Aspects & Modular Reasoning

Robby Findler
University of Chicago

Observable Equivalence

- When can you replace one expression with another?

1+1 **2**

Observable Equivalence

- When can you replace one expression with another?

For any context, **C**

$$\mathbf{C[1+1]} \quad \cong \quad \mathbf{C[2]}$$

That is, both expressions produce the same results, same errors, same output, same everything, no matter where you put them in the program.

Observable Equivalence

- When can you replace one expression with another?

$$\texttt{"1+1"} \not\cong \texttt{"2"}$$

Observable Equivalence

- When can you replace one expression with another?

For any expression context, **C**

$$\texttt{C[1+1]} \cong \texttt{C[2]}$$

That is, both expressions produce the same results, same errors, same output, same everything, no matter where you put them in the program.

Only consider well-typed programs.

Observable Equivalence

- When can you replace one expression with another?
- Compiler optimizations
- Refactoring
- To start, focus on Java

Observable Equivalence

- When can you replace one expression with another?

For any statement context, **C**

```
class D {
  int m() {
    return 1;
  }
}
```

```
C[D o = new D();        ≟        C[D o = new D();
  o.m()]                           1]
```

Observable Equivalence

- When can you replace one expression with another?

For any statement context, **C**

```
class D {
  int m() {
    return 1;
  }
}
```

```
C[D o = new D();          ≅      C[D o = new D();
  o.m()]                            1]
```

Observable Equivalence

- When can you replace one expression with another?

For any method context, **C**

```
class D {
  int m() {
    return 1;
  }
}
```

```
C[int n(D o) {
    if (o==null)
      return 1;
    return o.m();
  }]
```

$\stackrel{?}{\equiv}$

```
C[int n(D o) {
    return 1;
  }]
```

Observable Equivalence

- When can you replace one expression with another?

For any method context, **C**

```
class D {
  int m() {
    return 1;
  }
}
```

```
C[int n(D o) {
    if (o==null)
      return 1;
    return o.m();
  }]
```

$\not\equiv$

```
C[int n(D o) {
    return 1;
  }]
```

Subtyping

Subtyping disrupts observable equivalence

- Subtypes can change behavior (harder to reason)
- Subtypes allow dynamic binding (more expressive)
- Subtypes shape must match (preserve some equations)

Observable Equivalence

- When can you replace one expression with another?

For any expression context, **C**

```
class A {
  int x;
  A(int x) { this.x = x; }
}
```

C[new A(1).x] $\stackrel{?}{\equiv}$ **C[1]**

Observable Equivalence

- When can you replace one expression with another?

For any expression context, **C**

```
class A {
  int x;
  A(int x) { this.x = x; }
}
```

C[new A(1).x] $\cong$ **C[1]**

What if we add (AspectJ) aspects?

Observable Equivalence

- When can you replace one expression with another?

For any expression context, **C**

```
class A {
  int x;
  A(int x) { this.x = x; }
}
```

C[new A(1).x] $\overset{?}{\cong}$ **C[1]**

Observable Equivalence

- When can you replace one expression with another?

For any expression context, **C**

```
class A {
  int x;
  A(int x) { this.x = x; }
}
```

C[new A(1).x] $\not\equiv$ **C[1]**

```
aspect Get {
  int around() : get(int A.x) {
    return 77;
  }
}
```

Observable Equivalence

- When can you replace one expression with another?

For any expression context, **C**

```
class A {
  int x;
  A(int x) { this.x = x; }
}
```

```
C[new A(1).x]        ≢        C[1]
```

```
aspect Set {
  void around(int i) : set(int A.x) &&
                       args(i) {
    proceed(88);
  }
}
```

Observable Equivalence

- When can you replace one expression with another?

For any expression context, **C**

```
class A {
  int x;
  A(int x) { this.x = x; }
}
```

C[new A(1).x] $\not\equiv$ **C[1]**

```
aspect Init {
  Object around(int x) : call(A.new(int)) &&
                                       args(x) {
    return proceed(99);
  }
}
```

Aspects distrupt observable equivalence

- Introduce many new observations
- More quantifiction => fewer equations
- More oblivious => fewer equations
- Are there any equations left?

Try to restrict aspects

Method contract checking

- Pre- and post-condition checking
- Blaming caller and method, resp.

Aspect contract checking

- `around` advice on methods only (for simplicity)
- blame caller for pre-condition
- At proceed, blame aspect for pre-condition
- After proceed, blame method for post-condition
- After advice, blame aspect for post-condition

Claim: if a program has no contract violations, adding contract checked aspects can

- produce no violations, or
- blame an aspect.

Never blames existing code.

Claim: if a program has no contract violations, adding contract checked aspects can

- produce no violations, or
- blame an aspect.

Never blames existing code.

Wrong

Claim: if a program has no contract violations, adding contract checked aspects can

- produce no violations, or
- blame an aspect.

Never blames existing code.

Additional restrictions

- No advice on contract checking code itself
- Advice on public methods only
- Original program bug-free (no Java-only context can force a contract violation)

Claim: if a program has no contract violations, adding contract checked aspects can

- produce no violations, or
- blame an aspect.

Never blames existing code.

Wrong

Additional restrictions

- No advice on contract checking code itself
- Advice on public methods only
- Original program bug-free (no Java-only context can force a contract violation)

Pre-conditions alone do not guarantee post-conditions

The internals of the method also matter

```
class PosNegSet {
  IList posnums = new Null();
  IList negnums = new Null();

  int removePos() { ... }
    @pre { !(posnums instanceof Null) }
    @post { @ret > 0 }

  int removeNeg() { ... }
    @pre { !(negnums instanceof Null) }
    @post { @ret < 0 }

  void add(int x) {
    if (x < 0)
      negnums = new Cons(x, negnums);
    else
      posnums = new Cons(x, posnums); }
    @pre { x != 0 }
    @post { !(posnums instanceof Null &&
              negnums instanceof Null) }
}
```

```
aspect Get {
  Cons around(PosNegSet o, int i, IList l) :
      call(Cons.new(int,IList)) &&
      args(i,l) && this(o) &&
      withincode(void PosNegSet.add(int)) {

        return proceed(o, i, o.posnums);

  }
}


  void add(int x) {
    if (x < 0)
      negnums = new Cons(x, posnums);
    else
      posnums = new Cons(x, posnums);
  }
```