Proof Obligations Preserving Compilation
Extended abstract

Gilles Barthe[1] and Tamara Rezk[1] and Ando Saabas[2]

[1] INRIA Sophia Antipolis, France, {Gilles.Barthe,Tamara.Rezk}@sophia.inria.fr
[2] Institute of Cybernetics, Tallinn University of Technology, Estonia, ando@cs.ioc.ee

Abstract. The objective of this work is to study the interaction between program verification and program compilation, and to show that the proof that a source program meets its specification can be reused to show that the corresponding compiled program meets the same specification. More concretely, we introduce a core imperative language, and a bytecode language for a stack-based abstract machine, and a non-optimizing compiler. Then we consider for both languages verification condition generators that operate on programs annotated with loop invariants and procedure specifications. In such a setting, we show that compilation preserves proof obligations, in the sense that the proof obligations generated for the source annotated program are the same that those generated for the compiled annotated program (using the same loop invariants and procedure specifications). Furthermore, we discuss the relevance of our results to Proof Carrying Code.

1 Introduction

1.1 Background and contribution

Interactive verification techniques provide a means to guarantee that programs are correct with respect to a formal specification, and are increasingly being supported by interactive verification environments that can be used to prove the correctness of safety critical or security sensitive software. For example, interactive verification environments are being used to certify the correctness of smartcard software, both for platforms and applications.

However interactive verification environments typically operate on source code programs whereas it is clearly desirable to obtain correctness guarantees for compiled programs, especially in the context of mobile code where code consumers may not have access to the source program. Therefore it seems natural to study the relation between interactive program verification and compilation.

In this paper, we focus on the interaction between compilation and verification condition generators (VC generators), which are used in many interactive verification environments to guarantee the correctness of source programs, and by several proof carrying code (PCC) architectures to check the correctness of compiled programs. Such VC generators operate on annotated programs that carry loop invariants and procedure specifications expressed as preconditions

and postconditions, and yield a set of proof obligations that must be discharged in order to establish the correctness of the program.

The main technical contribution of the paper is to show in a particular setting that compilation preserves proof obligations, in the sense that the set of proof obligations generated for an annotated source code program P is equal to the set of proof obligations generated for the corresponding annotated compiled program $\mathcal{C}(P)$ (we let $\mathcal{C}(.)$ be some compilation function), where annotations for $\mathcal{C}(P)$ are directly inherited from annotations in P. The immediate practical consequence of the equivalence is that the results of interactive source program verification (i.e. the proofs that are built interactively) can be reused for checking compiled programs, and hence that it is possible to bring the benefits of interactive program verification (at source code level) to the code consumer.

One important question is whether preservation of proof obligations can be derived from the semantical correctness of the compiler (in the sense that compiled programs have the same semantics as their source counterpart), and thus can be established independently of the exact definition of the compiler. The answer is negative: our results hold for a specific compiler that does not perform any optimization, and simple program optimizations invalidate preservation of proof obligations. We return to this point in Section 5.

Another question is the choice of the source and target languages: our source and target languages are loosely inspired from Java (e.g. we handle procedure calls differently), as our main application scenario deals with Java-enabled mobile phones.

1.2 Application scenarios

In this paragraph, we propose a scenario that exploits preservation of proof obligations to bring the benefits of interactive source program verification to the code consumer. The scenario may be viewed as an instance of Proof Carrying Code (PCC) [9], from which it inherits benefits including its robustness under the code/specification being modified while transiting from producer to consumer and/or under the assumption of a malicious producer, and issues including the difficulty of expressing security policies for applications, etc.

Scenario Consider a mobile phone operator that is keen of offering its customers a new service and has the possibility to do so by deploying a program $\mathcal{C}(P)$ originating from an untrusted software company. The operator is worried about the negative impact on its business if the code is malicious or simply erroneous, and wants to be given guarantees for $\mathcal{C}(P)$. For liability reasons, the operator does not want to see the source code, and for intellectual property reasons, the software company does not want to disclose its source code nor does it authorize the operator to modify the compiled code to insert additional checks.

The equivalence of proof obligations can be used to justify the following scenario: the operator provides a partial specification of the program, e.g. a precondition ϕ and a postcondition ψ for the program **main** procedure, and requires the company to show that the program meets this partial specification. There are two possibilities: either the software company verifies directly $\mathcal{C}(P)$, which

is definitely a possibility but not the most comfortable one, or thanks to preservation of proof obligations, it can also set to verify P, and benefits from the structured nature of modern programming languages in which we assume that P has been written. To verify P, the software company suitably annotates the program, leading to an annotated program P'. Then it generates the set of proof obligations for P' and discharges each proof obligation using some verification tool that produces proofs. The compiled annotated program is sent to the operator, together with the set of proof obligations and their proofs. Upon reception, the operator checks that the compiled annotated program provided by the software company matches the partial specification it formulated in the first place (here it has to check that the precondition and postcondition are unchanged), and then run its own verification condition generator, and checks with the help of the proofs provided by the software company that these proof obligations can be discharged.

Our application scenario is being considered for specific application domains, such as midlets, where operators currently dispose of a large number of GSM applications that they do not want to distribute to their customers due to a lack of confidence in the code. Of course, we do not underestimate that our approach is costly, both by the infrastructure it requires, and by the effort involved in using it (notably by involving program verification). However, the pay-off is that our approach enables to prove precisely properties of programs, i.e. in particular correct programs will not be rejected because of some automatic method which is overly conservative (i.e. rejects correct programs).

Our approach can also be used in other mobile code scenarios. Consider for example a repository of certified algorithms; the algorithms have been written in different programming languages, but they are stored in the directory as compiled programs, e.g. as CLR programs. Prior to adding a new algorithm, say an efficient algorithm to verify square root, the maintainer of the repository asks for a certificate that the algorithm indeed computes the square root. The correctness of the algorithm must be established through interactive verification, say by the implementer of the algorithm. The implementer has the choice to write a proof using a program logic for the language in which the algorithm was developed, or using an appropriate bytecode logic. Once again, it seems likely that the first approach would be favored, and therefore that proof obligation preserving compilation would be useful.

1.3 Related work

There are several lines of work concerned with establishing a relation between source programs and compiled programs. The most established line of work is undoubtedly compiler verification [7], which aims at showing that a compiler preserves the semantics of programs.

A more recent line of work is translation validation, proposed by A. Pnueli, M. Siegel and E. Singerman [11], and credible compilation, proposed by M. Rinard [13], aim at showing for each individual run of the compiler that the resulting target program implements correctly the source program, i.e. has the same

semantics. This is achieved by the automatic generation of invariants for each program point in the source code that must be satisfied at the corresponding program points in the source code. This technique does not allow to verify that a given specification is satisfied. Related work has also been done by X. Rival [14, 15], who uses abstract interpretation techniques to infer invariants at the source level and compile these invariants for the target level.

Our work is complementary to approaches to Proof-Carrying Code based on certifying compilation. In [10], Necula and Lee propose to focus on safety properties which can be proved automatically through an extended compiler that synthesizes annotations from the information it gathers about a program, and a checker that discharges proof obligations generated by the verification condition generator. Certifying compilers are very important for the scalability of PCC, but of course the requirement of producing certificates automatically reduces the scope of properties it can handle.

There are also some recent works on program specification and verification that involve at source level and target levels: the Spec# project [3] has defined an extension of C# with annotations and type support for nullity discrimination. Such annotated programs are then compiled (with their specifications) to extended .NET files, which can be run using the .NET platform. Specifications are checked at run-time or verified using a static checker (called Boogie). This work does not consider explicitly the relationship between source and compiled program verification (but the Spec# methodology implicitly assumes some relation between the two, otherwise letting users to specify source code and have Boogie verifying the corresponding compiled program would be meaningless). In a similar line of work, L. Burdy and M. Pavlova [5] have extended the proof environment Jack, which provides a verification condition generator for JML-annotated sequential Java programs, with a verification condition generator for extended Java class files that accommodate compiled JML annotations. However, they do not establish any formal relation between the two VC generators. Independently of this work, F. Bannwart and P. Müller [2] have considered proof compilation for a substantial fragment of sequential Java, and have discuss the translation of proofs from source code to bytecode. However, their work does not discuss automatic proof verification, neither establishes the correctness of proof compilation in their setting. None of these works discusses optimizations.

For completeness, we also mention the existence of many Hoare-like logics and weakest precondition calculi for low-level languages such as the JVM or .NET or assembly languages, see e.g. [1, 4, 8, 12, 16]; many of these works have been proposed in the context of PCC.

Contents The remaining of the paper is organized as follows. Section 2 introduces syntax and annotation language, and VC generators for the assembly and source languages. Preservation of proof obligations is addressed in Section 3. Section 4 illustrates how our approach can be applied to guarantee program correctness. Finally, we conclude in Section 5 with related work and directions for future research.

2 Language and Proof Systems Definitions

In the sequel, we let $\mathcal{V}$ be the set of values that are manipulated by programs (here $\mathcal{V} = \mathbb{Z}$), and assume given a set $\mathbb{A} \subseteq \mathcal{V} \times \mathcal{V} \to \mathcal{V}$ of arithmetic operations and a set $\mathbb{C} \subseteq \mathcal{V} \times \mathcal{V} \to \{0, 1\}$ of comparison operators. Furthermore, we assume given a set $\mathcal{M}$ of procedure names and a set $\mathcal{X}$ of program variables.

2.1 The assembly language

The assembly language SAL is a stack-based language with conditional and unconditional jumps, procedure calls and exceptions. It is powerful enough to compile the core imperative language described in Section 2.2.

$instr$ ::=	$\mathsf{prim}\ op$	primitive arithmetic operation
|	$\mathsf{push}\ n$	push n on stack
|	$\mathsf{load}\ x$	load value of x on stack
|	$\mathsf{store}\ x$	store top of stack in x
|	$\mathsf{if}\ cmp\ j$	conditional jump
|	$\mathsf{goto}\ j$	unconditional jump
|	$\mathsf{assert}\ \Phi$	assertion Φ
|	nop	no operation
|	$\mathsf{invoke}\ m$	procedure invocation
|	throw	throw an exception
|	return	end of program

where $op : \mathbb{A}$, and $cmp : \mathbb{C}$, and $x : \mathcal{X}$, and $n : \mathcal{V}$, $j : \mathbb{N}$, $m : \mathcal{M}$ and Φ is an assertion.

Fig. 1. Instruction set

SAL programs are sets of procedures with a distinguished procedure **main**. Each procedure m consists of a function from its set $\mathcal{P}_m$ of program points to instructions where the set of instructions is defined in Figure 1, and of a partial function $\mathsf{Handler}_m : \mathcal{P}_m \rightharpoonup \mathcal{P}_m$ which specifies for each program point its handler, if any. We write $\mathsf{Handler_m}(l) \uparrow$ if $\mathsf{Handler_m}(l)$ is undefined, and $\mathsf{Handler_m}(l) \downarrow$ otherwise. Program states are pairs consisting of a global register map, and a stack of frames, which correspond to the execution context of a procedure, and which consist of an operand stack, a program counter and the name of the procedure being executed. The operational semantics is standard (except for assert that does not change the state, i.e. it is like a no operation instruction). Note that upon a procedure invocation, a new frame is created with an empty operand stack and with the program pointer set to 1 (the initial instruction of a procedure). As to exception handling, the intuitive meaning is that if the execution at program point l in procedure m raises an exception and $\mathsf{Handler_m}(l) = t$, then control is transfered to t with an empty operand stack. If on the contrary $\mathsf{Handler_m}(l)$ is not defined, then the top frame is popped from the stack and the exception is transfered to the next frame.

In the sequel, we use the successor relation $\mapsto \subseteq \mathcal{P}_m \times \mathcal{P}_m$ which relates instruction to their successors. We assume that the successor of an assert instruction always belongs to the instructions of the procedure (we need this assumption for the sake of simplicity of definition of proof obligations further on).

Assertion language The assertion language is a standard-first order language that contains comparison between arithmetic expressions as base assertions, and is closed under conjunction and implication. One unusual feature of arithmetic expressions is that there are two special constants *st* and *top* for reasoning about the stack. The constant *top* represents the size of the stack in the current state, while the constant *st* can be thought of as an array used for an abstract representation of the operand stack. Thus we can refer to the elements of an array via expressions of the form $st(top - i)$. The set of arithmetic expressions is defined inductively as follows:

$$\begin{aligned} se \quad &::= top \mid se - 1 \\ aexpr &::= n \mid x \mid aexpr\ op\ aexpr \mid st(se) \end{aligned}$$

where $op : \mathbb{A}$.

The semantics of assertions is standard, except that assertions that refer to an undefined arithmetic expression, i.e. that contain a reference to an element outside the stack bounds, are considered to be false.

The definition of the VC generator relies extensively on substitution operators. Besides the rules for substituting variables, which are standard, we also have substitution rules for *top* and for the non-atomic expressions, namely $st(top)$ and $top - 1$.

Well-annotated programs Verification condition generators compute from partially annotated programs a fully annotated program, in which all program points of each procedure of the program have an explicit precondition attached to them. VCGens are partial functions that require programs to be sufficiently annotated in the first place. We call such programs well-annotated.

The property of being well-annotated can be formalized through an induction principle that is reminiscent of the accessible fragment of a binary relation: that is, given a procedure P_m, a predicate R on $\mathcal{P}_m$, we define **ext** R inductively by the clauses: i) if $i \in R$ then $i \in$ **ext** R; ii) if for all $j \in \mathcal{P}_m$ such that $i \mapsto j$, we have $j \in$ **ext** R, then $i \in$ **ext** R. Informally, **ext** R is the set of points from which all paths eventually arrive at R.

Definition 1 (Well-annotated program).

1. *Let* $\mathcal{P}_m^{\text{assert}}$ *and* $\mathcal{P}_m^{\text{return}}$ *be the set of program points* i *such that* $P_m[i]$ *is an* assert *instruction and* return *instruction respectively. Then* P_m *is a* well-annotated *procedure code iff* **ext** $(\mathcal{P}_m^{\text{assert}} \cup \mathcal{P}_m^{\text{return}}) = \mathcal{P}_m$.
2. *A program is well-annotated if it comes equipped with functions* $EPost : \mathcal{M} \to \mathsf{Assn}$ *and* $NPost : \mathcal{M} \to \mathsf{Assn}$ *that give the exceptional and normal postcondition of each procedure, and a function* $Pre : \mathcal{M} \to \mathsf{Assn}$ *which gives the precondition of a procedure, preconditions and postconditions assertions do not contain* st *or* top*, and each procedure is well-annotated.*

Given a well-annotated program, one can generate a precondition for each program point. Indeed, the assertion at any given program point can be computed from the assertions for all its successors; the latter may either be given initially (as part of the partially annotated program), or have been computed previously. Note that the definition of well-annotated program does not require programs to have any particular structure, e.g. unlike [12], they do not rule out overlapping loops.

Verification condition generator The verification condition generator for assembly programs, vcg_a, is defined as a function that takes as input a well-annotated program P and returns an assertion for each program point in P. This assertion represents the weakest liberal precondition that an initial state before the execution of the corresponding program point should satisfy for the method to terminate in a state satisfying its postcondition, that is $NPost(m)$ in case of normal termination or $EPost(m)$ in case the method terminates with an unhandled exception.

The computation of vcg_a proceeds in a modular way, i.e. procedure by procedure, and uses annotations from the procedure under consideration, as well as the preconditions and post-conditions of procedures called by m. Concretely for each program point, vcg_a is defined by a case analysis on the instruction $P_m[i]$.

Its definition is given in Figure 2. Notice that we use -2, that does not belong to the assertion language, instead of $-1-1$ as syntactic sugar in the definition. After calculating vcg_a of the procedure P_m (w.r.t. the annotations of P_m), we define the set of proof obligations PO_m as

$$\begin{aligned}
&PO_m(P_m, NPost(m), EPost(m)) \\
&= \{\Phi_i \Rightarrow \mathrm{vcg}_a(i+1) \mid i \in \mathcal{P}_m^{\mathrm{assert}}\} \\
&\cup \{NPost(m') \Rightarrow \mathrm{vcg}_a(i+1) \mid P_m[i] = \mathsf{invoke}\ m'\} \\
&\cup \{Pre(m) \Rightarrow \mathrm{vcg}_a(1)\} \cup \mathcal{M}_h(m) \cup \mathcal{M}_{\overline{h}}(m)
\end{aligned}$$

where

$$\begin{aligned}
\mathcal{M}_h(m) &= \{EPost(m') \Rightarrow \mathrm{vcg}_a(t) \mid \quad P_m[i] = \mathsf{invoke}\ m' \wedge \mathsf{Handler}_m(i) = t\} \\
\mathcal{M}_{\overline{h}}(m) &= \{EPost(m') \Rightarrow EPost(m) \mid \quad P_m[i] = \mathsf{invoke}\ m' \wedge \mathsf{Handler}_m(i) \uparrow\}
\end{aligned}$$

Proof obligations fall in one of the following categories:

- proof obligations that correspond to assertions in code;
- proof obligations triggered by procedure calls, where one has to verify that the postcondition of the invoked procedure implies the normal precondition computed for the program point that corresponds to the program point of the procedure invocation;
- the proof obligation that establishes that the normal precondition computed for the first program point follows from the procedure precondition;
- proof obligations triggered by procedure calls for the case that such calls raise an exception that is handled by the procedure m. Here one has to verify

that the exceptional postcondition of m implies the normal precondition computed for the handler of the program point where procedure invocation occurs;
– proof obligations triggered by procedure calls for the case that such calls raise an exception that is not handled by the procedure m. Here one has to verify that the exceptional postcondition of the procedure called implies the exceptional postcondition of m.

We define the set of proof obligation of a program as the union of the proof obligations of all its methods:

$$PO(P) = \bigcup_{m \in \mathcal{M}} PO_m(P_m, NPost(m), EPost(m))$$

One can prove that the verification condition generator is sound, in the sense that if the program P is called with registers set to values that verify the precondition of the procedure main, and P terminates normally, then the final state will verify the normal postcondition of main. Likewise, if P terminates abnormally, that is if an exception is thrown and there is no handler, then the final state will verify the exceptional postcondition of main. Soundness is proved first for one step of execution, and then extended to execution traces by induction on the length of the execution.

push n :	$\mathrm{vcg}_a(i) = \mathrm{vcg}_a(i+1)[n/st(top), top/top-1]$
prim op :	$\mathrm{vcg}_a(i) = \mathrm{vcg}_a(i+1)[st(top-1)\ op\ st(top)/st(top), top-1/top]$
load x :	$\mathrm{vcg}_a(i) = \mathrm{vcg}_a(i+1)[x/st(top), top/top-1]$
store x :	$\mathrm{vcg}_a(i) = \mathrm{vcg}_a(i+1)[top-1/top, st(top)/x]$
if $cmp\ j$:	$\mathrm{vcg}_a(i) = st(top-1)\ cmp\ st(top) \Rightarrow \mathrm{vcg}_a(i+j)[top-2/top]$ $\wedge \neg(st(top-1)\ cmp\ st(top)) \Rightarrow \mathrm{vcg}_a(i+1)[top-2/top]$
goto j :	$\mathrm{vcg}_a(i) = \mathrm{vcg}_a(i+j)$
assert Φ :	$\mathrm{vcg}_a(i) = \Phi,$
nop :	$\mathrm{vcg}_a(i) = \mathrm{vcg}_a(i+1)$
throw :	$\mathrm{vcg}_a(i) = EPost(m)$ if $\mathsf{Handler}_m(i) \uparrow$
throw :	$\mathrm{vcg}_a(i) = \mathrm{vcg}_a(t)$ if $\mathsf{Handler}_m(i) = t$
invoke m' :	$\mathrm{vcg}_a(i) = Pre(m')$
return :	$\mathrm{vcg}_a(i) = NPost(m)$

Fig. 2. Verification Condition Generator for SAL Procedures

2.2 Source language

The source language IMP is an imperative language with loops and conditionals, procedures and exceptions.

Definition 2. *1. The set* AExpr *of* arithmetic expressions*, and* AProg_{IMP} *of* commands *are given by the following syntaxes:*

$$\begin{array}{lll} expr & ::= & x \mid n \mid expr\ op\ expr \\ cmpexpr & ::= & expr\ cmp\ expr \\ comm & ::= & \mathsf{skip} \mid x := expr \mid comm; comm \mid \mathsf{while}\ \{I\}\ cmpexpr\ \mathsf{do}\ comm \mid \\ & & \mathsf{if}\ cmpexpr\ \mathsf{then}\ comm\ \mathsf{else}\ comm \mid \mathsf{try}\ comm\ \mathsf{catch}\ comm \mid \\ & & \mathsf{throw} \mid \mathsf{call}\ m \end{array}$$

where op and cmp are as in Section 2.1 and I is an assertion as defined in Section 2.1, but without the constants top and st.

2. We define a program P in IMP *as a set of procedures (we use m to name a procedure), and their corresponding bodies, which are a command from* AProg_{IMP} *(we use* P_m *to name a procedure code).*

We define a standard verification condition generator vcg, which takes as input a command and an assertion, and returns an assertion. The function is implicitly parameterized by assertions; concretely, we assume that all procedures are annotated with a precondition, a normal postcondition, and an exceptional postcondition.

$$\begin{array}{l} \mathsf{vcg}(\mathsf{skip}, Q, R) = Q \\ \mathsf{vcg}(x := e, Q, R) = Q[e/x] \\ \mathsf{vcg}(c_1; c_2, Q, R) = \mathsf{vcg}(c_1, \mathsf{vcg}(c_2, Q, R), R) \\ \mathsf{vcg}(\mathsf{while}\ \{I\}\ e\ \mathsf{do}\ c_1, Q, R) = I \\ \mathsf{vcg}(\mathsf{if}\ e_1\ cmp\ e_2\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, Q, R) = \\ \qquad (e_1\ cmp\ e_2) \Rightarrow \mathsf{vcg}(c_1, Q, R) \wedge \\ \qquad \neg(e_1\ cmp\ e_2) \Rightarrow \mathsf{vcg}(c_2, Q, R) \\ \mathsf{vcg}(\mathsf{try}\ c\ \mathsf{catch}\ c', Q, R) = \mathsf{vcg}(c, Q, \mathsf{vcg}(c', Q, R)) \\ \mathsf{vcg}(\mathsf{throw}, Q, R) = R \\ \mathsf{vcg}(\mathsf{call}\ m'), Q, R) = Pre(m') \end{array}$$

Fig. 3. Verification Condition Generator for *IMP* Procedures

We also define inductively the set PO_c of proof obligations for a command as follows:

$$
\begin{aligned}
&PO_c(\mathsf{skip}, Q, R) = \emptyset\\
&PO_c(x := e, Q, R) = \emptyset\\
&PO_c(c_1; c_2, Q, R) = PO_c(c_1, \mathsf{vcg}(c_2, Q, R), R) \cup PO_c(c_2, Q, R)\\
&PO_c(\mathsf{while}\ \{I\}\ e\ \mathsf{do}\ c_1, Q, R) =\\
&\qquad\qquad PO_c(c_1, Q, R) \cup \{I \Rightarrow (e \Rightarrow \mathsf{vcg}(c_1, I, R) \wedge \neg e \Rightarrow Q)\}\\
&PO_c(\mathsf{if}\ e_1\ cmp\ e_2\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, Q, R) =\\
&\qquad\qquad\qquad\qquad\qquad PO_c(c_1, Q, R) \cup PO_c(c_2, Q, R)\\
&PO_c(\mathsf{throw}, Q, R) = \emptyset\\
&PO_c(\mathsf{call}\ m', Q, R) = \{EPost(m') \Rightarrow R\} \cup \{NPost(m') \Rightarrow Q\}\\
&PO_c(\mathsf{try}\ c\ \mathsf{catch}\ c', Q, R) = PO_c(c, Q, \mathsf{vcg}(c', Q, R)) \cup PO_c(c', Q, R)
\end{aligned}
$$

As in SAL, proof obligations fall in one of the following categories:

- proof obligations that correspond to annotations in while loops;
- proof obligations triggered by procedure calls,
- proof obligations triggered by procedure calls for the case that such calls raise an exception that is handled by the procedure m.

We define for every procedure m with body c, the set of proof obligations $PO_m(c, NPost(m), EPost(m))$ as:

$$
\begin{gathered}
PO_c(c, NPost(m), EPost(m)) \cup\\
\{Pre(m) \Rightarrow \mathsf{vcg}(c, NPost(m), EPost(m))\}
\end{gathered}
$$

That is, the proof obligations of a method are those generated by the body of the methods plus the proof obligation that establishes that the precondition computed for the body of the methods follows from the procedure precondition.

Finally, the set of proof obligation for a program P is defined as the union of proof obligations for each method in P:

$$
PO(P) = \bigcup_{m \in \mathcal{M}} PO_m(P_m, NPost(m), EPost(m))
$$

3 Proof obligations preserving compilation

This section shows that the sets of proof obligations are preserved by a standard non-optimizing compiler. The consequence of this result is that having annotations and proofs of proof obligations for the source code, the same evidence can be used to prove automatically the correctness of its corresponding compiled program.

Definition 3. *The compilation function $\mathcal{C}_p : \mathsf{AProg}_{IMP} \rightarrow \mathsf{AProg}_{SAL}$ is defined in Figure 4, using an auxiliary function $\mathcal{C}_e : \mathsf{AExpr} \rightarrow \mathsf{AProg}_{SAL}$ (also defined in Figure 4), and another auxiliary function to define exception tables (defined in Figure 5).*

The compilation of exception tables defines handlers for program points of instructions enclose in the "try" part of try-catch commands as the first program point of the code enclose in their "catch" part.

Throughout this section, we use $\mathsf{vcg}(p, Q, R)$ to denote both verification condition generator at source code and bytecode. For the bytecode, $\mathsf{vcg}(p, Q, R)$ is $\mathrm{vcg}_a(i)$ where the normal and exceptional postconditions are Q and R resp. and where i is the first program point in p.

We begin with an auxiliary lemma about expressions. Given a list P of instructions, we use the notation $P[i...j]$ to denote the list of instructions from instruction at i up to j.

Lemma 1. *Let e be an arithmetic expression in* AExpr *which appears in program P, and suppose that we have that $\mathcal{C}_e(e) = \mathcal{C}_c(P)[i...j]$. Let Q be an assertion in* Assn *that includes an arithmetic expression $st(top)$. Assume $\mathrm{vcg}_a(j + 1) = Q$. Then $\mathsf{vcg}(i) = Q[e/st(top), top/top - 1]$.*

The following lemma states that if there exists a handler c' at source level for a command c, then any exception thrown in the compilation of c will have a handler that corresponds to the compilation of c'.

Lemma 2 (Handler Preserving Compiler). *Let command* $\mathsf{try}\ c\ \mathsf{catch}\ c'$ *s.t. it is the inner-most try-catch command enclosing c and let $P_m[i \ldots j] = \mathcal{C}_c(c)$ and $P_m[i' \ldots j'] = \mathcal{C}_c(c')$ be compilations of c and c'. Then for any $h \in \{i \ldots j\}$ that can throw an exception in P_m, $\mathsf{Handler}_m(h) = i'$ and if c is not enclosed in a try-catch command $\mathsf{Handler}_m(h)\uparrow$.*

The following proposition establishes that compilation "commutes" with verification condition generation.

Proposition 1. $\mathsf{vcg}(\mathcal{C}_c(c), Q, R) = \mathsf{vcg}(c, Q, R)$

The following theorem claims that the set of proof obligations of the original program are the same of the proof obligations generated after compilation.

Theorem 1 (Proof Obligation Preserving Compilation).

$$PO_m(\mathcal{C}_c(c), Q, R) = PO_m(c, Q, R)$$

4 Example

The purpose of this section is to illustrate how the application scenario from the introduction can be applied to guarantee that compiled applications meet high-level security properties, such as the absence of uncaught exceptions, as well as specific security properties, such as non-interference; the latter is encoded in our language using self-composition as described in [6]. Here the operator will determine which program variables (in a more realistic language one would focus on method parameters) of the program P to be certified are to be considered confidential. In turn, this choice sets the precondition and the postcondition,

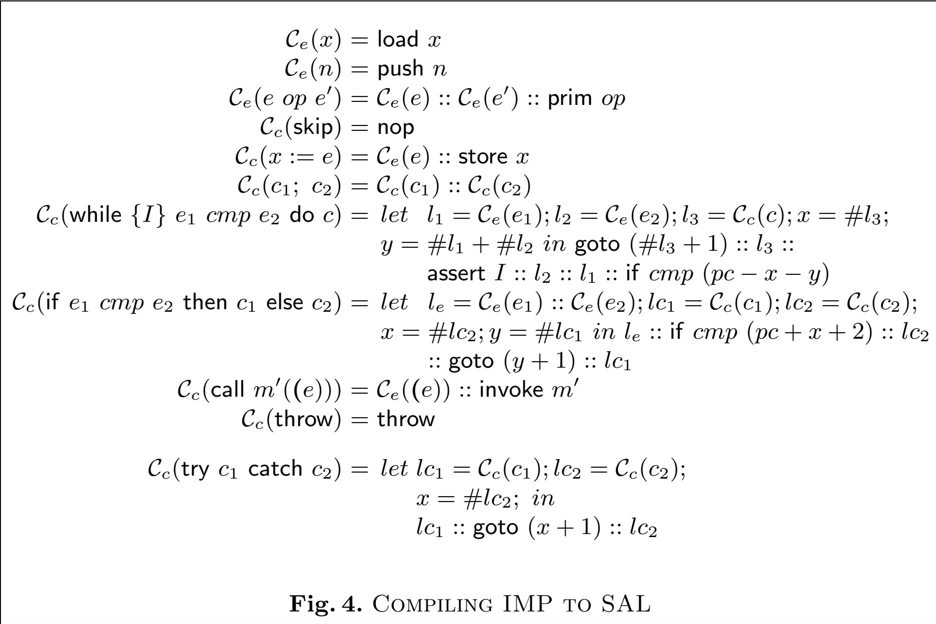

Fig. 4. Compiling IMP to SAL

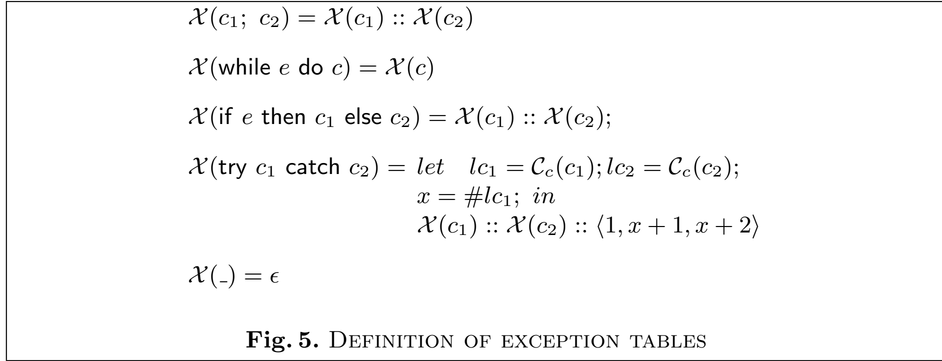

Fig. 5. Definition of exception tables

namely $\boldsymbol{x} = \boldsymbol{x'}$, where $\boldsymbol{x}$ are the low variables of P, and $\boldsymbol{x'}$ is a renaming of the low variables of P. Suppose in addition that the operator does not want the program to raise uncaught exceptions. Then the code producer must establish

$$\{\boldsymbol{x} = \boldsymbol{x'}\}P; P'\{\boldsymbol{x} = \boldsymbol{x'}, \texttt{false}\}$$

where P' is a renaming of P with fresh variables $\boldsymbol{x'}$ for low variables, and $\boldsymbol{y'}$ for high-variables. False as the exceptional postcondition denotes that an exception should not be thrown. To make matter precise, consider that P is the program constituted of two procedures main and aux that take one public parameter x

$\{x = x'\}$verif $==$ $x := y;$ call aux; $x' := y';$ call aux$'$
$\{x = x', false\}$

$\{true\}$
aux $==$ $x := 3;$ while $\{0 \leq x\}$ $x \geq 1$ do $y := y * x; x := x - 1$
$\{x = 0, false\}$

$\{x = 0\}$
aux$'$ $==$ $x' := 3;$ while $\{0 \leq x' \wedge x = 0\}$$x' \geq 1$ do $y' := y' * x'; x' := x' - 1$
$\{x = 0 \wedge x' = 0, false\}$

Proof Obligations for main:
$x = x' \Rightarrow true$
$false \Rightarrow false,\ x = 0 \Rightarrow x = 0$
$false \Rightarrow false\ x = 0 \wedge x' = 0 \Rightarrow x = x'$

Proof Obligations for aux:
$true \Rightarrow 0 \leq 3$
$0 \leq x \Rightarrow (x \geq 1 \Rightarrow 0 \leq x - 1 \wedge x < 1 \Rightarrow x = 0)$

Proof Obligations for aux':
$x = 0 \Rightarrow 0 \leq 3 \wedge x = 0$
$0 \leq x' \wedge x = 0 \Rightarrow (x' \geq 1 \Rightarrow 0 \leq x' - 1 \wedge\ x = 0 \wedge x < 1 \Rightarrow x = 0 \wedge x' = 0)$

Fig. 6. Example: Program with specification of Non-Interference

and one private parameter y, with main and aux defined as

$$\begin{aligned} \text{main} &== x := y;\ \text{call aux} \\ \text{aux} &== x := 3;\ \text{while } x \geq 1 \text{ do } y := y * x; x := x - 1 \end{aligned}$$

(Note that the program is non-interfering, since it always return with $x = 0$. However, the program is typically rejected by a type system.)

In order to prove the required properties, the software company must provide appropriate precondition and postcondition for the method aux, as well as appropriate loop invariants, and discharge the resulting proof obligations for the program **verif** defined as

$$\text{verif} == x := y;\ \text{call aux};\ x' := y';\ \text{call aux}'$$

The annotated program is given in Figure 6, where we use red to denote the specification provided by the operator, and green to denote the specification provided by the software company. We denote with blue the set of proof obligations. In Figure 7, we show the annotated compiled program.

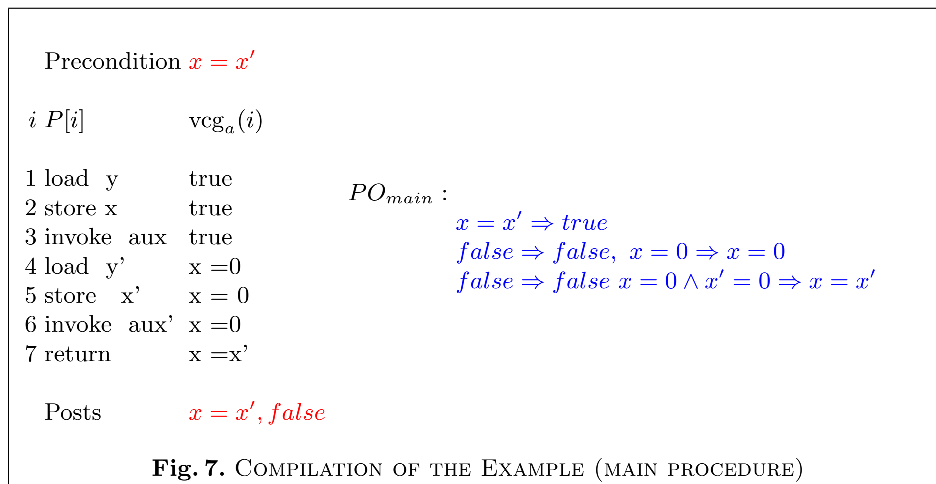

Fig. 7. Compilation of the Example (main procedure)

5 Concluding remarks

This paper shows, in a simple context, that it is possible to transfer evidence of program correctness from a source program to its compiled counterpart. Furthermore, we have shown on simple examples the possible uses of our results, and discussed some possible application domains. Although not reported here, we have also implemented a small prototype compiler and proof obligation generators to experiment our approach small examples.

We now intend to extend our results to (non-optimizing compilers for) programming languages such as Java and C#. Furthermore, we intend to extend our results to optimizing compilers. However, preservation of proof obligations may be destroyed by simple program optimizations. If we allow optimizations, it is necessary to focus on a more general property that involves an explicit representation of proofs.

Property of Proof Compilation For every annotated program P, a proof compiler is given by:

- a function f that gives for every proof obligation at the assembly level a corresponding proof obligation at the source level;
- a function that transforms, for every proof obligation ξ at the assembly level, proofs of $f(\xi)$ into proofs of ξ.

Proof compilation is a generalization of preservation of proof obligations and allows to bring the benefits of source code verification to code consumers. Like preservation of proof obligations, it is tied to a specific compiler; additionally, it is tied to a representation of proofs (although some degree of generality is possible here).

Preliminary investigations indicate that proof compilation is feasible for most common program optimizations. These results will be reported elsewhere.

Furthermore, we would like to explore further scenarios in which proof compilation could be used advantageously. We only mention two particularly interesting scenarios: the compilation of aspect-oriented programming, and the compilation of domain-specific languages DSLs into general purpose programs. The latter application domain seems particularly relevant since one could hope to exploit the features of DSLs to achieve easy proofs at the source code level.

Another item for future work is an evaluation of the usefulness of preservation of proof obligations and proof compilation on larger case studies. In the short term, the most promising application of our technique concerns high-level security properties that are often found in security policies for mobile applications; many of such properties are either recommended internally by the security experts to developers, or by external companies with strong security expertise (e.g. some certification authority) to solution providers (e.g. our telecom operator in the scenario of Subsection 1.2). In the longer term, it would be interesting to investigate the applicability of our method to the problem of performing dynamic updates of mobile devices infrastructures; indeed, such a scenario will probably require to establish that components behave according to their specification.

Acknowledgments We thank Benjamin Grégoire, César Kunz, Dante Zanarini and the anonymous referees for valuable comments on a preliminary version of this paper. This work was partially supported by the Estonian-French cooperation program Parrot, the EU projects APPSEM II, eVikings II, and INSPIRED, the Estonian Science Foundation grant no 5567, and the French ACI Sécurité SPOPS.

References

1. D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*, pages 1–27, 2005.
2. F. Bannwart and P. Müller. A program logic for bytecode. In F. Spoto, editor, *Proceedings of Bytecode'05*, Electronic Notes in Theoretical Computer Science. Elsevier Publishing, 2005.
3. M. Barnett, K.R.M. Leino, and W. Schulte. The spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceeings of CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*, pages 50–71. Springer-Verlag, 2005.
4. N. Benton. A typed logic for stacks and jumps. Manuscript, 2004.
5. L. Burdy and M. Pavlova. Java bytecode specification and verification. In *Proceedings of SAC'06*, 2006. To appear.
6. P. D'Argenio G. Barthe and T. Rezk. Secure information flow by self-composition. In R. Foccardi, editor, *Proceedings of CSFW'04*, pages 100–114. IEEE Press, 2004.
7. Joshua D. Guttman and Mitchell Wand. Special issue on VLISP. *Lisp and Symbolic Computation*, 8(1/2), March 1995.
8. N.A. Hamid and Z. Shao. Interfacing hoare logic and type systems for foundational proof-carrying code. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Proceedings of TPHOLs'04*, volume 3223 of *Lecture Notes in Computer Science*, pages 118–135. Springer-Verlag, 2004.

9. G.C. Necula. Proof-Carrying Code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.
10. G.C. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of PLDI'98*, pages 333–344, 1998.
11. A. Pnueli, E. Singerman, and M. Siegel. Translation validation. In B. Steffen, editor, *Proceedings of TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer-Verlag, 1998.
12. C.L. Quigley. A Programming Logic for Java Bytecode Programs. In D. Basin and B. Wolff, editors, *Proceedings of TPHOLs'03*, volume 2758, pages 41–54, 2003.
13. M. Rinard. Credible compilation. Manuscript, 1999.
14. X. Rival. Abstract Interpretation-Based Certification of Assembly Code. In L.D. Zuck, P.C. Attie, A.Cortesi, and S. Mukhopadhyay, editors, *Proceedings of VMCAI'03*, volume 2575 of *Lecture Notes in Computer Science*, pages 41–55, 2003.
15. X. Rival. Symbolic Transfer Functions-based Approaches to Certified Compilation. In *Proceedings of POPL'04*, pages 1–13. ACM Press, 2004.
16. M. Wildmoser and T. Nipkow. Asserting bytecode safety. In S. Sagiv, editor, *Proceedings of ESOP'05*, volume 1210 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.