Proving Backward Compatibility for Object-Oriented Libraries

Yannick Welsch and Arnd Poetzsch-Heffter

University of Kaiserslautern

15.07.2014

“Interfaces of systems are collections of classes rather than methods”

[Tony Hoare, Meeting of the IFIP WG 1.9, Vienna, 15.7.14]

General motivation

- practical motivation:
  → software maintenance and evolution

General motivation

- practical motivation:
  → software maintenance and evolution

- principle motivation:
  → behavioral subtyping at the code level

General motivation

- practical motivation:
  → software maintenance and evolution

- principle motivation:
  → behavioral subtyping at the code level

- conceptual motivation:
  → components with complex interfaces

General motivation

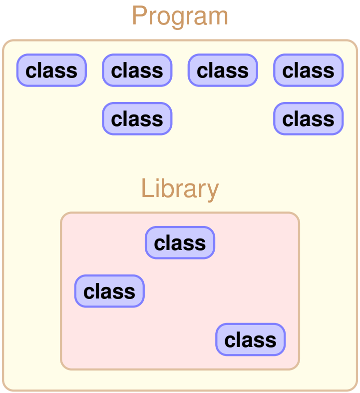
- practical motivation:
  → software maintenance and evolution
- principle motivation:
  → behavioral subtyping at the code level
- conceptual motivation:
  → components with complex interfaces

```
package p;

public interface IT { IT m(); }

public class UseIT {
  public IT runM( IT x ) {
    return x.m();
  }
}
```

Overview

- Introduction to backward compatibility
- A fully abstract semantics of LPJava
- Proving backward compatibility

Introduction to backward compatibility



Program

class

class

class

class

class

class

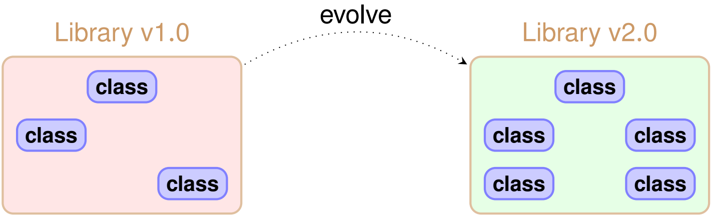
Library

class

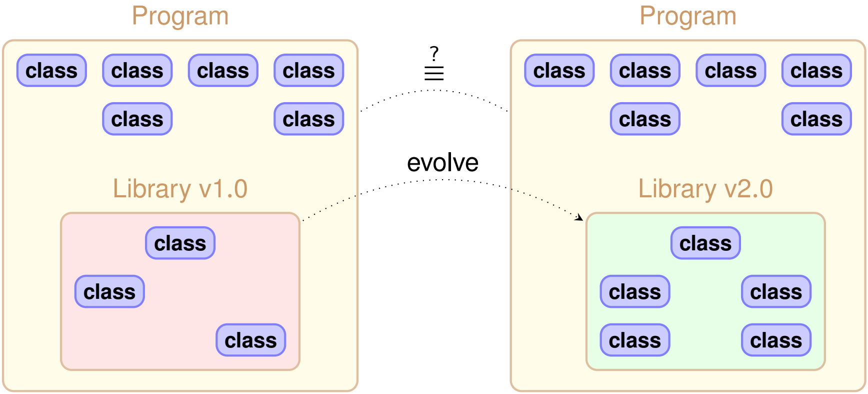
class

class

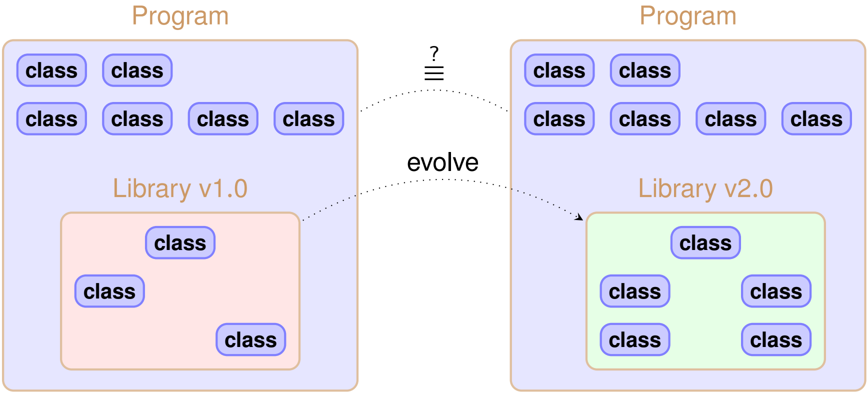
Introduction to backward compatibility



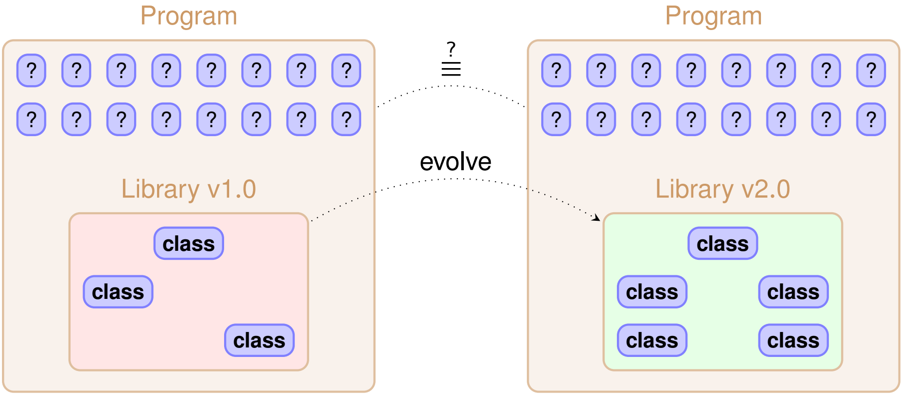
Introduction to backward compatibility



Introduction to backward compatibility



Introduction to backward compatibility



Example: Is library v2.0 backward compatible with v1.0?

Library v1.0

```
package cells;

public interface Val {}

public class Cell {
  private Val v;
  public void set(Val nv) {
    v = nv;
  }
  public Val get() {
    return v;
  }
}
```

Library v2.0

```
package cells;

public interface Val {}

public class Cell {
  private Val v1, v2;
  private boolean f;
  public void set(Val nv) {
    f = !f ;
    if (f) v1 = nv; else v2 = nv;
  }
  public Val get() {
    if (f) return v1; else return v2;
  }
  public Val getPrevious() {
    if (f) return v2; else return v1;
  }
}
```

Backward compatibility: Two aspects

backward compatibility

=

source compatibility

+

behavioral compatibility

Source compatibility: Separation by compiling

Library v1.0

```
package problem1;

interface I {
    ...
}

public abstract class C
            implements I {
    public I f;
    protected C g;
}
```

Library v2.0

```
package problem1;

public class C {
    public D f;
    public C g;
    public C m() { ... }
}

class D {
    ...
}
```

Source compatibility: Separation by used libraries

Library v1.0

```
package problem2;

public class C {
  public p.D f;
}
```

Library v2.0

```
package problem2;

public class C {
  public p.D f;
  private p.E g;
}
```

Behavioral compatibility: Separation by application code

Library v1.0

```
package problem3;

public interface A {
   int m1();
   int m2();
}

public class B implements A {
   public int m1() { return 42; }
   public int m2() { return m2(); }
}
```

Library v2.0

```
package problem3;

public interface A {
   int m1();
   int m2();
}

public class B implements A {
   public int m1() { return 42; }
   public int m2() { return 42; }
}
```

A fully abstract semantics of LPJava

Challenge and approach

Definition (Backward compatibility)

A library Y is **backward compatible** with X if for *any* program context K of X: $KX_{init}\downarrow$ implies $KY_{init}\downarrow$ (adopted from [Morris 68])

Challenge and approach

Definition (Backward compatibility)

A library Y is **backward compatible** with X if for *any* program context K of X: $KX_{init}\downarrow$ implies $KY_{init}\downarrow$

Proving backward compatibility is challenging:

1. Heaps and stacks in program configurations significantly different
2. Infinitely many possible contexts

Challenge and approach

Definition (Backward compatibility)

A library Y is **backward compatible** with X if for *any* program context K of X: $KX_{init}\downarrow$ implies $KY_{init}\downarrow$

Proving backward compatibility is challenging:

1. Heaps and stacks in program configurations significantly different

   → Use trace-based semantics that abstracts from internal representation of library

   ### Theorem (Trace semantics captures all relevant information)

   Y is backward compatible with X if and only if
   for any program context K of X: $Traces(KX) \subseteq Traces(KY)$.

2. Infinitely many possible contexts

Challenge and approach

Definition (Backward compatibility)

A library Y is **backward compatible** with X if for *any* program context K of X: $KX_{init}{\downarrow}$ implies $KY_{init}{\downarrow}$

Proving backward compatibility is challenging:

1. Heaps and stacks in program configurations significantly different

   → Use trace-based semantics that abstracts from internal representation of library

   ### Theorem (Trace semantics captures all relevant information)

   Y is backward compatible with X if and only if for any program context K of X: $Traces(KX) \subseteq Traces(KY)$.

2. Infinitely many possible contexts

   → Construct *most general context* κ_X that simulates all contexts of X

   ### Theorem (Most general context generates all possible behaviors)

   $$Traces(\kappa_X X) = \bigcup_K Traces(KX)$$

Setting - LPJava

$$
\begin{array}{rcl}
K, X, Y & ::= & \overline{Q} \\
Q, R & ::= & \textbf{package}\ p\ ;\ \overline{D} \\
D & ::= & [\textbf{public}]\ \textbf{class}\ c\ \textbf{extends}\ p.c\ \textbf{implements}\ \overline{p.i}\ \{\ \overline{F}\ \overline{M}\ \} \\
 & | & [\textbf{public}]\ \textbf{interface}\ i\ \textbf{extends}\ \overline{p.i}\ \{\ \overline{M}\ \} \\
F & ::= & \textbf{private}\ p.t\ f\ ; \\
M & ::= & \textbf{public}\ p.t\ m(\overline{p.t\ v})\ \Big(\ ;\ \mid\ \{\ E\ \}\ \Big) \\
E & ::= & x \mid \text{null} \mid \text{new}\ p.c() \mid E.f \mid E.f = E \mid E.m(\overline{E}) \\
 & | & \text{let}\ p.t\ x = E\ \text{in}\ E \mid E == E\ ?\ E : E \mid (p.t)E : E \\
t & ::= & c \mid i
\end{array}
$$

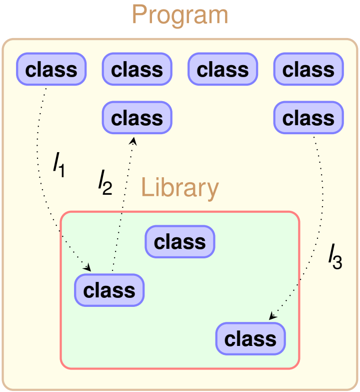
where $c \in$ class names, $i \in$ interface names, $p, q \in$ package names, $f \in$ field names, $m \in$ method names and $x \in$ variable names.

From program runs to traces

- Start with standard small-step operational semantics (similar to FJ)
  - $(KX, Heap, Stack) \rightsquigarrow (KX, Heap', Stack')$
- Characterize library behavior by the interactions between code belonging to library (X) and code belonging to program context (K)
- Generate a *label* if control flow passes from K to X or vice-versa
- Augment configurations
- Program runs then generate traces (i.e. sequences of labels)

From program runs to traces

- Start with standard small-step operational semantics (similar to FJ)
- Characterize library behavior by the interactions between code belonging to library (X) and code belonging to program context (K)



- Generate a *label* if control flow passes from K to X or vice-versa
- Augment configurations
- Program runs then generate traces (i.e. sequences of labels)

From program runs to traces

- Start with standard small-step operational semantics (similar to FJ)
- Characterize library behavior by the interactions between code belonging to library (X) and code belonging to program context (K)
- Generate a *label* if control flow passes from K to X or vice-versa
  - Only method calls and returns relevant
  - Label records all relevant information:
    - direction and method name
    - method call / return
    - exposed objects
    - abstraction of types of exposed objects
- Augment configurations
- Program runs then generate traces (i.e. sequences of labels)

From program runs to traces

- Start with standard small-step operational semantics (similar to FJ)
- Characterize library behavior by the interactions between code belonging to library (X) and code belonging to program context (K)
- Generate a *label* if control flow passes from K to X or vice-versa
- Augment configurations
  - Tag stack frames whether code originates from K or X
- Program runs then generate traces (i.e. sequences of labels)

From program runs to traces

- Start with standard small-step operational semantics (similar to FJ)
- Characterize library behavior by the interactions between code belonging to library (X) and code belonging to program context (K)
- Generate a *label* if control flow passes from K to X or vice-versa
- Augment configurations
- Program runs then generate traces (i.e. sequences of labels)

(Simplified) trace examples

```
// Program context K

public class ValueImpl
                implements Val { ... }

public class Main {
  public void main() {
    Cell c = new Cell();
    Val v = new ValueImpl();
    c.set(v);
    Val v2 = c.get();
  }
}
```

```
// Library X

public interface Val {}
public class Cell {
  private Val v;
  public void set(Val nv) {
    v = nv;
  }
  public Val get() {
    return v;
  }
}
```

(Simplified) trace examples

```
// Program context K

public class ValueImpl
             implements Val { ... }

public class Main {
  public void main() {
    Cell c = new Cell();
    Val v = new ValueImpl();
    c.set(v);
    Val v2 = c.get();
  }
}
```

```
// Library X

public interface Val {}
public class Cell {
  private Val v;
  public void set(Val nv) {
    v = nv;
  }
  public Val get() {
    return v;
  }
}
```

$\text{Traces}(KX) = \{\ \text{call } o_1.\mathit{set}(o_2)$⍗ $\cdot$ $\text{rtrn } \_$⍐ $\cdot$ $\text{call } o_1.\mathit{get}()$⍗ $\cdot$ $\text{rtrn } o_2$⍐ $\}$

(where $o_1 \neq o_2$ are arbitrary object identifier)

(Simplified) trace examples

```
// Program context K

public class ValueImpl
                implements Val { ... }

public class Main {
  public void main() {
    Cell c = new Cell();
    Val v = new ValueImpl();
    c.set(v);
    Val v2 = c.get();
  }
}
```

```
// Library Y

public interface Val {}
public class Cell {
  private Val v1, v2;
  private boolean f;
  public void set(Val nv) {
    f = !f ;
    if (f) v1 = nv; else v2 = nv;
  }
  public Val get() {
    if (f) return v1; else return v2;
  } ...
}
```

$$\text{Traces}(KX) = \{\ \text{call } o_1.set(o_2)\downarrow \ \cdot\ \text{rtrn } \_\uparrow \ \cdot\ \text{call } o_1.get()\downarrow \ \cdot\ \text{rtrn } o_2\uparrow\ \}$$

(where $o_1 \neq o_2$ are arbitrary object identifier)

$= \text{Traces}(KY)$

Construction of Most General Context

- Extend LPJava by nondeterministic expression ($E ::= \ldots \mid$ nde)

Construction of Most General Context

- Extend LPJava by nondeterministic expression ($E ::= \ldots \mid$ nde)
- Evaluation of *nde* leads to sequences of:

  or normal/abrupt program termination

Construction of Most General Context

- Extend LPJava by nondeterministic expression ($E ::= \ldots \mid \text{nde}$)
- Evaluation of *nde* leads to sequences of:
  - creation of new objects (of public class type)

or normal/abrupt program termination

Construction of Most General Context

- Extend LPJava by nondeterministic expression ($E ::= \ldots \mid \mathrm{nde}$)
- Evaluation of *nde* leads to sequences of:
  - creation of new objects (of public class type)
  - cross-border method call or return using exposed objects

  or normal/abrupt program termination

Construction of Most General Context

- Extend LPJava by nondeterministic expression ($E ::= \ldots \mid \text{nde}$)
- Evaluation of *nde* leads to sequences of:
    - creation of new objects (of public class type)
    - cross-border method call or return using exposed objects

  or normal/abrupt program termination
- Augment configurations

Construction of Most General Context

- Extend LPJava by nondeterministic expression ($E ::= \ldots \mid \mathrm{nde}$)
- Evaluation of *nde* leads to sequences of:
  - creation of new objects (of public class type)
  - cross-border method call or return using exposed objects

  or normal/abrupt program termination
- Augment configurations
  - Tag objects whether they have been created by code of K or X

Construction of Most General Context

- Extend LPJava by nondeterministic expression ($E ::= \ldots \mid \text{nde}$)
- Evaluation of *nde* leads to sequences of:
  - creation of new objects (of public class type)
  - cross-border method call or return using exposed objects

  or normal/abrupt program termination
- Augment configurations
  - Tag objects whether they have been created by code of K or X
  - Tag objects whether they have been exposed / internal

Construction of Most General Context

- Extend LPJava by nondeterministic expression ($E ::= \ldots \mid \text{nde}$)
- Evaluation of *nde* leads to sequences of:
    - creation of new objects (of public class type)
    - cross-border method call or return using exposed objects

  or normal/abrupt program termination
- Augment configurations
    - Tag objects whether they have been created by code of *K* or *X*
    - Tag objects whether they have been exposed / internal
- Construction of program context κ_X is solely based on library *X*:

```
public class Main { public void main() { nde; } }

public class Cell_1 extends Cell {}
public class Cell_2 extends Cell { public void set(Val nv) { nde; } }
public class Cell_3 extends Cell { public Val get() { return nde; } }
public class Cell_4 extends Cell {
  public void set(Val nv) { nde; }
  public Val get() { return nde; }
}
...
```

Full abstraction

1. Traces capture all relevant information about the behavior
2. κ_X represents exactly all possible program contexts for X

Theorem (Full abstraction)

Y is backward compatible with X if and only if $Traces(\kappa_X X) \subseteq Traces(\kappa_Y Y)$.

- More details in Welsch/Poetzsch-Heffter. *A fully abstract trace-based semantics for reasoning about backward compatibility of class libraries* (Science of Computer Prog. 92, pp. 129-161, Oct. 2014)
- Related work:
  - Java Jr. (Jeffrey/Rathke 2005)
  - Reasoning about class behavior (Koutavas/Wand 2007)
  - Ownership confinement ensures representation independence for object-oriented programs (Banerjee/Naumann 2005)
  - ...

Proving backward compatibility

Proving backward compatibility and equivalence

Two Approaches

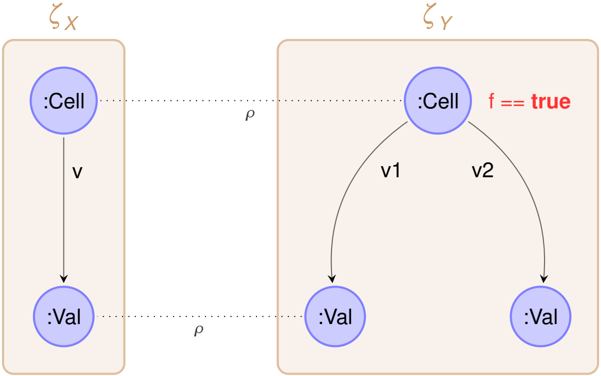
1. Simulation proof based on abstract models:
    - Develop (or mine) abstract models of the libraries
    - Prove models correct vs. code (Hoare-logic)
    - Prove equivalence on the model level
    - First experiences using ITP

Proving backward compatibility and equivalence

Two Approaches

1. Simulation proof based on abstract models:
    - Develop (or mine) abstract models of the libraries
    - Prove models correct vs. code (Hoare-logic)
    - Prove equivalence on the model level
    - First experiences using ITP
2. Simulation proof based on coupling relation:
    - Coupling relation between runtime configs of $\kappa_X X$ and $\kappa_X Y$
    - Prove simulation for all possible input messages
    - Automatic checking based on an embedding into Boogie (FTfJP'12)

Coupling relation for Cell example



Specification:

invariant forall old Cell o1, **new** Cell o2 :: o1 ~ o2
==> **if** o2.f then o1.c ~ o2.c1 **else** o1.c ~ o2.c2;

Checking technique and tool

BCVerifier:

- Specification language for coupling invariants

Checking technique and tool

BCVerifier:

- Specification language for coupling invariants
- Check source compatibility

Checking technique and tool

BCVerifier:

- Specification language for coupling invariants
- Check source compatibility
- Generate verification conditions for Boogie to prove that coupling invariant is a simulation:
  - Corresponding inputs lead to corresponding outputs
  - Coupling invariant preserved by interactions

Checking technique and tool

BCVerifier:

- Specification language for coupling invariants
- Check source compatibility
- Generate verification conditions for Boogie to prove that coupling invariant is a simulation:
  - Corresponding inputs lead to corresponding outputs
  - Coupling invariant preserved by interactions

Remark:

Needs a bit of twisting as Boogie is not designed for simulations

Coupling in Boogie

Coupling invariant:

```
function Inv(heap1:Heap, heap2:Heap, related:Bij) returns (bool) {
  ( forall o1,o2:Ref :: related[o1,o2] && heap2[o2,f]
     ==> RelNull(heap1[o1,c], heap2[o2,c1], related) ) &&
  ( forall o1,o2:Ref :: related[o1,o2] && !heap2[o2,f]
     ==> RelNull(heap1[o1, c], heap2[o2,c2], related) )
}
function RelNull(r1:Ref, r2:Ref, related:Bij) returns (bool) {
  (r1 == null && r2 == null) || (r1 != null && r2 != null && related[r1,r2])
}
```

allows to verify Cell example

BCVerifier example: OneOfLoop

```
public class C {
    public int m(int n){
        int x = 0;
        for(int i=0; i<n; i++){
            x += i;
        }
        return x;
    }
}
```

```
public class C {
    public int m(int n){
        int x = 0;
        int i = 1;
        while(i<n){
            x += i;
            i++;
        }
        return x;
    }
}
```

BCVerifier example: OneOfLoop

```
1  public class C {
2      public int m(int n){
3          int x = 0;
4          for(int i=0; i<n; i++){
5              x += i;
6          }
7          return x;
8      }
9  }
```

```
1  public class C {
2      public int m(int n){
3          int x = 0;
4          int i = 1;
5          while(i<n){
6              x += i;
7              i++;
8          }
9          return x;
10     }
11 }
```

```
local place inLoop1 = line 5 of old C when i > 0;
local place inLoop2 = line 6 of new C;

local invariant at(inLoop1) && at(inLoop2) ==>
      eval(inLoop1, n) == eval(inLoop2, n)
   && eval(inLoop1, x) == eval(inLoop2, x)
   && eval(inLoop1, i) == eval(inLoop2, i);
```

The END

Conclusions:

- Principles of proving backward compatibility

The END

Conclusions:

- Principles of proving backward compatibility
- Backward compatibility needs no specs: can be transferred to behavioral subtyping

The END

Conclusions:

- Principles of proving backward compatibility
- Backward compatibility needs no specs: can be transferred to behavioral subtyping
- Abstract semantics of packages/components

The END

Conclusions:

- Principles of proving backward compatibility
- Backward compatibility needs no specs: can be transferred to behavioral subtyping
- Abstract semantics of packages/components

The END

Conclusions:

- Principles of proving backward compatibility
- Backward compatibility needs no specs: can be transferred to behavioral subtyping
- Abstract semantics of packages/components

Aspects for the future:

- Design languages such that source compatibility is automatically checkable
- Develop refined forms of backward compatibility

Questions?