instruction at pc and in d. Some of the rules are:

– if $instructionAt(pc) = \texttt{getfield}$

$$t_{\mathsf{NP}(pc,pc+1)}(e :: s, l) = (const\ \top :: s, [\![e = nonnull]\!](l)),$$
$$t_{\mathsf{NP}(pc,pc_{\mathrm{exc}})}(e :: s, l) = (const\ nonnull :: [], [\![e = null]\!](l));$$

– if $instructionAt(pc) = \texttt{ireturn}$, $t_{\mathsf{NP}(pc,pc_N)}(v :: s, l) = (v :: s, l)$;
– if $instructionAt(pc) = \texttt{invokevirtual}$,

$$t_{\mathsf{NP}(pc,pc+1)}(args +\!\!+ loc :: s, l) = (const\ \top :: s, [\![e = nonnull]\!](l)),$$
$$t_{\mathsf{NP}(pc,pc_E)}(args +\!\!+ loc :: s, l) = (const\ nonnull :: [], l);$$

– if $instructionAt(pc) = \texttt{iload}$, $t_{\mathsf{NP}(pc,pc+1)}(s, l) = (localvar\ x :: s, l)$.

Given $e : E$, the expression $[\![e]\!](l) : NP$ evaluates e using the map l. Given $e : E$, $n : NP$, and $l : \mathbb{Z} \to NP$, the expression $[\![e = n]\!](l) : \mathbb{Z} \to NP$ is a mapping that updates l using the fact that $e = n$. Note the way this expression is used for the `getfield` instruction: in the transfer for normal execution we can update the local variables, knowing that the reference is non-null, and for exceptional execution, we know the reference is null. The second rule for `invokevirtual` indicates that it may throw an uncaught exception.

Another example of a static analysis is provided by the weakest precondition defined for the VCgen.

Example 2. The weakest precondition can be viewed as a backward analysis (see [7]), $\mathcal{A}_{\mathsf{WP}} = (D_{\mathsf{WP}}, t_{\mathsf{WP}}, I_{\mathsf{WP}}, \uparrow)$, where $D_{\mathsf{WP}} = \mathsf{Assertion}$. We have

$$d_1 \sqsubseteq d_2 = (\models d_1 \underline{\Rightarrow} d_2),$$

and $\top$, $\bot$, $\sqcap$, $\sqcup$ correspond with $\underline{\mathsf{true}}$, $\underline{\mathsf{false}}$, $\underline{\wedge}$, $\underline{\vee}$, respectively.

The transfer function is defined by:

$$t_{\mathsf{WP}(pc,pc')}(e) = \lambda s_0.\lambda s.C_{(pc,pc')}(s) \underline{\Rightarrow} P_{(pc,pc')}(e, s_0, s),$$

and finally the initial value $I_{\mathsf{WP}}(pc_{\mathrm{N}}) = \lambda s_0.\lambda s.\mathsf{subst}_{\mathsf{Post}}(Post_{Nrml}, s_0, s)$ and $I_{\mathsf{WP}}(pc_{\mathrm{E}}) = \lambda s_0.\lambda s.\mathsf{subst}_{\mathsf{Post}}(Post_{Exc}, s_0, s)$.

The function $\mathrm{wp}_{\mathsf{annot}}$ computes a solution for this analysis. To check that is in fact a solution, we need to prove that for all pc,

$$\mathrm{wp}_{\mathsf{annot}}(pc) \sqsubseteq \bigwedge_{(pc,pc')\in\mathcal{G}} t_{\mathsf{WP}(pc,pc')}(\mathrm{wp}_{\mathsf{annot}}(pc')) =$$
$$\lambda s_0.\lambda s. \bigwedge_{(pc,pc')\in\mathcal{G}} C_{(pc,pc')}(s) \underline{\Rightarrow} P_{(pc,pc')}(\mathrm{wp}_{\mathsf{annot}}(pc'), s_0, s) = \mathrm{wp}_{\mathsf{instr}}(pc)\ .$$

Note that this is stated in Lemma 3.

A static analysis simulates the execution of a program in its domain. To prove that an analysis is sound, we need to prove that a step in the operational semantics, correspond to a transfer function in the domain. We define a *correctness relation* that relates states, with the elements of the domain of the analysis.

Definition 4. *A* correctness relation *for an analysis* $\mathcal{A} = (D, t, I, f)$ *is a relation* $\vdash \subseteq State \times D$*, such that the following holds:*

- *for all* $d_1, d_2 \in D$*, if* $s \vdash d_1$ *and* $d_1 \sqsubseteq d_2$*, then* $s \vdash d_2$*, and*
- *if* $(\forall d \in D' \subseteq D, s \vdash d)$*, then* $s \vdash (\bigsqcap D')$*.*

The relation $s \vdash d$ should be read as: d is a safe approximation of s.

Definition 5. *A static analysis* $\mathcal{A} = (D, t, I, f)$ *with correctness relation* $\vdash$*, is* sound *if for every solution* S*, the following holds:* $(pc, s) \longrightarrow (pc', s')$ *and* $s \vdash S(pc)$*, implies* $s' \vdash S(pc')$*.*

The usual way to prove that an analysis is sound is to prove that the transfer functions preserve the semantics. For a forward analysis, this means that if $(pc, s) \longrightarrow (pc', s')$ and $s \vdash d$, then $s' \vdash t_{(pc,pc')}(d)$. For a backward analysis, the transfer functions preserve the semantics if $(pc, s) \longrightarrow (pc', s')$ and $s \vdash t_{(pc,pc')}(d)$ implies $s' \vdash d$.

If we prove for a given analysis that the transfers functions preserve the semantics, then the soundness of the analysis follows from the properties of the correctness relation, and the definition of a solution.

Continuing with the examples, we define a correctness relation for the null-pointer analysis and the weakest precondition.

Example 3. For the analysis defined in Example 1, we define a correctness relation, $\vdash_{\mathsf{NP}}$, by translating the elements of D_{NP} to Assrt, and using the validity relation of Assrt. First, we define the function $tr : \mathsf{V} \times NP \to \mathsf{Assrt}$, where $tr(e, \bot) = \underline{\mathsf{false}}$, $tr(e, \top) = \underline{\mathsf{true}}$, $tr(e, null) = (e \underline{=} \mathsf{null})$, and $tr(e, nonnull) = (e \underline{\neq} \mathsf{null})$.

This function is extended to $\overline{tr} : D_{\mathsf{NP}} \to \mathsf{Assrt}$. For example, $\overline{tr}(localvar\ 0 :: [], [0 \mapsto nonnull, 1 \mapsto \top]) = tr(\mathsf{St}\,0, nonnull) \underline{\wedge} tr(\mathsf{Lv}\,0, nonnull) \underline{\wedge} tr(\mathsf{Lv}\,1, \top)$.

The correctness relation is defined as $(s \vdash_{\mathsf{NP}} d) = (s \models \overline{tr}(d))$ (note that we do not need an initial state). It can be shown that the transfer functions for this analysis preserve the semantics, and therefore, that the analysis is sound.

Example 4. A correctness relation for the analysis defined in Example 2 is:

$$(s_0, s \vdash_{\mathsf{WP}} d) = (s_0, s \models d) \ .$$

This definition depends on a fixed initial state s_0. Note that the soundness of this analysis is stated in Theorem 1.

3.2 Combining a Static Analysis with the VCgen

We show how the VCgen can use the results of the analysis to reduce the proof obligations. The main idea is to use the solution of the analysis as a parameter for the VCgen. When computing the function $\mathsf{wp}_{\mathsf{instr}}$ at a particular point pc, we can use the information given by the analysis at pc to remove some branch.

Assume we have an analysis $\mathcal{A} = (D, t, I, f)$ with correctness relation $\vdash$, and a solution $S : PC \to D$. Further, assume we have a function $\gamma : D \to \mathsf{Assrt}$ that translates the results of the analysis to assertions in the VCgen language that reference to the local state, with the following property: if $s \vdash d$, then $s \models \gamma(d)$.

We redefine the function $\mathrm{wp}_{\mathsf{instr}}$. The general form is now

$$\mathrm{wp}_{\mathsf{instr}}(pc, s_0, s) = \bigwedge\!\!\!\!\!\!\underline{}_{(pc,pc')\in\mathcal{G}} F_{(pc,pc')}(s_0, s), \tag{2}$$

where the F is defined as

$$F_{(pc,pc')}(s_0, s) = \begin{cases} \underline{\mathsf{true}} & \text{if } \models \mathsf{subst}(s, \gamma(S(pc))) \mathrel{\underline{\Rightarrow}} \underline{\neg} C_{(pc,pc')}(s), \\ WP(pc, pc', s_0, s) & \text{otherwise,} \end{cases}$$

and $WP(pc, pc', s_0, s) = \big(C_{(pc,pc')}(s) \mathrel{\underline{\Rightarrow}} P_{(pc,pc')}(\mathrm{wp}_{\mathsf{annot}}(pc'), s_0, s)\big)$.

Intuitively, if we can infer $\underline{\neg} C_{(pc,pc')}(s)$ from $S(pc)$, then the path going from pc to pc' cannot be taken at runtime, since taking this path would imply that the condition $C_{(pc,pc')}(s)$ is valid. In that case, the proof obligation corresponding to this branch can be removed, replacing it by $\underline{\mathsf{true}}$.

The condition $\models \mathsf{subst}(s, \gamma(S(pc))) \mathrel{\underline{\Rightarrow}} \underline{\neg} C_{(pc,pc')}(s)$ may not be decidable; in that case we have to replace it with a decidable test, $test(S(pc), C_{(pc,pc')}(s))$, that is a sound approximation, i.e. if $test(S(pc), C_{(pc,pc')}(s))$, it implies that $\models \mathsf{subst}(s, \gamma(S(pc))) \mathrel{\underline{\Rightarrow}} \underline{\neg} C_{(pc,pc')}(s)$.

The definition of F depends on the domain of the analysis, so we will illustrate with the null-pointer analysis defined above.

Example 5. To remove proof obligations using the null-pointer analysis, we look on the instructions that could generate a null-pointer exception. For instance, let us take `getfield`. If $instructionAt(pc) =$ `getfield`, and $S(pc) = (e :: s, l)$, then $F_{(pc,pc_{\mathrm{exc}})}$ is defined by:

$$F_{(pc,pc_{\mathrm{exc}})}(s_0, s) = \begin{cases} \underline{\mathsf{true}} & \text{if } [\![e]\!](l) = nonnull, \\ WP(pc, pc_{\mathrm{exc}}, s_0, s) & \text{otherwise;} \end{cases}$$

This says that if the analysis guarantees that the top of the stack will contain a non-null pointer, then we do not need to check the branch corresponding to the null-pointer exception handler. In the same way, we can remove the proof obligation corresponding to normal execution if the analysis guarantees that the pointer is null.

A similar definition applies to other instructions such as `putfield`, and `invokevirtual`, i.e. all instructions that take a pointer parameter from the stack, and throw a `NullPointer` if the pointer is null.

3.3 Combining Static Analyses and Specifications

The VCgen presented above generates fewer proof obligations by using static analysis to reduce the control flow graph. However, there are situations where

the analysis cannot ensure enough information to make some reduction possible. Consider the following excerpt of Java bytecode:

$$
\begin{array}{ll}
pc_1 & \ldots \qquad\qquad\qquad\qquad A(pc_1) = \mathsf{Lv}\, x \neq \underline{\mathsf{null}} \mathbin{\underline{\wedge}} \ldots \\
\ldots & \\
pc_2 & \texttt{iload}\ x \\
pc_3 & \texttt{getfield}\ f
\end{array}
$$

Assume that the local variable x does not change between pc_1 and pc_2, and that the annotation table contains the assertion that x is not null at pc_1. Therefore, at pc_3, the `getfield` instruction is accessing a non-null pointer. If the analysis is not able to ensure this, then the VCgen will generate two proof obligations. The one corresponding to exceptional execution is proved by contradiction using the assertion at pc_1. If there is more that one access to x such as the one at pc_3, the VCgen will generate two proof obligations for each access.

In this section, we propose a way to transfer the assertions contained in the specification to the domain of the analysis, so that the analysis can produce more accurate results. In the example above, if the information contained in $A(pc_1)$ is transferred, the analysis can propagate it to point pc_3, where it can ensure that the object accessed is non-null. Then, only one proof obligation would have been generated.

We will assume an annotated method m with specification $\mathcal{S}$ and an analysis $\mathcal{A} = (D, t, I, f)$. In order to translate the assertions contained in the specification to the domain of the analysis, we assume a function $\alpha : \mathsf{Assrt} \to D$, with the following property: $s_0, s \models e \Rightarrow s \vdash \alpha(e)$.

We extend the annotation table A into a total function $\overline{A} : PC \to \mathsf{Assrt}$, where we complete with the value $\underline{\mathsf{true}}$ the elements that are not in the domain.

We redefine the meaning of a solution for the analysis, to use the specification. To differentiate from the previous definition, we call this *combined solution*, and refer to the previous as *simple solution*.

Definition 6. *A* combined solution *(or* combined table*) for a forward analysis* $\mathcal{A} = (D, t, I, \downarrow)$ *is a function* $S : PC \to D$*, such that* $I(pc_0) \sqcap \alpha(Pre) \sqsubseteq S(pc_0)$ *and*

$$\forall pc \in PC, \bigsqcup_{(pc,pc') \in \mathcal{G}} t_{(pc,pc')}(S(pc) \sqcap \alpha(\overline{A}(pc))) \sqsubseteq S(pc') \ .$$

A combined solution *(or* combined table*) for a backward analysis* $\mathcal{A} = (D, t, I, \uparrow)$ *is a function* $A : PC \to D$*, such that* $S(pc_{\mathrm{N}}) \sqcap \alpha(Post_{Nrml}) \sqsubseteq I(pc_{\mathrm{N}})$*,* $S(pc_{\mathrm{E}}) \sqcap \alpha(Post_{Exc}) \sqsubseteq I(pc_{\mathrm{E}})$ *and*

$$\forall pc \in PC, \alpha(\overline{A}(pc)) \sqcap S(pc) \sqsubseteq \bigsqcap_{(pc,pc') \in \mathcal{G}} t_{(pc,pc')}(S(pc')) \ .$$

Note that, since transfer functions and the meet operator ($\sqcap$) are monotone, any simple solution for the analysis is also a combined solution. To find combined solutions, we can use the same methods used to find simple solutions.

Again, we will exemplify the approach using the null-pointer analysis.

Example 6. To define the function α for the analysis $\mathcal{A}_{\mathsf{NP}}$, we first define the function $split : \mathsf{Assrt} \to list\,\mathsf{Assrt}$ such that $split(e_1 \mathbin{\underline{\wedge}} e_2) = split(e_1) +\!\!+ split(e_2)$, and $split(e) = e$ if e is not of the form $e_1 \mathbin{\underline{\wedge}} e_2$.

Then α is defined as: $\alpha(e) = filter(split(e))$, where *filter* looks in the list produced by *split* for expressions of the form $\mathsf{St}\,k \mathbin{\underline{=}} \mathsf{null}$, $\mathsf{St}\,k \mathbin{\underline{\neq}} \mathsf{null}$, $\mathsf{Lv}\,k \mathbin{\underline{=}} \mathsf{null}$, $\mathsf{Lv}\,k \mathbin{\underline{\neq}} \mathsf{null}$, or their symmetric, and translate them to the domain D_{NP}. For instance, $\alpha(\mathsf{Lv}\,0 \mathbin{\underline{\neq}} \mathsf{null} \mathbin{\underline{\wedge}} \mathsf{null} \mathbin{\underline{=}} \mathsf{St}\,1) = (const\ \top :: const\ null :: [], [0 \mapsto nonnull])$.

Soundness of the VCgen Revisited. Now we focus on the proof of soundness for the VCgen described in this section. We assume an analysis $\mathcal{A}$ with correctness relation $\vdash$ and a combined solution S. Stated in the terms defined in this section, to prove the soundness of the VCgen we need to prove:

$$(pc, s) \longrightarrow (pc', s') \wedge (s_0, s \vdash_{\mathsf{WP}} \mathrm{wp}_{\mathsf{annot}}(pc)) \Rightarrow (s_0, s' \vdash_{\mathsf{WP}} \mathrm{wp}_{\mathsf{annot}}(pc')) \ . \quad (3)$$

However, since the WP of an instruction depends on the combined solution for the analysis, and the solution depends on the validity of the specification, to prove (3) we have to prove the following:

$$(pc, s) \longrightarrow (pc', s') \wedge (s_0, s \vdash_{\mathsf{WP}} \mathrm{wp}_{\mathsf{annot}}(pc)) \wedge (s \vdash S(pc)) \Rightarrow (s_0, s' \vdash_{\mathsf{WP}} \mathrm{wp}_{\mathsf{annot}}(pc')) \wedge (s' \vdash S(pc')) \ . \quad (4)$$

The proof of (4) is divided in two parts. We need to prove that for all $(pc, s) \longrightarrow (pc', s')$ we have:

$$(s \vdash S(pc)) \wedge (s_0, s \vdash_{\mathsf{WP}} \overline{A}(pc)) \Rightarrow (s' \vdash S(pc')), \quad (5)$$

$$(s \vdash S(pc)) \wedge (s_0, s \vdash_{\mathsf{WP}} \mathrm{wp}_{\mathsf{annot}}(pc)) \Rightarrow (s_0, s' \vdash_{\mathsf{WP}} \mathrm{wp}_{\mathsf{annot}}(pc')) \ . \quad (6)$$

Equation (5) states that the analysis is sound (for combined solutions) assuming that the specification is verified. The proof is similar to the soundness proof for simple solutions. We first prove that the transfer functions preserve the semantics (this does not depend on any type of solution), and then conclude using properties of the correctness, and monotony of the transfer function and meet ($\sqcap$) and join ($\sqcup$) operators.

Equation (6) states that the VCgen is sound assuming that the analysis is sound. The proof follows the lines of Theorem 1, however, in this case we cannot prove that $(pc, s) \longrightarrow (pc', s')$ and $s_0, s \models \mathrm{wp}_{\mathsf{instr}}(pc)$ implies

$$P_{(pc,pc')}(\mathrm{wp}_{\mathsf{annot}}(pc'), s_0, s)),$$

since the proof obligation corresponding to the branch (pc, pc') may have been removed (changed to $\underline{\mathsf{true}}$) because of $S(pc)$. However, in that case, we can prove that $s \models \underline{\neg} C_{(pc,pc')}(s)$. On the other hand, from Lemma 1 we know that $s \models C_{(pc,pc')}(s)$, therefore we have a contradiction and the result follows.

4 Certificate Generation and Checking

In the typical PCC architecture, the producer runs the VCgen on the annotated code. This generates proof obligations, whose proof provides the certificate that is packaged along with the code and sent to the consumer.

For the VCgen described in the previous section, this framework largely applies. The difference lies in the generation of proof obligations. The analyses are performed on the code, using the specification of the methods. For this stage, any fixpoint algorithm can be used to generate the results of the analysis. The algorithm itself does not need to be verified, since we can check that the results given are correct.

The results of these analyses are then given to the VCgen, that returns the proof obligations. These can be proven by automatic methods or in a proof assistant (Coq in our case). The certificate given to the consumer consists on the proofs obtained and the results of the analysis.

Checking the certificate, on the consumer side, consists of three stages. First, the results of the analyses are checked. This involves a simple procedure that can be done very efficiently in one pass through the code [2]. Second, once the results are checked, they are given to the VCgen that generates the proof obligations. Third, the proofs given as part of the certificate are checked to correspond with the obligations generated by the VCgen. If all the checking goes well, the code can be safely executed.

5 Related Work

The use of abstract interpretation as a tool to verify safety policies in PCC has been proposed by Albert, Puebla and Hermenegildo in their *Abstraction-Carrying Code* (ACC) framework [2], where abstract interpretation is used to represent safety policies. The abstraction of a program is the certificate sent to the consumer alongside the code. We do not use analysis to express safety policies, but to reduce the control flow graph of a program. In [1], Albert et al. develop a technique to compress certificates for ACC. The main idea is to remove redundant information that can be easily reconstructed in one pass through the code. Their work can be readily applied to our case for compressing the results of the analyses.

Another compression technique is presented by Besson, Jensen and Pichardie in [3]. They develop an extensible PCC framework based on abstract interpretation. The compression is done through a set of commands that allows the reconstruction of the solution from partial information. Using these commands, different strategies for reconstruction can be encoded and adapted to each particular program. This can also be directly applied to our case.

Nipkow et al. developed the VeryPCC framework in Isabelle/HOL. They define a generic VCgen that can be instantiated with different programming languages, safety logics and safety policies. In [12], Wildmoser, Chaieb and Nipkow use trusted and untrusted analyses to verify a safety policy incrementally. A

VCgen is used to verify the results of the untrusted analyses, using the results of the trusted analyses as hypothesis.

Proof-producing program analysis (PPPA) [5,11] is a technique to generate Hoare-logic proof derivations from program analyses solutions. The advantage of this approach is that the consumer does not need to have a special procedure to check the results of the analysis. On the other hand, the size of the proofs (even if small compared with the program) can be bigger than using compression techniques mentioned above. Nevertheless, it should be possible to use PPPA techniques in our approach to combine the results of the analysis and the proof terms, into a proof term that ensures both properties.

6 Conclusions and Future Work

We have presented a technique based on static analysis to reduce the number of proof obligations generated by a VCgen for Java bytecode, by reducing the control flow graph of a program. The reduction and simplification of the proof obligations have the advantage that leaves the developer with fewer goals left to prove, which as a consequence, generate smaller proof term that can be more rapidly checked. We have exemplified the approach with a simple null-pointer analysis. We have chosen this type of analysis, because many instructions in the JVM can throw null-pointer exceptions, which allows for large reductions in the proof obligations. A recent study by Chalin and James [6] shows that in 2/3 of the cases, reference variables are meant to be non-null (based on design intent).

We have formalized in Coq the VCgen described in Sect. 2 including the proof of soundness (Theorem 1).[1] We plan to complete the formalization (null-pointer analysis and combination), and apply other type of analyses to our approach. Obvious candidates are interval analysis used for array-bound checking and escaping-exception analysis.

The VCgen does not use the complete solution of the analysis to reduce proof obligations. Removing unused parts could help to further compress the certificates. A good starting point should be [4].

Acknowledgments. We would like to thank David Pichardie for his insightful suggestions and for the help he provided with the formalization in Coq.

References

1. Albert, E., Arenas-Sánchez, P., Puebla, G., Hermenegildo, M.V.: Reduced certificates for abstraction-carrying code. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 163–178. Springer, Heidelberg (2006)
2. Albert, E., Puebla, G., Hermenegildo, M.V.: Abstraction-carrying code. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 380–397. Springer, Heidelberg (2005)

[1] Available online at `http://www-sop.inria.fr/everest/personnel/Benjamin.Gregoire/Code/Certified_vcgen.tgz`

3. Besson, F., Jensen, T.P., Pichardie, D.: Proof-carrying code from certified abstract interpretation and fixpoint compression. Theor. Comput. Sci. 364(3), 273–291 (2006)
4. Besson, F., Jensen, T.P., Turpin, T.: Small witnesses for abstract interpretation-based proofs. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 268–283. Springer, Heidelberg (2007)
5. Chaieb, A.: Proof-producing program analysis. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) ICTAC 2006. LNCS, vol. 4281, pp. 287–301. Springer, Heidelberg (2006)
6. Chalin, P., James, P.: Non-null references by default in java: Alleviating the nullity annotation burden. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, Springer, Heidelberg (2007)
7. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282 (1979)
8. Necula, G.C.: Proof-carrying code. In: POPL, pp. 106–119 (1997)
9. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
10. Pichardie, D.: Bicolano – Byte Code Language in Coq (2006), `http://mobius.inia.fr/bicolano`
11. Seo, S., Yang, H., Yi, K.: Automatic construction of hoare proofs from abstract interpretation results. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, pp. 230–245. Springer, Heidelberg (2003)
12. Wildmoser, M., Chaieb, A., Nipkow, T.: Bytecode analysis for proof carrying code. Electr. Notes Theor. Comput. Sci. 141(1), 19–34 (2005)
13. Wildmoser, M., Nipkow, T.: Certifying machine code safety: Shallow versus deep embedding. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 305–320. Springer, Heidelberg (2004)