Preservation of proof obligations for hybrid verification methods

Gilles Barthe
IMDEA Software, Spain

César Kunz
INRIA Sophia Antipolis, France

David Pichardie
INRIA Rennes/IRISA, France

Julián Samborski-Forlese
IMDEA Software, Spain

Abstract

Program verification environments increasingly rely on hybrid methods that combine static analyses and verification condition generation. While such verification environments operate on source programs, it is often preferable to achieve guarantees about executable code. We show that, for a hybrid verification method based on numerical static analysis, it is possible to transfer evidence from source to compiled programs. More concretely, we formalize a hybrid verification method for compiled programs and show that compilation preserves proof obligations. The crux of our result is a proof that compilation preserves solutions of analysis; this is achieved by relying on a bytecode analysis that performs symbolic execution of stack expressions in order to overcome the loss of precision incurred by performing static analyses on compiled (rather than source) code. Then, we show that hybrid verification methods are sound by proving that every program provable by hybrid methods is also provable (at a higher cost) by standard methods. Our results generalize to hybrid methods our earlier work on preservation of proof obligations, and warrant their use in Proof Carrying Code scenarios where certificates are generated from source code verification.

1 Introduction

Program verification techniques are widely used to reason about the correctness of applications, and play an important role in fields such as mobile code and embedded systems where strong guarantees are required. However, program verification, and in particular deductive program verification, is traditionally applied to source code, whereas guarantees are often required about executable code. This discrepancy is particularly acute in the context of mobile code, where code consumers do not trust code producers and may not have access to the source code. Therefore, it is of interest to develop methods to transfer evidence from source code verification to the code consumers. In earlier work [3], we focus on transferring proofs from source to compiled programs in the context of verification methods based on verification condition generators, that are commonly used in Proof Carrying Code [16] and program verification environments. We have shown that non-optimizing compilation preserves proof obligations, and therefore that the certificates of source programs can be reused directly to validate compiled programs. However, state-of-the-art verification tools do not use plain verification condition generation; instead, they rely on hybrid methods, that combine static analyses and verification condition generation.

The objective of the paper is to extend preservation of proof obligations to hybrid verification methods. For concreteness, we consider a small imperative language with arrays, and we focus on a hybrid method based on a generic numerical analysis, inspired by [14, 6], and that can be instantiated to several numeric domains, including polyhedra.

We first define a hybrid verification method in which programs are subjected to static analysis, and then to verification condition generation. The VCgen exploits the information of the analysis in two useful ways: on the one hand, verification conditions that originate from spurious edges in the control-flow graph are discarded: more precisely, the VCgen ignores the case of out-of-bound accesses whenever the analysis ensures that accesses are within bounds. This leads to fewer, smaller verification conditions. Furthermore, the VCgen adds the results of the analysis as additional assumptions to help the user prove the verification conditions. This is particularly useful for the relational analyses considered as they can provide part of the invariants required to prove programs correct.

Then, we prove preservation of proof obligations using the techniques of [5, 3]. The proof relies on knowing that the solutions of the analysis are preserved by compilation. Although analyzing compiled programs is known to be less precise than analyzing source programs, see e.g. [13], we achieve preservation of solutions by defining at byte-

An extended version of the paper is available online at:
http://www-sop.inria.fr/everest/research/Hybrid/

code level an analysis that performs a symbolic execution of stacks, as in [22, 21, 6].

Finally, we relate hybrid verification to standard verification. We show that programs that are provably correct using our hybrid method, remain provably correct using standard verification condition generation; to this end, we define a compiler that transforms a hybrid specification (combing logical assertions and analysis results) into a logical one by giving a logical interpretation of the analysis results.

2 Setting

This section introduces the source language (an imperative language with arrays of integers), the target language (a stack-based language with jumps), and the compiler.

We assume given two disjoint sets V_s of scalar variables and V_a of array variables, and let V denote $V_s + V_a$. Each variable in V_a has an associated size. Furthermore, we assume given two sets V_s^{old} and V_a^{old} in 1-1 correspondence with V_s and V_a, which are used to store initial values. We also consider a special variable res, which is used to represent the value of the program result. Finally, we assume given a set $\mathbf{Lab} \subset \mathbb{N}$ of labels.

Source Language

Programs are defined as commands, and are decorated with labels in order to express analysis results:

$$\begin{array}{lcl} e & ::= & e \;\mathsf{op}\; e \mid n \mid x \mid a[e] \\ c & ::= & \mathtt{Skip} \mid [x{:=}e]^k \mid [a[e]{:=}e]^k \mid c;c \mid [\mathtt{return}\; e]^k \\ & & \mid \mathtt{if}\; [e \bowtie e]^k \;\mathtt{then}\; c \;\mathtt{else}\; c \\ & & \mid \mathtt{while}\; [e \bowtie e]^k \;\mathtt{do}\; c \end{array}$$

where x, a, n and k respectively range over V_s, V_a, $\mathbb{Z}$ and $\mathbf{Lab}$, op ranges over (binary) arithmetic operations, and $\bowtie$ over arithmetic comparisons. We assume that labels occur at most once in commands.

The semantics of source programs is formalized by a small-step transition relation between states. States may be intermediate, in which case they consist of a statement and of a memory, or final, in which case they consist of a memory, and possibly a tag to denote abnormal termination. Memories are modeled as pairs of mappings respectively from variables to values and from arrays to indices to values. We assume that each array a comes equipped with its size $|a|$ and define the semantic domains of the source language as follows:

$$\begin{array}{rcl} VMem & = & V_s \to \mathbb{Z} \\ AMem & = & \Pi a \in V_a.\, \{i \mid 1 \leq i \leq |a|\} \to \mathbb{Z} \\ Mem & = & VMem \times AMem \\ State_I^s & = & Stmt \times VMem \times AMem \\ State_F^s & = & VMem \times AMem \times (\mathbb{Z} + \{\mathbf{AOB}\}) \\ State^s & = & State_I^s + State_F^s \end{array}$$

The operational semantics of programs is standard and, thus, omitted. (See the next subsection for the semantics of instructions that manipulate arrays.)

Bytecode Language

A bytecode programs is defined as a list of instructions. Instructions either manipulate the memory that stores the values of variables and the contents of arrays, or manipulate the operand stack, or perform a conditional or unconditional jump. The set of instructions is defined by the following grammar:

$$\begin{array}{rl} ins ::= & \mathtt{prim\;op} \mid \mathtt{push}\; v \mid \mathtt{load}\; x \mid \mathtt{store}\; x \mid \mathtt{return} \\ & \mid \mathtt{aload}\; a \mid \mathtt{astore}\; a \mid \mathtt{cjmp} \bowtie l \mid \mathtt{jmp}\; l \mid \mathtt{nop} \end{array}$$

We denote by $\dot{p}[l]$ the instruction at position l of a bytecode program $\dot{p}$. The semantics of bytecode programs is formalized using a transition relation between states. States may either be intermediate or final; intermediate states consist of a program counter, an operand stack, that stores the results of intermediate computations, and a memory. The semantic domains of the bytecode language are defined as follows, where we implicitly assume that the program counter is within the bounds of programs.

$$\begin{array}{rcl} Stack & = & \mathbb{Z}^\star \\ State_I^b & = & \mathbb{N} \times VMem \times AMem \times Stack \\ State_F^b & = & VMem \times AMem \times (\mathbb{Z} + \{\mathbf{AOB}\}) \\ State^b & = & State_I^b + State_F^b \end{array}$$

The operational semantics of programs is standard. We only provide the operational semantics of the instructions $\mathtt{aload}$ and $\mathtt{astore}$; these instructions may cause abrupt termination if array accesses are out-of-bound. The rules are given in Figure 2, where we use the notation $[f \mid s \to r]$ to refer the function that is identical to f everywhere except in r that returns s, for any sets R and S and any function $f : R \to S$.

Compiler

The compiler is standard, and defined in Figure 1; we use the function $init : \mathbf{Stm} \to \mathbf{Lab}$ to associate to each statement its initial label. We assume *label compatibility*, i.e. that the label of a source statement is the same as the label of the program point for its compilation.

Throughout the rest of the paper we let P be a source program, and the bytecode program $\dot{p}$ the result of the compilation of program P.

3 Preservation of solutions

It is folklore that compilation potentially yields a loss of precision for relational analyses. The purpose of this section is to show that solutions of abstract interpretations are

$[\![n]\!]_{\mathrm{e}} = \texttt{push}\ n$
$[\![x]\!]_{\mathrm{e}} = \texttt{load}\ x$
$[\![x[e]]\!]_{\mathrm{e}} = [\![e]\!]_{\mathrm{e}};\ \mathsf{aload}\ x$
$[\![e_1\ \mathsf{op}\ e_2]\!]_{\mathrm{e}} = [\![e_2]\!]_{\mathrm{e}};\ [\![e_1]\!]_{\mathrm{e}};\ \texttt{prim op}$

$[\![[x{:=}e]^k]\!] = k : [\![e]\!]_{\mathrm{e}}; \texttt{store}\ x$
$[\![[a[e_1]{:=}e_2]^k]\!] = k : [\![e_2]\!]_{\mathrm{e}};\ [\![e_1]\!]_{\mathrm{e}};\ \texttt{astore}\ a$
$[\![s1; s2]\!] = [\![s_1]\!];\ [\![s_2]\!]$
$[\![[\texttt{return}\ e]^k]\!] = k : [\![e]\!]_{\mathrm{e}};\ \texttt{return}$
$[\![[\texttt{Skip}]^k]\!] = k : \texttt{nop}$
$[\![\texttt{if}\ [e_1 \bowtie e_2]^k\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!] =$
$k : [\![e_2]\!]_{\mathrm{e}}; [\![e_1]\!]_{\mathrm{e}};\ \texttt{cjmp} \bowtie k_1; k_2 : [\![s_2]\!]; \texttt{jmp}\ l; k_1 : [\![s_1]\!]$
where $k_1 = init(s_1) = k_2 + |[\![s_2]\!]| + 1$
$k_2 = init(s_2) = k + |[\![e_2]\!]_{\mathrm{e}}; [\![e_1]\!]_{\mathrm{e}}| + 1$
$l = k_1 + |[\![s_1]\!]|$
$[\![\texttt{while}\ [e_1 \bowtie e_2]^k\ \texttt{do}\ s]\!] =$
$k : [\![e_2]\!]_{\mathrm{e}}; [\![e_1]\!]_{\mathrm{e}}; \texttt{cjmp} \bowtie k_1; \texttt{jmp}\ l; k_1 : [\![s]\!]; \texttt{jmp}\ k$
where $k_1 = k + |[\![e_2]\!]_{\mathrm{e}}| + |[\![e_1]\!]_{\mathrm{e}}| + 2$
$l = k_1 + |[\![s]\!]| + 1$

Figure 1. Compiler

$$\frac{P[i] = \mathsf{aload}\ a \qquad 0 \le n < |a|}{\langle i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, n :: s\rangle \rightsquigarrow \langle i+1, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, \rho_{\mathsf{a}}\ a\ n :: s\rangle}$$

$$\frac{P[i] = \mathsf{aload}\ a \qquad \neg\, 0 \le n < |a|}{\langle i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, n :: s\rangle \rightsquigarrow_{EX} \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, \mathsf{AOB}\rangle}$$

$$\frac{P[i] = \mathsf{astore}\ a \qquad 0 \le n < |a|}{\langle i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, n :: v :: s\rangle \rightsquigarrow \langle i{+}1, \rho_{\mathsf{v}}, [\rho_{\mathsf{a}} \mid a \to [\rho_{\mathsf{a}}\ a \mid n \to v]], s\rangle}$$

$$\frac{P[i] = \mathsf{astore}\ a \qquad \neg\, 0 \le n < |a|}{\langle i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, n :: v :: s\rangle \rightsquigarrow_{EX} \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, \mathsf{AOB}\rangle}$$

$$\frac{P[i] = \mathsf{return}}{\langle i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, n : s\rangle \rightsquigarrow \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, n\rangle}$$

Figure 2. Semantics of bytecode (excerpts)

preserved by compilation, provided one uses symbolic expressions, as done in [22, 21, 6], to mitigate the presence of the operand stack and to recover the loss of precision incurred by compilation.

Symbolic Expressions

Expressions and guards serve as the interface with the numerical relational domain in the analysis for bytecode. Below we let x range over V.

$$\begin{array}{rl} Expr \ni e ::= & n \mid x \mid x[e] \mid ? \mid ?[e] \mid e\ \mathsf{op}\ e \quad x \in V \\ Guard \ni t ::= & e \bowtie e \end{array}$$

The expression $?$ represents an unknown value; therefore, expressions are interpreted as sets of possible values. Formally, the semantics $[\![e]\!]_\rho$ and $[\![t]\!]_\rho$ of expressions with respect to an environment $\rho = \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}\rangle$ are defined by the clauses:

$$[\![n]\!]_\rho = \{n\} \quad [\![x]\!]_\rho = \rho_{\mathsf{v}}\ x \quad [\![?]\!]_\rho = \mathbb{Z} \quad [\![?[e]]\!]_\rho = \mathbb{Z}$$
$$\begin{array}{rcl} [\![x[e]]\!]_\rho & = & \{\rho_{\mathsf{a}}\ x\ v \mid v \in [\![e]\!]_\rho\} \\ [\![e_1\ \mathbf{op}\ e_2]\!]_\rho & = & \{n_1\ \mathbf{op}\ n_2 \mid n1 \in [\![e_1]\!]_\rho,\ n_2 \in [\![e_2]\!]_\rho\} \\ [\![e_1 \bowtie e_2]\!]_\rho & \iff & \exists\, n_1 \in [\![e_1]\!]_\rho,\ n_2 \in [\![e_2]\!]_\rho \bullet n_1 \bowtie n_2 \end{array}$$

Note that the expression $?$ is not required for analyzing bytecode programs that are achieved by compilation of the source program, since the stack is empty after storing a value in an array. However, it provides more precision when dealing with programs that are not obtained by compilation.

Abstract domain

Following Miné [14], we assume given an abstract numerical domain interface, which can be instantiated with standard relational abstract domains. The interface consists of a domain $\mathbb{D}$ equipped with a partial order $\sqsubseteq\ \subseteq \mathbb{D} \times \mathbb{D}$, meet and join operators $\sqcap, \sqcup : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$, a least element $\bot$ and a greater element $\top$. We also assume given abstract assignment functions $[\![x{:=}e]\!]^\sharp, [\![x[e_1]{:=}e_2]\!]^\sharp : \mathbb{D} \to \mathbb{D}$, and a function $assume^\sharp$ that maps guards to abstract elements.

Finally, we assume given a monotone concretization function $\gamma : \mathbb{D} \to \mathcal{P}(VMem \times AMem)$ mapping abstract elements to sets of environments in $VMem \times AMem$, and satisfying the following properties:

$$\gamma(d_1 \sqcap d_2) \supseteq \gamma(d_1) \cap \gamma(d_2)$$
$$\gamma(d_1 \sqcup d_2) \supseteq \gamma(d_1) \cup \gamma(d_2)$$

$$\gamma([\![x{:=}e]\!]^\sharp(d)) \supseteq \{\langle \rho_{\mathsf{v}}[x \mapsto v], \rho_{\mathsf{a}}\rangle \mid \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}\rangle \in \gamma(d) \wedge v \in [\![e]\!]_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}\rangle}\}$$
$$\gamma([\![x[e_1]{:=}e_2]\!]^\sharp(d)) \supseteq \{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}[x \mapsto v_1 \mapsto v_2]\rangle \mid \rho = \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}\rangle \in \gamma(d) \wedge v_1 \in [\![e_1]\!]_\rho \wedge v_2 \in [\![e_2]\!]_\rho\}$$

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

We define the abstract test $[\![t]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ of a guard $t \in Guard$ by $[\![t]\!]^\sharp(l^\sharp) = assume^\sharp(t) \sqcap l^\sharp$.

Source Code Analysis

The source code analysis is specified by abstract transfer functions that map elements of the abstract domain into elements of the abstract domain.

Definition 1 (Abstract Domain for High-Level) *A result of the analysis for the source program P is described by a mapping Loc in the lattice*

$$State^\sharp = \mathbf{Lab} \to \mathbb{D}\ .$$

stm	F_{stm}
`Skip`	$l^\sharp \to l^\sharp$
$x{:=}e$	$l^\sharp \to [\![x{:=}e]\!]^\sharp(l^\sharp)$
$a[x]{:=}y$	$l^\sharp \to [\![a[x]{:=}y]\!]^\sharp(l^\sharp)$

$$\frac{stm \notin \{\texttt{if}\ t\ \texttt{then}\ s_1\ \texttt{else}\ s_2,\ \texttt{while}\ t\ \texttt{do}\ c,\ s_1\ ;\ s_2, \texttt{return}\ e\}}{Loc \vdash \{Loc(i)\}\ [stm]^i\ \{F_{stm}(Loc(i))\}} \qquad \frac{}{Loc \vdash \{Loc(i)\}\ [\texttt{return}\ e]^i\ \{\bot\}}$$

$$\frac{Loc \vdash \{[\![t]\!]^\sharp(Loc(i))\}\ s_1\ \{l_1^\sharp\} \qquad Loc \vdash \{[\![\neg t]\!]^\sharp(Loc(i))\}\ s_2\ \{l_2^\sharp\}}{Loc \vdash \{Loc(i)\}\ \texttt{if}\ [t]^i\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{l_1^\sharp \sqcup l_2^\sharp\}}$$

$$\frac{Loc \vdash \{[\![t]\!]^\sharp(Loc(i))\}\ s\ \{l^\sharp\} \qquad l^\sharp \sqsubseteq Loc(i)}{Loc \vdash \{Loc(i)\}\ \texttt{while}\ [t]^i\ \texttt{do}\ s\ \{[\![\neg t]\!]^\sharp(Loc(i))\}} \qquad \frac{Loc \vdash \{l^\sharp\}\ s_1\ \{l_1^\sharp\} \qquad Loc \vdash \{l_1^\sharp\}\ s_2\ \{l_2^\sharp\}}{Loc \vdash \{l^\sharp\}\ s_1\ ;\ s_2\ \{l_2^\sharp\}}$$

$$\frac{Loc \vdash \{\top\}\ P\ \{l^\sharp\}}{Loc \vdash P}$$

Figure 3. Definition of the constraint system for the source code analysis.

Definition 2 (Solution) *A mapping Loc for the source program P is a solution of the analysis if it verifies the constraint system defined in Figure 3, i.e. $Loc \vdash P$ holds.*

Byte Code Analysis

As for the source code analysis, the bytecode analysis is defined by abstract transfer functions that map abstract states into abstract states. In this case, the abstract states are pairs of the form $(s^\sharp, l^\sharp)$ where $l^\sharp$ is an element of the abstract domain, and the list of symbolic expressions $s^\sharp$ abstracts the operand stack. The symbolic abstract domain for stacks is $Expr^\star$, where for any set A, $A^\star$ denotes the domain of lists with elements in A. The set of variables considered by the bytecode analysis is the same as in the source code analysis.

Definition 3 (Bytecode Abstract Domain) *A result of the analysis for $\dot{p}$ is described by a mapping $\dot{loc}$ in the lattice*

$$\dot{state}^\sharp = \mathbf{Lab} \to (Expr_L^\star \times \mathbb{D}) \ .$$

An analysis result is a solution of the analysis if it satisfies the constraint system associated to each program. The constraint system is defined in Figure 4. For instructions other than branching or return instructions, the constraint is defined by partial transfer functions in $Expr^\star \times \mathbb{D} \rightharpoonup (Expr^\star \times \mathbb{D})$, most of them defined as a symbolic execution affecting the abstract representation of the operand stack.

Definition 4 (Solution) *A mapping $\dot{loc}$ for the bytecode program $\dot{p}$ is a solution of the analysis if it satisfies the constraint system of Figure 4, i.e. if $\dot{loc} \vdash \dot{p}$ holds.*

Preservation of Solutions

We define first the compilation of a source code analysis solution and then show that it is a solution for the byte code analysis. For notational convenience, we denote by $\dot{f}_{s_1;\cdots;s_n}(s^\sharp, l^\sharp)$ the composition $\dot{f}_{s_n}(\cdots(\dot{f}_{s1}(s^\sharp, l^\sharp))\cdots)$, where $s_1;\cdots;s_n$ is a sequence of byte code instructions. Let $\mathsf{succ}(l)$ denotes the set of successors of a label l, e.g. $\mathsf{succ}(l) = \emptyset$ and $\mathsf{succ}(l) = \{l+1, l'\}$ respectively for $\dot{p}[l] = \texttt{return}$ and $\dot{p}[l] = \texttt{cjmp} \bowtie l'$. The set $\mathsf{pred}(l)$ is defined as $\{l' \mid l \in \mathsf{succ}(l')\}$.

Remark 5 *For each byte code program $\dot{p}$, we can extract from the previous constraint system a set of transfer functions $(\dot{g}_{i,j})_{(i,j)\in\mathbf{Lab}^2}$ such that $\dot{loc} \vdash \dot{p}$ if and only if $\bigsqcup_{k'\in\mathsf{pred}(k)} \dot{g}_{k',k}(\dot{loc}(k')) \sqsubseteq \dot{loc}(k)$ for all $k \in \mathsf{dom}(\dot{p})$.*

We can extend a partial function $\dot{loc}_{partial} \in \dot{state}^\sharp$ to a total function $\dot{loc}$ on $\mathsf{dom}(\dot{p})$ if we set $\dot{loc}(k)$ equal to:

if $k \in \mathsf{dom}(\dot{loc}_{partial})$ then $\dot{loc}_{partial}(k)$
else if $k \in \mathsf{dom}(\dot{p})$ then $\bigsqcup_{k'\in\mathsf{pred}(k)} \dot{g}_{k',k}(\dot{loc}(k'))$
else undef

This definition only make sens if, by considering the control flow graph of $\dot{p}$ whose edges are $\{(i,j) | i \in \mathsf{dom}(\dot{p}) \wedge j \in \mathsf{succ}(i)\}$, every loop contain a label in $\mathsf{dom}(\dot{loc}_{partial})$. We call completion *of $\dot{loc}_{partial}$ this function $\dot{loc}$.*

Definition 6 (Compiled analysis results) *Given an analysis result Loc for the program P, an analysis result compiled from Loc is the completion of the function $\dot{loc}_{partial}$ defined on each $k \in \mathsf{dom}(Loc)$ by $\dot{loc}_{partial}(k) = ([\,], Loc(k))$.*

This definition can be shown to be well defined from the facts that Loc annotates every loop in P and each loop in the control flow graph of $\dot{p}$ contains a label of a loop in P.

$instr$	$\dot{f}_{instr}$
$\texttt{prim}\, op$	$(e_1 :: e_2 :: s^\sharp, l^\sharp) \to (\llcorner e_1\ \underline{op}\ e_2 \lrcorner :: s^\sharp, l^\sharp)$
$\texttt{push}\, n$	$(s^\sharp, l^\sharp) \to (n :: s^\sharp, l^\sharp)$
$\texttt{load}\, r$	$(s^\sharp, l^\sharp) \to (\llcorner r \lrcorner :: s^\sharp, l^\sharp)$
$\texttt{store}\, r$	$(e :: s^\sharp, l^\sharp) \to (s^\sharp[?/r], [\![r{:=}e]\!]^\sharp(l^\sharp))$
$\texttt{aload}\, a$	$(e :: s^\sharp, l^\sharp) \to (\llcorner a[e] \lrcorner :: s^\sharp, l^\sharp)$
$\texttt{astore}\, a$	$(e_1 :: e_2 :: s^\sharp, l^\sharp) \to (s^\sharp[?/a], [\![a[e_1]{:=}e_2]\!]^\sharp(l^\sharp))$
$\texttt{nop}$	$(s^\sharp, l^\sharp) \to (s^\sharp, l^\sharp)$

$$\frac{\mathsf{Instr} \notin \{\,\texttt{jmp}\, i', \texttt{cjmp} \bowtie i', \texttt{return}\} \qquad \dot{f}_{instr}(\dot{loc}(i)) \sqsubseteq \dot{loc}(i+1)}{\dot{loc} \vdash i : \mathsf{Instr}} \qquad \frac{}{\dot{loc} \vdash i : \texttt{return}}$$

$$\frac{\dot{loc}(i) = (e_1 :: e_2 :: s^\sharp, l^\sharp) \qquad (s^\sharp, [\![\neg(e_1 \bowtie e_2)]\!]^\sharp(l^\sharp)) \sqsubseteq \dot{loc}(i+1) \qquad (s^\sharp, [\![e_1 \bowtie e_2]\!]^\sharp(l^\sharp)) \sqsubseteq \dot{loc}(j)}{\dot{loc} \vdash i : \texttt{cjmp} \bowtie j} \qquad \frac{\dot{loc}(i) \sqsubseteq \dot{loc}(j)}{\dot{loc} \vdash i : \texttt{jmp}\, j}$$

$$\frac{\top \sqsubseteq \dot{loc}(0) \qquad \forall i \in \mathsf{dom}(\dot{p}) \bullet\ \dot{loc} \vdash i : \dot{p}[i]}{\dot{loc} \vdash \dot{p}}$$

Figure 4. Definition of the constraint system for the byte code analysis.

Lemma 7 *Let $\dot{p}_1$, $\dot{p}_2$ and e such that $\dot{p} = \dot{p}_1 :: l : [\![e]\!]_{\mathrm{e}} :: l' : \dot{p}_2$. Then, $\dot{loc}(l') = f_{i_1;\ldots;i_k}(s^\sharp, l^\sharp) = (e :: s^\sharp, l^\sharp)$ where $(s^\sharp, l^\sharp) = \dot{loc}(l)$ and $[i_1; \ldots; i_k] = [\![e]\!]_{\mathrm{e}}$.*

PROOF. *We prove the lemma by structural induction over expression e.*

Lemma 8 *Let $\dot{p}_1$, $\dot{p}_2$ and e such that $\dot{p} = \dot{p}_1 :: k_1 : [\![e]\!]_{\mathrm{e}} :: k_2 : \dot{p}_2$. Then*

$$\forall k \in [k_1, k_2),\ \dot{loc} \vdash k : \dot{p}[k].$$

PROOF. *We shall prove it by induction over expression e. Note that $k_2 \notin \mathsf{dom}(Loc)$ since $\dot{p}$ is a compiled program.*

Case $e = n$. *In this case $[\![e]\!]_{\mathrm{e}} = \texttt{push}\ n$, $[k_1, k_2) = \{k_1\}$ and since $\mathsf{pred}(k_1 + 1) = \{k_1\}$, $\dot{loc}(k_1 + 1) = \dot{f}_{\texttt{push}\ n}(\dot{loc}(k_1))$, which implies that $\dot{loc} \vdash k_1 : \texttt{push}\ n$.*

Case $e = x$. *We have $[\![e]\!]_{\mathrm{e}} = \texttt{load}\ x$, $[k_1, k_2) = \{k_1\}$ and since $\mathsf{pred}(k_1{+}1) = \{k_1\}$, $\dot{loc}(k_1{+}1) = \dot{f}_{\texttt{load}\ x}(\dot{loc}(k_1))$, which implies that $\dot{loc} \vdash k_1 : \texttt{load}\ x$.*

Case $e = a[e']$. *Here $[\![e]\!]_{\mathrm{e}} = [\![e']\!]_{\mathrm{e}};\ \texttt{aload}\ a$. Let $k' = k_1 + |[\![e']\!]_{\mathrm{e}}|$. By induction hypothesis we have that $\forall k \in [k_1, k')$, $\dot{loc} \vdash k : \dot{p}[k]$ and since $\mathsf{pred}(k'{+}1) = \{k'\}$ then $\dot{f}_{\texttt{aload}\ a}(\dot{loc}(k'))$ which implies that $\dot{loc} \vdash k' : \texttt{aload}\ a$.*

Case $e = e_1 \mathsf{op}\ e_2$. *This give $[\![e]\!]_{\mathrm{e}} = [\![e_2]\!]_{\mathrm{e}};\ [\![e_1]\!]_{\mathrm{e}};\ \texttt{prim}\,\mathsf{op}$. Let $k'' = k_1 + |[\![e'']\!]_{\mathrm{e}}|$ and $k' = k'' + |[\![e']\!]_{\mathrm{e}}|$. By induction hypothesis, $\forall k \in [k_1, k'') \cup [k'', k')$, $\dot{loc} \vdash k : \dot{p}[k]$ and since the only predecessor of $k_1 + 1$ is k, $\dot{loc}(k' + 1) = \dot{f}_{\texttt{prim}\ \mathsf{op}}(\dot{loc}(k'))$, which means that $\dot{loc} \vdash k' : \texttt{prim}\,\mathsf{op}$.*

Thus, we have proved that $\forall k \in [k_1, k_2)$, $\dot{loc} \vdash k : \dot{p}[k]$.

The following lemma states the main result of this section: compilation preserves analysis solutions.

Lemma 9 *If Loc is s.t. $Loc \vdash P$, then the analysis result $\dot{loc}$ compiled from Loc is s.t. $\dot{loc} \vdash \dot{p}$, i.e. it is a solution of the bytecode analysis.*

PROOF. *Let suppose that $\dot{p} = \dot{p}_1 :: k_1 : [\![s]\!] :: k_2 : \dot{p}_2$ and that there exists $l^\sharp$ such that $Loc \vdash \{Loc(k_1)\}\ s\ \{l^\sharp\}$ and $([], l^\sharp) \sqsubseteq \dot{loc}(k_2)$. We shall prove that $\forall k \in [k_1, k_2)$, $\dot{loc} \vdash k : \dot{p}[k]$. In order to do that, we proceed by induction over statement s. In this proof we omit the calculus of the primed labels.*

Case $s = [\texttt{Skip}]^{k_1}$. *We have $[\![s]\!] = \texttt{nop}$. Since*

$$\begin{aligned}
\dot{f}_{\texttt{nop}}(\dot{loc}(k_1)) &= \dot{loc}(k_1)\\
&= ([], Loc(k_1))\\
&= ([], F_{\texttt{Skip}}(\dot{loc}(k_1)))\\
&= ([], l^\sharp)\\
&\sqsubseteq \dot{loc}(k_2)\\
&= \dot{loc}(k_1 + 1),
\end{aligned}$$

then $\dot{loc} \vdash k_1 : \texttt{nop}$.

Case $s = [x{:=}e]^{k_1}$. *Here $[\![s]\!] = [\![e]\!]_{\mathrm{e}}; k_1' : \texttt{store}\ x$. By Lemma 8, $\forall k \in [k_1, k_1')$, $\dot{loc} \vdash k : \dot{p}[k]$ and since*

$$\begin{aligned}
\dot{f}_{\texttt{store}\ x}(\dot{loc}(k_1')) &= \dot{f}_{\texttt{store}\ x}(([e], Loc(k_1')))\\
&= ([], [\![x{:=}e]\!]^\sharp(Loc(k_1)))\\
&= ([], F_{x:=e}(\dot{loc}(k_1)))\\
&= ([], l^\sharp)\\
&\sqsubseteq \dot{loc}(k_2)\\
&= \dot{loc}(k_1' + 1),
\end{aligned}$$

also $\dot{loc} \vdash k_1' : x{:=}e$.

Case $s = [a[e_1]{:=}e_2]^{k_1}$. $[\![s]\!] = [\![e_2]\!]_{\mathrm{e}}; [\![e_1]\!]_{\mathrm{e}}; k_1'{:}\texttt{astore}\ a$. *By Lemma 8,* $\forall k \in [k_1, k_1'),\ \dot{loc} \vdash k : \dot{p}[k]$ *and since*

$$\begin{aligned}\dot{f}_{\texttt{astore}\ a}(\dot{loc}(k_1')) &= \dot{f}_{\texttt{astore}\ a}(([e_1, e_2], Loc(k_1'))) \\ &= ([], [\![a[e_1]{:=}e_2]\!]^\sharp(Loc(k_1))) \\ &= ([], F_{a[e_1]:=e_2}(\dot{loc}(k_1))) \\ &= ([], l^\sharp) \\ &\sqsubseteq \dot{loc}(k_2) \\ &= \dot{loc}(k_1' + 1),\end{aligned}$$

also $\dot{loc} \vdash k_1' : a[e_1]{:=}e_2$.

Case $s = s_1; s_2$. *In this case* $[\![s]\!] = [\![s_1]\!]; k_1' : [\![s_2]\!]$. *By Induction Hypothesis we know that* $\forall k \in [k_1, k_1') \cup [k_1', k_2),\ \dot{loc} \vdash k : \dot{p}[k]$.

Case $s = [\texttt{return}\ e]^{k_1}$. *Here* $[\![s]\!] = [\![e]\!]_{\mathrm{e}}; k_1'{:}\texttt{return}$. *By Lemma 8,* $\forall k \in [k_1, k_1'),\ \dot{loc} \vdash k : \dot{p}[k]$. *Also,* $\dot{loc} \vdash k_1' : \texttt{return}$ *is always true.*

Case $s = \texttt{if}\ [e_1 \bowtie e_2]^{k_1}\ \texttt{then}\ s_1\ \texttt{else}\ s_2$. *We have* $[\![s]\!] = [\![e_2]\!]_{\mathrm{e}}; [\![e_1]\!]_{\mathrm{e}}; k_1'{:}\texttt{cjmp} \bowtie k_4'; k_2'{:}[\![s_2]\!]; k_3'{:}\texttt{jmp}\ k_2; k_4'{:}[\![s_1]\!]_{\mathrm{e}}$. *By Lemma 8 and Induction Hypothesis we know that* $\forall k \in [k_1, k_1') \cup [k_2', k_3') \cup [k_4', k_2),\ \dot{loc} \vdash k : \dot{p}[k]$

Assuming our hypothesis,

$$\begin{aligned} & Loc \vdash \{[\![e_1 \bowtie e2]\!]^\sharp(Loc(k_1))\}\ s_1\ \{l_1^\sharp\} \\ \text{and}\quad & Loc \vdash \{[\![\neg(e_1 \bowtie e2)]\!]^\sharp(Loc(k_1))\}\ s_2\ \{l_2^\sharp\}\end{aligned}$$

where $l_1^\sharp \sqcup l_2^\sharp = l^\sharp$.

It can be proved that for every judgement of the form $Loc \vdash \{d_1^\#\}\ s\ \{d_2^\#\}$, $d_1^\# = Loc(init(s))$. *Therefore,* $\dot{loc}(k_4') = ([], Loc(k_4')) = ([], [\![e_1 \bowtie e_2]\!]^\sharp(Loc(k_1)))$ *and* $\dot{loc}(k_2') = ([], Loc(k_2')) = ([], [\![\neg(e_1 \bowtie e_2)]\!]^\sharp(Loc(k_1)))$. *Additionally,* $Loc(k_1') = ([e_1, e_2], Loc(k_1))$ *by Lemma 7. Thus,* $\dot{loc} \vdash k_1' : \texttt{cjmp} \bowtie k_4'$.

One can show that for all s s.t. $\dot{p} = \dot{p}_1 :: k : [\![s]\!] :: k' : \dot{p}_2$ *and* $k' \notin \mathsf{dom}(Loc)$ *and* $Loc \vdash \{Loc(init(s))\}\ s\ \{l^\sharp\}$, $\dot{loc}(k') \sqsubseteq ([], l^\sharp)$. *Since* $k_3' \notin \mathsf{dom}(Loc)$, $\dot{loc}(k_3') \sqsubseteq ([], l_2^\sharp)$. *Also,* $([], l_2^\sharp) \sqsubseteq ([], l^\sharp) \sqsubseteq \dot{loc}(k_2)$. *Then,* $\dot{loc}(k_3') \sqsubseteq \dot{loc}(k_2)$ *implies* $\dot{loc} \vdash k_3' : \texttt{jmp}\ k_2$, *which complete de proof for this case.*

Case $s = \texttt{while}\ [e_1 \bowtie e_2]^{k_1}\ \texttt{do}\ s'$. $[\![s]\!] = [\![e_2]\!]_{\mathrm{e}}; [\![e_1]\!]_{\mathrm{e}}; k_1' : \texttt{cjmp} \bowtie k_3'; k_2' : \texttt{jmp}\ k_2; k_3' : [\![s']\!]; k_4'{:}\texttt{jmp}\ k_1$. *By Lemma 8 and Induction Hypothesis we know that* $\forall k \in [k_1, k_1') \cup [k_3', k_4'),\ \dot{loc} \vdash k : \dot{p}[k]$.

Using Lemma 7,

$$\dot{loc}(k_1') = ([e_1, e_2], Loc(k_1)) \tag{1}$$

Since $k_2' \notin Loc$ *and* $\mathsf{pred}(k_2') = \{k_1'\}$,

$$\begin{aligned}\dot{loc}(k_2') &= \dot{g}_{k_1', k_2'}(\dot{loc}(k_1')) \\ &= ([], [\![\neg(e_1 \bowtie e_2)]\!]^\sharp(Loc(k_1)))\end{aligned} \tag{2}$$

Given that $Loc \vdash \{[\![e_1 \bowtie e_2]\!]^\sharp(Loc(k_1))\}\ s'\ \{l_{s'}^\sharp\}$ *holds assuming our hypothesis and* $k_3' = init(s')$,

$$\dot{loc}(k_3') = ([], [\![e_1 \bowtie e_2]\!]^\sharp(Loc(k_1))) \tag{3}$$

As we said, $\dot{loc}(k_4') \sqsubseteq l_{s'}^\sharp$, *because* $k_4' \notin Loc$. *This gives us*

$$\dot{loc}(k_4') \sqsubseteq \dot{loc}(k_2) \tag{4}$$

Also, by hypothesis,

$$[\![\neg(e_1 \bowtie e_2)]\!]^\sharp(Loc(k_1)) \sqsubseteq \dot{loc}(k_2) \tag{5}$$

Then, (1), (2) and (3) implies $\dot{loc} \vdash k_1' : \texttt{cjmp} \bowtie k_3'$. *(2) and (5) implies* $\dot{loc} \vdash k_2' : \texttt{jmp}\ k_2$, *and (4) implies* $\dot{loc} \vdash k_4' : \texttt{jmp}\ k_1$, *which complete the proof for this last case.*

Therefore, we showed that $\forall k_1[k_1, k_2),\ \dot{loc} \vdash k : \dot{p}[k]$.

4 Preservation of proof obligations

In this section we define two verification frameworks, respectively for source programs and for unstructured bytecode of previous sections. As a specification language we consider first order formulae, namely the domain of assertions $\mathcal{A}$. The validity of an assertions in a particular execution state $\eta \in State^s$ is standard. In particular, an assertion that contains the expression $a[e]$ is invalid in those execution states in which e is out of the bounds of the array a.

We consider as a program specification a tuple $(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$, where the assertion pre is a precondition, post and post are respectively normal and abnormal postconditions, and the partial function $\mathsf{annot} : \mathbf{Lab} \rightharpoonup \mathcal{A}$ maps program labels to internal points specifications. The special variable res may only occur in post, and pre only refers to variables from V. When specifying a bytecode program, assertions may refer to the special variable s representing the operand stack.

We say that a program satisfies the specification $(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$, if every execution starting in a state that satisfies pre only reaches normal final states satisfying post or abnormal states satisfying χ, and only reaches intermediate l-labeled points satisfying $\mathsf{annot}(l)$. Given a program specification $(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$, a verification condition generator (VCgen) framework provides a set of sufficient proof obligations that ensures that the program satisfies the specification.

The VCgen's defined in this section are hybrid in the sense that they take advantage of a previously computed analysis to reduce the size of proof obligations. We assume that the result of a relational analysis (Loc and $\dot{loc}$ respectively for source and bytecode programs) is given as input to the VCgen. For the abstract domain $\mathbb{D}$, we consider a relation $\models \subseteq \mathbb{D} \times \mathcal{A}$ such that for any guard b and any $d \in \mathbb{D}$, $d \models b$ indicates that the interpretation of the abstract element d ensures the validity of the condition b. For example, when accessing an array in the expression $a[x]$ we shall check that the value of the variable x is within the bounds of the array a. If we instantiate $\mathbb{D}$ with the domain of convex polyhedra, each element $d \in \mathbb{D}$ represents a set of linear constraints from which we can discover whether the condition $0 \leq x < |a|$ is satisfied.

A further improvement over standard VCgen's consists of reusing the result of the analysis to strengthen loop invariants. This technique helps reducing the size of annotations and the burden of interactive specification. To that end, we assume a concretization function $\gamma_{\mathsf{a}} : \mathbb{D} \to \mathcal{A}$ to interpret abstract elements $d \in \mathbb{D}$ as assertions.

VCgen for Source Programs

Consider a specification $(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$ for the source program P. Throughout this section, we assume that annot sufficiently annotates the program P, that is, for every subprogram $\texttt{while}\ [t]^l\ \texttt{do}\ c$ of P, we have that $l \in \mathsf{dom}(\mathsf{annot})$.

A VCgen for source programs is defined by the set of proof obligations:

$$\mathsf{PO} = \{\mathsf{pre} \Rightarrow \phi[^{\vec{V}}/_{\vec{V}^{old}}]\} \cup \theta$$

where $\langle \phi, \theta \rangle = \mathsf{WP}(P, \mathsf{post})$, $\phi[^{\vec{V}}/_{\vec{V}^{old}}]$ represents the result of substituting in ϕ any array or scalar variable x^{old} in $V_s^{old} + V_a^{old}$ by x, and the function WP is defined in Figure 5. In the figure, the assertion $\mathsf{inB}(e)$ stands for the condition that must satisfy an execution state to ensure that every array access in e is within bounds. For instance, if e does not contain array expressions $\mathsf{inB}(e)$ is defined as true and $\mathsf{inB}(a[e])$ as $0 \leq e < |a|$. We follow the simplifying assumption that expressions contain no more than one array access. For any array variable a and expressions e_1 and e_2, $\mathsf{upd}(a, e_1, e_2)$ is interpreted as the array a' such that $a'[e]$ is evaluated to e_2 if $e_1 = e$ and to $a[e]$ otherwise. To simplify the presentation of examples, proof obligations for $\texttt{while}$ statements are split into two assertions corresponding to the true and false branches.

The function WP considers the result of the analysis Loc to reduce the size of proof obligations. That is, if the abstract value $Loc(l)$ associated to the program point under consideration indicates that any array access in the statement is within bounds, the returned predicate is simplified by omitting the exceptional postcondition. Consider the program of Figure 6. If the analysis is able to compute at label k_1 an abstract value d such that $d \models 0 \leq i < |A|$, the WP function will return the assertion $\mathsf{upd}(A, i, A[0])[i + 1 - 1] = A[0]$, which together with the loop invariant at label k yields the proof obligation

$$\begin{aligned} &A[i-1] = 0 \land \boxed{0 \leq i \leq |A|} \Rightarrow \\ &\quad i < |A| \Rightarrow \mathsf{upd}(A, i, A[0])[i + 1 - 1] = A[0] \end{aligned}$$

where the boxed assertion $\boxed{0 \leq i \leq |A|}$ represents the interpretation of the result of the analysis at the loop entry point.

In contrast, if we do not take advantage of the result of the analysis we are due to prove the equivalent but bigger formula:

$$\begin{aligned} &A[i-1] = 0 \land \boxed{0 \leq i \leq |A|} \Rightarrow \\ &\quad i < |A| \Rightarrow \\ &\qquad (0 \leq i < |A| \Rightarrow \mathsf{upd}(A, i, A[0])[i + 1 - 1] = A[0] \\ &\qquad \land \neg(0 \leq i < |A|) \Rightarrow \mathsf{false}) \ . \end{aligned}$$

As can be seen from the definition of WP, proof obligations are of the form $\phi_1 \land \boxed{\gamma_{\mathsf{a}}(d)} \Rightarrow \phi_2$, whereas a standard VCgen outputs the stronger proof obligation $\phi_1 \Rightarrow \phi_2$. In consequence, one can provide the code with a weaker invariant ϕ_1 as long as the analyzer is able to eventually infer the missing information $\gamma_{\mathsf{a}}(d)$. For instance, for the simple program of Figure 6, a standard VCgen will return the invalid proof obligation

$$A[i-1] = A[0] \Rightarrow \neg(i < |A|) \Rightarrow A[|A| - 1] = A[0]$$

for the path that does not enter the loop. It is sufficient to provide a stronger invariant, i.e. in conjunction with the condition $i \leq |A|$, to prove the program correct. However, as an alternative to increasing the size of the program annotations, assuming the condition $i \leq |A|$ is inferred by the analysis, the hybrid VCgen generates the weaker (and valid) proof obligation

$$\begin{aligned} &A[i-1] = A[0] \land \boxed{0 \leq i \leq |A|} \Rightarrow \\ &\quad \neg(i < |A|) \Rightarrow A[|A| - 1] = A[0] \ . \end{aligned}$$

VCgen for Bytecode Programs

Let $(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$ be a specification for the bytecode program $\dot{p}$. As with the VCgen for source programs defined above, the precondition pre and the internal annotations $\mathsf{annot}(l)$ are strengthened with the result of the analysis. To that end, we interpret the result of the analysis with the aid of the concretization functions $\gamma_{\mathsf{a}} : \mathbb{D} \to \mathcal{A}$ and $\bar{\gamma_{\mathsf{a}}} : (Expr^\star \times \mathbb{D}) \to \mathcal{A}$. A VCgen for bytecode is defined by extracting the set of proof obligations:

$$\begin{aligned} \mathsf{po} = \{&\mathsf{pre} \Rightarrow \mathsf{wpi}(0)[^V/_{V^{\vec{old}}}]\} \cup \\ &\{\mathsf{annot}(l) \land \bar{\gamma_{\mathsf{a}}}(\dot{loc}(l)) \Rightarrow \mathsf{wpi}(l) \mid l \in \mathsf{dom}(\mathsf{annot})\} \end{aligned}$$

$$\frac{}{\mathsf{WP}(\mathtt{Skip}, \phi) = \langle \phi, \emptyset \rangle} \qquad \frac{}{\mathsf{WP}([\mathtt{return}\ e]^l, \phi) = \langle \mathsf{ckB}(e, \mathsf{post}[{}^{e}\!/_{\mathsf{res}}]), \emptyset \rangle} \qquad \frac{}{\mathsf{WP}([x{:=}e]^l, \phi) = \langle \mathsf{ckB}(e, \phi[{}^{e}\!/_{x}]), \emptyset \rangle}$$

$$\frac{}{\mathsf{WP}([a[e_1]{:=}e_2]^l, \phi) = \langle \mathsf{ckB}(e_2, \mathsf{ckB}(a[e_1], \phi[{}^{\mathsf{upd}(a,e_1,e_2)}\!/_{a}])), \emptyset \rangle} \qquad \frac{\mathsf{WP}(c_1, \phi) = \phi_1, \theta_1 \qquad \mathsf{WP}(c_2, \phi) = \phi_2, \theta_2}{\mathsf{WP}(\mathtt{if}\ [t]^l\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \phi) = \langle \mathsf{ckB}(t, t \Rightarrow \phi_1 \wedge \neg t \Rightarrow \phi_2), \theta_1 \cup \theta_2 \rangle}$$

$$\frac{\mathsf{WP}(c, \phi) = \langle \phi_1, \theta \rangle \qquad \Phi = (t \Rightarrow \phi_1) \wedge (\neg t \Rightarrow \phi)}{\mathsf{WP}(\mathtt{while}\ [t]^l\ \mathtt{do}\ c, \phi) = \langle \mathsf{annot}(l), \{\mathsf{annot}(l) \wedge \gamma_{\mathsf{a}}(Loc(l)) \Rightarrow \mathsf{ckB}(t, \Phi)\} \cup \theta_1 \rangle} \qquad \frac{\mathsf{WP}(c_1, \phi_2) = \langle \phi_1, \theta_1 \rangle \qquad \mathsf{WP}(c_2, \phi) = \langle \phi_2, \theta_2 \rangle}{\mathsf{WP}(c_1; c_2, \phi) = \langle \phi_1, \theta_1 \cup \theta_2 \rangle}$$

where the expression $\mathsf{ckB}(e, \varphi)$ stands for φ if $Loc(l) \models \mathsf{inB}(e)$ and the formula $\mathsf{inB}(e) \Rightarrow \varphi \wedge \neg\mathsf{inB}(e) \Rightarrow \chi$ otherwise.

Figure 5. Definition of WP function

```
//   pre : true,  χ : false
[i := 1]^{k0};
  //   A[i − 1] = A[0]
while [i < |A|]^k do {
  [A[i] := A[0]]^{k1};  [i := i + 1]^{k2}
}
 //   A[|A| − 1] = A[0]
...
```

Figure 6. Program example

$$
\begin{array}{ll}
\mathsf{wpi}(l) = \mathsf{wp}(l+1)[{}^{\mathbf{s}[0]\ \mathsf{op}\ \mathbf{s}[1]::\uparrow^2\mathbf{s}}\!/_{\mathbf{s}}] & \dot{p}[l] = \mathtt{prim}\ \mathtt{op} \\
\mathsf{wpi}(l) = \mathsf{wp}(l+1)[{}^{v::\mathbf{s}}\!/_{\mathbf{s}}] & \dot{p}[l] = \mathtt{push}\ v \\
\mathsf{wpi}(l) = \mathsf{wp}(l+1)[{}^{\mathbf{s}[0],\uparrow\mathbf{s}}\!/_{x,\mathbf{s}}] & \dot{p}[l] = \mathtt{store}\ x \\
\mathsf{wpi}(l) = \mathsf{wp}(l+1)[{}^{x::\mathbf{s}}\!/_{\mathbf{s}}] & \dot{p}[l] = \mathtt{load}\ x \\
\mathsf{wpi}(l) = \mathsf{ckB}(\mathsf{wp}(l+1)[{}^{\mathsf{upd}(a,\mathbf{s}[0],\mathbf{s}[1]),\uparrow^2\mathbf{s}}\!/_{a,\mathbf{s}}]) & \dot{p}[l] = \mathtt{astore}\ a \\
\mathsf{wpi}(l) = \mathsf{ckB}(\mathsf{wp}(l+1)[{}^{a[\mathbf{s}[0]]::\uparrow\mathbf{s}}\!/_{\mathbf{s}}]) & \dot{p}[l] = \mathtt{aload}\ a \\
\mathsf{wpi}(l) = \mathbf{s}[0] \bowtie \mathbf{s}[1] \Rightarrow \mathsf{wp}(l')[{}^{\uparrow^2\mathbf{s}}\!/_{\mathbf{s}}] & \dot{p}[l] = \mathtt{cjmp} \bowtie l' \\
\qquad \wedge\neg(\mathbf{s}[0] \bowtie \mathbf{s}[1]) \Rightarrow \mathsf{wp}(l+1)[{}^{\uparrow^2\mathbf{s}}\!/_{\mathbf{s}}] & \\
\mathsf{wpi}(l) = \mathsf{wp}(l') & \dot{p}[l] = \mathtt{jmp}\ l' \\
\mathsf{wpi}(l) = \mathsf{wp}(l+1) & \dot{p}[l] = \mathtt{nop} \\
\mathsf{wpi}(l) = \mathsf{post}[{}^{\mathbf{s}[0]}\!/_{\mathsf{res}}] & \dot{p}[l] = \mathtt{return}
\end{array}
$$

where $\mathsf{ckB}(\psi)$ stands for ψ if $\dot{loc}(l) \models \mathsf{inB}(x[\mathbf{s}[0]])$ and $\mathsf{inB}(x[\mathbf{s}[0]]) \Rightarrow \psi \wedge \neg\mathsf{inB}(x[\mathbf{s}[0]]) \Rightarrow \chi$ otherwise.

$$\mathsf{wp}(l) = \begin{cases} \mathsf{annot}(l) & \text{if } l \in \mathsf{dom}(\mathsf{annot}) \\ \mathsf{wpi}(l) & \text{otherwise} \end{cases}$$

Figure 7. VCgen for bytecode programs

where the predicate transformer wp is shown in Figure 7. If the program point is annotated, the function wp returns $\mathsf{annot}(l)$. Otherwise it applies the weakest precondition transformer wpi, defined in terms of the instruction at program point l, taking as parameters the annotations computed for the successor program points. The definition of wp and wpi is done by induction along the control flow paths of the program. We say that a program $\dot{p}$ is sufficiently annotated if the control flow graph of the program $\dot{p}$ does not contain unannotated loops. The induction principle following from the definition of sufficiently annotated programs is sufficient to ensure that wp and wpi are well defined For a list s, $s[0]$ and $s[1]$ represent the first and second element of s, and $\uparrow s$ denotes the result of removing the first element from s.

Preservation of Proof Obligations

Consider the specification $(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$ for source program P, and assume that annot is a sufficient annotation for P, i.e. every loop is annotated. Let $(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$ define as well the specification for the bytecode program $\dot{p}$. From previous results [5], we know that if annot is a sufficient annotation for P then it is also a sufficient annotation for the result of the compilation $\dot{p}$. Let Loc be a solution of the analysis for the source program P, and $\dot{loc}$ a solution of the analysis for the bytecode program $\dot{p}$, compiled from Loc as described in Section 3.

We assume that the concretization functions satisfy the property $\bar{\gamma_{\mathsf{a}}}([], d) = \gamma_{\mathsf{a}}(d)$, so that the interpretation of abstract analysis results in the source and bytecode sides coincides (recall that by definition $\dot{loc}(l) = ([\,], Loc(l))$ for every l in $\mathsf{dom}(Loc)$.) In addition, for any expression e and any $d \in \mathbb{D}$, if e does not contain array expressions, i.e. $\mathsf{inB}(e) = \mathsf{true}$, then $d \models \mathsf{inB}(e)$.

The following auxiliary result about the compilation of expressions is helpful to prove the preservation of proof obligations:

Lemma 10 *Assume that $\dot{p}$ is of the form $\dot{p}_1 :: l_1 : [\![e]\!]_{\mathrm{e}} :: l_2 : \dot{p}_2$. Then $\mathsf{wpi}(l_1)$ is equal to $\mathsf{wpi}(l_2)[{}^{e::\mathbf{s}}\!/_{\mathbf{s}}]$ if $\dot{loc}(l_1) \models \mathsf{inB}(e)$ and equal to $\mathsf{inB}(e) \Rightarrow \mathsf{wpi}(l_2)[{}^{e::\mathbf{s}}\!/_{\mathbf{s}}] \wedge \neg\mathsf{inB}(e) \Rightarrow \chi$ otherwise.*

PROOF. *The result holds under the assumption stated before that the expression e contains at most a variable ac-*

```
k0 : push 1          load i
     store i         prim +
 k : jmp k'          store i
k1 : push 0     k' : push |A|
     aload A         load i
     load i          cjmp < k1
     astore A  k'' : ...
k2 : push 1
```

Figure 8. Program example

cess. Otherwise, the syntactic equality of predicates does not hold, but it is straightforward to show a logical equivalence. The proof proceeds by structural induction on the expression e.

The coincidence of the sets of proof obligations PO and po is stated in the following lemma, from the fact that the bytecode program $\dot{p}$ is the result of compiling the source program P.

Proposition 11 *For every subprogram c of P, proof obligations corresponding to the subprogram c are equal to the proof obligations in $\dot{p}$ that correspond to the subsequence $[\![c]\!]$.*

PROOF. *Assume $\dot{p}$ is of the form $\dot{p}_1 :: l{:}[\![c]\!]_{\mathrm{e}} :: l'{:}\dot{p}_2$. Let $\langle \phi, \theta \rangle = \mathsf{WP}(c, \mathsf{wp}(l'))$ then one can prove by structural induction on c that $\mathsf{wp}(l) = \phi$ and that θ is equal to*

$$\{\mathsf{annot}(k) \wedge \bar{\gamma}_{\mathsf{a}}(\dot{loc}(k)) \Rightarrow \mathsf{wpi}(k) \mid k \in \mathsf{dom}(\mathsf{annot}) \cap \mathbf{Lab}_c\},$$

where $\mathbf{Lab}_c$ denotes the set of labels in the statement c.

Consider, the bytecode program of Figure 8 compiled from the example in Figure 6. One can see that the proof obligation at label k is

$$\begin{array}{l} A[i-1] = A[0] \wedge \boxed{0 \leq i \leq |A|} \Rightarrow \\ \quad (i < |A| \Rightarrow (A[i-1] = A[0])[^{\mathsf{upd}(A,i,A[0]),i+1}/_{A,i}]) \wedge \\ \quad (\neg(i < |A|) \Rightarrow A[|A|-1] = A[0]) \end{array}$$

which is equal to the proof obligation at label k for the source program of Figure 6.

5 From hybrid VCgen to VCgen

In this section we show a correspondence between the hybrid VCgen for bytecode of previous sections with a standard VCgen that does not take advantage of the result of the analysis. More precisely, interpreting the abstract result as logical formulae, we show an equivalence between the proof obligations of both VCgen's. Assuming that the relation $\models$ satisfies a correctness condition, soundness of the hybrid VCgen follows from soundness of the standard VCgen. In addition, soundness of the VCgen for source programs follows if the compiler is semantics preserving.

$$\begin{array}{ll} \hat{\mathsf{wpi}}(l) = \mathsf{ckB}(\hat{\mathsf{wp}}(l+1)[^{\mathsf{upd}(x,\mathbf{s}[0],\mathbf{s}[1]),\uparrow^2\mathbf{s}}/_{x,\mathbf{s}}]) & \dot{p}[l] = \mathtt{astore}\ x \\ \hat{\mathsf{wpi}}(l) = \mathsf{ckB}(\hat{\mathsf{wp}}(l+1)[^{x[\mathbf{s}[0]]::\uparrow\mathbf{s}}/_{\mathbf{s}}]) & \dot{p}[l] = \mathtt{aload}\ x \\ \hat{\mathsf{wpi}}(l) = \mathsf{wpi}(l) & \text{otherwise} \end{array}$$

where $\mathsf{ckB}(\psi)$ stands for

$$\mathsf{inB}(x[\mathbf{s}[0]]) \Rightarrow \psi \wedge \neg\mathsf{inB}(x[\mathbf{s}[0]]) \Rightarrow \chi$$

regardless of whether $\dot{loc}(l) \models \mathsf{inB}(x[\mathbf{s}[0]])$ is satisfied.

$$\hat{\mathsf{wp}}(l) = \begin{cases} \mathsf{annot}(l) & \text{if } l \in \mathsf{dom}(\mathsf{annot}) \\ \hat{\mathsf{wpi}}(l) & \text{otherwise} \end{cases}$$

$$\hat{\mathsf{po}} = \{\hat{\mathsf{annot}}(l) \Rightarrow \hat{\mathsf{wpi}}(l) \mid l \in \mathsf{dom}(\hat{\mathsf{annot}})\}$$

Figure 9. Non-hybrid bytecode VCgen

Given a specification $(\mathsf{pre}, \hat{\mathsf{annot}}, \mathsf{post}, \chi)$ for the bytecode program $\dot{p}$, a non-hybrid VCgen extracts the set of proof obligations:

$$\hat{\mathsf{po}} \cup \{\mathsf{pre} \Rightarrow \hat{\mathsf{wpi}}(0)[^{V}/_{V^{old}}]\}$$

where $\hat{\mathsf{wpi}}$ and $\hat{\mathsf{po}}$ are defined in Figure 9. To avoid ambiguity, in the sequel we make explicit some parameters needed in the definition of wpi, wp, $\hat{\mathsf{wpi}}$ and $\hat{\mathsf{wp}}$. We write for instance $\hat{\mathsf{wpi}}(l, \hat{\mathsf{annot}}, \mathsf{post}, \chi)$ instead of simply $\hat{\mathsf{wpi}}(l)$.

Let $\dot{loc}$ be a result of the analysis for the bytecode program $\dot{p}$. Consider the specifications $(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$ and $(\mathsf{pre}, \hat{\mathsf{annot}}, \mathsf{post}, \chi)$ for program $\dot{p}$, such that for all l in $\mathsf{dom}(\mathsf{annot})$, $\hat{\mathsf{annot}}(l)$ is defined as $\mathsf{annot}(l) \wedge \bar{\gamma}_{\mathsf{a}}(\dot{loc}(l))$. We say that the relation $\models \subseteq \mathbb{D} \times Guard$ is valid if for every abstract element $d \in \mathbb{D}$ and $b \in Guard$ we have that $d \models b$ implies the universal validity of $\gamma_{\mathsf{a}}(d) \Rightarrow b$. The result of the analysis $\dot{loc}$ is said *verifiable* if the set of proof obligations $\mathsf{po}(\mathsf{true}, \bar{\gamma}_{\mathsf{a}} \circ \dot{loc}, \mathsf{true}, \mathsf{true})$ are provable.

Lemma 12 *For every label l in the program $\dot{p}$:*

$$\mathsf{wpi}(l, \mathsf{annot}, \mathsf{post}, \chi) \wedge \bar{\gamma}_{\mathsf{a}}(\dot{loc}(l)) \Rightarrow \hat{\mathsf{wpi}}(l, \hat{\mathsf{annot}}, \mathsf{post}, \chi)$$

provided the relation $\models \subseteq \mathbb{D} \times Guard$ is valid, and the analysis $\dot{loc}$ is verifiable.

PROOF. *Following the induction principle induced by the definition of sufficiently annotated programs, for every label l we prove the goal above simultaneously with:*

$$\mathsf{wp}(l, \mathsf{annot}, \mathsf{post}, \chi) \wedge \bar{\gamma}_{\mathsf{a}}(\dot{loc}(l)) \Rightarrow \hat{\mathsf{wp}}(l, \hat{\mathsf{annot}}, \mathsf{post}, \chi)$$

The soundness of the VCgen po follows from the following result and the hypothesis that the standard VCgen $\hat{\mathsf{po}}$ is sound:

Proposition 13 *The provability of the set of proof obligations* $\hat{\mathsf{po}}(\mathsf{pre}, \hat{\mathsf{annot}}, \mathsf{post}, \chi)$ *follows from the provability of* $\mathsf{po}(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$.

Consider for instance the sequence of bytecode in Figure 8. Recall that annot is defined as $A[i-1] = A[0]$ and $A[|A|-1] = A[0]$ in k and k'' respectively. Let $\hat{\mathsf{annot}}$ be defined by strengthening annot with the result of the analysis, i.e. $\hat{\mathsf{annot}}(k) = \mathsf{annot}(k) \wedge 0 \leq i \leq |A|$ (we can let $\hat{\mathsf{annot}}(k'') = \mathsf{annot}(k'')$). Let Ψ be the weakest precondition computed by the non hybrid VCgen at label k_1:

$$\begin{array}{l} 0 \leq i < |A| \Rightarrow (\mathsf{upd}(A, i, A[0])[i+1-1] = A[0] \\ \qquad\qquad\qquad\qquad \wedge\, 0 \leq i+1 \leq |A|) \\ \wedge \neg(0 \leq i < |A|) \Rightarrow \mathsf{false} \end{array}$$

which, from Lemma 12 is implied by the hybrid wp and the result of the analysis, i.e. by

$$\mathsf{upd}(A, i, A[0])[i+1-1] = A[0] \wedge \boxed{0 \leq i < |A|}\ .$$

As stated in Proposition 13, if the proof obligations returned by the hybrid VCgen are valid, and assuming the analysis is verifiable, we have that

$$\begin{array}{l} A[i-1] = A[0] \wedge \boxed{0 \leq i \leq |A|} \Rightarrow i < |A| \Rightarrow \\ \quad \mathsf{upd}(A, i, A[0])[i+1-1] = A[0] \end{array}$$

and

$$0 \leq i \leq |A| \Rightarrow i < |A| \Rightarrow 0 \leq i < |A|$$

are provable. Then, it follows that the verification condition returned by the standard VCgen

$$A[i-1] = A[0] \wedge 0 \leq i \leq |A| \Rightarrow i < |A| \Rightarrow \Psi$$

is provable.

The above results establish that hybrid verification methods can be mapped to standard verification methods. In the context of Proof Carrying Code, one would like to establish the stronger result that hybrid certificates can be compiled into standard certificates. It is in fact possible to prove such a result, using the framework of [4]. However, the compilation of hybrid certificates into standard certificates requires using a certifying analyzer, that generates automatically logical proofs of correctness of the results of the analysis. While it is possible to avoid hybrid methods altogether, e.g. to rely on standard Proof Carrying Code architectures, hybrid methods are beneficial both for the code producer because they reduce significantly the number of proof obligations required to certify code, and for the code consumer, because they yield certificates that are more compact and more efficient to check. Translating certificates of proof obligations from a hybrid to a standard VCgen is interesting to complete a certificate translation process [2] in which original proof obligations are generated by a hybrid VCgen, but in which the targeted Trusted Computed Base has no support for hybrid certificates.

6 Related work

There are two approaches to hybrid methods. In the explicit approach, the user provides safety annotations that are used by the verification condition generator, and checked by a annotation checker. In contrast, the implicit approach advocates that the safety annotations are inferred by a static analyzer, and then used by verification condition generation. Both approaches are used (sometimes in conjunction) in deductive program verification, as well as in type-based analyses.

In addition, some authors have formalized and proved the soundness of hybrid verification methods. For example, Wildmoser, Chaieb and Nipkow [20] have used Isabelle/HOL to prove the soundness of hybrid methods for Java bytecode; they rely on interval analyses to detect arrays out of bounds, and implement a proof-producing version of the analysis that generates proofs that the results of the analysis is correct. More recently, Grégoire and Sacchini [9] have formalized in Coq a hybrid verification method for JVM programs. They focus on a null-pointer analysis; although the analyzer is not formalized in Coq, Hubert and co-workers [12] provides a good starting point for carrying such an implementation. The method of [9] supports bidirectional interaction between the analysis and verification condition generation, as the static analysis can extract useful information from the program annotations, at the same time as the results of the analysis are exploited by the verification condition generator to reduce the number of proof obligations (although we have not done so, it is possible and relatively easy to integrate such bidirectional interaction in our work).

It is folklore that deductive verification methods can be viewed as abstract interpretation [8, 7]. Logical abstract interpretation [10] explores the interaction between analysis and verification from the perspective of using theorem proving to improve the precision of abstract interpretations, and combinations of them.

Finally, the paper is closely related to previous works on proof-transforming compilation, and proof-producing program analyses. Saabas and Uustalu [18] provide an algorithm to transform proofs in Hoare logic into proofs in compositional proof systems for assembly programs. Moving to more realistic languages, Müller and co-authors define proof-transforming compilation for Java and Eiffel [1, 15, 17],

The aforementioned works, as the current paper, focus on non-optimizing compilation. Compiling proofs along program optimizations require using proof-producing analyses, that produce formal proofs of their correctness. Such analyses are also required to extend the results of Section 2 to certificates. Proof-producing analyses have been studied by several authors, including Wildmoser, Chaieb and Nipkow [20] in the context of verification condition generation for a bytecode language and Seo, Yang and Yi [19] in the context of a Hoare logic for a simple imperative language.

We refer to [3, 2] for a more detailed account of related work in this area.

7 Conclusion

Program verification environments increasingly rely on hybrid methods to prove correctness of software. Motivated by applications to Proof Carrying Code, we have shown the coincidence of hybrid verification methods at source and bytecode levels. Additionally, we have shown that hybrid verification methods can be "compiled" into methods based on verification condition generation, which ensure that hybrid methods are sound.

Our next goal is to extend our results to more realistic languages and analyses. Modern languages include features, e.g. exceptions, that can potentially yield very large control flow graphs, making hybrid certificates particularly necessary; we expect that our results on preservation of proof obligations for Java [3] will scale without difficulty to hybrid methods, making it possible to leverage the proof carrying code architecture of the Mobius project (see `mobius.inria.fr`) to hybrid methods.

Besides, it would be beneficial to allow hybrid methods to rely on more advanced static analyses that provide valuable information for proving properties of programs. We intend to focus on the recent analysis of [11], and to develop a hybrid verification method based on this analysis.

References

[1] F. Y. Bannwart and P. Müller. A program logic for bytecode. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 255–273. Elsevier, 2005.

[2] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In K. Yi, editor, *Static Analysis Symposium*, number 4134 in Lecture Notes in Computer Science, pages 301–317, Seoul, Korea, Aug. 2006. Springer-Verlag.

[3] G. Barthe, B. Grégoire, and M. Pavlova. Preservation of proof obligations for Java. In *International Joint Conference on Automated Reasoning*, Lecture Notes in Computer Science. Springer-Verlag, 2008. To appear.

[4] G. Barthe and C. Kunz. Certificate translation in abstract interpretation. In S. Drossopoulou, editor, *European Symposium on Programming*, volume 4960 of *Lecture Notes in Computer Science*, pages 368–382, Budapest, Hungary, Apr. 2008. Springer-Verlag.

[5] G. Barthe, T.Rezk, and A. Saabas. Proof obligations preserving compilation. In T. Dimitrakos, F. Martinelli, P. Ryan, and S. Schneider, editors, *Proceedings of FAST'05*, volume 3866 of *Lecture Notes in Computer Science*, pages 112–126. Springer-Verlag, 2005.

[6] F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Result certification for relational program analysis. Research Report 6333, IRISA, Sept. 2007.

[7] P. Cousot. Semantic foundations of program analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

[9] B. Grégoire and J. Sacchini. Combining a verification condition generator for a bytecode language with static analyses. In *Trustworthy Global Computing*, volume 4912 of *Lecture Notes in Computer Science*, pages 23–40. Springer-Verlag, 2007.

[10] S. Gulwani and A. Tiwari. Combining abstract interpreters. In M. Schwartzbach and T. Ball, editors, *PLDI*, pages 376–386. ACM, 2006.

[11] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *Programming Languages Design and Implementation*, ACM Sigplan Notices. ACM Press, June 2008. To appear.

[12] L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *International Conference on Formal Methods for Open Object-based Distributed Systems*, Lecture Notes in Computer Science. Springer-Verlag, 2008. To appear.

[13] F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In L. Hendren, editor, *CC*, volume 4959 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2008.

[14] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, December 2004. `http://www.di.ens.fr/~mine/these/these-color.pdf`.

[15] P. Müller and M. Nordio. Proof-transforming compilation of programs with abrupt termination. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems*, pages 39–46, New York, NY, USA, 2007. ACM.

[16] G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[17] M. Nordio, P. Müller, and B. Meyer. Proof-transforming compilation of eiffel programs. In R. Paige, editor, *TOOLS-EUROPE*, Lecture Notes in Business and Information Processing. Springer-Verlag, 2008.

[18] A. Saabas and T. Uustalu. A compositional natural semantics and Hoare logic for low-level languages. *Theoretical Computer Science*, 373(3):273–302, 2007.

[19] S. Seo, H. Yang, and K. Yi. Automatic Construction of Hoare Proofs from Abstract Interpretation Results. In A. Ohori, editor, *Asian Programming Languages and Systems Symposium*, volume 2895 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2003.

[20] M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation*, volume 141 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2005.

[21] M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping proof carrying code. In J.-J. Levy, E. W. Mayr, and J. C. Mitchell, editors, *Theoretical Computer Science*, pages 333–347. Kluwer Academic Publishing, Aug. 2004.

[22] H. Xi and S. Xia. Towards array bound check elimination in java tm virtual machine language. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 14. IBM Press, 1999.