Proof-Transforming Compilation of Programs with Abrupt Termination

Peter Müller
Microsoft Research, USA
mueller@microsoft.com

Martin Nordio
ETH Zurich, Switzerland
Martin.Nordio@inf.ethz.ch

ABSTRACT

The execution of untrusted bytecode programs can produce undesired behavior. A proof on the bytecode programs can be generated to ensure safe execution. Automatic techniques to generate proofs, such as certifying compilation, can only be used for a restricted set of properties such as type safety. Interactive verification of bytecode is difficult due to its unstructured control flow. Our approach is verify programs on the source level and then translate the proof to the bytecode level. This translation is non-trivial for programs with abrupt termination. We present proof transforming compilation from Java to Java Bytecode. This paper formalizes the proof transformation and present a soundness result.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Program Verification—*Correctness proofs*; D.3.4 [**Programming Languages**]: Processors—*Compilers*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Verification, Languages

Keywords

Trusted Components, Proof-Transforming Compiler, Proof-Carrying Code

1. INTRODUCTION

Proof-Carrying Code (PCC) [8, 9] has been developed with the goal of solving the problems produced by the unsafe execution of mobile code. In PCC, the code producer provides a *proof*, a certificate that the code does not violate the security properties of the code consumer. Before the code execution, the proof is checked by the code consumer. Only if the proof is correct, the code is executed.

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

The certificate proves the properties that are satisfied by the bytecode program. With the goal of generating certificates automatically, Necula [9] has developed certifying compilers. *Certifying compilers* are compilers that take a program as input and produce bytecode and its proof. Unfortunately, certifying compilers only work with a restricted set of provable properties such as type safety.

Another approach to solve the problem caused by mobile code is *interactive verification of bytecode*. This approach is applicable to a wide range of properties, but is difficult due to the bytecode's unstructured control flow. Contrary, source verification is simpler, but does not generate a certificate for the bytecode program.

The approach we propose here is the use of a Proof - Transforming Compiler (PTC). PTCs are similar to certifying compilers in PCC, but take a source proof as input and produce the bytecode proof. Figure 1 shows the architecture of this approach. The code producer develops a program. A proof of the source program is developed using a prover. Then, the PTC translates the proof producing the bytecode and its proof, which are sent to the code consumer. The proof checker verifies the proof. If the source proof or the translation were incorrect, the checker would reject the code.

An important property of Proof-Transforming Compilers is that they do not have to be trusted. If the compiler produces a wrong specification or a wrong proof for a component, the proof checker will reject the component. This approach has the strengths of both above mentioned approaches.

If the source and target languages are close, the proof translation is simple. However, if they are not close and the compilation function is complex, the translation can be hard. For example, proof-transformation from a subset of Java with `try-catch`, `try-finally` and `break` statements to Java Bytecode is not simple. Compiling these statements in isolation is simple, but the compilation of their interplay is not.

A `try-finally` statement is compiled using *code duplication*: the `finally` block is put after the `try` block. If `try-finally` statements are used inside of a `while` loop, the compilation of `break` statements first duplicates the `finally` blocks and then inserts a jump to the end of the loop. Furthermore, the generation of exception tables is also harder. The code duplicated before the `break` may have exception handlers different from those of the enclosing `try` block. Therefore, the exception table must be changed so that exceptions are caught by the appropriate handlers. In this paper, we present the first PTC that handles these complications.

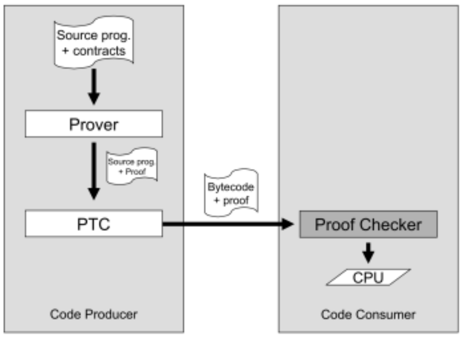

Figure 1: General architecture.

Outline. The source language and its Hoare-style logic are introduced in Section 2. We present the Bytecode language and its logic in Section 3. In Section 4, we define the proof transformation. Section 5 illustrates proof transformations by an example. Section 6 states a soundness theorem. Related work is discussed in Section 7. Section 8 summarizes and gives directions for future work.

2. SOURCE LANGUAGE AND LOGIC

The source language we consider is similar to a Java subset. Its definition is the following:

$$\begin{array}{lcl} exp & ::= & literal \mid var \mid exp\ op\ exp \\ stm & ::= & x = exp \mid stm; stm \mid \texttt{while}\ (exp)\ stm \\ & \mid & \texttt{break}\ ; \mid \texttt{if}\ (exp)\ stm\ \texttt{else}\ stm \\ & \mid & \texttt{try}\ stm\ \texttt{catch}\ (type\ var)\ stm \\ & \mid & \texttt{try}\ stm\ \texttt{finally}\ stm \mid \texttt{throw}\ exp\ ; \end{array}$$

To avoid `return` statements, we assume that the return value of every method is assigned to a special local variable named **result** (this is the only discordance with respect to Java). Moreover, we assume that the expressions are side-effect-free and cannot throw exceptions.

The subset of Java is small, but the combination of `while`, `break`s, `try-catch` and `try-finally` statements produces an interesting subset especially from the point of view of compilation. The code duplication used by the compiler for `try-finally` statements increases the complexity of the compilation and translation functions, specially the formalization and its soundness proof.

In our technical report [7], the source languages also includes object-oriented features such as cast, new, read and write field, and method invocation. In this paper, we only present the most interesting features.

2.1 Method and statement specifications

The logic is based on the programming logic introduced in [6, 12, 13]. We have modified it and proposed new rules for `while` including `break` and exceptions, `try-catch` and `try-finally`. In [13], a special variable χ is used to capture the status of the program such as normal or exceptional status. This variable is not necessary in the bytecode proof since non-linear control flow is implemented via jumps. To eliminate the χ variable, we use Hoare triples with two or three postconditions to encode the status of the program execution. This simplifies not only the translation but also the presentation.

Properties of methods are expressed by Hoare triples of the form $\{P\}\ \ T.m\ \ \{\ Q_n\ ,\ Q_e\ \}$, where P, Q_n, Q_e are first-order formulas and $T.m$ is a method m declared in class T. The third component of the triple consists of a normal postcondition (Q_n), and an exceptional postcondition (Q_e). We call such a triple *method specification*.

Properties of statements are specified by Hoare triples of the form $\{P\}\ \ S\ \ \{Q_n, Q_b, Q_e\}$, where P, Q_n, Q_b, Q_e are first-order formulas and S is a statement. For statements, we have a normal postcondition (Q_n), a postcondition after the execution of a `break` (Q_b), and an exceptional postcondition (Q_e).

The triple $\{P\}\ \ S\ \ \{Q_n, Q_b, Q_e\}$ defines the following refined partial correctness property: if S's execution starts in a state satisfying P, then (1) S terminates normally in a state where Q_n holds, or S executes a `break` statement and Q_b holds, or S throws an exception and Q_e holds, or (2) S aborts due to errors or actions that are beyond the semantics of the programming language, e.g., memory allocation problems, or (3) S runs forever.

2.2 Rules

Figure 2 shows the rules for compositional, `while`, `break`, `try-catch`, and `throw` statements. In the compositional statement, the statement s_1 is executed first. The statement s_2 is executed if and only if s_1 has terminated normally.

In the `while` rule, the execution of the statement s_1 can produce three results: either (1) s_1 terminates normally and I holds, or (2) s_1 executes a `break` statement and Q_b holds, or (3) s_1 throws an exception and R_e holds. The postcondition of the `while` statement expresses that either the loop terminates normally and $(I \wedge \neg e) \vee\ Q_b$ holds or throws an exception and R_e holds. The `break` postcondition is false, because after a `break` within the loop, execution continues normally after the loop.

The `break` rule sets the normal and exception postcondition to false and the `break` postcondition to P due to the execution of a `break` statement.

In the `try-catch` rule, the execution of the statement s_1 can produce three different results: (1) s_1 terminates normally and Q_n holds or terminates with a `break` and Q_b holds. In these cases, the statement s_2 is not executed and the postcondition of the `try-catch` is the postcondition of s_1; (2) s_1 throws an exception and the exception is not caught. The statement s_2 is not executed and the `try-catch` finishes in an exception mode. The postcondition is $Q_e'' \wedge \tau(excV) \not\preceq T$, where τ yields the runtime type of an object, $excV$ is a variable that stores the current exception, and $\preceq$ denotes subtyping; (3) s_1 throws an exception and the exception is caught. In the postcondition of s_1, $Q_e' \ \wedge\ \tau(excV) \preceq T$ specifies that the exception is caught. Finally, s_2 is executed producing the postcondition. Note that the postcondition is not only a normal postcondition: it also has to take into account that s_2 can throw an exception or can execute a `break`.

Similar to `break`, the `throw` rule modifies the postcondition P by updating the exception component of the state with the just evaluated reference.

To define the rule for `try-finally`, we have to treat a special case, illustrated through the example in Figure 3.

The exception thrown in the `try` block is never caught. However, the loop terminates normally due to the execution

compositional

$$\frac{\{P\}\ \ s_1\ \ \{Q_n, R_b, R_e\} \qquad \{Q_n\}\ \ s_2\ \ \{R_n, R_b, R_e\}}{\{P\}\ \ s_1; s_2\ \ \{R_n, R_b, R_e\}}$$

while

$$\frac{\{e\ \wedge\ I\}\ \ s_1\ \ \{I, Q_b, R_e\}}{\{I\}\ \ \texttt{while}\ (e)\ s_1\ \ \{((I \wedge \neg e) \vee\ Q_b), \mathit{false}, R_e\}}$$

break

$$\frac{}{\{P\}\ \ \texttt{break}\ \ \{\mathit{false}, P, \mathit{false}\}}$$

try-catch

$$\frac{\{P\}\ \ s_1\ \ \{Q_n, Q_b, Q\} \qquad \{Q'_e[e/excV]\}\ \ s_2\ \ \{Q_n, Q_b, R_e\}}{\{P\}\ \ \texttt{try}\ s_1\ \texttt{catch}\ (T\ e)\ s_2\ \ \{Q_n, Q_b, R\}}$$

where

$Q \equiv (\ (Q''_e\ \wedge\ \tau(excV) \npreceq T) \vee (Q'_e\ \wedge\ \tau(excV) \preceq T)\ )$

$R \equiv (R_e\ \vee\ (Q''_e\ \wedge\ \tau(excV) \npreceq T)\ )$

throw

$$\frac{}{\{P[e/excV]\}\ \ \texttt{throw}\ e\ \ \{\mathit{false}, \mathit{false}, P\}}$$

Figure 2: Rules for composition, `while`, `break`, `try-catch`, and `throw`.

```
void foo () {
   int b = 1;
   while (true) {
      try { throw new Exception(); }
      finally { b++; break; }
   }
   b++;
}
```

Figure 3: The exception raised in the `try` block is not handled, yet the method terminates normally.

of the `break` statement in the `finally` block. Thus, the value of b at the end of *foo* is 3.

If an exception occurs in a `try` block, it will be re-raised after the execution of the `finally` block. If both the `try` and the `finally` block throw an exception, the latter takes precedence. The following table summarizes the status of the program after the execution of the `try-finally`:

		`finally`		
		normal	**break**	**$\mathbf{exc_2}$**
`try`	**normal**	normal	break	exc_2
	break	break	break	exc_2
	$\mathbf{exc_1}$	exc_1	break	exc_2

We use the fresh variable $eTmp$ to store the exception occurred in s_1 because another exception might be raised and caught in s_2. In this case, we still need to have access to the first exception of s_1 because this exception is the result of that statement [13]. We use the fresh variable $\mathcal{X}Tmp$ to store the status of the program after the execution of s_1. The possible values of $\mathcal{X}Tmp$ are: *normal*, *break*, and *exc*. Depending on the status after the execution of s_2, we need to propagate an exception or change the status of the program to *break*. The rule is the following:

$$\frac{\{P\}\ \ s_1\ \ \{Q_n, Q_b, Q_e\} \qquad \{Q\}\ \ s_2\ \ \{R, R'_b, R'_e\}}{\{P\}\ \ \texttt{try}\ s_1\ \texttt{finally}\ s_2\ \ \{R'_n, R'_b, R'_e\}}$$

where

$$Q \equiv \left(\begin{array}{l}(Q_n \wedge \mathcal{X}Tmp = normal)\ \vee (Q_b \wedge \mathcal{X}Tmp = break) \vee \\ (\ \ Q_e[eTmp/excV] \wedge \mathcal{X}Tmp = exc \wedge eTmp = excV\ \ )\end{array}\right)$$

$$R \equiv \left(\begin{array}{l}(R'_n \wedge \mathcal{X}Tmp = normal)\ \vee\ (R'_b \wedge \mathcal{X}Tmp = break) \vee \\ (R'_e \wedge \mathcal{X}Tmp = exc)\end{array}\right)$$

Furthermore, the logic contains language-independent rules such as the rule of consequence. Due to space limitations, we do not present them here.

3. BYTECODE LANGUAGE AND LOGIC

The bytecode language consists of classes with fields and methods. Methods are implemented as method bodies consisting of a sequence of labeled bytecode instructions. Bytecode instructions operate on the operand stack, local variables (which also include parameters), and heap. The bytecode instructions used to compile the source language are: pushc v, pushv x, pop x, bin_{op}, goto l, brtrue l, and athrow. pushc v pushes constant v onto the stack. pushv x pushes the value of a variable x onto the stack. pop x pops the topmost element off the stack and assigns it to the local variable x. bin_{op} removes the two topmost values from the stack and pushes the result of applying bin_{op} to these values. goto l transfers control to the point l. brtrue l transfers control to the point l if the topmost element of the stack is true and unconditionally pops it. athrow takes the topmost value from the stack, assumed to be an exception, and throws it. To simplify the translation of source programs, we assume the bytecode language has a type boolean.

The bytecode logic is a Hoare-style program logic which allows one to formally verify that implementations satisfy interface specifications given as pre- and postconditions. We use the bytecode logic developed by Bannwart and Müller [1].

3.1 Method and Instruction Specifications

To make proof transformation feasible, it is essential that the source logic and the bytecode logic are similar in their structure. In particular, they treat methods in the same way, they contain the same language-independent rules, and triples have a similar meaning.

Analogously to the source logic, properties of methods are expressed by method specifications of the form form {P} $T.mp$ $\{Q_n,\ Q_e\}$. Properties of method bodies are expressed by Hoare triples of the form {P} *comp* {Q}, where P, Q are first-order formulas and *comp* is a method body. The triple {P} *comp* {Q} expresses the following refined partial correctness property: if the execution of *comp* starts in a state satisfying P, then (1) *comp* terminates in a state where Q holds, or (2) *comp* aborts due to errors or actions that are beyond the semantics of the programming language, or (3) *comp* runs forever.

The unstructured control flow of bytecode programs makes it difficult to handle instruction sequences, because jumps can transfer control into and from the middle of a sequence. Therefore, the logic treats each instruction individually: each individual instruction I_l in a method body p has a precondition E_l. An instruction with its precondition is called an *instruction specification*, written as $\{E_l\}\ l : I_l$.

The meaning of an instruction specification $\{E_l\}\ l : I_l$ cannot be defined in isolation. $\{E_l\}\ l : I_l$ expresses that if the precondition E_l holds when the program counter is at position l, the precondition $E_{l'}$ of I_l's successor instruction I'_l holds after normal termination of I_l.

3.2 Rules

All the rules for instructions, except for method calls, have the following form:

$$\frac{E_l \Rightarrow wp_p^1(I_l)}{\mathrm{A} \vdash \{E_l\}\ l : I_l}$$

where $wp_p^1(I_l)$ denotes the *local weakest precondition* of instruction I_l. Such a rule specifies that the precondition of I_l has to imply the weakest precondition of I_l with respect to all possible successor instructions of I_l. The definition of wp_p^1 is shown in Table 1.

Within an assertion, the current stack is referred to as s and its elements are denoted by non-negative integers: element 0 is the topmost element, etc. The interpretation $[E_l] : State \times Stack \rightarrow Value$ for s is

$$\begin{array}{l} [s(0)]\langle S, (\sigma, v)\rangle \ = v \ \text{ and} \\ [s(i+1)]\langle S, (\sigma, v)\rangle = [s(i)]\langle S, \sigma\rangle \end{array}$$

The functions *shift* and *unshift* define the substitutions that occur when values are pushed onto and popped from the stack, respectively:

$$\begin{array}{ll} shift(E) & = E[s(i+1)/s(i) \mid \forall i \in \mathbb{N}] \\ unshift & = shift^{-1} \end{array}$$

I_l	$wp_p^1(I_l)$
pushc v	$unshift(E_{l+1}[v/s(0)])$
pushv x	$unshift(E_{l+1}[x/s(0)])$
pop x	$(shift(E_{l+1}))[s(0)/x]$
bin_{op}	$(shift(E_{l+1}))[s(1)\ op\ s(0)/s(1)]$
goto l'	$E_{l'}$
brtrue l'	$(\neg s(0) \Rightarrow shift(E_{l+1})) \wedge (s(0) \Rightarrow shift(E_{l'}))$

Table 1: Definition of function wp_p^1.

4. PROOF TRANSLATION

Our proof-transforming compiler is based on two *transformation functions*, ∇_S and ∇_E, for statements and expressions, respectively. Both functions yield a sequence of bytecode instructions and their specification. The PTC takes a list of classes with their proofs and returns the bytecode classes with their proofs.

The function ∇_E generates a bytecode proof from a source expression and a precondition for its evaluation. The function ∇_S generates a bytecode proof and an exception table from a source proof. These functions are defined as a composition of the translations of its sub-trees. The signatures are the following:

$$\begin{array}{ll} \nabla_E & : Precondition \times Expression \times Postcondition \times \\ & \quad Label \rightarrow BytecodeProof \end{array}$$

$$\begin{array}{ll} \nabla_S & : ProofTree \times Label \times Label \times Label \times List[Finally] \times \\ & \quad ExcTable \rightarrow [BytecodeProof \times ExcTable] \end{array}$$

In ∇_E, the label is used as the starting label of the translation. *ProofTree* is a derivation in the source logic. In ∇_S, the three labels are: (1) l_{start} for the first label of the resulting bytecode; (2) l_{next} for the label after the resulting bytecode; this is for instance used in the translation of an `else` branch to determine where to jump at the end; (3) l_{break} for the jump target for `break` statements.

Type	Typical use
Precondition ∪ *Postcondition*	P, Q, R, U, V
ProofTree (for source language only)	$T_{S_1}, T_{S_2}, Tree_i$
ProofTree (for `finally` only)	T_{F_i}
List[*Finally*]	f
ExceptionTable	et_i
ExceptionTable (for `finally` only)	et_i'
BytecodeProof	B_{S_1}, B_{S_2}
InstrSpec	$b_{\mathsf{pushc}}, \ldots, b_{\mathsf{brtrue}}$
Label	$l_{start}, l_{next}, l_{break}, l_b, l_c, \ldots, l_g$

Table 2: Naming conventions.

The *BytecodeProof* type is defined as a list of *InstrSpec*, where *InstrSpec* is an instruction specification. The *Finally* type, used to translate `finally` statements, is defined as a tuple [*ProofTree*, *ExcTable*]. Furthermore, the ∇_S takes an exception table as parameter and produces an exception table. This is necessary because the translation of `break` statements can lead to a modification of the exception table as described above. (more details are presented in Section 4.3).

The *ExcTable* type is defined as follows:

$$\begin{array}{ll} ExcTable & := List[ExcTableEntry] \\ ExcTableEntry & := [Label, Label, Label, Type] \end{array}$$

In the *ExcTableEntry* type, the first label is the *starting label* of the exception line, the second denotes the *ending label*, and the third is the *target label*. An exception of type T_1 thrown at line l is caught by the exception entry $[l_{start}, l_{end}, l_{targ}, T_2]$ if and only if $l_{start} \leq l < l_{end}$ and $T_1 \preceq T_2$. Control is then transferred to l_{targ}.

In the following, we present the proof translation for compositional rule, `while`, `try-finally`, and `break`. Table 2 comprises the naming conventions we use in the rest of this paper.

4.1 Compositional Statement

Let T_{S_1} and T_{S_2} be the following proof trees:

$$T_{S_1} \equiv \frac{Tree_1}{\{P\}\ \ s_1\ \ \{Q_n, R_b, R_e\}}$$

$$T_{S_2} \equiv \frac{Tree_2}{\{Q_n\}\ \ s_2\ \ \{R_n, R_b, R_e\}}$$

$$T_{S1;S2} \equiv \frac{T_{S_1} \qquad T_{S_2}}{\{P\}\ \ s_1; s_2\ \ \{R_n, R_b, R_e\}}$$

In the translation of T_{S_1}, the label l_{next} is the start label of the translation of s_2, say l_b. The translation of T_{S_2} uses the exception table produced by the translation of T_{S_1}, et_1. The translation of $T_{S1;S2}$ yields the concatenation of the bytecode proofs for the sub-statements and the exception table produced by the translation of T_{S_2}.

Let $[B_{S_1}, et_1]$ and $[B_{S_2}, et_2]$ be of type [*BytecodeProof*, *ExcTable*]:

$$\begin{array}{l} [B_{S_1}, et_1] = \nabla_S\left(T_{S_1}, l_{start}, l_b, l_{break}, f, et\right) \\ [B_{S_2}, et_2] = \nabla_S\left(T_{S_2}, l_b, l_{next}, l_{break}, f, et_1\right) \end{array}$$

The translation is defined as follows:

$$\nabla_S \left(\ T_{S1;S2}, \ l_{start}, l_{next}, l_{break}, f, \ et \right) = [B_{S_1} + B_{S_2} \ , \ et_2]$$

The bytecode for s_1 establishes Q_n, which is the precondition of the first instruction of the bytecode for s_2. Therefore, the concatenation $B_{S_1} + B_{S_2}$ produces a sequence of valid instruction specifications. We will formalize soundness in Section 6.

4.2 While Statement

Let T_{S_1} and T_{while} be the following proof trees:

$$T_{S_1} \equiv \frac{Tree_1}{\{e \ \wedge \ I\} \ \ s_1 \ \ \{I, Q_b, R_e\}}$$

$$T_{while} \equiv \frac{T_{S_1}}{\{I\} \ \ \texttt{while} \ (e) \ s_1 \ \ \{(I \wedge \neg e) \vee \ Q_b, false, R_e\}}$$

In this translation, first the loop expression is evaluated at l_c. If it is true, control is transferred to l_b, the start label of the loop body. In the translation of T_{S_1}, the start label and next labels are l_b and l_c. The break label is the end of the loop (l_{next}). Furthermore, the finally list is set to $\emptyset$, because a `break` inside the loop jumps to the end of the loop without executing any `finally` blocks.

Let b_{goto} and b_{brtrue} be instruction specifications and B_{S_1} and B_e be bytecode proofs:

$$\begin{aligned} b_{\mathsf{goto}} &= \ \ \{I\} \quad l_a : \mathsf{goto} \ l_c \\ [B_{S_1}, et_1] &= \ \ \nabla_S \left(T_{S_1}, \ l_b, l_c, l_{next}, \ \emptyset, et \right) \\ B_e &= \ \ \nabla_E \ (\ I, \ e, \ (shift(I) \ \wedge \ s(0) = e) \ , _c \ ) \\ b_{\mathsf{brtrue}} &= \ \ \{shift(I) \ \wedge \ s(0) = e\} \quad l_d : \ \mathsf{brtrue} \ l_b \end{aligned}$$

The definition of the translation is the following:

$$\nabla_S \left(T_{while}, \ l_{start}, l_{next}, l_{break}, \ f, \ et \right) = [\ b_{\mathsf{goto}} + B_{S_1} + B_e + b_{\mathsf{brtrue}} \ , \ et_1 \ ]$$

The instruction b_{goto} establishes I, which is the precondition of the successor instruction (the first instruction of B_e). B_e establishes $shift(I) \ \wedge \ s(0) = e$ because the evaluation of the expression pushes the result on top of the stack. This postcondition implies the precondition of the successor instruction b_{brtrue}. b_{brtrue} establishes the preconditions of both possible successor instructions, namely $e \ \wedge \ I$ for the successor l_b (the first instruction of B_{S_1}), and $I \ \wedge \ \neg e$ for l_{next}. Finally, B_{S_1} establishes I, which implies the precondition of its successor B_e, I. Therefore, the produced bytecode proof is valid.

4.3 Try-Finally Statement

Sun's newer Java compilers translate `try-finally` statements using code duplication. Consider the following example:

```
while (i < 20) {
    try {
        try {
            try { ... break; ... }
            catch (Exception e) { i = 9; }
        }
        finally { throw new Exception(); }
    }
    catch (Exception e) { i = 99; }
}
```

The `finally` body is duplicated before the `break`. But the exception thrown in the `finally` bock must be caught by the outer `try-catch`. To achieve that, the compiler creates, in the following order, exception lines for the outer `try-catch`, for the `try-finally`, and for the inner `try-catch`. When the compiler reaches the `break`, it divides the exception entry of the inner `try-catch` and `try-finally` into two parts so that the exception is caught by the outer `try-finally`. To be able to divide the exception table the compiler needs to compare the exception entries. This is why our *Finally* type consists of a proof tree (for the duplicated code) and an exception table. Note that we have a list of *Finally* to handle nested `try-finally` statements.

Let T_{S_1}, T_{S_2} and $T_{try-finally}$ be the following proof trees:

$$T_{S_1} \equiv \frac{Tree_1}{\{P\} \ \ s_1 \ \ \{Q_n, Q_b, Q_e\}}$$

$$T_{S_2} \equiv \frac{Tree_2}{\{Q\} \ \ s_2 \ \ \{R, R'_b, R'_e\}}$$

$$T_{try-finally} \equiv \frac{T_{S_1} \qquad T_{S_2}}{\{P\} \ \ \texttt{try} \ s_1 \ \texttt{finally} \ s_2 \ \ \{R'_n, R'_b, R'_e\}}$$

where

$$Q \equiv \left(\begin{array}{l} (Q_n \wedge \mathcal{X}Tmp = normal) \ \vee (Q_b \wedge \mathcal{X}Tmp = break) \ \vee \\ (\ Q_e[eTmp/excV] \wedge \mathcal{X}Tmp = exc \wedge eTmp = excV \ ) \end{array} \right)$$

$$R \equiv \left(\begin{array}{l} (R'_n \wedge \mathcal{X}Tmp = normal) \ \vee \ (R'_b \wedge \mathcal{X}Tmp = break) \ \vee \\ (R'_e \wedge \mathcal{X}Tmp = exc) \end{array} \right)$$

In this translation, the bytecode for s_1 is followed by the bytecode for s_2. In the translation of T_{S_1}, the `finally` block is added to the finally-list f with T_{S_2}'s source proof tree and its associated exception table. The corresponding exception table is retrieved using the function $getExcLines : Label \times Label \times ExcTable \rightarrow ExcTable$. Given two labels and an exception table et, $getExcLines$ returns, per every exception type in et, the first et's exception entry (if any) for which the interval made by the starting and ending labels includes the two given labels. Furthermore, a new exception entry, for the `finally` block, is added to the exception table et. Then, the bytecode proof for the case when s_1 throws an exception is created. The exception table of this translation is produced by the predecessor translations.

Let et', et'' be the following exception tables:

$$\begin{aligned} et_1 &= et + [l_{start}, l_b, l_d, any] \\ et' &= getExcLines(l_a, l_b, et_1) \end{aligned}$$

Let b_{goto}, b_{pop}, b_{pushv}, and b_{athrow} be instructions specifications and B_{S_1}, B_{S_2}, and B'_{S_2} be bytecode proofs:

$$\begin{aligned} [B_{S_1}, et_2] &= \ \nabla_S \ \left(T_{S_1}, \ l_{start}, \ l_b, \ l_{break}, \ [T_{S_2} \ , et'] + f, \ et_1 \right) \\ [B_{S_2}, et_3] &= \ \nabla_S \ \left(T_{S_2}, \ l_b, \ l_c, \ l_{break}, \ f, \ et_2 \right) \\ b_{\mathsf{goto}} &= \ \{Q'_n\} \qquad\qquad l_c : \mathsf{goto} \ l_{next} \\ b_{\mathsf{pop}} &= \ \left\{ \begin{array}{l} shift(Q_e) \ \wedge \\ excV \neq null \\ \wedge \ s(0) = excV \end{array} \right\} \quad l_d : \mathsf{pop} \ eTmp \\ [B_{S'_2}, et_4] &= \ \nabla_S \ \left(T_{S_2}, l_e, l_f, l_{break}, \ f, \ et_3 \right) \\ b_{\mathsf{pushv}} &= \ \left\{ \ Q'_n \ \vee \ Q'_b \ \vee \ Q'_e \ \right\} \quad l_f : \mathsf{pushv} \ eTmp \\ b_{\mathsf{athrow}} &= \ \left\{ \begin{array}{l} (Q'_n \ \vee \ Q'_b \ \vee \ Q'_e) \\ \wedge \ s(0) = eTmp \end{array} \right\} \quad l_g : \mathsf{athrow} \end{aligned}$$

The translation is defined as follows:

$$\nabla_S\ (\ T_{try-finally},\ l_{start}, l_{next}, l_{break},\ f,\ et) = \\ [\ B_{S_1} + B_{S_2} + b_{\mathsf{goto}} + b_{\mathsf{pop}} + B_{S_2'} + b_{\mathsf{pushv}} + b_{\mathsf{athrow}}\ ,\ et_4\ ]$$

It is easy to see that the instruction specifications b_{goto}, b_{pop}, b_{pushv}, and b_{athrow} are valid (by applying the definition of the weakest precondition). However, the argument for the translation of T_{S_1} and T_{S_2} is more complex. Basically, the result is a valid proof because the proof tree inserted in f for the translation of T_{S_1} is a valid proof and the postcondition of each finally block implies the precondition of the next one. Furthermore, for normal execution, the postcondition of B_{S_1} (Q_n) implies the precondition of B_{S_2} (Q).

4.4 Break Statement

To specify the rules for `break`, we use the following recursive function: *divide*: *ExcTable* × *ExcTableEntry* × *Label* × *Label* → *ExcTable*. Its definition assumes that the exception entry is in the given exception table and the two given labels are in the interval made by the exception entry's starting and ending labels. Given an exception entry y and two labels l_s and l_e, *divide* compares every exception entry, say x, of the given exception table to y. If the interval defined by x's starting and ending labels is included in the interval defined by y's starting and ending labels, then x must be divided to have the appropriate behavior of the exceptions. Thus, the first and the last interval of the three intervals defined by x's starting and ending labels, l_s, and l_e are returned, and the procedure is continued for the next exception entry. If x and y are equal, then recursion stops as *divide* reached the expected entry. The formal definition of *divide* is the following:

$$\begin{array}{l} \mathit{divide} : \mathit{ExcTable} \times \mathit{ExcTableEntry}\ \times \mathit{Label} \times \\ \quad \mathit{Label} \rightarrow \mathit{ExcTable} \\ \mathit{divide} : ([\,], e', l_s, l_e) = [\ e'\ ] \\ \mathit{divide} : (e : et, e', l_s, l_e) = \\ \quad [\ l_{start},\ l_s,\ l_{targ},\ T_1\ ] + [\ l_e,\ l_{end},\ l_{targ},\ T_1\ ] + \\ \qquad \mathit{divide}(et, e', l_s, l_e)\ \ \textbf{if}\ e \subseteq e' \ \wedge\ e \neq e' \\ \quad \mid e : et\ \ \textbf{if}\ \ e = e' \\ \quad \mid e : \mathit{divide}(et, e', l_s, l_e)\ \ \textbf{otherwise} \end{array}$$

where

$$e \equiv [l_{start}, l_{end}, l_{targ}, T_1] \text{ and } e' \equiv [l'_{start}, l'_{end}, l'_{targ}, T_2]$$

$$\begin{array}{l} \subseteq : \mathit{ExcTableEntry} \times \mathit{ExcTableEntry} \rightarrow \mathit{Boolean} \\ \subseteq : ([l_{start}, l_{end}, l_{targ}, T_1], [l'_{start}, l'_{end}, l'_{targ}, T_2]) = \\ \quad \mathit{true}\ \ \ \textbf{if}\ (l'_{st} \leq l_{st})\ \wedge\ (l'_{end} \geq l_{end}) \\ \quad \mid\ \ \mathit{false}\ \textbf{otherwise} \end{array}$$

When a `break` statement is encountered, the proof tree of every `finally` block the `break` has to execute upon exiting the loop is translated. Then, control is transferred to the end of the loop using the label l_{break}. Let $f_i = [T_{F_i}, et'_i]$ denote the i-th element of the list f, where

$$T_{F_i} = \frac{\mathit{Tree}_i}{\{U^i\}\ \ s_i\ \ \{V^i\}}$$

and U^i and V^i have the following form, which corresponds to the Hoare rule for `try-finally` (see Section 2):

$$U^i\ \equiv \left\{ \begin{array}{l} (U^i_n\ \wedge\ \mathcal{X}Tmp = normal) \vee \\ (U^i_b\ \wedge\ \mathcal{X}Tmp = break) \vee \\ \left(\begin{array}{l} U^i_e[eTmp/excV]\ \wedge\ \mathcal{X}Tmp = exc\ \wedge \\ eTmp = excV \end{array} \right) \end{array} \right\}$$

$$V^i \equiv \left\{ \left(\begin{array}{l} (V'^i_n\ \wedge\ \mathcal{X}Tmp = normal) \vee \\ (V'^i_b\ \wedge\ \mathcal{X}Tmp = break) \vee \\ (V'^i_e\ \wedge\ \mathcal{X}Tmp = exc) \end{array} \right),\ V^i_b,\ V^i_e \right\}$$

Let B_{Fi} be a *BytecodeProof* for T_{Fi} such that

$$[B_{F_i}, et_{i+1}] = \ \nabla_S \left(\begin{array}{l} T_{F_i},\ l_{start+i}, l_{start+i+1}, l_{br}, f_{i+1}...f_k, \\ \mathit{divide}(et_i, et'_i[0], l_{start+i}, l_{start+i+1}) \end{array} \right)$$

$$b_{\mathsf{goto}} = \quad \{B^k_b\} \quad l_{start+k+1} : \mathsf{goto}\ l_{br}$$

The definition of the translation is the following:

$$\nabla_S \left(\frac{}{\{P\}\ \ \mathtt{break}\ \ \{\mathit{false}, P, \mathit{false}\}}, l_{start}, l_{next}, l_{br}, f, et_0 \right) \\ = [\ B_{F_1} + B_{F_2} + ...B_{F_k} + b_{\mathsf{goto}}, et_k]$$

To argue that the bytecode proof is valid, we have to show that the postcondition of B_{F_i} implies the precondition of $B_{F_{i+1}}$ and that the translation of every block is valid. This is the case because the source rule requires the break postcondition of s_1 to imply the normal precondition of s_2.

The exception table has two important properties that hold during the translation. The first one (Lemma 1) states that the exception entries, whose starting labels appear after the last label generated by the translation, are kept unchanged. The second one (Lemma 2) expresses that the exception entry is not changed by the division. These properties are used to prove soundness of the translation.

LEMMA 1. *If* $\nabla_S(\{P_n\}\ s\ \{Q\},\ l_a,\ l_{b+1},\ l_{break},\ f,\ et) = [(I_{l_a}...I_{l_b}), et']$ *and* $l_{start} \leq l_a < l_b \leq l_{end}$ *then for every* $l_s, l_e \in$ *Label such that* $l_b < l_s < l_e \leq l_{end}$ *and for every* $T \in$ *Type such that* $T \preceq$ *Throwable* $\vee\, T \equiv$ *any, the following holds:* $et[l_{start}, l_{end}, T] = et'[l_s, l_e, T]$.

LEMMA 2. *Let* $r \in$ *ExcTableEntry and* $et' \in$ *ExcTable be such that* $r \in et'$. *If* $et \in$ *ExcTable and* $l_s, l_e \in$ *Label are such that* $et = \mathit{divide}(et', r, l_s, l_e)$, *then* $et[l_s, l_e, T] = r[2]$

5. EXAMPLE

Figure 4 exemplifies the translation. The source proof of the example in Figure 3 is presented on the left-hand side and the corresponding bytecode proof on the right. An exception is thrown in the `try` block with precondition $b = 1$. The `finally` block increases b and then executes a `break` changing the status of the program to break mode (the postcondition is $b = 2$). In the bytecode proof, the body of the loop is between lines 09 and 18. Lines 17 and 18 re-throw the exception produced at line 10. Due to the execution of a `break` instruction, the code from 17 to 18 is not reachable (this is the reason for their *false* precondition). The `break` translation yields at line 16 a goto instruction whose target is the end of the loop, *i.e.*, line 23.

```
void foo () {
    { true }
    int b = 1;
    { b = 1, false, false }
    while (true) {
        { b = 1, false, false }
        try {
            { b = 1, false, false }
            throw new Exception();
            { false, false, b = 1 }
        }
        finally {
            { b = 1 ∧Xtmp = exc }
            b = b+1;
            { b = 2 ∧Xtmp = exc, false, false }
            break;
            { false, b = 2 ∧Xtmp = exc, false }
        }
        { false, b = 2, false }
    }
    { b = 2, false, false }
    b = b+1;
    { b = 3, false, false }
}
```

Precondition	Instruction
$\{$ **true** $\}$	00 : **pushc** 1
$\{s(0) = 1\}$	01 : **pop** b
$\{b = 1\}$	02 : **goto** 20
$\{b = 1\}$	09 : **newobj** *Exception*
$\{b = 1\}$	10 : **athrow**
$\{b = 1 \wedge excV \neq null \wedge s(0) = excV\}$	11 : **pop** $eTmp$
$\{b = 1 \wedge eTmp = excV\}$	12 : **pushc** 1
$\{b = 1 \wedge s(0) = 1\}$	13 : **pushv** b
$\{b = 1 \wedge s(1) = 1 \wedge s(0) = b\}$	14 : **binop**$_+$
$\{b = 1 \wedge s(0) = b + 1\}$	15 : **pop** b
$\{b = 2\}$	16 : **goto** 23
$\{$ **false** $\}$	17 : **pushv** $eTmp$
$\{$ **false** $\}$	18 : **athrow**
$\{b = 1\}$	20 : **pushc true**
$\{b = 1 \wedge s(0) =$ **true** $\}$	21 : **brtrue** 04
$\{b = 2\}$	23 : **pushc** 1
$\{b = 2 \wedge s(0) = 1\}$	24 : **pushv** b
$\{b = 2 \wedge s(1) = 1 \wedge s(0) = b\}$	25 : **binop**$_+$
$\{b = 2 \wedge s(0) = 1 + b\}$	26 : **pop** b

Exception Table

From	*to*	*target*	*type*
0	7	10	*any*

Figure 4: Example of source and bytecode proofs generated by the PTC.

6. SOUNDNESS THEOREM

In a PCC environment, a soundness proof is required only for the trusted components. PTCs are not part of the trusted code base: If the PTC generates an invalid proof, the proof checker would reject it. But from the point of view of the code producer, we would like to have a compiler that always generates valid proofs. Otherwise, it would be useless.

We prove the soundness of the translations, *i.e.*, the translation produces valid bytecode proofs. It is, however, not enough to prove that the translation produces a valid proof, because the compiler could generate bytecode proofs where every precondition is false. The theorem states that if (1) we have a valid source proof for the statement s_1, and (2) we have a proof translation from the source proof that produces the instructions $I_{l_{start}} ... I_{l_{end}}$, their respective preconditions $E_{l_{start}} ... E_{l_{end}}$, and the exception table et, and (3) the exceptional postcondition in the source logic implies the precondition at the target label stored in the exception table for all types T such that $T \preceq Throwable \vee T \equiv any$ but considering the value stored in the stack of the bytecode, and (4) the normal postcondition in the source logic implies the next precondition of the last generated instruction (if the last generated instruction is the last instruction of the method, we use the normal postcondition in the source logic), (5) the break postcondition implies *finallyProperties*. Basically, the *finallyProperties* express that for every triple stored in f, the triple holds and the break postcondition of the triple implies the break precondition of the next triple. And the exceptional postcondition implies the precondition at the target label stored in the exception table et_i but considering the value stored in the stack of the bytecode. Then, we have to prove that every bytecode specification holds ($\vdash \{E_l\}\ I_l$).

In the soundness theorem, we use the following abbreviation: for an exception table et, two labels l_a, l_b, and a type T, $et[l_a, l_b, T]$ returns the target label of the first et's exception entry whose starting and ending labels are less or equal and greater or equal than l_a and l_b, respectively, and whose type is a supertype of T.

Due to space limitations, we present the theorem without the details of the properties satisfied by the finally function f. The proof runs by induction on the structure of the derivation tree for $\{P\}\ s_1\ \{Q_n, Q_b, Q_e\}$. The proof and the complete theorem can be found in our technical report [7].

THEOREM 1.

$$\begin{pmatrix} \vdash \dfrac{Tree}{\{P\}\ \ s_1\ \ \{Q_n, Q_b, Q_e\}} \equiv T_{S_1} \quad \wedge \\ \left[(I_{l_{start}} ... I_{l_{end}}), et\right] = \nabla_S \left(T_{S_1},\ l_{start}, l_{end+1}, l_{break}, f, et'\right) \wedge \\ (\forall\ T : Type : (T \preceq Throwable \vee T \equiv any) : \\ (Q_e \wedge excV \neq null \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{et'[l_{start}, l_{end}, T]}) \wedge \\ \left(Q_n \quad \Rightarrow \quad E_{l_{end+1}}\right) \quad \wedge \\ (Q_b \ \Rightarrow finallyProperties) \end{pmatrix}$$
$$\Rightarrow$$
$$\forall\ l\ \in\ l_{start}\ ...\ l_{end} : \vdash \{E_l\}\ I_l$$

7. RELATED WORK

Necula and Lee [9] have developed certifying compilers, which produce proofs for basic safety properties such as type safety. Since our approach supports interactive verification of source programs, we can handle more complex properties such as functional correctness.

The open verifier framework for foundational verifiers [4] verifies untrusted code using customized verifiers. The approach is based on foundation proof carrying code. The architecture consists of a trusted checker, a fixpoint module, and an untrusted extension (a new verifier developed by untrusted users). However, the properties that can be proved are still limited.

A certified compiler [5, 11] is a compiler that generates a proof that the translation from the source program to the assembly code preserves the semantics of the source program. Together with a source proof, this gives an indirect correctness proof for the bytecode program. Our approach generates the bytecode proof directly, which leads to smaller certificates.

Barthe *et al.* [3] show that proof obligations are preserved by compilation (for a non-optimizer compiler). They prove the equivalence between the verification condition (VC) generated over the source code and the bytecode. The source language is an imperative language which includes method invocation, loops, conditional statements, `throw` and `try-catch` statements. However, they do not consider `try-finally` statements, which make the translation significantly more complex. Our translation supports `try-finally` and `break` statements.

Pavlova [10] extends the aforementioned work to a subset of Java (which includes `try-catch`, `try-finally`, and `return` statements). She proves equivalence between the VC generated from the source program and the VC generated from the bytecode program. The translation of the above source language has a similar complexity to the translation presented in this paper. However, Pavlova avoided the code duplication for `finally` blocks by disallowing `return` statements inside the `try` blocks of `try-finally` statements. This simplifies not only the verification condition generator, but also the translation and the soundness proof.

Furthermore, Barthe *et al.* [2] translate certificates for optimizing compilers from a simple interactive language to an intermediate RTL language (Register Transfer Language). The translation is done in two steps: first the source program is translated into RTL and then optimizations are performed building the appropriate certificate. Barthe *et al.* use a source language that is simpler than ours. We will investigate optimizing compilers as part of future work.

This work is based on Müller and Bannwart's work [1]. They present a proof-transforming compiler from a subset of Java which includes loops, conditional statements and object oriented features. We have extended the source language including exception handling and `break` statements. Moreover, we have also proved soundness.

8. CONCLUSION

We have defined proof transformation from a subset of Java to bytecode. The PTC allows us to develop the proof in the source language (which is simpler), and transforms it into a bytecode proof. Since Java source and bytecode are very similar, proof transformation is simple for many language features. In this paper, we focused on one of the most complex translations, namely the interaction between `try-finally` and `break` statements. We showed that our translation is sound, that is, it produces valid bytecode proofs.

To show the feasibility of our approach, we implemented a PTC for a language similar to the Java subset considered here. The compiler takes a proof in XML format and produces the bytecode proof.

As future work, we plan to extend the source language with statements like `return` and `continue`. Also, we plan to develop a proof checker that tests the bytecode proof. Moreover, we plan to analyze how proofs can be translated using an optimizing compiler.

Moreover, we will investigate proof-transforming compilation for language features that cannot by directly mapped to bytecode such as multiple inheritance and Eiffel's once methods. This extension will lead to a more general transformation framework.

9. ACKNOWLEDGMENTS

We would like to thank Nicu Georgian Fruja for reviewing and providing helpful comments on drafts of this paper. Müller's work was carried out at ETH Zurich. It was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. Nordio's work was funded in part by ETH under the Heterogeneous Proof-Carrying Components project.

10. REFERENCES

[1] F. Y. Bannwart and P. Müller. A Logic for Bytecode. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, volume 141 of *ENTCS*, pages 255–273. Elsevier, 2005.

[2] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate Translation for Optimizing Compilers. In *13th International Static Analysis Symposium (SAS)*, LNCS, Seoul, Korea, August 2006. Springer-Verlag.

[3] G. Barthe, T. Rezk, and A. Saabas. Proof obligations preserving compilation. In *Third International Workshop on Formal Aspects in Security and Trust, Newcastle, UK*, pages 112–126, 2005.

[4] B. Chang, A. Chlipala, G. Necula, and R. Schneck. The Open Verifier Framework for Foundational Verifiers. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDIS05)*, 2005.

[5] G. Goos and W. Zimmermann. Verification of Compilers. LNCS, pages 201–230. Springer-Verlag, 2005.

[6] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[7] P. Müller and M. Nordio. Proof-Transforming Compilation of Programs with Abrupt Termination. Technical Report 565, ETH Zurich, 2007.

[8] G. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1998.

[9] G. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Programming Language Design and Implementation (PLDI)*, pages 333–344. ACM Press, 1998.

[10] M. Pavlova. *Java Bytecode verification and its applications*. PhD thesis, University of Nice Sophia-Antipolis, 2007.

[11] A. Poetzsch-Heffter and M. J. Gawkowski. Towards Proof Generating Compilers. *ENTCS*, 132(1):37–51, 2005.

[12] A. Poetzsch-Heffter and P. Müller. A Programming Logic for Sequential Java. In S. D. Swierstra, editor, *European Symposium on Programming Languages and Systems (ESOP'99)*, volume 1576 of *LNCS*, pages 162–176. Springer-Verlag, 1999.

[13] A. Poetzsch-Heffter and N. Rauch. Soundness and Relative Completeness of a Programming Logic for a Sequential Java Subset. Technical report, Technische Universität Kaiserlautern, 2004.