Asserting Bytecode Safety

Martin Wildmoser and Tobias Nipkow

Technische Universität München, Institut für Informatik
{wildmosm, nipkow}@in.tum.de

Abstract. We instantiate an Isabelle/HOL framework for proof carrying code to Jinja bytecode, a downsized variant of Java bytecode featuring objects, inheritance, method calls and exceptions. Bytecode annotated in a first order expression language can be certified not to produce arithmetic overflows. For this purpose we use a generic verification condition generator, which we have proven correct and relatively complete.

1 Proof Carrying Code

In mobile code applications, e.g. applets, grid computing, dynamic drivers, or ubiquitous computing, safety is a primary concern. Proof carrying code (PCC) aims at certifying that low level code adheres to some safety policy, such as type safety [6], bounded array accesses [13], or limited memory consumption [4]. When such properties are checked statically sandbox mechanisms and error recovery become obsolete. In classical PCC a verification condition generator (VCG) reduces annotated machine code to proof obligations that guarantee safety. Proofs, usually obtained automatically with a theorem prover, are then shipped to the code consumer, who checks them. The whole setup is sound if the VCG and the proof checker can be trusted. In Foundational Proof Carrying Code [3] the VCG is eliminated by proving safety directly on the machine semantics, typically assisted by a source level type system. Our approach is to formalize and verify PCC in a theorem prover. In [19] we present an Isabelle/HOL [15] framework for PCC. The essential part is a generic, executable and verified VCG. This turns out to be feasible as only a small part of a VCG needs to be trusted. Many parts can be outsourced in form of parameters that can be customized to the programming language, safety policy and safety logic at hand. In this paper we instantiate a PCC system for Jinja bytecode and a safety policy that prohibits arithmetic overflow. We verified that this instantiation meets all the requirements our framework demands for the VCG to be correct and relatively complete. Verifying programs at the bytecode level has clear advantages. First, one does not have to trust a compiler. Second, the source code, which is often kept secret, is not required. Third, many safety policies are influenced by the machine design. For example verifying sharp runtime bounds even requires going down to the processor level and considering pipeline and caching activities. In case of bytecode, we need a safety logic that can adequately model JVM states. Over the years various logics for object oriented programs have been proposed.

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 326–341, 2005.

For instance [2] defines a Hoare Logic based on a combination of a type system and first order logic with equality. In [16] a shallow embedding of Isabelle/HOL is used to define a Hoare logic for Java. A very prominent annotation language for Java is JML [10] or the downsized version of it used in ESC Java [7]. However, all of the logics have been designed for Java, not its bytecode. This paper introduces a first order expression language with JVM specific constructs. This language is expressive enough for weakest preconditions of Jinja instructions and a safety policy against arithmetic overflow. Although this is undecidable, many programs produce proof obligations that are easy enough to be handled by Isabelle/HOL's decision procedures, such as Cooper's algorithm [14].

2 Jinja Bytecode

Jinja bytecode is a downsized version of Java bytecode. Although it only has 16 instructions, it covers most object oriented features: object creation, inheritance, dynamic method calls and exceptions.

datatype *instr* =

`Load` *nat*	load from register
| `Store` *nat*	store into register
| `Push` *val*	push a constant
| `New` *cname*	create object on heap
| `Getfield` *vname cname*	fetch field from object
| `Putfield` *vname cname*	set field in object
| `Checkcast` *cname*	check if object is of class *cname*
| `Invoke` *mname nat*	invoke instance method with *nat* parameters
| `Return`	return from method
| `Pop`	remove top element
| `IAdd`	integer addition
| `Goto` *int*	goto relative address
| `CmpEq`	equality comparison
| `IfFalse` *int*	branch if top of stack false
| `IfIntLeq` *int*	take integers a and b from stack, branch if $a \leq b$
| `Throw`	throw exception

Jinja programs are interpreted by the Jinja virtual machine, which closely models the Java VM. States consist of a flag indicating whether an exception is raised (if yes, a reference to the exception object), a heap and a method frame stack.

types *jvm-state* = *addr option* × *heap* × *frame list*

The heap is a partial map from addresses (natural numbers) to objects. We use the polymorphic type *'a option* = *None* | *Some 'a* to model partiality in Isabelle/HOL, a logic of total functions. Using *the* one can extract the content,e.g. *the* (*Some a*) = *a*.

types *heap* = *addr* ⇒ *obj option*
obj = *cname* × *fields*

fields = (*vname* × *cname*) ⇒ *val option*
cname = *vname* = *mname* = *string*

Jinja has values for booleans, e.g. *Bool True*, integers, e.g. *Intg 5*, references, e.g. *Addr 3*, null pointers, e.g. *Null* or dummy elements, e.g. *Unit*. For some values, we use partially defined extractor functions.

datatype *val* = *Bool bool* | *Intg int* | *Addr addr* | *Null* | *Unit*
the-Intg (*Intg i*) = *i*, *the-Bool* (*Bool b*) = *b*, *the-Addr* (*Addr a*) = *a*

Each value has a type associated with it:

datatype *ty* = *Boolean* | *Integer* | *Class cname* | *NT* | *Void*

Whenever a method is called a new frame is allocated on the method frame stack. This frame contains registers, an operand stack and the program counter. In the registers the Jinja VM stores the **this** reference, the method's arguments and its local variables. The operand stack is used to evaluate expressions. Both are modelled as lists, e.g. $l = [Null, Unit]$, for which Isabelle/HOL provides many operators. For example, there is concatenation, e.g. $l@[Intg\ 2] = [Null, Unit, Intg\ 2]$, indexed lookup $l\ !\ 1 = Unit$, head,i.e. $hd\ l = Null$ and tail, $tl\ l = [Unit]$.

types *frame* = *opstack* × *registers* × *pos*
opstack = *val list*
registers = *val list*
pos = *cname* × *mname* × *nat*

Instructions are identified with positions. For example (C,M,pc) points to instruction number pc in method M of class C. Each method is a tuple of the form (mxs,mxr,is,et), where *mxs* indicates the maximum operand stack height, *mxr* the number of used registers, *is* the method body and *et* the exception table.

types *jvm-method* = *nat* × *nat* × *instr list* × *ex-table*

The exception table is a list of tuples $(f,\ t,\ C,\ h,\ d)$:

types *ex-table* = (*nat* × *nat* × *cname* × *nat* × *nat*) *list*

Whenever an intruction within the *try* block bounded by $[f,\ t)$ throws an exception of type C the handler starting at h is executed. The parameter d, which is always *0* in our case, specifies the size of the stack the handler expects. This is used in [9] to handle exceptions within expression evaluation, but is not required for real Java programs. Jinja programs are lists of class declarations. Each class declaration (C,S,fs,ms) consists of the name of the class, the name of its direct superclass, a list of field declarations, which are pairs of field names and types, and a list of method declarations. Method declarations consist of the method's name, its argument types, its result type and its body.

types *jvm-prog* = (*cname* × *cname* × *fdecl list* × *mdecl list*) *list*
fdecl = *vname* × *ty*
mdecl = *mname* × *ty list* × *ty* × *jvm-method*

Our PCC system requires programs Π with annotations. These are added by finite maps from positions to logical expressions. Finite maps are lists of pairs,

e.g. $fm = [(1,1),(3,5),(3,6)]$, and have operations for lookup, e.g. $fm{\downarrow}0 = None$ or $fm{\downarrow}3 = Some\ 5$, domain, e.g. $dom\ fm = [1,3,3]$ and range, e.g. $ran\ fm = [1,5,5]$. Note that a pair (x,y) is overwritten by a pair (x,y') to the left of it.

types *jbc-prog* = *jvm-prog* × (*pos* ▷ *expr*)

To specify and verify safety properties we need to reason about Jinja VM states. A central issue of this paper is to suggest a formula language for this purpose. For some constructs of this language we require an extended version of the Jinja VM, one which memorizes additional information in a so called environment e, in order to define their semantics. In addition our PCC framework requires positions to be given as the first component of a state.

types *jbc-state* = *pos* × *jvm-state* × *env*

The environment e contains a virtual stack of call states *cs* e and a binding *lv* e for so called logical variables.

record *env* =*cs*:: *heap list*
lv:: *var* ⇒ *val*

Whenever a new frame is allocated on the method call stack, we record the current heap and register values in a call stack, which acts like a history variable in Hoare Logics. Whenever a frame is popped, we also pop an entry from the call stack. The bytecode semantics consists of two rules: One specifies normal, the other one exceptional execution.

nrml:
⟦ *P* = *fst Π*; *p* = (*C*,*M*,*pc*); *i* = *instrs-of P C M* ! *pc*;
σ = (*None*,*h*,(*stk*,*loc*,*p*)#*frs*); *check P* σ;
exec-instr i P h stk loc C M pc frs = (*None*,*h'*,*fr'*#*frs'*);
σ′ = (*None*,*h'*,*fr'* # *frs'*); *p'* = *snd* (*snd fr'*);
e' = *e*(|*cs*:=*if* ∃ *M n*. *i* = *Invoke M n then h*#(*cs e*)
else if i = *Return then tl* (*cs e*) *else cs e*|)
⟧ ⟹ ((*p*,σ,*e*),(*p'*,σ′,*e'*)) ∈ *effS Π*

expt:
⟦ *P* = *fst Π*; *p* = (*C*,*M*,*pc*); *i* = *instrs-of P C M* ! *pc*;
σ = (*None*,*h*,(*stk*,*loc*,*p*)#*frs*); *check P* σ;
exec-instr i P h stk loc C M pc frs = (*Some xa*,-,-);
find-handler P xa h ((*stk*,*loc*,*p*)#*frs*) = σ′;
σ′ = (*None*,*h*,([*Addr xa*],*loc'*,*p'*)#*frs'*);
e' = *e*(|*cs*:=*drop* (*length frs* − *length frs'*) (*cs e*)|)
⟧ ⟹ ((*p*,σ,*e*),(*p'*,σ′,*e'*)) ∈ *effS Π*

In both rules we use *instrs-of* to retrieve the instruction list of the current method. The actual execution for single instructions is delegated to *exec-instr*, whose full definition can be found in [9]. Here we only give one example:

exec-instr IAdd P h stk loc C_0 M_0 *pc frs* =
(*let* i_2 = *the-Intg* (*hd stk*); i_1 = *the-Intg* (*hd* (*tl stk*))
in (*None*, *h*, (*Intg* (i_1+i_2)#(*tl* (*tl stk*)), *loc*, C_0, M_0, *pc*+1)#*frs*))

The function *exec-instr* returns triples, whose first component indicates whether an exception occurs. If yes (second rule), we use the function *find-handler*) to do exception handling similar to the Java VM: it looks up the exception table in the current method, and sets the program counter to the first handler that protects *pc* and that matches the exception class. If there is no such handler, the topmost call frame is popped, and the search continues recursively in the invoking frame. If no exception handler is found, the exception flag remains set and the machine halts. If this procedure does find an exception handler $(f,\ t,\ C,\ h,\ 0::'e)$ it sets the *pc* to *h* and empties the operand stack except for reference to the exception object. Additional safety checks in the semantics, i.e. *check P* σ, ensure that arguments of proper type and number are used. This simplifies the verification of the soundness and completness requirements our PCC framework demands. As proven in [9] the bytecode verifier only accepts programs for which these checks hold. Hence, it is sound to work with this defensive version of the Jinja VM. We require Jinja Bytecode programs to have a main method. For simplicity we assume that this method is named *main*, has no arguments, and belongs to a class called *Start*. This means we start a program at position (*Start*,*main*,*0*). The initial operand stack is empty and the registers are initialized with arbitrary values. The initial heap (*start-heap* (*fst* Π)) contains three entries for system exceptions *NullPointer*, *ClassCast* and *OutOfMemory*. The initial environment contains an empty call stack. The binding of logical variables is unrestricted.

initS $\Pi \equiv \{\ (p,\sigma,e).\ p = (Start,main,0) \wedge cs\ e = []$
$\wedge\ \sigma = (None,start\text{-}heap\ (fst\ \Pi),[([],Null\ \#\ replicate\ mxr\ arbitrary,p)])\ \}$

3 Assertion Logic

Many aspects of Java are identical in the bytecode, but an important concept of the Java VM cannot be found at the source level: The method frame stack. This stack is used to store local data of methods in evaluation. Almost all bytecode instructions affect it. If we want to simulate these effects at a syntactic level, we need a language that can describe this stack. Fig. 1 shows the assertion language we use for this purpose. It is a variant of first order arithmetic with special constructs for adequate modeling of JVM states. All these constructs are used to express our safety policy (no arithmetic overflow) and weakest preconditions for Jinja VM instructions. For simplicity this language is untyped. We do not even distinguish between formulas and expressions. Earlier instantiations [18] showed that this leads to duplication of functions and lemmas for both types. The uniform representation avoids this and still allows categorization with type checking functions. Next, we explain the semantics of all language constructs. Each expression can be evaluated for a given *jbc-state* and yields some Jinja value.

eval:: *jbc-state* $\Rightarrow$ *expr* $\Rightarrow$ *val*

The expressions come in two categories. From *Rg nat* to *Catch expr* we have JVM specific constructs. These are needed to access various parts of Jinja states

datatype *expr* =

Rg nat	register
| *St nat*	operand stack cell
| *Cn val*	constant value
| *NewA nat*	address for nth new object
| *Gf vname cname expr*	get field value
| *Ty expr ty*	type check
| *FrNr*	height of method frame stack
| *Pos pos*	position check
| *Call expr*	evaluation in call state
| *Catch cname expr*	evaluation in catch state
| *expr* ⌞+⌟ *expr* | *expr* ⌞−⌟ *expr* | *expr* ⌞*⌟ *expr*	arithmetic
| ⌞*if* *expr* ⌞*then*⌟ *expr* ⌞*else*⌟ *expr*	conditional
| *expr* ⌞=⌟ *expr* | *expr* ⌞≤⌟ *expr* | *expr* ⌞<⌟ *expr*	relational
| ⌞¬⌟ *expr*	negation
| *expr* ⌞⇒⌟ *expr*	implication
| ⌞⋀⌟ *expr list*	conjunction
| *Lv nat*	logical variable
| ⌞∀⌟ *nat expr*	quantification (logical vars)

Fig. 1. Jinja bytecode assertion language

in annotations. The remaining constructs are purely logical and are required to construct verification conditions.

3.1 JVM Specific Constructs

Since we want to use the annotation language to abstract Jinja states, we need various constructs to access different parts of such states. Most instructions only manipulate the topmost method frame. Instead of making the whole frame stack accessible in the language, which would complicate the evaluation range, we decided to use constructs for individual parts only. With *Rg k* and *St k* we access the *k*th register or element on the operand stack.

evalE Π (*p*,σ,*e*) (*Rg k*) = (*let* (*x*,*h*,*fs*)=σ; (*st*,*rg*,*p*)=*hd fs in rg*!*k*)
evalE Π (*p*,σ,*e*) (*St k*) = (*let* (*x*,*h*,*fs*)=σ; (*st*,*rg*,*p*)=*hd fs in st*!*k*)

Constants *Cn v* evaluate to their values *v*, i.e. *evalE Π s* (*Cn v*) = *v*. For better readability we abbreviate constants like *Cn* (*Intg 5*) or *Cn* (*Bool True*) with ⌞5⌟ or ⌞*True*⌟.

The *NewA n* expression returns the reference that is allocated in the heap for the *n*th object. In many cases this operator can be avoided as [5] shows. In our *wpF* operation, we will also replace field accesses of newly created objects by their default values. Eliminating all instances of *NewA* could be achieved at the level of formulas. However, this involves complex transformations of formulas and should be delegated to a post-processing function. In the definition we use the auxiliary function *new-Addr h*, which yields either *Some a* if *a* is the reference that is allocated next, or *None* if the heap is full.

evalE Π (p,σ,e) (NewA n) = (let (x,h,frs)=σ in evalNewA h n)
evalNewA h 0 = (case new-Addr h of None ⇒ Null | Some a ⇒ Addr a)
evalNewA h (Suc n) = evalNewA (h(the (new-Addr h):= Some arbitrary)) n

To evaluate *Gf F C ex*, which corresponds to $(C)ex.F$ in Java, we first check whether *ex* evaluates to an address value. If so, we fetch the value of the corresponding object field, otherwise we return *Unit*.

evalE Π s (Gf F C ex) = (case (evalE Π s ex)
of Addr a⇒(let (p,σ,e)=s;(x,h,frs)=σ;(D,fs)=the (h a)
in the (fs (F,C))))
| - ⇒ Unit)

The evaluation of *FrNr*, *Pos p* and *Ty ex tp* is straightforward, hence we skip the formal definitions. The *FrNr* expression yields the length of the method frame stack and *Pos p* evaluates to $\lfloor True \rfloor$ only if the current program position is *p*. To check the exact type of some expression we use *Ty ex tp*. Note that this check does not take the class hierarchy into account. Subtyping can be expressed by a disjunction of *Ty ex stp* expressions. With *Call* and *Catch* we evaluate formulas in previous states. The *Call ex* expression evaluates *ex* in the call state of the current method. This helps to specify postconditions of methods modularly. For example, annotating a `Return` instruction with $Rg\ 1 \sqsupseteq Call\ (Rg\ 1) \sqsupset \lfloor 1 \rfloor$ means that the returning method has incremented register 1. This techniqe is related to primed variables in VDM [8], except that we can set entire expressions into a different temporal context, just like temporal logic operators do. This is important, as some method postconditions need old values of object fields or other parts of the heap. Another reason is that we use this operator to restore the call context when we compute the verification condition of a method return. For example, if register 1 had value *Intg 5* before the method call, the programmer might annotate the call position with $Rg\ 1 \sqsupseteq \lfloor 5 \rfloor$ and the return position with $Rg\ 1 \sqsupseteq \lfloor 6 \rfloor$. Our VCG would then produce the following proof obligation, where we have to show that the postcondtion together with the call annotation (evaluated in the call state!) imply the annotation at the return position: $(Rg\ 1 \sqsupseteq Call\ (Rg\ 1) \sqsupset \lfloor 1 \rfloor \bigwedge Call\ (Rg\ 1 \sqsupseteq \lfloor 5 \rfloor)) \Rightarrow Rg\ 1 \sqsupseteq \lfloor 6 \rfloor$. Details about this construction can be found in [19]. We use the auxiliary function *call* to restore the call state of the current method. The program counter, registers and operand stack at call time are taken from the method frame beneath. The heap can be restored from the recordings in the environment. In case of a `main` method state (no caller), *Call ex* evaluates to *arbitrary*.

evalE Π s (Call ex) = (let (p,σ,e)=s; (x,h,frs)=σ in
(if length frs ≤ 1 then arbitrary else evalE Π (call s) ex))
call (p,(x,h,f#(s,r,p′)#frs),e) = (p′,(None,hd (cs e),(s,r,p′)#frs),e(|cs:=tl(cs e)|))

Exceptions impose a similar problem than method returns. However, since the number of frames chopped off the frame stack by exception handling is hard to determine statically, we need a special operator for this purpose. Just like *Call ex* the construct *Catch X ex* evaluates *ex* in a previous state. In this case we

restore the state under which we have last been in the `try` block that has a catching handler for exception X. The auxiliary function *catch* chops off frames until a catching handler is found. Simultaneously it restores the corresponding heap from the environment. Note that the resulting state is not the state under which the handler is entered, because the heap remains the same in this case.

$$\begin{array}{l} evalE\ \Pi\ s\ (Catch\ X\ ex) = (let\ (p,\sigma,e)=s;\ (x,h,frs)=\sigma\ in \\ if\ length\ frs \leq 1\ then\ arbitrary\ else\ evalE\ \Pi\ (catch\ (fst\ \Pi,X,s,ex))) \end{array}$$

$$\begin{array}{l} catch\ (P,X,(p,(x,h,fr\#(st,rg,p)\#frs),e)) = \\ (let\ (C,M,pc)=p\ in\ (case\ (match\text{-}ex\text{-}table\ P\ X\ pc\ (ex\text{-}table\text{-}of\ P\ C\ M)) \\ of\ None \Rightarrow catch\ (P,X,(p,(None,hd\ (cs\ e),(st,rg,p)\#frs),e(\!|cs:=tl(cs\ e)|\!))) \\ |\ Some\ pc' \Rightarrow (p,(None,hd\ (cs\ e),(st,rg,p)\#frs),e(\!|cs:=tl\ (cs\ e)|\!)))) \end{array}$$

3.2 Logical Constructs

The arithmetic, relational, conditional and logical expressions are evaluated recursively. First, we evaluate the argument expressions, then we apply the corresponding arithmetic, relational or conditional operator on the results. If any argument value has not the expected type the result becomes *arbitrary*. A logical expression *ex* is true if it evaluates to *Bool True*, otherwise it is false. From Winskel [20] we take the idea of distinguishing program and logical variables. The first (registers, stack ...) depend on the *jvm-state* and may be modified by instructions. The latter are evaluated in a separate binding *lv e*, which we made part of the environment e in *jbc-state*, and are unaffected by instructions. In the substitutions we use later on to express the effect of instructions, we will neither transform nor introduce logical variables. Hence, no renaming of bound variables is required.

$$evalE\ \Pi\ (p,\sigma,e)\ (Lv\ k) = (lv\ e)\ k$$

Quantification only binds logical variables. The formula $\llbracket\forall\rrbracket\ v.\ ex$ holds, if ex holds no matter what value v' the logical variable $Lv\ v$ is bound to.

$$\begin{array}{l} evalE\ \Pi\ (p,\sigma,e)\ (\llbracket\forall\rrbracket\ v\ ex) = \\ Bool\ (\forall\, v'.\ the\text{-}Bool\ (evalE\ \Pi\ (p,\sigma,e(\!|lv:=((lv\ e)(v:=v'))|\!))\ ex)) \end{array}$$

3.3 Validity and Provability

To use this expresssion language as a logic, we need judgements for validity and provability. Models of formulas are program states under which a formula evaluates to *Bool True*.

$$\Pi,s \models ex = the\text{-}Bool\ (evalE\ \Pi\ s\ ex)$$

Provabiliy $\vdash$ is usually defined by giving a set of axioms and inference rules. However, we can also define provability semantically and use the inference system of HOL for proofs. We regard a formula as provable if we can prove in HOL that it holds for all states in the safety closure *safeP* Π of a program Π. This set of states is defined relative to some safety policy *safeF*, which we are going to instantiate in the next section.

$\Pi \vdash ex = \forall\ s{\in}\ safeP\ \Pi.\ \Pi{,}s \models ex$

The safety closure is defined inductively: All initial states are in $safeP\ \Pi$. If a state $(p{,}\sigma{,}e)$ is in $safeP\ \Pi$, satisfies the safety formula and annotation at p (if there is any), then every successor state $(p'{,}\sigma'{,}e')$, i.e. $((p{,}\sigma{,}e){,}(p'{,}\sigma'{,}e')) \in (effS\ \Pi)$, that satisfies the safety formula and annotation at p' is also in $safeP\ \Pi$.

4 Safety Policy

The assertion language serves three purposes: First, it provides a means to specify machine states and thus to annotate programs. Second, we use it to express verification conditions. Third, we use it to specify the safety policy. Our PCC framework expects the safety policy to be given as a function $safeF{::}\ jbc\text{-}prog \Rightarrow pos \Rightarrow expr$, which defines a safety formula for each position p in a given program Π. This formula expresses conditions that must hold whenever we reach p at runtime. In our case we instantiate a safety policy that prohibits arithmetic overflow. The result of *IAdd* must not exceed *MAXINT*, which stands for Java's highest 32 bit integer $\lfloor 2147483647 \rfloor$.

$$safeF\ \Pi\ p = (if\ cmd\ \Pi\ p = Some\ IAdd\ \ then\ St\ 0 \lfloor{+}\rfloor St\ 1 \lfloor{\leqslant}\rfloor MAXINT\ \ else\ \lfloor True \rfloor)$$

A safety policy can also be lifted to programs. A **program is safe** if and only if every reachable state satisfies its safety formula.

$$isSafe\ \Pi = (\forall\, p_0\ \sigma_0\ e_0\ p\ m\ e.\ (p_0{,}\sigma_0{,}e_0) \in (initS\ \Pi) \wedge ((p_0{,}\sigma_0{,}e_0){,}(p{,}\sigma{,}e)) \in (effS\ \Pi)^* \longrightarrow \Pi{,}(p{,}\sigma{,}e) \models safeF\ \Pi\ p)$$

5 Verification Conditions

Our generic VCG analyses an annotated control flow graph and produces a formula in the assertion language. Details are in [19], here we only sketch the idea. Assume position p in program Π is annotated with A and has successor p', annotated with A'. A branch condition B specifies when p' is accessible from p. The verification condition for Π would contain the following proof obligation, ensuring a safe transition from p to p'.

$$(safeF\ \Pi\ p \lfloor\bigwedge\rfloor A \lfloor\bigwedge\rfloor B) \lfloor\Rightarrow\rfloor wpF\ \Pi\ p\ p'\ (safeF\ \Pi\ p' \lfloor\bigwedge\rfloor A')$$

We have to show that the safety formula at p, the annotation A and the branch condition B imply the weakest precondition for the safety formula and annotation at p'. The entire verification condition consists of various parts of this form. Not all positions must be annotated. It suffices if there is at least one annotation in each loop. For non-annotated positions the VCG constructs proof obligations by pulling back annotations of further successors using the weakest precondition function *wpF* and the successor function *succsF*. Relying on requirements for the parameter functions, we show in the PCC framework that the VCG is correct

and complete. The correctness theorem says that if we can prove the verification condition of a wellformed program, then this program is safe at runtime.

theorem vcg-correct: $wf\ \Pi \wedge \Pi \vdash vcg\ \Pi \longrightarrow isSafe\ \Pi$

Wellformedness means that there are enough annotations in the program. Completeness means that each wellformed and safe program with valid annotations yields a provable verification condition.

theorem vcg-complete: $wf\ \Pi \wedge correctAn\ \Pi \wedge isSafe\ \Pi \longrightarrow \Pi \vdash vcg\ \Pi$

Annotations are valid if they hold whenever the corresponding position is reached at runtime. For our instantiation to Jinja bytecode, we have proven both theorems by showing all the requirements on the parameter functions. The hardest part is to show that control flow function and the weakest precondition operator work correctly and precisely enough.

5.1 Jinja Bytecode Control Flow

To determine the control flow our VCG requires the function *succsF*, which given a program position yields a list of all direct successors paired with branch conditions. These specify the situations when a successor is accessible. In the definition we use separate functions for normal and exceptional successors. The auxiliary function *addPos* augments the branch conditions with a position formula *Pos p*. This connects verification conditions with the program structure and allows to weave in other properties (system invariants) in a post-processing step.

$$\begin{aligned} &succsF\ \Pi\ p = (case\ cmd\ \Pi\ p\ of\ None \Rightarrow [] \\ &\qquad\qquad \mid Some\ c \Rightarrow addPos\ p\ (succsNrm\ \Pi\ p\ c\ @\ succsExpt\ \Pi\ p\ c)) \\ &addPos\ p\ ss = map\ (\lambda\ (p',B).\ (p', \bigsqcap [B,Pos\ p]))\ ss \end{aligned}$$

The function *succsNrm* yields the successors for normal execution. For example *IfIntLeq* has two successors depending on whether the topmost stack entry *St 0* is less than or equal to *St 1*, the element beneath it.

$$\begin{aligned} &succsNrm\ \Pi\ (C,M,pc)\ (\texttt{IfIntLeq}\ t) = [((C,M,pc+t), St\ 0 \sqsubseteq St\ 1), \\ &\qquad\qquad\qquad\qquad\qquad\qquad ((C,M,pc+1), \sqsupset\ (St\ 0 \sqsubseteq St\ 1))] \end{aligned}$$

For instructions that might throw exceptions, *succsNrm* produces a branch condition that excludes this exception. We write $incA\ \ (C,M,pc)$ to increment positions, e.g. $(C,M,pc+1)$. The auxiliary function *xcpt-cond* generates a condition that ensures a particular exception.

$$\begin{aligned} &succsNrm\ \Pi\ p\ (\texttt{Getfield}\ F\ C\ ex) = [(incA\ p, \sqsupset (xcpt\text{-}cond\ \Pi\ NullPointer\ p))] \\ &cmd\ \Pi\ p = Some\ (\texttt{Getfield}\ F\ C) \longrightarrow xpct\text{-}cond\ \Pi\ X\ p = St\ 0 \sqsupseteq \lfloor Null \rfloor \end{aligned}$$

For `Putfield`, `New` and `Checkcast` the normal successors are determined analogously, only the type of exception differs. Method calls are more complicated, because overwriting opens multiple possibilities. It is hard to determine

statically the real type of the object whose method we are calling. However, we can ask the programmer or compiler to insert proper type annotations. Then we can select the corresponding method entry positions and construct sharp branch conditions.

succsNrm $\Pi\ p$ (`Invoke` $M\ n$) = *succsInvoke* (Π,M,n,p)

First, *succsInvoke* analyses the annotation at p to find out the types of the object reference on top of the stack. It expects this information to be given in form of a disjunction of *Ty ex tp* expressions. Then, it constructs exclusive branch conditions for each type.

succsInvoke (Π,M,n,p) = (*case anF* $\Pi\ p$ *of None* ⇒ []
| *Some A* ⇒ *concat* (*map* (λ *tp*. (*case tp*
of Class X ⇒ [((*X*,*M*,*0*),⌜(*xcpt-cond* Π *NullPointer p*) $\bigwedge$ *Ty* (*St n*) (*Class X*))⌝]
| - ⇒ [])) (*extractTy* (*A*,*St n*))))

For `Return` instructions we scan the code for all positions from which the current method could have been called. The name and class of the current method can be obtained from the position, say (C,M,pc), of the `Return` instruction. Then we scan the code for all positions p' with `Invoke` $M\ n$, which have *Ty* (*St n*) C in its annotation. For each of those call positions p' we construct a branch condition with the annotation, safety formula and position information of p'.

succsNrm $\Pi\ p$ `Return` =
map (λ p'. (*incA* p',*Call* (*And* [*assert* $\Pi\ p'$,*Pos* p']))) (*callers* $\Pi\ p$)
assert $\Pi\ p \equiv \bigwedge$ [*safeF* $\Pi\ p$]@(*case anF* $\Pi\ p$ *of None* ⇒[] | *Some A* ⇒ [*A*])

For the remaining instructions *succsNrm* can be defined analogously to the shown examples. When instructions throw exceptions control flows to an appropriate handler. The function *succsExpt* checks which exceptions each instruction may throw and invokes *succsXpt* to find potential handlers. Example:

succsExpt $\Pi\ p$ (`Getfield` $F\ C$) = *succsXpt* (Π,*NullPointer*,[p])

Handlers are searched by recursively climbing up the call tree and inspecting the exception tables of each call method. In *succsXpt* we keep a list of visited positions. When this list becomes too long or empty, *succsXpt* terminates by making all program positions potential successors. This means programs with uncaught exceptions usually yield unprovable verification conditions. However, adding a global exception handler to the main method always helps to avoid this problem. When *succsXpt* finds a handler it constructs a branch condition that specifies under which situation this handler is selected. When an exception is caught in the same method as it is thrown (pss = []), we get branch condition ⌞*True*⌟, otherwise we restore the call context using *Catch* on the annotation and safety formula of the call point.

succsXpt (Π,X,ps) = (*if length* (*domC* Π) ≤ *length ps* ∨ ps=[]
then map (λ*p*. (p,⌜*True*⌝)) (*domC* Π)
else (*let p*=*fst ps*; (*C*,*M*,*pc*)=*p*; *et*=*ex-table-of P C M*; *A*=*assert* $\Pi\ p$
in (*case match-ex-table P X pc et*

of None ⇒ *concat* (*map* (λ*p′*. *succsXpt* (*Π*,*X*,*p′*#*ps*)) (*callers Π p*))
Some h ⇒ [((*C*,*M*,*h*), ⋀ (*if pss*=[] *then* [] *else* [*Catch X A*])@[*xcpt-cond Π X* (*last ps*)])]

5.2 Weakest Preconditions

The purpose of the *wpF* operator is to express the semantics of the underlying programming language at the level of formulas. We have proven that our *wpF* operator satisfies the following lemma, which implies the requirements we need for correctness and completeness of our *VCG*.

lemma *wf Π* ∧ (*p*,*m*,*e*)∈ *safeP Π* ∧ ((*p*, *m*, *e*), (*p′*, *m′*,*e′*)) ∈ *effS Π*
⟶ *evalE Π* (*p*,*m*,*e*) (*wpF Π p p′ Q*) = *evalE Π* (*p′*, *m′*,*e′*) *Q*

Roughly speaking this lemma says that *wpF Π p p′ Q* transforms a postcondition *Q* such that it evaluates to the same value as *Q* does in the successor state. This can be done by substituting all expressions of a formula *Q* that change its value due to the effect of an instruction by another expressions that yields the same value in the predecessor state. The substitution function *substE*::(*expr* ▷ *expr*) ⇒ *expr* ⇒ *expr* maintains an expression map *em*. It traverses a given formula (not descending into temporal constructs) and simultaneously replaces all instances of expressions that appear on the left hand side of a maplet in *em* by the corresponding right hand side. Example:

substE [(*St 0*,*Rg 0*)] (*St 0* ⌞=⌟ *Call* (*St 0*)) = *Rg 0* ⌞=⌟ *Call* (*St 0*)

Usually substitution only replaces variables. However, Jinja Bytecode instructions may also change the heap. Hence, we sometimes have to substitute entire expressions. In the definition of *wpF* we analyse the postcondition and extract subexpressions of particular patterns. These are then used to build maplets for the substitution map.

wpF Π p p′ Q = (*let pm*=*map* (λ*q*. (*Pos q*,*q*=*p′*⌟)) (*getPosEx Q*)
in (*case cmd Π p of None* ⇒ *FF* | *Some ins* ⇒ (*case handlesEx* (*fst Π*) *p′*
of None ⇒ *wpFNrm Π p p′ Q pm ins* | *Some cn* ⇒ *wpFExpt Π p p′ Q pm ins*)))

When evaluating *wpF Π p p′ Q* we assume that the program counter changes from *p* to *p′*. This means we can eliminate position expressions *Pos q* in *Q*, which we extract with *getPosEx Q*. If *p′* is the start address of some handler for an exception *cn*, we assume that the transition from *p* to *p′* is due to an exception and delegate work to *wpFExpt*. Otherwise we use *wpFNrm*, which transforms *Q* according to normal execution. For example in case of an *IAdd* instruction *wpFNrm* replaces instances of *St 0* with *St 0* ⌞+⌟ *St 1*, which has the same value as *St 0* has in the successor state. Since *IAdd* reduces the stack, all other instances of *St k*, whose indexes are extracted by *stkIds*, get shifted.

wpFNrm Π p p′ pm IAdd = *substE* (*pm*@
(*map* (λ*k*. (*St k*,*if k*=*0 then St 0* ⌞+⌟ *St 1 else St* (*k*+*1*))) (*stkIds Q*))) *Q*

In case of `Getfield` *F C* we only have to transform *St 0*.

wpFNrm Π p p′ pm (*Getfield F C*) ≡ *substE* (*pm*@[(*St 0*,*Gf F C* (*St 0*))]) *Q*

Like other instructions that affect the heap, `Putfield` is more tricky to handle. Apart from shifting the stack, which gets reduced by two elements, we have to scan *Q* for all expressions that depend on the heap. These are all expressions of the form *Gf F C ex*, which we extract in subterm order with *getGfEx*. For each instance, we first transform *ex* using the maplets we have found so far. Then we build an expression that checks whether *ex′* equals the reference of the changed object (in *St 1*). If yes, we replace *Gf F C ex* with the new object field value, stored in *St 0*. Otherwise, we take the transformed version *Gf F C ex′*.

wpFNrm Π p p′ Q pm (*Putfield F C*) = (
let em=*pm*@(*map* (λ*k*. (*St k*,*St* (*k*+*2*))) (*stkIds Q*));
  gfe′=*foldl* (λ*mp ex*. *let x*=*substE mp ex*
    in (*Gf F C ex*,*IF x*=*St 1 THEN St 0 ELSE Gf F C x*)#*mp*)
    em (*getGfEx F C Q*)
in substE gfe′ Q)

Another tricky instruction is `Invoke` *M n*. Since the successor state has one frame more, we replace *FrNr* with *FrNr* +1. Since the call state of the successor state is the current state we replace instances of *Call ex* with *ex*. In case of *Catch X ex* we check wether the current method has an appropriate handler. If so, we can eliminate the *Catch*. Otherwise, we replace it with an expression that checks wether the current state only has one frame. In this case evaluation of *Catch X ex* equals *ex*, hence we eliminate *Catch* again. Otherwise, we leave it. Finally, we handle the argument passing. The first *n* elements of the stack are written into the registers *1* to *n+1* in reversed order. We create maplets that substitute each register with the corresponding stack position of the predecessor state. The stack is emptied, which amounts to replacing all references *St k* with ␣*arbitrary*␣.

wpFNrm Π p p′ Q pm (`Invoke` *M n*) = *substE* (*pm*@(*FrNr*,*FrNr* +1 *Cn* (*Intg 1*))#
(*map* (λ*k*. (*Rg k*,*if k* ≤ *n then St* (*n*−*k*) *else* ␣*arbitrary*␣)) (*rgIds Q*))@
(*map* (λ*k*. (*St k*,␣*arbitrary*␣)) (*stkIds Q*))@
(*map* (λ*x*. (*Call x*,*x*)) (*getCallEx Q*))@
(*concat* (*map* (λ(*c*,*x*).(*if catchesEx* (*fst Π*) *c p then* [(*Catch c x*,*x*)]
  else [(*Catch c x*,*IF FrNr* = 1 *THEN x*
    ELSE Catch c x)])) (*getCatchEx Q*))) *Q*

In case of `Return` the successor state has one frame less. Hence, the evaluation of *Call* and *Catch* expressions needs to be adjusted again. Adding an additional *Call* to such expressions amounts to the same as chopping off the topmost frame of the current state. We skip the formal defintion for *Return* and the remaining instructions, as the same techniques apply as before. It turns out that the instructions that affect the heap or the structure of the frame stack are significantly more difficult to handle. Exception handling works similar for all instructions, but `Throw`, which needs to be treated special because we do not know from the code which exception is thrown. Similarly to *Invoke M n* we require that potential classes are annotated in form of *Ty ex tp* expressions.

```
wpFExpt Π p p' Q pm cn = (let mp=pm@(map (λk. (St k,if 1≤k then ⌞arbitrary⌟
else (if (cmd Π p = Some Throw)
then (IF St 0 ≔ ⌞Null⌟ THEN ⌞addr-of-sys-xcpt NullPointer⌟ ELSE St 0)
else ⌞addr-of-sys-xcpt cn⌟))) (stkIds Q))@
(let (C,M,pc)=p; (C',M',pc')=p'; (P,An)=Π
in (if match-ex-table P cn pc (ex-table-of P C M) = Some pc' then [] else
    let rgm=map (λk. (Rg k,Catch cn (Rg k))) (rgIds Q);
        om=map (λex. (Call ex,Catch cn (Call ex))) (getCallEx Q);
        cm=map (λ(c,x). (Catch c x, Catch cn (Catch c x))) (getCatchEx Q)
        in (FrNr,Catch cn FrNr)#rgm@om@cm))
in substE mp Q)
```

6 Example Program

This section presents a small example program, which we have proven safe.

```
Cl ::jvm-method cdecl
Cl ≡ (A,                    class A {
(Object,[(n,Integer)],[     int n;
(sum,[],Integer,(2,2,[      int sum() {
Push (Intg 0) − ''pre'',    {int k = 0;
Store k,
Push (Intg 0),                int r = 0;
Store r,
Load k,
Load this − ''inv'',
Getfield n A,
IfIntLeq 10,                while (k < n)
Load k,                       {
Push (Intg 1),
IAdd,
Store k,                      k = k + 1;
Load r,
Load k,
IAdd,
Store r,                      r = r + k;
Goto −12                      }
Load r,
Return − ''post'' ,           return r; }
],[]))]))                   }
```

Fig. 2. Example Program

Method *sum* in class *A* adds the numbers from *0* to field value *n*. To verify this program we need annotations. The precondition says field *n* has not changed since call time and ranges between *0* and *65535*, the highest input for which the sum does not overflow.

$$pre \equiv \bigwedge [Rg\ 0 \sqsupseteq Call\ (St\ 0),\ Gf\ n\ A\ Rg\ 0 \sqsupseteq Call\ (Gf\ n\ A\ St\ 0),\ Gf\ n\ A\ Rg\ 0 \leqslant \lfloor 65535 \rfloor,\ \lfloor 0 \rfloor \leqslant Gf\ n\ A\ Rg\ 0]$$

The invariant contains type restrictions and the Gaussian summation formula. It also says that *n* does not change and that *k* ranges between *Intg 0* and the value of *n*.

$$inv \equiv \bigwedge [Ty\ Rg\ k\ Integer,\ Ty\ Rg\ r\ Integer,\ \lfloor 2 \rfloor \sqcup^* Rg\ r \sqsupseteq Rg\ k \sqcup^* (Rg\ k \sqcup \lfloor 1 \rfloor),\ Gf\ n\ A\ Rg\ 0 \sqsupseteq Call\ (Gf\ n\ A\ St\ 0),\ Gf\ n\ A\ Rg\ 0 \leqslant \lfloor 65535 \rfloor,\ \lfloor 0 \rfloor \leqslant Gf\ n\ A\ Rg\ 0,\ Rg\ k \leqslant Gf\ n\ A\ Rg\ 0,\ \lfloor 0 \rfloor \leqslant Rg\ k]$$

The postcondition contains type information again, and a formula that relates the result value to the input value *n*, which still has the same value as at call time.

$$post \equiv \bigwedge [St\ 0 \sqsupseteq Rg\ r,\ Ty\ Rg\ r\ Integer,\ Gf\ n\ A\ Rg\ 0 \sqsupseteq Call\ (Gf\ n\ A\ St\ 0),\ \lfloor 2 \rfloor \sqcup^* Rg\ r \sqsupseteq Call\ (Gf\ n\ A\ St\ 0) \sqcup^* (Call\ (Gf\ n\ A\ St\ 0)) \sqcup \lfloor 1 \rfloor]$$

Proving the verification condition of this program is automatic using a tactic for bounded arithmetics. The type annotations are used to trigger simplification rules that translate the arithmetic expressions of type *expr* turning up in the verification conditions to arithmetic expressions in Isabelle/HOL. For the latter powerful decision procedures, such as Presburger Arithmetics are available. We have tested Jinja programs that call this method up to the size of *10.000* instructions. Up to this size the decision procedures and the simplifier scale pretty well. Details can be found online [1].

7 Generating Annotations

The annotations in the example program have been added manually. They have been designed to make the verification go through automatically with a general setup of simplification rules and decision procedures. In practice a fully automatic approach would be desirable. In many cases program analysis can help to find proper annotations. For example the type information of the annotations above could be directly transferred from the bytecode verifier's analysis. To find out upper and lower bounds of expressions, we can employ interval analysis for Java bytecode [17]. More complex invariants could be gained by advanced analysis techniques, such as polyhedra or affine relations [12]. For annotations involving analysis of pointer structures TVLA [11] can help. Since the generation of annotations need not be trusted, a wide range of options are available at this point.

8 Conclusion

To our knowledge the literature on Java does not propose a logic to annotate and verify bytecode. In this paper we tried to fill this gap for a safety policy against arithmetic overflow and a bytecode subset that covers the essential object oriented features. This and various other instantiations of our PCC framework [1] show that a PCC system with formally verified trusted components is feasible. The infrastructure an interactive theorem prover like Isabelle/HOL provides is very useful. In particular the ability to generate proofs with decision procedures or interactively with the full power of HOL available, turns out to be a good strategy in a field where Rice's theorem shatters the dream of complete automation.

References

1. VeryPCC project website in Munich, http://isabelle.in.tum.de/verypcc/, 2003.
2. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In *Verification: Theory and Practice*, volume 2772 of *Lect. Notes in Comp. Sci.*, pages 11–41. Springer-Verlag, 2004.
3. A. W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258, June 2001.

4. D. Aspinall, L. Beringer, M. Hoffman, and H.-W. Loidl. A resource-aware program logic for a jvm-like language. In S. Gilmore, editor, *Trends in Functional Programming*. Edinburgh, 2003.
5. F. D. Boer and C. Pierik. A syntax-directed hoare logic for object-oriented programming concepts. In *Proceedings of Formal Methods for Open Object-based Distributed Systems (FMOODS)*, LNCS. Springer, 2003.
6. J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound tal for back-end optimization. In *Programming Languages Design and Implementation (PLDI)*. ACM Sigplan, 2003.
7. D. L. Detlefs, K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical report, Compaq Systems Research Center, 1998.
8. C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 2nd edition, 1990.
9. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Research report, National ICT Australia, Sydney, 2004.
10. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. Jml reference manual (draft). Technical report, 2004.
11. T. Lev-Ami, T. Reps, M. Sagiv, and T. Wilhelm. Putting static analysis to work for verification: A case study in issta 2000. Technical report, 2000.
12. M. Mueller-Olm and H. Seidl. Program analysis through linear algebra. In *31st Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 330–341, 2004.
13. G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.
14. T. Nipkow and A. Chaieb. Generic proof synthesis for presburger arithmetic – draft. Technical report, Technische Universitaet Muenchen, 2004.
15. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer, 2002.
16. D. v. Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.
17. M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. *Proceedings of the 1st Workshop on Bytecode (Bytecode05)*, 2005. submitted for publication.
18. M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In *Proc. 17th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs 2004)*. Springer Verlag, 2004. 16 pages.
19. M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping proof carrying code. In *Proc. 3rd IFIP Int. Conf. Theoretical Computer Science (TCS 2004)*, 2004.
20. G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.