Proof-Transforming Compilation of Programs with Abrupt Termination

Peter Müller and Martin Nordio

Microsoft Research (USA)  ETH Zurich
Proof-Carrying Code

Source Prog

Certifying compiler

VC Generator

VC

Proof Generator

Code Producer

code Annotations

Proof

VC Generator

Proof Checker

CPU

Code Consumer
Develop the proof for the Bytecode

- Logics for intermediate languages such as Java Bytecode and CIL were developed (Müller and Bannwart)

- Pro: It can produce the certificate needed
- Con: It is difficult and expensive
Proof-Transforming Compilers (PTC)

Source prog. + contracts

Prover

PTC

Source prog. + proof

Bytecode + proof

Proof Checker

CPU

Code Producer

Code Consumer
PTC Elements

Source Language: Java

Logic: Hoare-Style

Bytecode Logic

Bytecode Language: Java Bytecode

structured control flow variables

unstructured control flow
operand stack

unstructured control flow
operand stack

translation functions
The bytecode Language

```
bytecodeInstr ::= pushc v
                | pushv x
                | pop x
                | op op
                | goto /
                | brtrue /
                | nop
                | athrow
```
The bytecode Logic

- We use the bytecode logic developed by F. Bannwart and P. Müller
- Instruction specification

\[ \{ E_l \} \downarrow : I_l \]
The Source Language

- Similar to a **Java** subset

\[
\begin{align*}
exp & ::= \text{literal} \mid \text{var} \mid \text{exp op exp} \\
stm & ::= x = \text{exp} \mid \stm; \stm \mid \text{while (exp) stm} \\
& \quad \mid \text{break ;} \mid \text{if (exp) stm else stm} \\
& \quad \mid \text{try stm catch (type var) stm} \\
& \quad \mid \text{try stm finally stm} \mid \text{throw exp ;}
\end{align*}
\]
Logic for Java subset

- The logic is based on the programming logic developed by A. Poetzsch-Heffter and N. Rauch.

- Properties of method bodies are expressed by Hoare triples of the form

\[
\{ P \} \text{ comp } \{ Q_n, Q_b, Q_e \}
\]

normal break exception
Example: try-finally statements

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

b = 4 Normal
Compilation: try-finally statements

```
try {
  s1
}
finally {
  s2
}
```

\[ \nabla_S (s_1) \]

\[ \nabla_S (s_2) \]

\[ l_c : \text{goto} \ l_h \]

\[ l_d : \text{pop} \ eTmp \]

\[ \nabla_S (s_2) \]

\[ l_f : \text{pushv} \ eTmp \]

\[ l_g : \text{athrow} \]

\[ l_h : \ldots \]

Exception Table

<table>
<thead>
<tr>
<th>From</th>
<th>to</th>
<th>target</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l_a )</td>
<td>( l_b )</td>
<td>( l_d )</td>
<td>any</td>
</tr>
</tbody>
</table>
Example: try-finally statements

\[ \nabla_S \left( b++; \right. \\
\left. \text{throw new Exception() } \right) \]

\[ \nabla_S \left( b++; \right. \\
\left. \text{break } \right) \]

l\textsubscript{c} : goto l\textsubscript{h}

l\textsubscript{d} : pop eTmp

\[ \nabla_S \left( b++; \right. \\
\left. \text{break } \right) \rightarrow \text{goto l\textsubscript{i}} \]

l\textsubscript{f} : pushv eTmp

l\textsubscript{g} : athrow

l\textsubscript{h}...

l\textsubscript{i} : \nabla_S \left( b++; \right)

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

\[
\begin{array}{c}
\{ P \} s_1 \{ Q_n, Q_b, Q_e \} \\
\{ Q \} s_2 \{ R, R'_b, R'_e \} \\
\{ P \} \text{try } s_1 \text{ finally } s_2 \{ R'_n, R'_b, R'_e \}
\end{array}
\]

where

\[
Q \equiv \left( (Q_n \land \chi Tmp = \text{normal}) \lor (Q_b \land \chi Tmp = \text{break}) \lor \left( Q_e[eTmp/excV] \land \chi Tmp = \text{exc} \land eTmp = \text{excV} \right) \right)
\]

and

\[
R \equiv \left( (R'_n \land \chi Tmp = \text{normal}) \lor (R'_b \land \chi Tmp = \text{break}) \lor \left( R'_e \land \chi Tmp = \text{exc} \right) \right)
\]
Example 2: Exception Table

```java
while (i < 20) {
    try {
        try {
            try {
                ...
                break;
                ...
            }
            catch (Exception e) {
                i = 9;
            }
        }
        finally {
            throw new Exception();
        }
    }
    catch (Exception e) {
        i = 99;
    }
}
```
Example 2: Exception Table (cont.)

```java
while (i < 20) {
    try {
        try {
            try {
                ...  
                ...  
            }
            catch (Exception e) {
                i = 9;  
            }
        }
        catch (Exception e) {
            i = 99;  
        }
    }
    finally {
        throw new Exception();
    }
}
```
Example 2: Exception Table (cont.)

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

\[ \nabla_E : \text{Precondition} \times \text{Expression} \times \text{Postcondition} \times \text{Label} \rightarrow \text{BytecodeProof} \]

\[ \nabla_S : \text{ProofTree} \times \text{List[Finally]} \times \text{ExceptionTable} \rightarrow [\text{BytecodeProof} \times \text{ExceptionTable}] \]

Finally is defined as a tuple of \([\text{ProofTree}, \text{ExceptionTable}]\)
PTC

- **Compositional statement**
  \[
  [B_{S_1}, et_1] = \nabla_S (T_{S_1}, f, et) \\
  [B_{S_2}, et_2] = \nabla_S (T_{S_2}, f, et_1) \\
  \]
  \[
  \{ [B_{S_1} + B_{S_2}, et_2] \}
  \]

- **While**
  \[
  \text{Finally} := \emptyset
  \]

- **try-finally**
  \[
  \text{Finally} := [\text{ProofTree}, \text{ExceptionTable}] + \text{Finally}
  \]

- **Break**
  - Translate the finally blocks dividing the exception table
  - Add a goto end-while
Summary

- **Source Language:**
  - Subset of Java
  - while, break,
  - try-catch, try-finally, throw

- **Soundness proof**