Proof Transformations for Object-Oriented Programs

Martin Nordio
ETH Zurich

Joint work with Cristiano Calcagno, Bertrand Meyer, and Peter Müller
Mobile Code

Problem: How to verify mobile code?
Solution: Proof-Carrying Code
Verification Process based on Proof-Transforming Compilation (PTC)

- **Source prog. + contracts**
  - **Prover**
  - **PTC**
    - **Source prog. + proof**
      - **Bytecode + proof**
        - **Code Producer**
          - **untrusted tool**
          - **trusted tool**
        - **Proof Checker**
          - **CPU**
            - **Code Consumer**
Verification Process based on Proof-Transforming Compilation (PTC)

Advantages:
• Verification of Functional Properties
• PTCs are not part of the Trusted Computing Base
• Small Trusted Computing Base: Proof Checker
• Verification on the Source Language
Basics of our PTC

Source Language: C#, Eiffel, and Java

Logic: Hoare-Style

Translation Functions

Bytecode Language: CIL Bytecode

Bytecode Logic

structured control flow

variables

unstructured control flow
operand stack
The bytecode language and its Logic

- Bytecode language similar to CIL bytecode
  - Boolean type
  - Instead of using an array of local variables like in CIL, we use the name of the source variable

- Bytecode Logic:
  - Logic developed by F. Bannwart and P. Müller
  - Instruction specification

\[ \{ E_l \} \quad l : I_l \]
Outline

Part 1: Java
- Semantics for Java
- PTC for Java

Part 2: Eiffel
- Semantics for Eiffel
- PTC for Eiffel
The subset of Java

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

```java
foo () {
    int b = 1;
    b++;
}
```

`b = 2`
Java: while and break

```java
foo () {
    int b = 1;
    while (true) {
        b++;
        break;
    }
}
```

b = 2
Java: throw and try-finally

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

b = 2   Exception
Why do we need semantics for Java?

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

Does this program compile in C#?
Semantics for Java

- Operational and Axiomatic Semantics
- The logic is based on the programming logic developed by P. Müller and A. Poetzsch-Heffter
- Properties of method bodies are expressed by Hoare triples of the form

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

- Properties of methods:
  - Normal postcondition
  - Exception postcondition
  - Break postcondition
Compiling the subset of Java

```java
while (i < 20) {
    try {
        try {
            try {
                ...
                break;
                ...
            }
        }
        catch (Exception e) {
            i = 9;
        }
    }
    finally {
        (throw new Exception();)
    }
    catch (Exception e) {
        i = 99;
    }
}
```
Compiling the subset of Java

```java
while (i < 20) {
    try {
        try {
            try {
                ...
            }
            ...
        } catch (Exception e) {
            i = 9;
        }
        finally {
            throw new Exception();
        }
    } catch (Exception e) {
        i = 99;
    }
}
```
Compiling the subset of Java

```java
class Main {
  public static void main(String[] args) {
    int i = 0;
    while (i < 20) {
      try {
        try {
          ... break;
          ...
        } catch (Exception e) {
          i = 9;
        }
      } finally {
        throw new Exception();
      }
      catch (Exception e) {
        i = 99;
      }
    }
  }
}
```
Proof-Transforming Compilation for Java

- Translation of Methods

\[ \nabla_P : Proof \rightarrow CILProof \]
\[ \nabla_B : ProofTree \rightarrow CILProofTree \]

ProofTree is a derivation in the Java logic
CILProofTree is a derivation in the bytecode logic

- Translation of Expressions and Instructions:

\[ \nabla_E : Precondition \times Expression \times Postcondition \times Label \rightarrow BytecodeProof \]
\[ \nabla_S : ProofTree \times Label \times Label \times Label \rightarrow BytecodeProof \]

BytecodeProof is a list of instruction specifications
Eiffel

Part 1: Java

Semantics for Java
Proof

PTC for Java
Proof Transformation

Part 2: Eiffel

Semantics for Eiffel
Proof

PTC for Eiffel
Proof Transformation
The subset of Eiffel

- The subset includes:
  - Basic instructions such as assignments, and loops
  - Exception handling: rescue clauses
  - Multiple inheritance
  - Once routines
Operational and Axiomatic Semantics

Based on the logic by P. Müller and A. Poetzsch-Heffter

Properties of routines and routine bodies are expressed by Hoare triples of the form

$$\{ P \} \text{COMP} \{ Q_n, Q_e \}$$

Normal postcondition

Exception postcondition
merge (other: LINKED_LIST [G]): LINKED_LIST [G]
-- Merge other into current structure returning a new LINKED_LIST

require
is_linked_list: other.type.conforms_to(LINKED_LIST [G].type)
same_type: Current.type.is_equal(other.type)
ensure
result_type: Result.type.is_equal(LINKED_LIST_CLASS [G].type)
Proof-Transforming Compilation for Eiffel

- Contract Translator
  - Deep embedding of contracts, pre- and postconditions
  - Translation functions
    - Input: Deep embedding of Boolean expressions
    - Output: First Order Logic

- Proof Translator

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

Eiffel Proof-Transforming Compiler

- XML Parser
- Proof Translator
- Contract Translator
- Isabelle Generator

Input: .XML
Output: .thy
Summary

Java Proof-Transforming Compiler:
• Bytecode: CIL and Java Bytecode
• Translation of try-catch, try-finally, while and break Instructions
• Soundness Proof

Eiffel Proof-Transforming Compiler:
• Bytecode: CIL
• Logic for Eiffel: Hoare-style logic
• Translation of Multiple Inheritance to Single Inheritance
• Exception Handling
• Soundness Proof

Prototype:
• Implementation Eiffel PTC
• Proof Checker (formalized in Isabelle)