For More Usable *This* Type

(*This* Type as a Hidden Type Variable)

Hyunik Na

(Joint Work with Sukyoung Ryu)

PL Lab@KIAST

2011.1.6~9

ROSAEC 5th Workshop

Overview

- Introduction to *This* type and *Exact* type
  - with ‘equals()’ method example
- Problems and existing solution
- Our solution
- Conclusion

Writing '`equals`()' Method Is Not Easy

```
class Point {
  int x, y;

  boolean equals( ? ) {
    ?
  }
}

class ColorPoint extends Point {
  RGB color;

  boolean equals( ? ) {
    ?
  }
}
```

Writing '`equals`()' Method Is Not Easy

First Naive Version of '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 );
  }
}
```

Writing '`equals`()' Method Is Not Easy

Pitfall 1 : Unwanted Overloading

```
Stack<Object> st = new Stack<Object>();
Point p1 = new Point( 1, 2 );
Point p2 = new Point( 1, 2 );

st.add( p1 );
st.contains( p2 );  // returns false !
```

Writing '`equals`()' Method Is Not Easy

Pitfall 1 : Unwanted Overloading - Solution

```
class Point {
  int x, y;

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

```
class ColorPoint extends Point {
  RGB color;

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

Writing '`equals`()' Method Is Not Easy

Second Version, But Still Incomplete

```
class Point {
  int x, y;

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

```
class ColorPoint extends Point {
  RGB color;

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

Writing '`equals`()' Method Is Not Easy :
Pitfall 2 : Not Transitive Equality Relation

```
Point p1      = new Point( 1, 2 );
ColorPoint p2 = new ColorPoint( 1, 2, Red );
ColorPoint p3 = new ColorPoint( 1, 2, Blue );

p1.equals( p2 );    // returns true
p1.equals( p3 );    // returns true
p2.equals( p3 );    // returns false !
```

Writing '`equals`()' Method Is Not Easy

Pitfall 2 : Not Transitive Equality Relation - Solution

```
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 );
  }
}
```

Writing '`equals`()' Method Is Not Easy
Final Version

```
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 );
  }
}
```

Possible Solutions

- Multi-methods (Dynamic Overloading)
  - Pros: more flexible, can handle heterogeneous collections
  - Cons: runtime overhead and exception
  - Care needed for binary methods
    - Best match is not the solution, e.g. intransitive equality relation again...
- *This* Type and Exact Type
  - Cons: less flexible
  - Pros: no runtime overhead

‘equals()’ Using *This* Type and Exact Type

```
class Object {
  boolean equals( @This other ) { ... }
  ...
}

class Point {
  int x, y;

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

class ColorPoint extends Point {
  RGB color;

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

- ***This*** : type of ***this***
  - changes its meaning along inheritance
- ***@This*** : exact ***This***
  - disallows proper subtypes

Typing Rules about Exact Types

- '**`new C(...)`**' has **`@C`** type

```
@C c = new C();
```

- @C is compatible to C, but not vice versa

```
C c   = new C();  // OK
@C c2 = c;        // Not OK
```

- Binary methods can be called on only exactly typed expressions

```
@Point p = ...
p.equals(...); // OK
```

but

```
Point p = ...
p.equals(p2)  // Not OK
```

Why Is the 3rd Rule Required?

- Otherwise, following problematic code

```
Point p1 = new ColorPoint(...);
@Point p2 = new Point(...);
p1.equals(p2);
    // tries to access p2's color field!
```

Problems

- Severely restricts dynamic dispatch of binary methods
  - Cannot type clearly type safe binary methods invocations such as follows

```
Point p = ...
p.equals(p);
```

and

```
LinkedNode n = ...
n.linkTo( n.next().next() );
```

- Cannot type factory methods definitions which have ***@This*** as the return type

```
class Point {
    @This clone() {  return ( new ? );  }
}
```

Existing Solution (Saito & Igarashi, SAC 2009)

- Local exactization
  - Locally capture the exact type using **`exact`** statement

```
Point p = ...
exact p as x, X in {
    x.equals(x);        // locally, x: @X, X<:Point
}
```

- Nonheritable methods

```
class Point {
    nonheritable @This clone() {
        // Under nonheritable, @Point <: @This
        return ( new Point(...) );  }}
```

Our Solution

- Implicit Exact Type Capture (IETC)
  - Exact type capture needs not boilerplate code
  - It's done by the type checker's internal process

```
Point p = ...
p.equals(p);  // just well-typed as is
```

- Virtual Constructor (*This*-constructor)

```
class Point {
    @This clone() {
        return ( new This(...) );
    }
}
```

Notions beneath IETC

- **`class C ...  == class C⟨This<:C⟨This⟩⟩ ...`**
  - **`C`** is a type constructor mapping (hidden) *This* type variable
  - Only exact types instantiate *This*
    - *This* is exact without @ notation
- **`@C`** == fixed point of **`C`**
  - that is, **`@C = C⟨@C⟩`**
- **`C == C⟨?⟩`**
  - ? denotes wildcard
  - Exact type capture becomes wildcard capture,
    which can be done by a type checker's internal process

How is '`p.equals(p)`' type checked

```
Point p = ...
  // p has type Point<?> which is captures by Point<X>
  // for a fresh type variable X.

p.equals(p);
  // 1. 'p.equals' has type '[X/This](This -> boolean)
  //     = (X -> boolean)'.
  // 2. Argument p has type Point<X> as described above,
  //     which is coompatible to X, since X = Point<X>.
  //     Therefore, well-typed!
```

Virtual Constructor (This-constructor)

- Not a new idea, but insufficiently explored
- Especially, following problem

```
class Foo {
  Foo( T fld ) {...}
  This genFactory() {
    ... new This( f ) ...
  }
}

class Goo {
  Goo( T fld, S fld2 ) {...}
  // inherited genFactory() becomes ill-typed
  // because of the addtional field 'fld2'.
}
```

- How to fill missing values for additional fields
  - Possibly from '*default values*' or from '*this*'

Conclusion

- *This* type becomes more usable with IETC and Virtual Constructor

- Todo
  - Writing paper - typing rules and manuscript. half done
  - Type soundness proof
  - Implementation – ThisJ
    - using a Java extension framework, e.g. PolyGlot

Thank you

- Q & A