Binary Methods Support Using Self Type Idiom in Fortress

Hyunik Na
(Joint work with Sukyoung Ryu and Victor Luchangco)
PL Lab@KIAST

2010.8.25~28

ROSAEC 4th Workshop

What Is a Binary Method?

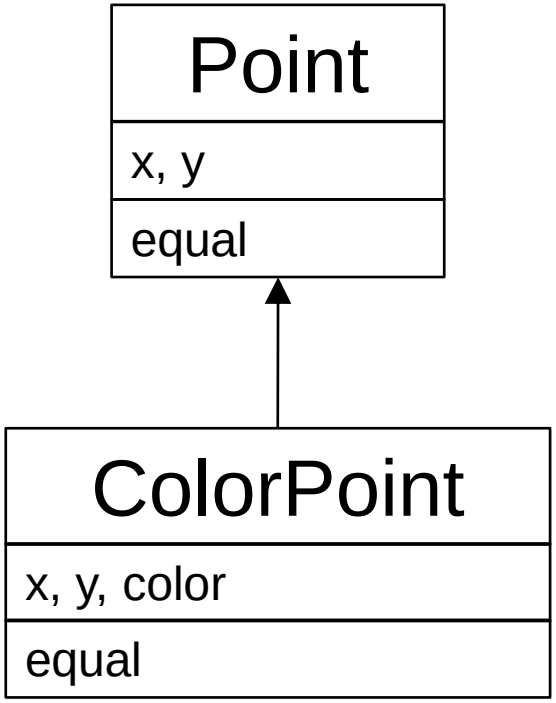
- Binary operation with two arguments of the same type
  - +, -, *, /
  - =, <
  - Set union and inclusion

- In OOP Lang, methods of the same parameter types
  - n1.add(n2), ~ n1 + n2
  - this.equals(that) ~ this == that
  - set1.includes(set2) ~ set1 INCL set2

What is the Binary Method Problem?

- At first, how can we represent a binary method?
  - How can we enforce two operands to have the same type?
  - Or, how can we represent the type of *this* (or *self*)?
- Given above, it prevents subtyping
  - Subclassing (class inheritance) does not mean subtyping in the presence of binary methods

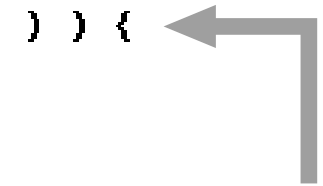
Binary Method Example




```
class Point {
    int x, y;

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

class ColorPoint extends Point {
    RGB color;

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

Problematic Code

```
void breakit( Point p ) {
    Point fixedPt = new Point( 3, 4 );
        if ( p.equal( fixedPt ) ) {
            ...
    }
}
```



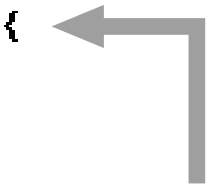
What if p is a ColorPoint?

For example,

```
breakit( new ColorPoint( 2, 7, YELLOW ) )
```

Problematic Code

```
void breakit( Point p ) {
    Point fixedPt = new Point( 3, 4 );
        if ( p.equal( fixedPt ) ) {
            ...
    }
}
```



What if p is a ColorPoint?

For example,

```
breakit( new ColorPoint( 2, 7, YELLOW ) )
```

⇒ Unhopefully,
Point.equal() is called

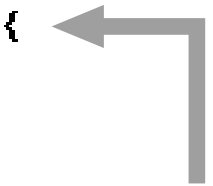
To Match and Override, Use '*This*' Keyword

- *This* : type of *this*
  - Type of the class in which it resides

```
class Point {
    int x, y;

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

class ColorPoint extends Point {
    RGB color;

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

Problematic Code, Again

```
void breakit( Point p ) {
    Point fixedPt = new Point( 3, 4 );
        if ( p.equal( fixedPt ) ) {
            ...
    }
}
```



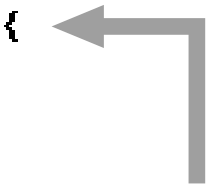
What if p is a ColorPoint?

For example,

```
breakit( new ColorPoint( 2, 7, YELLOW ) )
```

Problematic Code, Again

```
void breakit( Point p ) {
    Point fixedPt = new Point( 3, 4 );
        if ( p.equal( fixedPt ) ) {
            ...
    }
}
```



What if p is a ColorPoint?

For example,

```
breakit( new ColorPoint( 2, 7, YELLOW ) )
```

⇒ Runtime Error

Approach in Fortress: Self Type Idiom (1/2)

- Self Type Idiom (STI)
  - Letting a type variable represent the Self Type of a trait or object using the *comprises clause* and appropriate *typing rules*

- *Comprises* clause
  - Restricts downwards inheritance relations
    - contrasting upwards ones of *extends*
  - For example

```
trait Shape extends Drawable comprises {Circle, Triangle, Rectangle}
trait Integer extends Number comprises {Even, Odd}
trait List⟦E⟧ comprises {Empty⟦E⟧, Cons⟦E⟧}
```

Approach in Fortress: Self Type Idiom (2/2)

- Typing *self* in the presence of *comprises* clause

  $\texttt{trait}\ P\ \texttt{comprises}\ \{\texttt{C}_1, \texttt{C}_2, \ldots, \texttt{C}_n\}$

  $\Rightarrow$ *self* : P $\cap$ ($\cup$ C_i), C_i <: P

- Type of a type variable appearing in a *comprises* clause

  $\texttt{trait}\ P[\![\ldots, X\ \texttt{extends}\ N, \ldots]\!]\ \texttt{comprises}\ \{\ldots, X, \ldots\}$

  $\Rightarrow$ X <: P[[...]] $\cap$ N ; X is a subtype of P[[...]]

- In particular, when X is the only one in the *comprises* clause (Self Type Idiom),

  $\texttt{trait}\ P[\![\ldots, X\ \texttt{extends}\ N, \ldots]\!]\ \texttt{comprises}\ \{X\}$

  $\Rightarrow$ *self* : P[[...]] $\cap$ X $\equiv$ X ; X becomes the type of *self !*

Using Self Type Idiom

```
trait Eq⟦X⟧ comprises {X}
    equal(other : X): Boolean
end

trait Point⟦Y⟧ extends Eq⟦Y⟧ comprises {Y}
    x(): ℤ32
    y(): ℤ32
    equal(other : Y): Boolean =
        (x() = other.x()) ∧ (y() = other.y())
end

trait ColorPoint⟦Z⟧ extends Point⟦Z⟧ comprises {Z}
    x(): ℤ32
    y(): ℤ32
    color(): RGB
    equal(other : Z): Boolean =
        (x() = other.x()) ∧ (y() = other.y()) ∧ (color() = other.color())
end

object ColorPointObj(...) extends ColorPoint⟦ColorPointObj⟧
```

Problematic Code, Revisited

```
void breakit( Point p ) {
    Point fixedPt = new Point( 3, 4 );
        if ( p.equal( fixedPt ) ) {
            ...
    }
}
breakit( new ColorPoint( 2, 7, YELLOW ) )
```

Code like above does not pass Fortress type system

Which type can replace '?' in the following Fortress code?

$$
\begin{array}{l}
\mathit{breakit}(\,p\colon ?\,)\colon () = \texttt{do} \\
\quad \mathit{fixedPt}\colon \mathrm{PointObj} = \mathrm{PointObj}(3,4) \\
\quad \texttt{if } p.\mathit{equal}(\,\mathit{fixedPt}\,) \texttt{ then} \\
\qquad \ldots \\
\texttt{end} \\
\mathit{breakit}\big(\,\mathrm{ColorPointObj}(\,2,7,\texttt{YELLOW}\,)\,\big)
\end{array}
$$

Other Applications of STI : Simulating Scala's Self Type Annotation

- Scala's Self Type Annotation
  - Represents a kind of relation between classes.
    "A is not B, but A depends on B." , or more specifically,
    "Any class that extends A must also extend B."
  - For example, for a trade company

```
trait Commission { this : Trade => ... }
trait Tax { this : Trade => ... }
```

```
trait A { this : B => ... }                    // in Scala
```

```
trait A⟦ X extends B ⟧ comprises X ...         // in Fortress
```

Other Applications of STI : Neater Solution for Expression Problem (1/2)

```
interface Exp<V extends AleVisitor> {
    void accept(V v);
}
class Lit<V extends AleVisitor> implements Exp<V> {
    public int value;
    Lit(int v) { value = v; }
    public void accept(V v) { v.visitLit(this); }
}
class Add<V extends AleVisitor> implements Exp<V> {
    public Exp<V> left, right;
    Add(Exp<V> l, Exp<V> r) { left = l; right = r; }
    public void accept(V v) { v.visitAdd(this, v); }
}

interface AleVisitor {
    <V extends AleVisitor> void visitLit(Lit<V> lit);
    <V extends AleVisitor> void visitAdd(Add<V> add, V self);
}
class AlePrint implements AleVisitor {
    public <V extends AleVisitor> void visitLit(Lit<V> lit) {
        System.out.print(lit.value);
    }
    public <V extends AleVisitor> void visitAdd(Add<V> add, V self) {
        add.left.accept(self); System.out.print(' +' ); add.right.accept(self);
    }
}
```

Other Applications of STI : Neater Solution for Expression Problem (1/2)

```
interface Exp<V extends AleVisitor> {
    void accept(V v);
}
class Lit<V extends AleVisitor> implements Exp<V> {
    public int value;
    Lit(int v) { value = v; }
    public void accept(V v) { v.visitLit(this); }
}
class Add<V extends AleVisitor> implements Exp<V> {
    public Exp<V> left, right;
    Add(Exp<V> l, Exp<V> r) { left = l; right = r; }
    public void accept(V v) { v.visitAdd(this, v); }
}

interface AleVisitor {
    <V extends AleVisitor> void visitLit(Lit<V> lit);
    <V extends AleVisitor> void visitAdd(Add<V> add, V self);
}
class AlePrint implements AleVisitor {
    public <V extends AleVisitor> void visitLit(Lit<V> lit) {
        System.out.print(lit.value);
    }
    public <V extends AleVisitor> void visitAdd(Add<V> add, V self) {
        add.left.accept(self); System.out.print(' + '); add.right.accept(self);
    }
}
```

???

Other Applications of STI : Neater Solution for Expression Problem (1/2)

```
interface Exp<V extends AleVisitor> {
    void accept(V v);
}
class Lit<V extends AleVisitor> implements Exp<V> {
    public int value;
    Lit(int v) { value = v; }
    public void accept(V v) { v.visitLit(this); }
}
class Add<V extends AleVisitor> implements Exp<V> {
    public Exp<V> left, right;
    Add(Exp<V> l, Exp<V> r) { left = l; right = r; }
    public void accept(V v) { v.visitAdd(this, v); }
}

interface AleVisitor {
    <V extends AleVisitor> void visitLit(Lit<V> lit);
    <V extends AleVisitor> void visitAdd(Add<V> add, V self);
}
class AlePrint implements AleVisitor {
    public <V extends AleVisitor> void visitLit(Lit<V> lit) {
        System.out.print(lit.value);
    }
    public <V extends AleVisitor> void visitAdd(Add<V> add, V self) {
        add.left.accept(self); System.out.print(' + '); add.right.accept(self);
    }
}
```

???

`add.right.accept(this);`

⇒ Type error !

Other Applications of STI : Neater Solution for Expression Problem (2/2)

```
trait Exp⟦V extends AleVisitor⟦V⟧⟧
    accept(v:V):()
end

object Lit⟦V extends AleVisitor⟦V⟧⟧(val:ℤ32) extends Exp⟦V⟧
    accept(v:V):() = v.visitLit(self)
end

object Add⟦V extends AleVisitor⟦V⟧⟧(left:Exp⟦V⟧, right:Exp⟦V⟧) extends Exp⟦V⟧
    accept(v:V):() = v.visitAdd(self)
end

trait AleVisitor⟦X⟧ comprises {X}
    visitLit(lit:Lit⟦X⟧):()
    visitAdd(add:Add⟦X⟧):()
end

trait AlePrint⟦Y⟧ extends AleVisitor⟦Y⟧ comprises {Y}
    visitLit(lit:Lit⟦Y⟧) = print(lit.val)
    visitAdd(add:Add⟦Y⟧) = do
        add.left.accept(self); print("+"); add.right.accept(self);
    end
end
```

Using STI

Progress and Future Work

- Progress
  - Basic idea was proposed by Fortress team
  - Formalization
    - Constructing the formal calculi : typing rules
    - Proving type safety (half done)

- Future work
  - Less restrictive STI
  - Further application and meaning of STI
  - Interactions with other language concepts
    - (dynamic) overloading
    - functional methods of Fortress
    - ...

References

- On Binary Methods
  - Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce
  - TAPOS `95
- The Expression Problem Revisited
  - Mads Torgersen
  - ECOOP `04

Thank you

Q & A

Other Applications of STI : Neater Solution for Expression Problem (2/2)

```
interface Exp<V extends AleVisitor<V> > {
    void accept(V v);
}
class Lit<V extends AleVisitor<V> > implements Exp<V> {
    public int value;
    Lit(int v) { value = v; }
    public void accept(V v) { v.visitLit(this); }
}
class Add<V extends AleVisitor<V> > implements Exp<V> {
    public Exp<V> left, right;
    Add(Exp<V> l, Exp<V> r) { left = l; right = r; }
    public void accept(V v) { v.visitAdd(this, v); }
}

interface AleVisitor<X> comprises X {
    void visitLit(Lit<X> lit);
    void visitAdd(Add<X> add);
}
class AlePrint<Y> implements AleVisitor<Y> comprises Y {
    public void visitLit(Lit<Y> lit) {
        System.out.print(lit.value);
    }
    public void visitAdd(Add<Y> add) {
        add.left.accept(this); System.out.print(' +' ); add.right.accept(this);
    }
}
```