ThisJava:
An Extension of Java with Recursive Types

Hyunik Na
PL Lab@KAIST

ROSAEC 8th Workshop
2012.7.25~28

equals() Method

```
class Point {
  int x, y;

  boolean equals( Point other ) {
    return ( x == other.x &&
             y == other.y );
  }
}
```

```
class ColorPoint extends Point {
  RGB color;

  boolean equals( ColorPoint other ) {
    return ( x == other.x &&
             y == other.y &&
             color == other.color );
  }
}
```

equals() Method

```
class Point {
  int x, y;

  boolean equals( Object o ) {
    if ( o instanceof Point ) {
      Point other = ( Point ) o;
      return ( other.canEqual( this )
               && x == other.x
               && y == other.y );
    } else {
      return false;
    }
  }

  boolean canEqual( Object o ) {
    return ( o instanceof Point );
  }
}
```

```
class ColorPoint extends Point {
  RGB color;

  boolean equals( Object o ) {
    if ( o instanceof ColorPoint ) {
      ColorPoint other = ( ColorPoint ) o;
      return ( other.canEqual( this )
               && x == other.x
               && y == other.y
               && color == other.color );
    } else {
      return false;
    }
  }

  boolean canEqual( Object o ) {
    return ( o instanceof ColorPoint );
  }
}
```

equals() Method

```
class Point {
  int x, y;

  boolean equals( Object o ) {
    if ( o instanceof Point ) {
      Point other = ( Point ) o;
      return ( other.canEqual( this )
               && x == other.x
               && y == other.y );
    } else {
      return false;
    }
  }

  boolean canEqual( Object o ) {
    return ( o instanceof Point );
  }
}
```

```
class ColorPoint extends Point {
  RGB color;

  boolean equals( Object o ) {
    if ( o instanceof ColorPoint ) {
      ColorPoint other = ( ColorPoint ) o;
      return ( other.canEqual( this )
               && x == other.x
               && y == other.y
               && color == other.color );
    } else {
      return false;
    }
  }

  boolean canEqual( Object o ) {
    return ( o instanceof ColorPoint );
  }
}
```

```
class Object {
  ...
  boolean equals( Object o ) { ... }
}
```

`equals()` Method

```
class Object {
  ...
  boolean equals( ? o ) { ... }
}
```

`clone()` Method

```
C c = ...;

C c2 = c.clone();        // rejected
C c3 = (C) c.clone();    // accepted
```

clone() Method

```
C c = ...;

C c2 = c.clone();        // rejected
C c3 = (C) c.clone();    // accepted
```

```
class Object {
  ...
  Object clone() { ... }
}
```

clone() Method

```
C c = ...;

C c2 = c.clone();         // rejected
C c3 = (C) c.clone();     // accepted
```

```
class Object {
  ...
  Object clone() { ... }
}
```

```
class C {
  ...
  C clone() { ... }
}

  C c = ...;
  C c2 = c.clone();        // accepted
```

clone() Method

```
class Object {
  ...
  ? clone() { ... }
}
```

Recursive Types to Express Type Equality

P = { x: int, y: int, equals: P→boolean, clone: ()→P }

C = { x: int, y: int, equals: C→boolean, clone: ()→C, color: RGB }

Recursive Types to Express Type Equality

P = { x: int, y: int, equals: P→boolean, clone: ()→P }
C = { x: int, y: int, equals: C→boolean, clone: ()→C, color: RGB }

P = μX.{ x: int, y: int, equals: X→boolean, clone: ()→X }
C = μX.{ x: int, y: int, equals: X→boolean, clone: ()→X, color: RGB }

Recursive Types to Express Type Equality

P = { x: int, y: int, equals: P→boolean, clone: ()→P }
C = { x: int, y: int, equals: C→boolean, clone: ()→C, color: RGB }

P = μX.{ x: int, y: int, equals: X→boolean, clone: ()→X }
C = μX.{ x: int, y: int, equals: X→boolean, clone: ()→X, color: RGB }

```
class Point {
  int x, y;

  boolean equals( This other ) {
    return ( x == other.x &&
             y == other.y );
  }

  This clone() { ... }
}
```

```
class ColorPoint extends Point {
  RGB color;

  boolean equals( This other ) {
    return ( x == other.x &&
             y == other.y &&
             color == other.color );
  }

  This clone() { ... }
}
```

Problem of Recursive Types

- A recursive type breaks "subtyping-by-subclassing" when the recursion variable appears on a parameter type

- For example,

  $\mu X.\{$ x: int, y: int, equals: X→boolean, color: RGB $\}$

  is not a subtype of

  $\mu X.\{$ x: int, y: int, equals: X→boolean $\}$

Problem of Recursive Types

- A recursive type breaks "subtyping-by-subclassing" when the recursion variable appears on a parameter type

- For example,

  $\mu X.\{\ x: \text{int},\ y: \text{int},\ \text{equals}: X \rightarrow \text{boolean},\ \text{color}: \text{RGB}\ \}$

  is not a subtype of

  $\mu X.\{\ x: \text{int},\ y: \text{int},\ \text{equals}: X \rightarrow \text{boolean}\ \}$

  (cf. Cardelli, 1984, 1986)

Record Types:

$$\frac{S_i \leq T_i \quad \text{for} \quad i \in 1..n}{\{a_i : S_i\ ^{i \in 1..n..m}\} \ \leq\ \{a_i : T_i\ ^{i \in 1..n}\}}$$

Recursive Types:

$$\frac{\Sigma, X \leq Y \vdash S \leq T}{\Sigma \vdash \mu X.S \ \leq\ \mu Y.T}$$

If We Ignore It And Just Use Recursive Types, …

```
boolean compare( Point p, Point q ) {
  return p.equals(q);
}

...
  compare( new ColorPoint(1,2,BLUE), new Point(1,2) );  // type safety broken!
```

If We Ignore It And Just Use Recursive Types, …

```
boolean compare( Point p, Point q ) {
  return p.equals(q);
}

...
  compare( new ColorPoint(1,2,BLUE), new Point(1,2) );  // type safety broken!
```

Is inheritance(subclassing) subtyping?

So, Binary Method Problem

- No recursive types unless we abandon subtyping-by-subclassing

So, Binary Method Problem

- No recursive types unless we abandon subtyping-by-subclassing
- However, the latter is more valuable than the former

So, Binary Method Problem

- No recursive types unless we abandon subtyping-by-subclassing
- However, the latter is more valuable than the former
- So, no recursive types in object-oriented languages

So, Binary Method Problem

- No recursive types unless we abandon subtyping-by-subclassing
- However, the latter is more valuable than the former
- So, no recursive types in object-oriented languages
- Imprecise static typing for equal types
  ➔ Essence of Binary Method Problem

Our Solution: We Can Have Both!

- How to reject the problematic code?

```
boolean compare( Point p, Point q ) {
  return p.equals(q);
}

...
  compare( new ColorPoint(1,2,BLUE), new Point(1,2) );
```

Our Solution: We Can Have Both!

- How to reject the problematic code?

```
boolean compare( Point p, Point q ) {
  return p.equals(q);
}

...
  compare( new ColorPoint(1,2,BLUE), new Point(1,2) );
```

- Traditionally, `p.equals(q)` is allowed, because
  "q's *compile-time type* is a subtype of p's *compile-time type*"

- We reject it, because
  "p and q's *run-time classes* may be different"

Our Solution: We Can Have Both!

- How to reject the problematic code?

```
boolean compare( Point p, Point q ) {
  return p.equals(q);
}

...
  compare( new ColorPoint(1,2,BLUE), new Point(1,2) );
```

- Traditionally, `p.equals(q)` is allowed, because
  "q's *compile-time type* is a subtype of p's *compile-time type*"

- We reject it, because
  "p and q's *run-time classes* may be different"

- That is, **a modified notion of This type**

Is Our Type System Too Restrictive?

- When is an invocation of equals() method allowed?
  - When it is certain at compile-time that
    the run-time classes of the receiver and argument match exactly.

- For example,

```
new Point(1,2).equals( new Point(3,4) );
```

Exact Class Types and Named Wildcards

```
boolean compare( #Point p, #Point q ) {
  return p.equals(q);
}

boolean compare( #ColorPoint p, #ColorPoint q ) {
  return p.equals(q);
}

boolean compare( #ColorPointX p, #ColorPointX q ) {
  return p.equals(q);
}

boolean compare( #ColorPointXX p, #ColorPointXX q ) {
  return p.equals(q);
}

...
```

```
#Point p = new ColorPoint(1,2,BLUE);  // rejected
#Point p = new Point(1,2);            // accepted
```

Exact Class Types and Named Wildcards

```
boolean compare( #Point p, #Point q ) {
  return p.equals(q);
}

boolean compare( #ColorPoint p, #ColorPoint q ) {
  return p.equals(q);
}

boolean compare( #ColorPointX p, #ColorPointX q ) {
  return p.equals(q);
}

boolean compare( #ColorPointXX p, #ColorPointXX q ) {
  return p.equals(q);
}

...
```

⇓

```
boolean compareGeneric( Point</X/> p, Point</X/> q ) {
  return p.equals(q);
}
```

Exact Type Capture

```
Point p;    // local variable
...
p.equals(p);
```

Exact Type Capture

```
Point p;    // local variable
...
p.equals(p);
```

- Declared type of p (`Point`) is internally converted to `Point</X/>` where X is a fresh exact type variable
- Similar to Java's wildcard capture (unpacking of an existential type)
- Cannot capture exact types of a non-final field and an array element
  - due to multi-threading

Exact Type Inference

```
Point p, q;
if (...) {
  p = new Point(1,2);
  q = new Point(3,4);
} else {
  p = new ColorPoint(1,2,BLUE);
  q = new ColorPoint(3,4,RED);
}

p.equals(q);
```

Exact Type Inference

```
Point p, q;
if (...) {
  p = new Point(1,2);
  q = new Point(3,4);
} else {
  p = new ColorPoint(1,2,BLUE);
  q = new ColorPoint(3,4,RED);
}

p.equals(q);
```

- Exact type inference is a data-flow analysis
  - based on reaching-definition analysis
- It should consider each flow separately
- Restricted within a method, and ignited only when exact type matching is necessary

Run-time Type Recover Using ‘classesmatch’

```
boolean compare( Point p, Point q ) {
  return p.equals(q);      // rejected
}
```

Run-time Type Recover Using ‘classesmatch’

```
boolean compare( Point p, Point q ) {
  return p.equals(q);     // rejected
}
```

⇓

```
boolean compare( Point p, Point q ) {
  classesmatch ( p, q ) {
    return p.equals(q);   // allowed
  } else {
    return false;
  }
}
```

Run-time Type Recover Using 'classesmatch'

```
boolean compare( Point p, Point q ) {
  return p.equals(q);     // rejected
}
```

⇓

```
boolean compare( Point p, Point q ) {
  classesmatch ( p, q ) {
    return p.equals(q);   // allowed
  } else {
    return false;
  }
}
```

- Run-time type recover is used in many languages
  - e.g. type casting, pattern matching, typecase

ThisJava

- A conservative extension of Java with the features described so far (+ virtual constructors)

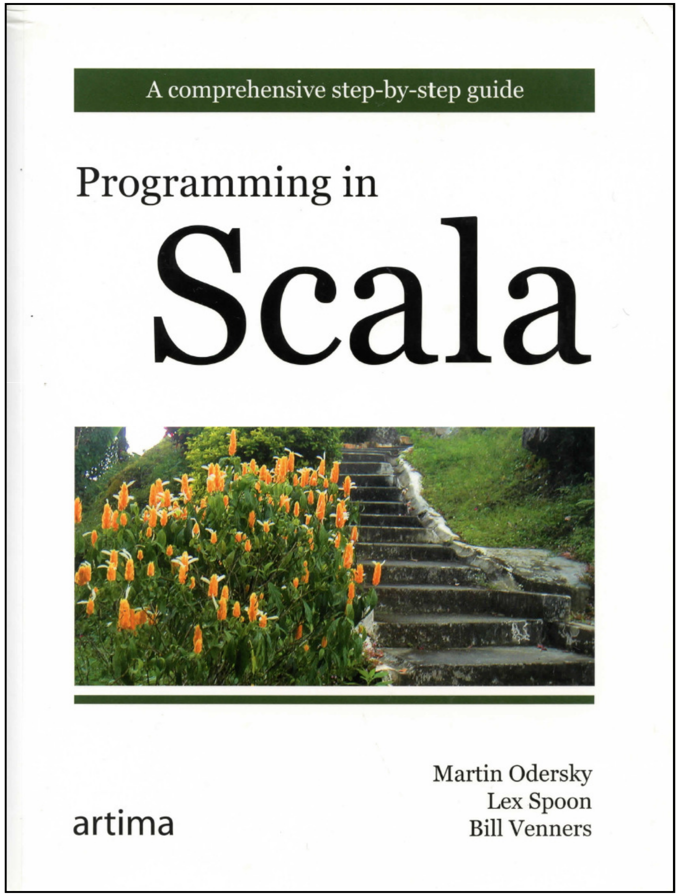
- Implementation
  - Extending JastAddJ compiler
    - JastAddJ 1.4/1.5 Frontend/Backend → ThisJava 1.4/1.5 Frontend/Backend

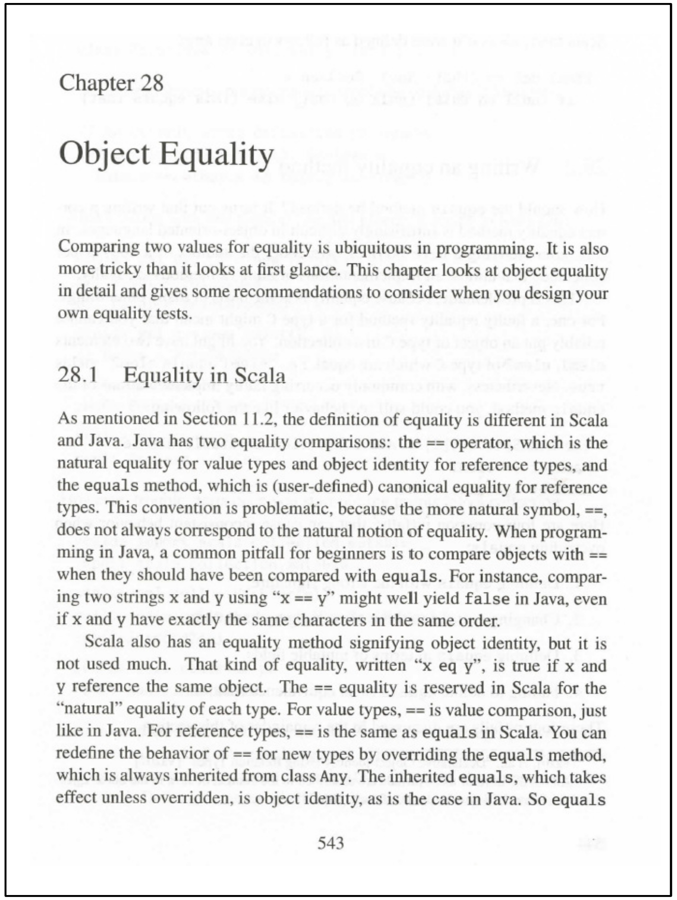
- Status
  - Implementation done
  - It well compiles Java class library (7636 .java files) obtained from OpenJDK 1.6
  - Execution is as expected for small test programs
    - Execution tests for big benchmark programs is to be done
  - Added features seem to work well with existing various Java features
    - mutable variables and arrays, nested classes, multi-threading, generics, etc

Conclusion

- Run-time class matching seems to be the most desirable notion of This type
  - Type safety is restored without further restriction
  - “Inheritance is subtyping” with the notion
- For flexible use of This type, various typing scheme is necessary
  - named wildcards
  - exact type capture
  - exact type inference
  - `classesmatch` construct
- We can have both recursive types and subtyping-by-inheritance in a non-toy object-oriented language

Reference on
“How to Write a Good Equals Method”




Chapter 28

Object Equality

Comparing two values for equality is ubiquitous in programming. It is also more tricky than it looks at first glance. This chapter looks at object equality in detail and gives some recommendations to consider when you design your own equality tests.

28.1 Equality in Scala

As mentioned in Section 11.2, the definition of equality is different in Scala and Java. Java has two equality comparisons: the == operator, which is the natural equality for value types and object identity for reference types, and the `equals` method, which is (user-defined) canonical equality for reference types. This convention is problematic, because the more natural symbol, ==, does not always correspond to the natural notion of equality. When programming in Java, a common pitfall for beginners is to compare objects with == when they should have been compared with `equals`. For instance, comparing two strings x and y using “x == y” might well yield `false` in Java, even if x and y have exactly the same characters in the same order.

Scala also has an equality method signifying object identity, but it is not used much. That kind of equality, written “`x eq y`”, is true if `x` and `y` reference the same object. The == equality is reserved in Scala for the “natural” equality of each type. For value types, == is value comparison, just like in Java. For reference types, == is the same as `equals` in Scala. You can redefine the behavior of == for new types by overriding the `equals` method, which is always inherited from class Any. The inherited `equals`, which takes effect unless overridden, is object identity, as is the case in Java. So `equals`

543