Software Verification

ETH Zurich, September-December 2010

-6-

Proof-Carrying Code & Proof-Transforming Compilation

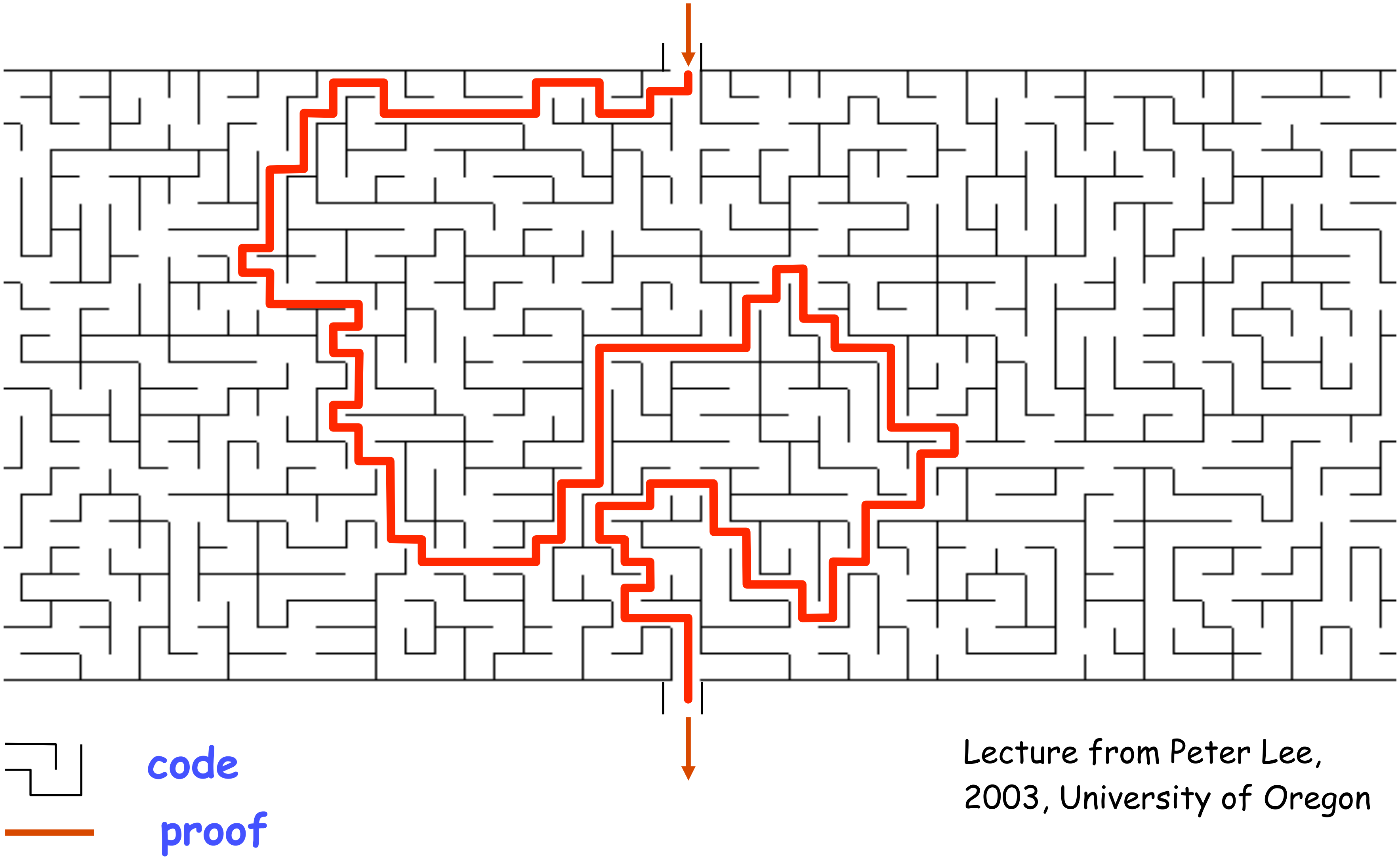
Overview

- Proof-Carrying Code
- Proof-Transforming Compilation
  - Semantics for Java and Eiffel
  - A Hoare-style logic for Bytecode
  - Proof Translation

Mobile Code

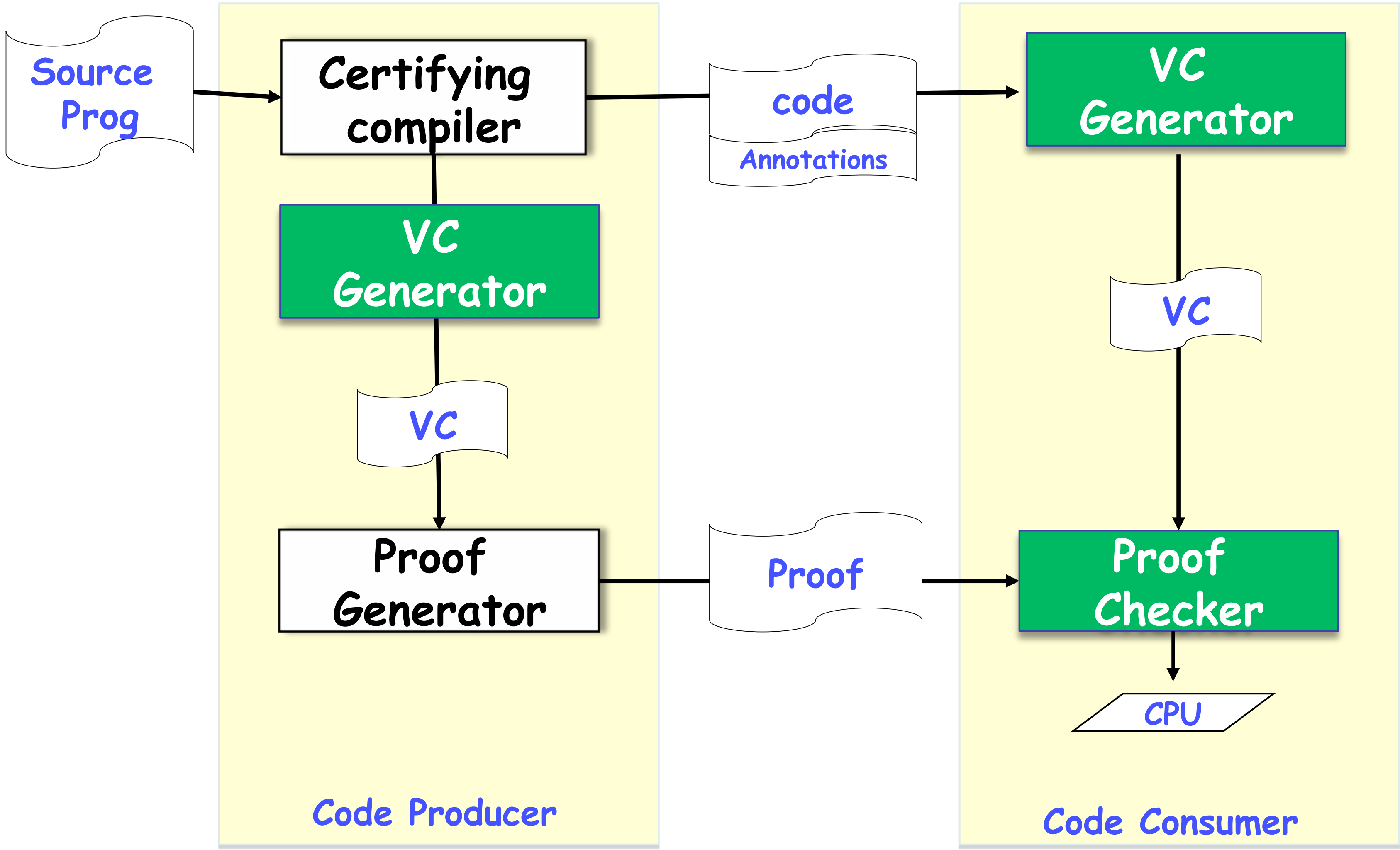
How to verify mobile code?

Proof-Carrying Code



Lecture from Peter Lee,
2003, University of Oregon

Proof-Carrying Code



What do we gain?

The process of checking the proof is fast and automatic

There is no loss of performance in the bytecode program

The overhead of developing the proof is done once and for all by the code producer

The code consumer does not need to trust the code producer

Limitations

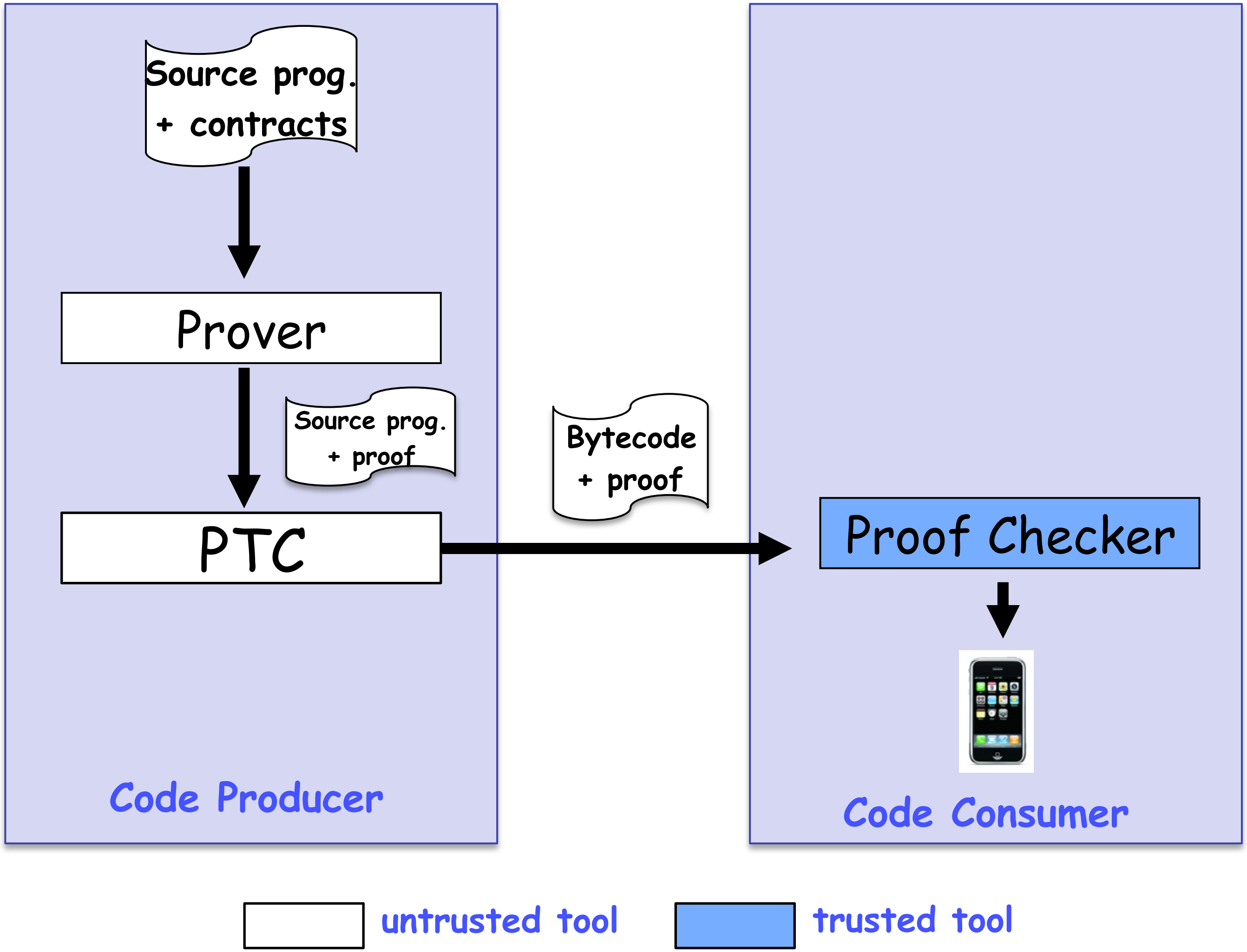
Proofs are big

Good for safety but not yet termination

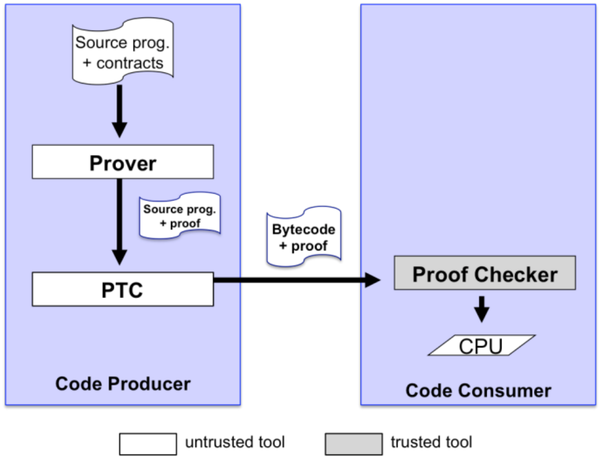
Certifying compilers can generate proof automatically only for a restricted set of properties

In Lee and Necula's implementation, they consider machine code… portability?

Verification Process based on Proof-Transforming Compilation (PTC)



Advantages



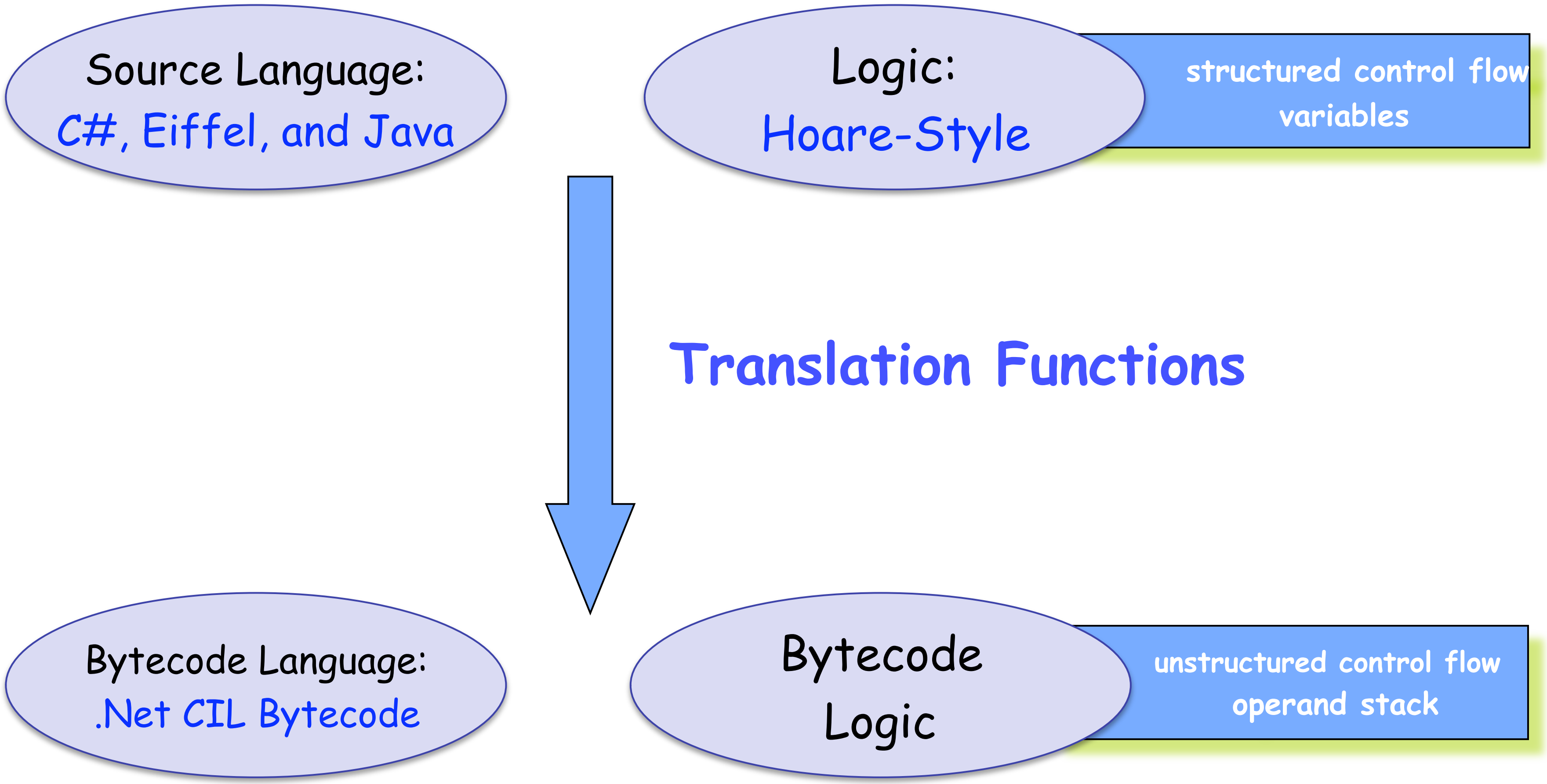
Verification of functional properties

PTCs are not part of the trusted computing base

Small trusted computing base: Proof Checker

Verification on the source language

Basics of our PTC



Overview

- Proof-Carrying Code
- Proof-Transforming Compilation
  - Semantics for Java and Eiffel
  - A Hoare-style logic for Bytecode
  - Proof Translation

The Subset of Java

Assignment and compound

```
foo () {
    int b=1;
    b++;
} b = 2
```

While and break

```
foo () {
    int b=1;
    while (true) {
        b++;
        break;
    }
} b = 2
```

Try-finally and throw

```
foo () {
    int b=1;
    try {
        throw new Exception();
    }
    finally {
        b++;
    }
} b = 2      Exception
```

Other features:
- **Try-catch**
- **If then else**
- **Read and write fields**
- **Routine invocation**
- **Single inheritance**

Why is this Subset of Java interesting?

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

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

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

Why is this Subset of Java interesting?

```
foo () {
    int b=1;                          b = 1
    while (true) {
        try {
            b++;                      b = 2
            throw new Exception();    b = 2
        }
        finally {
            b++;                      b = 3
            break;                    b = 3
        }
    }
    b++;                              b = 4
}
```

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

$$\{\ P\ \}\ \ \mathtt{S}\ \ \{\ Q_n\ ,\ Q_b\ ,\ Q_e\ \}$$

Normal postcondition

Break postcondition

Exception postcondition

- Properties of methods

$$\{\ P\ \}\ \ m\ \ \{\ Q_n\ ,\ Q_e\ \}$$

Normal postcondition

Exception postcondition

The subset of Eiffel

Basic instructions such as assignments, if then else, and loops

Exception handling: rescue clauses

Once routines

Multiple inheritance

Eiffel: Exception Handling

```
connect_to_server
  --Connect to Madrid, York, or Zurich.
 local
    i: INTEGER
 do
    if i = 0 then connect_to_madrid  end
    if i = 1 then connect_to_york    end
    if i = 2 then connect_to_zurich  end
 rescue
    if i < 3 then
      i := i + 1
      Retry := True
    else
      failed := True
    end
 end
```

Eiffel: Once Functions

```
f (i: INTEGER): INTEGER
    once
        Result := i + 1
    end
```

```
j := f (2)
        { j = 3 }
k := f (4)
        { j = 3 and k = 3 }
```

Semantics for Eiffel

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

$$\{\ P\ \}\quad S\quad \{\ Q_n\ ,\ Q_e\ \}$$

Normal postcondition

Exception postcondition

Proof of soundness and completeness

Logic: Assignment Rule

$$\dfrac{}{\left\{ \begin{array}{l} (safe(e) \ \wedge \ P[e/x]) \vee \\ (\neg safe(e) \ \wedge \ Q_e) \end{array} \right\} \ x := e \ \{ P \ , \ Q_e \}}$$

Logic: Compound

$$\frac{\{\ P\ \}\ \ s_1\ \ \{\ ???\ ,\ ???\ \}\qquad \{\ ???\ \}\ \ s_2\ \ \{\ ???\ ,\ ???\ \}}{\{\ P\ \}\ \ s_1; s_2\ \ \{\ ???\ ,\ ???\ \}}$$

Example

Assignment Rule

$$\{\ true\ \}\quad balance := b \setminus\setminus i\ \{\ i \neq 0\ \wedge\ balance = b \setminus\setminus i \wedge\ ,\quad i = 0\ \}$$

Assignment Rule

$$\{i \neq 0\ \wedge\ balance = b\ \setminus\setminus\ i\ \}\ \ credit := b + 10\ \left\{ \begin{array}{l} i \neq 0\ \wedge\ balance = b\ \setminus\setminus\ i \wedge \\ credit = b + 10 \end{array} ,\quad false \right\}$$

Compound Rule

$$\{\ true\ \}\ balance := b\ \setminus\setminus\ i\ ;\ credit := b + 10\ \left\{ \begin{array}{l} i \neq 0\ \wedge\ balance = b\ \setminus\setminus\ i \wedge \\ credit = b + 10 \end{array} ,\quad i = 0 \right\}$$

Rescue

"Retry invariant"

$$P \Rightarrow P'$$

$$\frac{\{INV \wedge P'\}\ \boxed{b}\ \{INV \wedge Q\ ,\ Q'\} \qquad \{Q'\}\ \boxed{r}\ \{INV \wedge (\mathbf{Retry} \Rightarrow P') \wedge (\neg\ \mathbf{Retry} \Rightarrow R)\ ,\ INV \wedge R\}}{\{INV \wedge P\}\ \boxed{\mathbf{do}\ b\ \mathbf{rescue}\ r\ \mathbf{end}}\ \{INV \wedge Q\ ,\ INV \wedge R\}}$$

Normal postcondition

Error postcondition

Example: rescue

```
safe_division (x,y: INTEGER): INTEGER
    local
        z: INTEGER
    do
        Result := x // (y+z)
    ensure
        y = 0 implies Result = x
        y /= 0 implies Result = x // y
    rescue
        z := 1
        Retry := true
    end
```

$$\{\ true\ \}\ \ \textit{MATH:safe\_division}\ \ \{\ Q\ ,\ \ false\ \ \}$$

where

$$Q \equiv (y = 0 \Rightarrow Result = x) \wedge (y/ = 0 \Rightarrow Result = x//y)$$

Example: rescue

```
safe_division (x,y: INTEGER): INTEGER
 local
     z: INTEGER
 do
```

$$\{(y \neq 0 \wedge z = 0) \vee (y = 0 \wedge (z = 1 \vee z = 0))\}$$

```
     Result := x // (y+z)
```

$$\left\{ \left(\begin{array}{l} (y = 0 \Rightarrow Result = x) \wedge \\ (y \neq 0 \Rightarrow Result = x/y) \end{array} \right), (y = 0 \wedge z = 0) \right\}$$

```
 ensure
     y = 0 implies Result = x
     y /= 0 implies Result = x // y
 rescue
```

$$\{ \; y = 0 \wedge z = 0 \; \}$$

```
     z := 1
```

$$\{ \; (y = 0 \wedge z = 1), \; false \; \}$$

```
     Retry := true
```

$$\left\{ \left(\; Retry \wedge (y = 0 \wedge z = 1) \; \right), \; false \right\}$$

```
 end
```

Retry invariant

$$(y \neq 0 \wedge z = 0) \vee (y = 0 \wedge (z = 1 \vee z = 0))$$

Overview

- Proof-Carrying Code
- Proof-Transforming Compilation
  - Semantics for Java and Eiffel
  - A Hoare-style logic for Bytecode
  - Proof Translation

The bytecode Language

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

$$
\begin{array}{lcl}
\mathsf{bytecodeInstr} & ::= & \mathsf{pushc}\ \mathsf{v} \\
 & | & \mathsf{pushv}\ x \\
 & | & \mathsf{pop}\ x \\
 & | & op_{op} \\
 & | & \mathsf{goto}\ l \\
 & | & \mathsf{brtrue}\ l \\
 & | & \mathsf{nop} \\
 & | & \mathsf{athrow}
\end{array}
$$

The Bytecode Language and its Logic

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

$$\{P\}\ l:\ \ Inst$$

Precondition

Label

CIL Instruction

The bytecode Logic

Rules for instructions

$$\frac{E_l \Rightarrow wp^1_p(I_l)}{\mathsf{A} \vdash \{E_l\}\ l : I_l}$$

The bytecode Logic

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'	$(\neg s(0) \Rightarrow shift(E_{l+1})) \wedge (s(0) \Rightarrow shift(E_{l'}))$
return	true
nop	E_{l+1}

$$shift(E) = E[s(i+1)/s(i) \text{ for all } i \in \mathrm{N}\ ]$$
$$unshift = shift^{-1}$$

Example Bytecode Proof

Source Program:

x := 5

y := 1

Compiled Program:

L00: push 5

L01: pop x

L02: push 1

L03: pop y

Overview

- Proof-Carrying Code
- Proof-Transforming Compilation
  - Semantics for Java and Eiffel
  - A Hoare-style logic for Bytecode
  - Proof Translation

Proof-Transforming Compilation for Eiffel

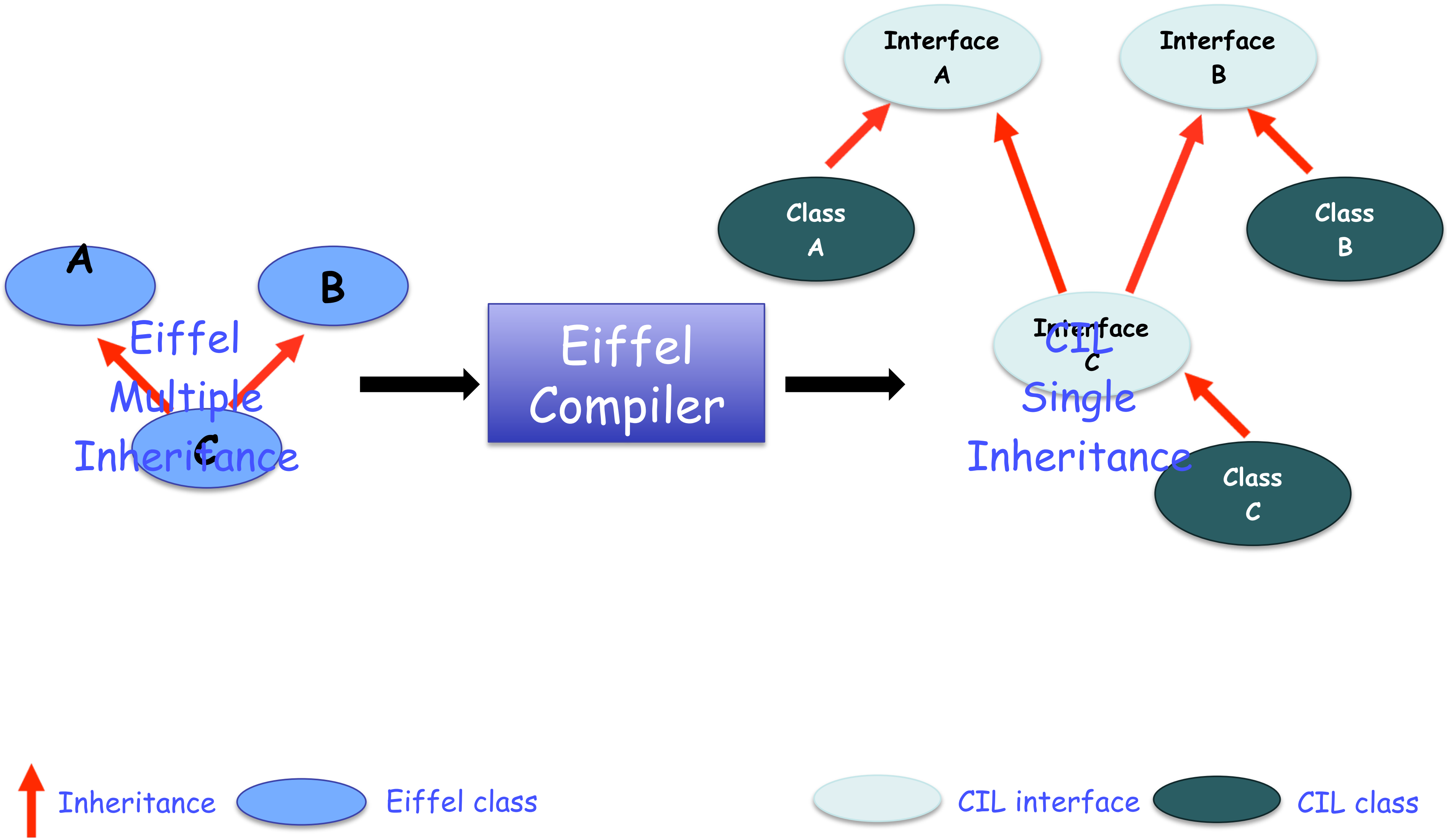
Contract Translator

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

$$
\begin{aligned}
\nabla_E &: Precondition \times Expression \times Postcondition \times Label \rightarrow BytecodeProof \\
\nabla_S &: ProofTree \times Label \times Label \times Label \rightarrow BytecodeProof
\end{aligned}
$$

- Soundness Proof

Compiling Eiffel to .Net CIL



Applications

Compound Rule

$$\nabla_S\left(\begin{array}{c} \{\,true\,\}\quad balance := b\;;\; credit := b + 10 \quad \left\{ \begin{array}{l} balance = b\;\wedge \\ credit = b + 10 \end{array} \right\} \end{array}, l_{00},\; l_{06} \right)$$

Assignment Rule

$$\nabla_S\left(\{\,true\,\}\; balance := b\; \{\, balance = b\,\}, l_{00}, l_{02}\right)$$

Assignment Rule

$$\nabla_S\left(\{balance = b\}\;\; credit := b + 10 \;\left\{ \begin{array}{l} balance = b\;\wedge \\ credit = b + 10 \end{array} \right\}, l_{02}, l_{06}\right)$$

CIL proof:

$$\nabla_E\left(\{\,true\,\},\; l_{00}:\ \text{ldloc } b,\ \{\, s(0) = b \,\}\right)$$

$$\{\, s(0) = b \,\}\quad l_{01}:\ \text{stloc } balance$$

$$\nabla_E\left(\{\, balance = b \,\},\; l_{02}:\ \text{ldloc } b,\ \{\, balance = b \wedge s(0) = b \,\}\right)$$

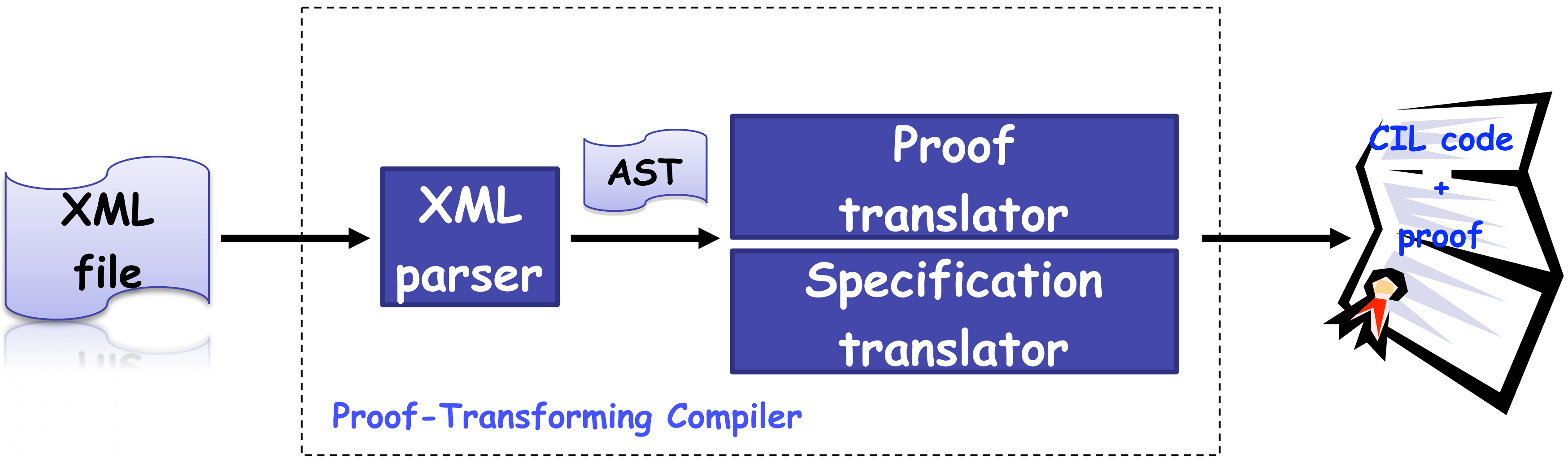
$$\nabla_E\left(balance = b,\; b + 10,\; \left(\begin{array}{l} balance = b\;\wedge \\ s(0) = b + 10 \end{array}\right)\right)$$

$$\nabla_E\left(\left(\{\, balance = b \wedge s(0) = b \,\},\; 10\right),\; \left(l_{03}:\ \text{ldc } 10,\ balance = b \wedge s(1) = b \;\wedge\; s(0) = 10\right)\right)$$

$$\{\, balance = b \;\wedge\; s(1) = b \wedge\; s(0) = 10 \,\}\quad l_{04}:\ \text{add}$$

$$\{\, balance = b \;\wedge\; s(0) = b + 10 \,\}\quad l_{05}:\ \text{stloc } credit$$

Tool Support



Experiments with PTC

Example	#Classes	#Routines	#lines Eiffel	#lines source proof
Boolean expressions	2	3	76	205
Attributes	3	5	83	167
Conditionals	1	2	55	154
Loops	1	1	31	73
Bank Account simple	1	3	57	108
Bank Account	1	5	57	130
Sum Integers	1	1	35	126
Subtyping	3	5	41	117
Demo	4	8	152	483
Total	**17**	**33**	**587**	**1563**

Size of the proof

Example	#lines Eiffel	#lines source proof	#lines in Isabelle
Boolean expressions	76	205	711
Attributes	83	167	1141
Conditionals	55	154	510
Loops	31	73	305
Bank Account simple	57	108	441
Bank Account	57	130	596
Sum Integers	35	126	358
Subtyping	41	117	756
Demo	152	483	1769
Total	**587**	**1563**	**6587**

Experiments Proof Checker

Isabelle Example	#lines in Isabelle	Simplifier Proof Script (in sec)	Optimized Proof Script (in sec)
Boolean expressions	711	3.4	1.9
Attributes	1141	3.6	2.2
Conditionals	510	7.3	3.8
Loops	305	14.1	3.2
Bank Account simple	441	5.5	2.4
Bank Account	596	12.8	4.6
Sum Integers	358	45.2	6.3
Subtyping	756	4.3	2.3
Demo	1769	92.2	27.5
Total	**6587**	**192.4 (~3')**	**54.2**